Python 之旅

Python 是一门面向对象，解释型的高级程序设计语言，它的语法非常简洁、优雅，而这也是 Python 的一些设计哲学。Python 自带了很完善的库，涵盖了数据库，网络，文件处理，GUI 等方方面面，通过这些库，我们可以…

目　录

致谢

当前文档 《Python 之旅》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-09-17。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：ethan-funny https://github.com/ethan-funny/explore-python

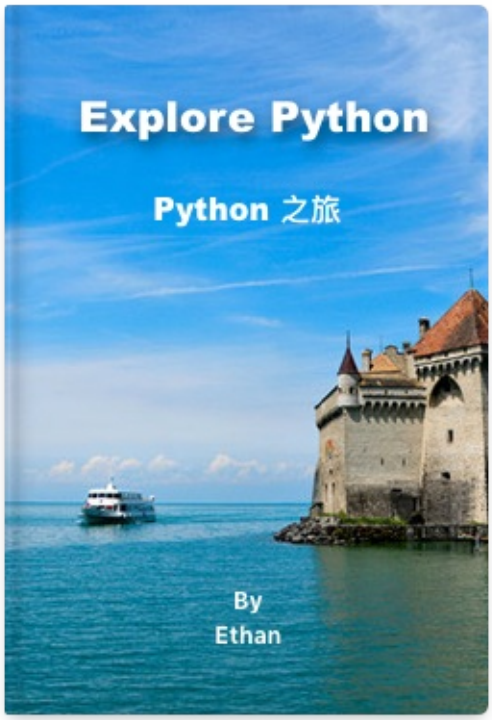
文档地址：http://www.bookstack.cn/books/explore-python

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Summary



Python 之旅

Python 简介

Python 诞生于 1989 年的圣诞期间，由 Guido van Rossum 开发而成，目前 Guido 仍然是 Python 的主要开发者，主导着 Python 的发展方向，Python 社区经常称呼他为『仁慈的独裁者』。

Python 是一门面向对象，解释型的高级程序设计语言，它的语法非常简洁、优雅，而这也是 Python 的一些设计哲学。Python 自带了很完善的库，涵盖了数据库，网络，文件处理，GUI 等方方面面，通过这些库，我们可以比较快速地解决一些棘手问题，也可以将其作为基础库，开发出一些高级库。

目前 Python 在大部分领域都占有一席之地，比如 Web 开发，机器学习，科学计算等。不少大型网站都是使用 Python 作为后台开发语言的，比如 YouTube、Pinterest、国内的豆瓣和知乎等。

另外，有不少知名的机器学习库也是使用 Python 开发的，比如，scikit-learn 是一个强大的机器学习库，PyTorch 是一个成熟的深度学习库。

当然了，Python 也有一些缺点。Python 经常被人们吐槽的一点就是：运行速度慢，和 C/C++ 相比非常慢。但是，除了像视频高清解码等计算密集型任务对运行速度有较高的要求外，在大部分时候，我们可能并不需要非常快的运行速度。比如，一个程序使用 C 来实现，运行时间只需 0.01 秒，而使用 Python 来实现，需要 0.1 秒，虽然 Python 的运行时间是 C 的 10 倍，显然很慢，但对我

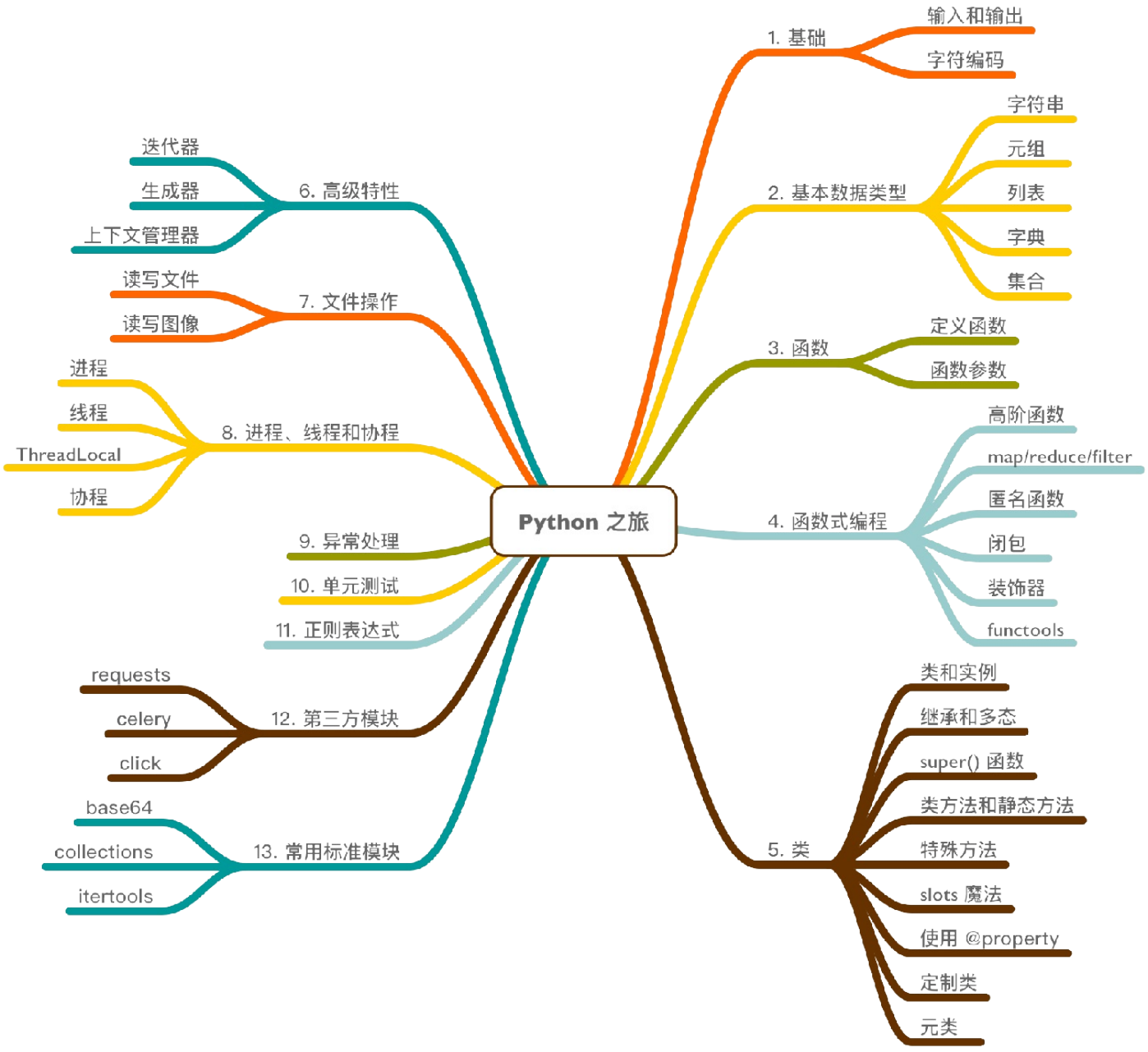
们而言，这压根不是问题。

关于本书

本书是我学习和使用 Python 的总结。在学习和使用 Python 的过程中，我作了不少笔记，并对一些笔记进行了加工和完善，发表在博客上。随着笔记的增加，我就萌生了写一本书的想法，希望能比较系统地总结相关知识，巩固自己的知识体系，而不是停留在『感觉好像懂了』的状态中。

有了想法之后，接下来就要开始写了。当然，从产生想法到付诸实践还是纠结了一段时间，毕竟，作笔记和写书很不一样啊。思想斗争过后，我下定决心要把它写出来。

首先，我参考一些相关的书籍，作了一个基础的思维导图，如下：



接下来，就要开始写作了，这也是最艰难的一关。

我没有按照从头到尾的顺序写，而是从最感兴趣的知识点入手，比如函数式编程、类的使用等等。就这样，一点一点地写，实在不想写了，就先搁置一下，过两天继续写。

我在写作的过程中，给自己提了一个要求：尽量深入浅出，条理清晰。至于是否达到了，希望读者们多多批评指正，并给我提意见和建议。

本书的每章基本上都是独立的，读者可以挑选感兴趣的章节进行阅读。目前本书有 15 个章节：

- 第 1 章：介绍一些基础知识，包括 Python 中的输入和输出，字符编码。
- 第 2 章：介绍常用数据类型，比如字符串、列表和字典等。
- 第 3 章：介绍函数的定义和函数参数魔法。
- 第 4 章：介绍 Python 中的函数式编程，包括匿名函数、闭包和装饰器等。
- 第 5 章：介绍 Python 中类的使用，包括类方法、静态方法、super 和元类的使用等。
- 第 6 章：介绍 Python 中的高级特性，比如生成器，上下文管理器。
- 第 7 章：介绍文件和目录操作，os 的使用。
- 第 8 章：介绍使用 Python 处理进程、线程和协程。
- 第 9 章：异常处理。
- 第 10 章：单元测试。
- 第 11 章：正则表达式，re 模块的使用。
- 第 12 章：HTTP 服务，requests 模块的使用。
- 第 13 章：一些标准模块的使用，比如 argparse、collections 和 datetime 等。
- 第 14 章：一些第三方模块的使用。
- 第 15 章：结束语。

本书的编码环境：

- Python 版本以 2.7 为主，同时也会指出在 Python3 中的相应变化
- 操作系统使用 macOS，代码结果，尤其是内存地址等由于运行环境的不同会存在差异

本书将会持续进行修订和更新，读者如果遇到问题，请及时向我反馈，我会在第一时间加以解决。

声明

更新记录

时间	说明
2017-01-03	发布版本 v1.0
2019-02-09	fix typo

支持我

如果你觉得本书对你有所帮助，不妨请我喝杯咖啡，感谢支持！

基础

本章主要介绍两个方面的内容：

- 字符编码
- 输入和输出

其中，字符编码的概念很重要，不管你用的是 Python2 还是 Python3，亦或是 C++ 等其他编程语言，希望读者厘清这个概念，当遇到 UnicodeEncodeError 和 UnicodeDecodeError 时才能从容应对，而不是到处查找资料。

字符编码

字符编码是计算机编程中不可回避的问题，不管你用 Python2 还是 Python3，亦或是 C++, Java 等，我都觉得非常有必要厘清计算机中的字符编码概念。本文主要分以下几个部分介绍：

- 基本概念
- 常见字符编码简介
- Python 的默认编码
- Python2 中的字符类型
- UnicodeEncodeError & UnicodeDecodeError 根源

基本概念

- 字符（Character）

在电脑和电信领域中，字符是一个信息单位，它是各种文字和符号的总称，包括各国家文字、标点符号、图形符号、数字等。比如，一个汉字，一个英文字母，一个标点符号等都是一个字符。

- 字符集（Character set）

字符集是字符的集合。字符集的种类较多，每个字符集包含的字符个数也不同。比如，常见的字符集有 ASCII 字符集、GB2312 字符集、Unicode 字符集等，其中，ASCII 字符集共有 128 个字符，包含可显示字符（比如英文大小写字符、阿拉伯数字）和控制字符（比如空格键、回车键）；GB2312 字符集是中国国家标准的简体中文字符集，包含简化汉字、一般符号、数字等；Unicode 字符集则包含了世界各国语言中使用到的所有字符，

- 字符编码（Character encoding）

字符编码，是指对于字符集中的字符，将其编码为特定的二进制数，以便计算机处理。常见的字符编码有 ASCII 编码，UTF-8 编码，GBK 编码等。一般而言，字符集和字符编码往往被认为是同义的概念，比如，对于字符集 ASCII，它除了有「字符的集合」这层含义外，同时也包含了「编码」的含义，也就是说，**ASCII** 既表示了字符集也表示了对应的字符编码。

下面我们用一个表格做下总结：

概念	概念描述	举例
字符	一个信息单位，各种文字和符号的总称	‘中’, ‘a’, ‘1’, ‘$’, ‘￥’, …
字符集	字符的集合	ASCII 字符集, GB2312 字符集, Unicode 字符集
字符编码	将字符集中的字符，编码为特定的二进制数	ASCII 编码，GB2312 编码，Unicode 编码
	计算机中存储数据的单元，一个 8 位（bit）的	

	二进制数	

常见字符编码简介

常见的字符编码有 ASCII 编码，GBK 编码，Unicode 编码和 UTF-8 编码等等。这里，我们主要介绍 ASCII、Unicode 和 UTF-8。

ASCII

计算机是在美国诞生的，人家用的是英语，而在英语的世界里，不过就是英文字母，数字和一些普通符号的组合而已。

在 20 世纪 60 年代，美国制定了一套字符编码方案，规定了英文字母，数字和一些普通符号跟二进制的转换关系，被称为 ASCII (American Standard Code for Information Interchange，美国信息互换标准编码) 码。

比如，大写英文字母 A 的二进制表示是 01000001（十进制 65），小写英文字母 a 的二进制表示是 01100001 （十进制 97），空格 SPACE 的二进制表示是 00100000（十进制 32）。

Unicode

ASCII 码只规定了 128 个字符的编码，这在美国是够用的。可是，计算机后来传到了欧洲，亚洲，乃至世界各地，而世界各国的语言几乎是完全不一样的，用 ASCII 码来表示其他语言是远远不够的，所以，不同的国家和地区又制定了自己的编码方案，比如中国大陆的 GB2312 编码 和 GBK 编码等，日本的 Shift_JIS 编码等等。

虽然各个国家和地区可以制定自己的编码方案，但不同国家和地区的计算机在数据传输的过程中就会出现各种各样的乱码（mojibake），这无疑是个灾难。

怎么办？想法也很简单，就是将全世界所有的语言统一成一套编码方案，这套编码方案就叫 Unicode，它为每种语言的每个字符设定了独一无二的二进制编码，这样就可以跨语言，跨平台进行文本处理了，是不是很棒！

Unicode 1.0 版诞生于 1991 年 10 月，至今它仍在不断增修，每个新版本都会加入更多新的字符，目前最新的版本为 2016 年 6 月 21 日公布的 9.0.0。

Unicode 标准使用十六进制数字，而且在数字前面加上前缀 `U+` ，比如，大写字母「A」的 unicode 编码为 `U+0041` ，汉字「严」的 unicode 编码为 `U+4E25` 。更多的符号对应表，可以查询 unicode.org，或者专门的汉字对应表。

UTF-8

Unicode 看起来已经很完美了，实现了大一统。但是，Unicode 却存在一个很大的问题：资源浪费。

为什么这么说呢？原来，Unicode 为了能表示世界各国所有文字，一开始用两个字节，后来发现两个字节不够用，又用了四个字节。比如，汉字「严」的 unicode 编码是十六进制数 `4E25`，转换成二进制有十五位，即 100111000100101，因此至少需要两个字节才能表示这个汉字，但是对于其他的字符，就可能需要三个或四个字节，甚至更多。

这时，问题就来了，如果以前的 ASCII 字符集也用这种方式来表示，那岂不是很浪费存储空间。比如，大写字母「A」的二进制编码为 01000001，它只需要一个字节就够了，如果 unicode 统一使用三个字节或四个字节来表示字符，那「A」的二进制编码的前面几个字节就都是 `0`，这是很浪费存储空间的。

为了解决这个问题，在 Unicode 的基础上，人们实现了 UTF-16, UTF-32 和 UTF-8。下面只说一下 UTF-8。

UTF-8 (8-bit Unicode Transformation Format) 是一种针对 Unicode 的可变长度字符编码，它使用一到四个字节来表示字符，例如，ASCII 字符继续使用一个字节编码，阿拉伯文、希腊文等使用两个字节编码，常用汉字使用三个字节编码，等等。

因此，我们说，**UTF-8 是 Unicode 的实现方式之一**，其他实现方式还包括 UTF-16（字符用两个或四个字节表示）和 UTF-32（字符用四个字节表示）。

Python 的默认编码

Python2 的默认编码是 ascii，Python3 的默认编码是 utf-8，可以通过下面的方式获取：

- Python2

```
Python 2.7.11 (default, Feb 24 2016, 10:48:05)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.getdefaultencoding()
'ascii'
```

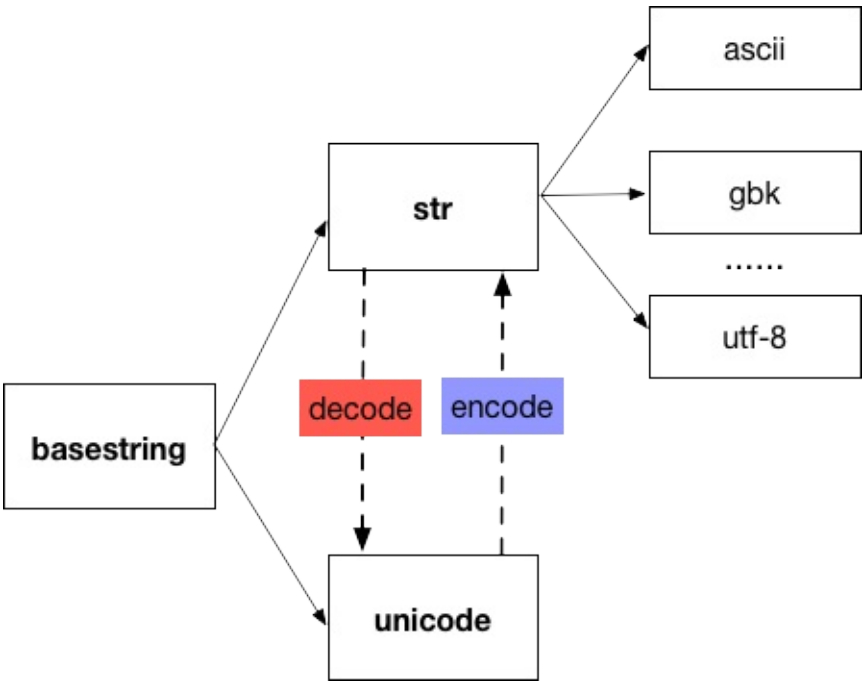
- Python3

```
Python 3.5.2 (default, Jun 29 2016, 13:43:58)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> sys.getdefaultencoding()
'utf-8'
```

Python2 中的字符类型

Python2 中有两种和字符串相关的类型：str 和 unicode，它们的父类是 basestring。其中，str 类型的字符串有多种编码方式，默认是 ascii，还有 gbk，utf-8 等，unicode 类型的字符串使用 `u'...'` 的形式来表示，下面的图展示了 str 和 unicode 之间的关系：



两种字符串的相互转换概括如下：

- 把 UTF-8 编码表示的字符串 ‘xxx’ 转换为 Unicode 字符串 u’xxx’ 用 `decode('utf-8')` 方法：

```
>>> '中文'.decode('utf-8')
u'\u4e2d\u6587'
```

- 把 u’xxx’ 转换为 UTF-8 编码的 ‘xxx’ 用 `encode('utf-8')` 方法：

```
>>> u'中文'.encode('utf-8')
'\xe4\xb8\xad\xe6\x96\x87'
```

UnicodeEncodeError & UnicodeDecodeError 根源

用 Python2 编写程序的时候经常会遇到 UnicodeEncodeError 和 UnicodeDecodeError，它们出现的根源就是如果代码里面混合使用了 **str** 类型和 **unicode** 类型的字符串，**Python** 会默认使用 **ascii** 编码尝试对 **unicode** 类型的字符串编码 **(encode)**，或对 **str** 类型的字符串解码 **(decode)**，这时就很可能出现上述错误。

下面有两个常见的场景，我们最好牢牢记住：

- 在进行同时包含 str 类型和 unicode 类型的字符串操作时，Python2 一律都把 str 解码（decode）成 unicode 再运算，这时就很容易出现 UnicodeDecodeError。

让我们看看例子：

```
>>> s = '你好'    # str 类型，utf-8 编码
>>> u = u'世界'   # unicode 类型
>>> s + u        # 会进行隐式转换，即 s.decode('ascii') + u
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe4 in position 0: ordinal
not in range(128)
```

为了避免出错，我们就需要显示指定使用 ‘utf-8’ 进行解码，如下：

```
>>> s = '你好'                # str 类型，utf-8 编码
>>> u = u'世界'
>>>
>>> s.decode('utf-8') + u     # 显示指定 'utf-8' 进行转换
u'\u4f60\u597d\u4e16\u754c'   # 注意这不是错误，这是 unicode 字符串
```

- 如果函数或类等对象接收的是 str 类型的字符串，但你传的是 unicode，Python2 会默认使用 ascii 将其编码成 str 类型再运算，这时就很容易出现 UnicodeEncodeError。

让我们看看例子：

```
>>> u_str = u'你好'
>>> str(u_str)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
ordinal not in range(128)
```

在上面的代码中，u_str 是一个 unicode 类型的字符串，由于 `str()` 的参数只能是 str 类型，此时 Python 会试图使用 ascii 将其编码成 ascii，也就是：

```
u_str.encode('ascii')   // u_str 是 unicode 字符串
```

上面将 unicode 类型的中文使用 ascii 编码转，肯定会出错。

再看一个使用 raw_input 的例子，注意 raw_input 只接收 str 类型的字符串：

```
>>> name = raw_input('input your name: ')
input your name: ethan
>>> name
'ethan'

>>> name = raw_input('输入你的姓名：')
输入你的姓名： 小明
>>> name
'\xe5\xb0\x8f\xe6\x98\x8e'
>>> type(name)
<type 'str'>

>>> name = raw_input(u'输入你的姓名: ')   # 会试图使用 u'输入你的姓名'.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-5: ordinal not in range(128)

>>> name = raw_input(u'输入你的姓名: '.encode('utf-8')) #可以，但此时 name 不是 unicode 类型
输入你的姓名: 小明
>>> name
'\xe5\xb0\x8f\xe6\x98\x8e'
>>> type(name)
<type 'str'>

>>> name = raw_input(u'输入你的姓名: '.encode('utf-8')).decode('utf-8') # 推荐
输入你的姓名:  小明
>>> name
u'\u5c0f\u660e'
>>> type(name)
<type 'unicode'>
```

再看一个重定向的例子：

```
1. hello = u'你好'
2. print hello
```

将上面的代码保存到文件 `hello.py` ，在终端执行 `python hello.py` 可以正常打印，但是如果将其重定向到文件 `python hello.py > result` 会发现 `UnicodeEncodeError` 。

这是因为：输出到控制台时，print 使用的是控制台的默认编码，而重定向到文件时，print 就不知道使用什么编码了，于是就使用了默认编码 ascii 导致出现编码错误。

应该改成如下：

```
1. hello = u'你好'
2. print hello.encode('utf-8')
```

这样执行 `python hello.py > result` 就没有问题。

小结

- UTF-8 是一种针对 Unicode 的可变长度字符编码，它是 Unicode 的实现方式之一。
- Unicode 字符集有多种编码标准，比如 UTF-8，UTF-7，UTF-16。
- 在进行同时包含 str 类型和 unicode 类型的字符串操作时，Python2 一律都把 str 解码（decode）成 unicode 再运算。
- 如果函数或类等对象接收的是 str 类型的字符串，但你传的是 unicode，Python2 会默认使用 ascii 将其编码成 str 类型再运算。

参考资料

- 字符 - 维基百科，自由的百科全书
- UTF-8 - 维基百科，自由的百科全书
- 字符，字节和编码 - Characters, Bytes And Encoding
- 字符编码笔记：ASCII，Unicode和UTF-8 - 阮一峰的网络日志
- 字符串和编码 - 廖雪峰的官方网站
- python - Dangers of sys.setdefaultencoding('utf-8') - Stack Overflow

输入和输出

Python2 提供了 `input` ， `raw_input` ， `print` 等用于输入输出，但在 Python3 中发生了一些改变， `raw_input` 已经没有了， `input` 的用法发生了变化， `print` 也从原来的语句变成了一个函数。本文将对这两种情况进行介绍。

输入

- 首先看 Python2 中的 `raw_input` ，它的用法如下：

```
raw_input(prompt)
```

其中，prompt 表示输入提示。 `raw_input` 会读取控制台的输入，并返回字符串类型。

让我们看几个例子：

```
>>> name = raw_input('please enter your name: ')
please enter your name: ethan     # 输入一个字符串
>>> name
'ethan'
>>> type(name)
<type 'str'>
>>>
>>> num = raw_input('please enter your id: ')
please enter your id: 12345       # 输入一个数值
>>> num
'12345'
>>> type(num)
<type 'str'>
>>>
>>> sum = raw_input('please enter a+b: ')
please enter a+b: 3+6             # 输入一个表达式
>>> sum
'3+6'
>>> type(sum)
<type 'str'>
```

可以看到，不管我们输入一个字符串、数值还是表达式， `raw_input` 都直接返回一个字符串。

- 现在看一下 Pythn2 中的 `input` 。

`input` 的用法跟 `raw_input` 类似，形式如下：

```
input(prompt)
```

事实上， `input` 本质上是使用 `raw_input` 实现的，如下：

```
def input(prompt):
    return (eval(raw_input(prompt)))
```

也就说，调用 `input` 实际上是通过调用 `raw_input` 再调用 `eval` 函数实现的。

这里的 `eval` 通常用来执行一个字符串表达式，并返回表达式的值，它的基本用法如下：

```
>>> eval('1+2')
3
>>> a = 1
>>> eval('a+9')
10
```

现在，让我们看看 `input` 的用法：

```
>>> name = input('please input your name: ')
please input your name: ethan         # 输入字符串如果没加引号会出错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'ethan' is not defined
>>>
>>> name = input('please input your name: ')
please input your name: 'ethan'       # 添加引号
>>> name
'ethan'
>>>
>>> num = input('please input your id: ')
please input your id: 12345           # 输入数值
>>> num                               # 注意返回的是数值类型，而不是字符串
12345
>>> type(num)
<type 'int'>
>>>
>>> sum = input('please enter a+b: ')  # 输入数字表达式，会对表达式求值
please enter a+b: 3+6
```

```
>>> sum
9
>>> type(sum)
<type 'int'>
>>>
>>> sum = input('please enter a+b: ')    # 输入字符串表达式，会字符串进行运算
please enter a+b: '3'+'6'
>>> sum
'36'
```

可以看到，使用 `input` 的时候，如果输入的是字符串，必须使用引号把它们括起来；如果输入的是数值类型，则返回的也是数值类型；如果输入的是表达式，会对表达式进行运算。

- 再来看一下 Python3 中的 `input` 。

事实上，Python3 中的 `input` 就是 Python2 中的 `raw_input` ，也就是说，原 Python2 中的 `raw_input` 被重命名为 `input` 了。那如果我们想使用原 Python2 的 `input` 功能呢？你可以这样做：

```
eval(input())
```

也就是说，手动添加 `eval` 函数。

输出

Python2 中的 `print` 是一个语句（statement），而 Python3 中的 `print` 是一个函数。

Python2 中的 print

- 简单输出

使用 `print` 最简单的方式就是直接在 `print` 后面加上数字、字符串、列表等对象，比如：

```
# Python 2.7.11 (default, Feb 24 2016, 10:48:05)
>>> print 123
123
>>> print 'abc'
abc
>>> x = 10
>>> print x
10
```

```
>>> d = {'a': 1, 'b': 2}
>>> print d
{'a': 1, 'b': 2}
>>>
>>> print(123)
123
>>> print('abc')
abc
>>> print(x)
10
>>> print(d)
{'a': 1, 'b': 2}
```

在 Python2 中，使用 `print` 时可以加括号，也可以不加括号。

- 格式化输出

有时，我们需要对输出进行一些格式化，比如限制小数的精度等，直接看几个例子：

```
>>> s = 'hello'
>>> l = len(s)
>>> print('the length of %s is %d' % (s, l))
the length of hello is 5
>>>
>>> pi = 3.14159
>>> print('%10.3f' % pi)     # 字段宽度 10，精度 3
     3.142
>>> print('%010.3f' % pi)    # 用 0 填充空白
000003.142
>>> print('%+f' % pi)        # 显示正负号
+3.141590
```

- 换行输出

print 默认是换行输出的，如果不想换行，可以在末尾加上一个 `,’，比如：

```
>>> for i in range(0, 3):
...     print i
...
0
1
2
>>> for i in range(0, 3):
```

```
...     print i,            # 加了 ,
...
0 1 2                       # 注意会加上一个空格
```

Python3 中的 print

在 Python3 中使用 print 跟 Python2 差别不大，不过要注意的是在 Python3 中使用 print 必须加括号，否则会抛出 SyntaxError。

另外，如果不想 print 换行输出，可以参考下面的方式：

```
>>> for i in range(0, 3):
...     print(i)
...
0
1
2
>>> for i in range(0, 3):
...     print(i, end='')    # 加上一个 end 参数
...
012
```

小结

- 在 Python2 中，`raw_input` 会读取控制台的输入，并返回字符串类型。
- 在 Python2 中，如无特殊要求建议使用 raw_input() 来与用户交互。
- 在 Python3 中，使用 `input` 处理输入，如有特殊要求，可以考虑加上 `eval` 。

参考资料

- python - What’s the difference between raw_input() and input() in python3.x? - Stack Overflow

常用数据类型

在介绍 Python 的常用数据类型之前，我们先看看 Python 最基本的数据结构 - 序列（**sequence**）。

序列的一个特点就是根据索引（index，即元素的位置）来获取序列中的元素，第一个索引是 0，第二个索引是 1，以此类推。

所有序列类型都可以进行某些通用的操作，比如：

- 索引（indexing）
- 分片（sliceing）
- 迭代（iteration）
- 加（adding）
- 乘（multiplying）

除了上面这些，我们还可以检查某个元素是否属于序列的成员，计算序列的长度等等。

说完序列，我们接下来看看 Python 中常用的数据类型，如下：

- 列表（list）
- 元组（tuple）
- 字符串（string）
- 字典（dict）
- 集合（set）

其中，列表、元组和字符串都属于序列类型，它们可以进行某些通用的操作，比如索引、分片等；字典属于映射类型，每个元素由键（key）和值（value）构成；集合是一种特殊的类型，它所包含的元素是不重复的。

通用的序列操作

索引

序列中的元素可以通过索引获取，索引从 0 开始。看看下面的例子：

```
1.  >>> nums = [1, 2, 3, 4, 5]   # 列表
2.  >>> nums[0]
3.  1
4.  >>> nums[1]
```

```
2
>>> nums[-1]                # 索引 -1 表示最后一个元素
5
>>> s = 'abcdef'            # 字符串
>>> s[0]
'a'
>>> s[1]
'b'
>>>
>>> a = (1, 2, 3)           # 元组
>>> a[0]
1
>>> a[1]
2
```

注意到，`-1` 则代表序列的最后一个元素，`-2` 代表倒数第二个元素，以此类推。

分片

索引用于获取序列中的单个元素，而分片则用于获取序列的部分元素。分片操作需要提供两个索引作为边界，中间用冒号相隔，比如：

```
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> numbers[0:2]                # 列表分片
[1, 2]
>>> numbers[2:5]
[3, 4, 5]
>>> s = 'hello, world'          # 字符串分片
>>> s[0:5]
'hello'
>>> a = (2, 4, 6, 8, 10)        # 元组分片
>>> a[2:4]
(6, 8)
```

这里需要特别注意的是，分片有两个索引，第 **1** 个索引的元素是包含在内的，而第 **2** 个元素的索引则不包含在内，也就是说，`numbers[2:5]` 获取的是 `numbers[2]`, `numbers[3]`, `numbers[4]`，没有包括 `numbers[5]`。

下面列举使用分片的一些技巧。

- 访问最后几个元素

假设需要访问序列的最后 3 个元素，我们当然可以像下面这样做：

```
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> numbers[3:6]
[4, 5, 6]
```

有没有更简洁的方法呢？想到可以使用负数形式的索引，你可能会尝试这样做：

```
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> numbers[-3:-1]      # 实际取出的是 numbers[-3], numbers[-2]
[4, 5]
>>> numbers[-3:0]       # 左边索引的元素比右边索引出现得晚，返回空序列
[]
```

上面的两种使用方式并不能正确获取序列的最后 3 个元素，Python 提供了一个捷径：

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8]
>>> numbers[-3:]
[6, 7, 8]
>>> numbers[5:]
[6, 7, 8]
```

也就是说，如果希望分片包含最后一个元素，可将第 2 个索引置为空。

如果要复制整个序列，可以将两个索引都置为空：

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8]
>>> nums = numbers[:]
>>> nums
[1, 2, 3, 4, 5, 6, 7, 8]
```

- 使用步长

使用分片的时候，步长默认是 1，即逐个访问，我们也可以自定义步长，比如：

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8]
>>> numbers[0:4]
[1, 2, 3, 4]
>>> numbers[0:4:1]    # 步长为 1，不写也可以，默认为 1
[1, 2, 3, 4]
>>> numbers[0:4:2]    # 步长为 2，取出 numbers[0], numbers[2]
[1, 3]
```

```
>>> numbers[::3]      # 等价于 numbers[0:8:3], 取出索引为 0, 3, 6 的元素
[1, 4, 7]
```

另外，步长也可以是负数，表示从右到左取元素：

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8]
>>> numbers[0:4:-1]
[]
>>> numbers[4:0:-1]       # 取出索引为 4, 3, 2, 1 的元素
[5, 4, 3, 2]
>>> numbers[4:0:-2]       # 取出索引为 4, 2 的元素
[5, 3]
>>> numbers[::-1]         # 从右到左取出所有元素
[8, 7, 6, 5, 4, 3, 2, 1]
>>> numbers[::-2]         # 取出索引为 7, 5, 3, 1 的元素
[8, 6, 4, 2]
>>> numbers[6::-2]        # 取出索引为 6, 4, 2, 0 的元素
[7, 5, 3, 1]
>>> numbers[:6:-2]        # 取出索引为 7 的元素
[8]
```

这里总结一下使用分片操作的一些方法，分片的使用形式是：

```
# 左索引:右索引:步长
left_index:right_index:step
```

要牢牢记住的是：

- 左边索引的元素包括在结果之中，右边索引的元素不包括在结果之中；
- 当使用一个负数作为步长时，必须让左边索引大于右边索引；
- 对正数步长，从左向右取元素；对负数步长，从右向左取元素；

加

序列可以进行「加法」操作，如下：

```
>>> [1, 2, 3] + [4, 5, 6]     # 「加法」效果其实就是连接在一起
[1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> 'hello, ' + 'world!'
```

```
'hello, world!'
>>> [1, 2, 3] + 'abc'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

这里需要注意的是：两种相同类型的序列才能「加法」操作。

乘

序列可以进行「乘法」操作，比如：

```
>>> 'abc' * 3
'abcabcabc'
>>> [0] * 3
[0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

in

为了检查一个值是否在序列中，可以使用 `in` 运算符，比如：

```
>>> 'he' in 'hello'
True
>>> 'hl' in 'hello'
False
>>> 10 in [6, 8, 10]
True
```

参考资料

- 《Python 基础教程》

列表

字符串和元组是不可变的，而列表是可变（mutable）的，可以对它进行随意修改。我们还可以将字符串和元组转换成一个列表，只需使用 `list` 函数，比如：

```
>>> s = 'hello'
>>> list(s)
['h', 'e', 'l', 'l', 'o']
>>> a = (1, 2, 3)
>>> list(a)
[1, 2, 3]
```

本文主要介绍常用的列表方法：

- index
- count
- append
- extend
- insert
- pop
- remove
- reverse
- sort

index

index 方法用于从列表中找出某个元素的位置，如果有多个相同的元素，则返回第一个元素的位置。

看看例子：

```
>>> numbers = [1, 2, 3, 4, 5, 5, 7, 8]
>>> numbers.index(5)        # 列表有两个 5，返回第一个元素的位置
4
>>> numbers.index(2)
1
>>> words = ['hello', 'world', 'you', 'me', 'he']
>>> words.index('me')
3
>>> words.index('her')       # 如果没找到元素，则会抛出异常
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ValueError: 'her' is not in list
```

count

count 方法用于统计某个元素在列表中出现的次数。

看看例子:

```
>>> numbers = [1, 2, 3, 4, 5, 5, 6, 7]
>>> numbers.count(2)   # 出现一次
1
>>> numbers.count(5)   # 出现了两次
2
>>> numbers.count(9)   # 没有该元素，返回 0
0
```

append

append 方法用于在列表末尾增加新的元素。

看看例子:

```
>>> numbers = [1, 2, 3, 4, 5, 5, 6, 7]
>>> numbers.append(8)          # 增加 8 这个元素
>>> numbers
[1, 2, 3, 4, 5, 5, 6, 7, 8]
>>> numbers.append([9, 10])   # 增加 [9, 10] 这个元素
>>> numbers
[1, 2, 3, 4, 5, 5, 6, 7, 8, [9, 10]]
```

extend

extend 方法将一个新列表的元素添加到原列表中。

看看例子:

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

```
>>> a
[1, 2, 3, 4, 5, 6]
>>>
>>> a.extend(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> a.extend([3])
>>> a
[1, 2, 3, 4, 5, 6, 3]
```

注意到，虽然 append 和 extend 可接收一个列表作为参数，但是 append 方法是将其作为一个元素添加到列表中，而 extend 则是将新列表的元素逐个添加到原列表中。

insert

insert 方法用于将某个元素添加到某个位置。

看看例子:

```
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> numbers.insert(3, 9)
>>> numbers
[1, 2, 3, 9, 4, 5, 6]
```

pop

pop 方法用于移除列表中的一个元素（默认是最后一个），并且返回该元素的值。

看看例子:

```
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> numbers.pop()
6
>>> numbers
[1, 2, 3, 4, 5]
>>> numbers.pop(3)
4
>>> numbers
[1, 2, 3, 5]
```

remove

remove 方法用于移除列表中的某个匹配元素，如果有多个匹配，则移除第一个。

看看例子：

```
>>> numbers = [1, 2, 3, 5, 6, 7, 5, 8]
>>> numbers.remove(5)          # 有两个 5，移除第 1 个
>>> numbers
[1, 2, 3, 6, 7, 5, 8]
>>> numbers.remove(9)          # 没有匹配的元素，抛出异常
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

reverse

reverse 方法用于将列表中的元素进行反转。

看看例子：

```
>>> numbers = [1, 2, 3, 5, 6, 7, 5, 8]
>>> numbers.reverse()
>>> numbers
[8, 5, 7, 6, 5, 3, 2, 1]
```

sort

sort 方法用于对列表进行排序，注意该方法会改变原来的列表，而不是返回新的排序列表，另外，sort 方法的返回值是空。

看看例子：

```
>>> a = [4, 3, 6, 8, 9, 1]
>>> b = a.sort()
>>> b == None          # 返回值为空
True
>>> a
[1, 3, 4, 6, 8, 9]     # 原列表已经发生改变
```

如果我们不想改变原列表，而是希望返回一个排序后的列表，可以使用 sorted 函数，如下：

```
>>> a = [4, 3, 6, 8, 9, 1]
>>> b = sorted(a)          # 返回一个排序后的列表
>>> a
[4, 3, 6, 8, 9, 1]         # 原列表没有改变
>>> b
[1, 3, 4, 6, 8, 9]         # 这是对原列表排序后的列表
```

注意到，不管是 sort 方法还是 sorted 函数，默认排序都是升序排序。如果你想要降序排序，就需要指定排序参数了。比如，对 sort 方法，可以添加一个 reverse 关键字参数，如下：

```
>>> a = [4, 3, 6, 8, 9, 1]
>>> a.sort(reverse=True)    # 反向排序
>>> a
[9, 8, 6, 4, 3, 1]
```

该参数对 sorted 函数同样适用：

```
>>> a = [4, 3, 6, 8, 9, 1]
>>> sorted(a, reverse=True)
[9, 8, 6, 4, 3, 1]
```

除了 reverse 关键字参数，还可以指定 key 关键字参数，它为每个元素创建一个键，然后所有元素按照这个键来排序，比如我们想根据元素的长度来排序：

```
>>> s = ['ccc', 'a', 'bb', 'dddd']
>>> s.sort(key=len)          # 使用 len 作为键函数，根据元素长度排序
>>> s
['a', 'bb', 'ccc', 'dddd']
```

另外，我们还可以使用 sorted 进行多列（属性）排序。

看看例子：

```
>>> students = [
        ('john', 'B', 15),
        ('jane', 'A', 12),
        ('dave', 'B', 10),
        ('ethan', 'C', 20),
        ('peter', 'B', 20),
```

```
        ('mike', 'C', 16)
    ]
>>>
# 对第 3 列排序 (从小到大)
>>> sorted(students, key=lambda student: student[2])
[('dave', 'B', 10),
 ('jane', 'A', 12),
 ('john', 'B', 15),
 ('mike', 'C', 16),
 ('ethan', 'C', 20),
 ('peter', 'B', 20)]

# 对第 2 列排序（从小到大），再对第 3 列从大到小排序
>>> sorted(students, key=lambda student: (student[1], -student[2]))
[('jane', 'A', 12),
 ('peter', 'B', 20),
 ('john', 'B', 15),
 ('dave', 'B', 10),
 ('ethan', 'C', 20),
 ('mike', 'C', 16)]
```

如果你想了解更多关于排序的知识，可以参考此文。

小结

- 列表是可变的。
- 列表常用的方法有 `index`, `count`, `append`, `extend` 等。

参考资料

- 《Python 基础教程》
- HowTo/Sorting - Python Wiki

元组

在 Python 中，元组是一种不可变序列，它使用圆括号来表示：

```
>>> a = (1, 2, 3)    # a 是一个元组
>>> a
(1, 2, 3)
>>> a[0] = 6         # 元组是不可变的，不能对它进行赋值操作
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

空元组

创建一个空元组可以用没有包含内容的圆括号来表示：

```
>>> a = ()
>>> a
()
```

一个值的元组

创建一个值的元组需要在值后面再加一个逗号，这个比较特殊，需要牢牢记住：

```
>>> a = (12,)   # 在值后面再加一个逗号
>>> a
(12,)
>>> type(a)
<type 'tuple'>
>>>
>>> b = (12)    # 只是使用括号括起来，而没有加逗号，不是元组，本质上是 b = 12
>>> b
12
>>> type(b)
<type 'int'>
```

元组操作

元组也是一种序列，因此也可以对它进行索引、分片等。由于它是不可变的，因此就没有类似列表的 `append, extend, sort` 等方法。

小结

- 元组是不可变的。
- 创建一个值的元组需要在值后面再加一个逗号。

字符串

字符串也是一种序列，因此，通用的序列操作，比如索引，分片，加法，乘法等对它同样适用。比如：

```
>>> s = 'hello, '
>>> s[0]            # 索引
'h'
>>> s[1:3]          # 分片
'el'
>>> s + 'world'     # 加法
'hello, world'
>>> s * 2           # 乘法
'hello, hello, '
```

但需要注意的是，字符串和元组一样，也是不可变的，所以你不能对它进行赋值等操作：

```
>>> s = 'hello'
>>> s[1] = 'ab'    # 不能对它进行赋值
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

除了通用的序列操作，字符串还有自己的方法，比如 join, lower, upper 等。字符串的方法特别多，这里只介绍一些常用的方法，如下：

- find
- split
- join
- strip
- replace
- translate
- lower/upper

find

find 方法用于在一个字符串中查找子串，它返回子串所在位置的最左端索引，如果没有找到，则返回 -1。

看看例子：

```
>>> motto = "to be or not to be, that is a question"
>>> motto.find('be')            # 返回 'b' 所在的位置，即 3
3
>>> motto.find('be', 4)         # 指定从起始位置开始找，找到的是第 2 个 'be'
16
>>> motto.find('be', 4, 7)      # 指定起始位置和终点位置，没有找到，返回 -1
-1
```

split

split 方法用于将字符串分割成序列。

看看例子:

```
>>> '/user/bin/ssh'.split('/')         # 使用 '/' 作为分隔符
['', 'user', 'bin', 'ssh']
>>> '1+2+3+4+5'.split('+')             # 使用 '+' 作为分隔符
['1', '2', '3', '4', '5']
>>> 'that    is a   question'.split()  # 没有提供分割符，默认使用所有空格作为分隔符
['that', 'is', 'a', 'question']
```

需要注意的是，如果不提供分隔符，则默认会使用所有空格作为分隔符（空格、制表符、换行等）。

join

join 方法可以说是 split 的逆方法，它用于将序列中的元素连接起来。

看看例子:

```
>>> '/'.join(['', 'user', 'bin', 'ssh'])
'/user/bin/ssh'
>>>
>>> '+'.join(['1', '2', '3', '4', '5'])
'1+2+3+4+5'
>>> ' '.join(['that', 'is', 'a', 'question'])
'that is a question'
>>> ''.join(['h', 'e', 'll', 'o'])
'hello'
>>> '+'.join([1, 2, 3, 4, 5])          # 不能是数字
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: sequence item 0: expected string, int found
```

strip

strip 方法用于移除字符串左右两侧的空格，但不包括内部，当然也可以指定需要移除的字符串。

看看例子：

```
>>> '  hello world!   '.strip()                # 移除左右两侧空格
'hello world!'
>>> '%%%   hello world!!!  ####'.strip('%#')   # 移除左右两侧的 '%' 或 '#'
'   hello world!!!  '
>>> '%%%   hello world!!!  ####'.strip('%# ')  # 移除左右两侧的 '%' 或 '#' 或空格
'hello world!!!'
```

replace

replace 方法用于替换字符串中的所有匹配项。

看看例子：

```
>>> motto = 'To be or not To be, that is a question'
>>> motto.replace('To', 'to')        # 用 'to' 替换所有的 'To', 返回了一个新的字符串
'to be or not to be, that is a question'
>>> motto                            # 原字符串保持不变
'To be or not To be, that is a question'
```

translate

translate 方法和 replace 方法类似，也可以用于替换字符串中的某些部分，但 **translate** 方法只处理单个字符。

translate 方法的使用形式如下：

```
str.translate(table[, deletechars]);
```

其中，table 是一个包含 256 个字符的转换表，可通过 maketrans 方法转换而来，deletechars 是字符串中要过滤的字符集。

看看例子：

```
>>> from string import maketrans
>>> table = maketrans('aeiou', '12345')
>>> motto = 'to be or not to be, that is a question'
>>> motto.translate(table)
't4 b2 4r n4t t4 b2, th1t 3s 1 q52st34n'
>>> motto
'to be or not to be, that is a question'
>>> motto.translate(table, 'rqu')        # 移除所有的 'r', 'q', 'u'
't4 b2 4 n4t t4 b2, th1t 3s 1 2st34n'
```

可以看到，maketrans 接收两个参数：两个等长的字符串，表示第一个字符串的每个字符用第二个字符串对应位置的字符替代，在上面的例子中，就是 ‘a’ 用 ‘1’ 替代，’e’ 用 ‘2’ 替代，等等，注意，是单个字符的代替，而不是整个字符串的替代。因此，motto 中的 o 都被替换为 4，e 都被替换为 2，等等。

lower/upper

lower/upper 用于返回字符串的大写或小写形式。

看看例子：

```
>>> x = 'PYTHON'
>>> x.lower()
'python'
>>> x
'PYTHON'
>>>
>>> y = 'python'
>>> y.upper()
'PYTHON'
>>> y
'python'
```

小结

- 字符串是不可变对象，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回。
- translate 针对单个字符进行替换。

参考资料

- 《python 基础教程》

字典

字典是 Python 中唯一的映射类型，每个元素由键（key）和值（value）构成，键必须是不可变类型，比如数字、字符串和元组。

字典基本操作

这里先介绍字典的几个基本操作，后文再介绍字典的常用方法。

- 创建字典
- 遍历字典
- 判断键是否在字典里面

创建字典

字典可以通过下面的方式创建：

```
>>> d0 = {}    # 空字典
>>> d0
{}
>>> d1 = {'name': 'ethan', 'age': 20}
>>> d1
{'age': 20, 'name': 'ethan'}
>>> d1['age'] = 21          # 更新字典
>>> d1
{'age': 21, 'name': 'ethan'}
>>> d2 = dict(name='ethan', age=20)    # 使用 dict 函数
>>> d2
{'age': 20, 'name': 'ethan'}
>>> item = [('name', 'ethan'), ('age', 20)]
>>> d3 = dict(item)
>>> d3
{'age': 20, 'name': 'ethan'}
```

遍历字典

遍历字典有多种方式，这里先介绍一些基本的方式，后文会介绍一些高效的遍历方式。

```
>>> d = {'name': 'ethan', 'age': 20}
```

```
>>> for key in d:
...     print '%s: %s' % (key, d[key])
...
age: 20
name: ethan
>>> d['name']
'ethan'
>>> d['age']
20
>>> for key in d:
...     if key == 'name':
...         del d[key]         # 要删除字典的某一项
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
>>>
>>> for key in d.keys():  # python2 应该使用这种方式, python3 使用 list(d.keys())
...     if key == 'name':
...         del d[key]
...
>>> d
{'age': 20}
```

在上面，我们介绍了两种遍历方式： `for key in d` 和 `for key in d.keys()` ，如果在遍历的时候，要删除键为 key 的某项，使用第一种方式会抛出 RuntimeError，使用第二种方式则不会。

判断键是否在字典里面

有时，我们需要判断某个键是否在字典里面，这时可以用 `in` 进行判断，如下：

```
>>> d = {'name': 'ethan', 'age': 20}
>>> 'name' in d
True
>>> d['score']             # 访问不存在的键，会抛出 KeyError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'score'
>>> 'score' in d           # 使用 in 判断 key 是否在字典里面
False
```

字典常用方法

字典有自己的一些操作方法，这里只介绍部分常用的方法：

- clear
- copy
- get
- setdefault
- update
- pop
- popitem
- keys/iterkeys
- values/itervalues
- items/iteritems
- fromkeys

clear

clear 方法用于清空字典中的所有项，这是个原地操作，所以无返回值（或者说是 None）。

看看例子：

```
>>> d = {'name': 'ethan', 'age': 20}
>>> rv = d.clear()
>>> d
{}
>>> print rv
None
```

再看看一个例子：

```
>>> d1 = {}
>>> d2 = d1
>>> d2['name'] = 'ethan'
>>> d1
{'name': 'ethan'}
>>> d2
{'name': 'ethan'}
>>> d1 = {}         # d1 变为空字典
>>> d2
{'name': 'ethan'}   # d2 不受影响
```

在上面，d1 和 d2 最初对应同一个字典，而后我们使用 `d1 = {}` 使其变成一个空字典，但此时 d2 不受影响。如果希望 d1 变成空字典之后，d2 也变成空字典，则可以使用 clear 方法：

```
>>> d1 = {}
>>> d2 = d1
>>> d2['name'] = 'ethan'
>>> d1
{'name': 'ethan'}
>>> d2
{'name': 'ethan'}
>>> d1.clear()      # d1 清空之后，d2 也为空
>>> d1
{}
>>> d2
{}
```

copy

copy 方法实现的是浅复制（shallow copy）。它具有以下特点：

- 对可变对象的修改保持同步；
- 对不可变对象的修改保持独立；

看看例子：

```
# name 的值是不可变对象，books 的值是可变对象
>>> d1 = {'name': 'ethan', 'books': ['book1', 'book2', 'book3']}
>>> d2 = d1.copy()
>>> d2['name'] = 'peter'          # d2 对不可变对象的修改不会改变 d1
>>> d2
{'books': ['book1', 'book2', 'book3'], 'name': 'peter'}
>>> d1
{'books': ['book1', 'book2', 'book3'], 'name': 'ethan'}
>>> d2['books'].remove('book2')  # d2 对可变对象的修改会影响 d1
>>> d2
{'books': ['book1', 'book3'], 'name': 'peter'}
>>> d1
{'books': ['book1', 'book3'], 'name': 'ethan'}
>>> d1['books'].remove('book3')  # d1 对可变对象的修改会影响 d2
>>> d1
{'books': ['book1'], 'name': 'ethan'}
```

```
>>> d2
{'books': ['book1'], 'name': 'peter'}
```

和浅复制对应的是深复制（deep copy），它会创造出一个副本，跟原来的对象没有关系，可以通过 copy 模块的 deepcopy 函数来实现：

```
>>> from copy import deepcopy
>>> d1 = {'name': 'ethan', 'books': ['book1', 'book2', 'book3']}
>>> d2 = deepcopy(d1)              # 创造出一个副本
>>>
>>> d2['books'].remove('book2')    # 对 d2 的任何修改不会影响到 d1
>>> d2
{'books': ['book1', 'book3'], 'name': 'ethan'}
>>> d1
{'books': ['book1', 'book2', 'book3'], 'name': 'ethan'}
>>>
>>> d1['books'].remove('book3')    # 对 d1 的任何修改也不会影响到 d2
>>> d1
{'books': ['book1', 'book2'], 'name': 'ethan'}
>>> d2
{'books': ['book1', 'book3'], 'name': 'ethan'}
```

get

当我们试图访问字典中不存在的项时会出现 KeyError，但使用 get 就可以避免这个问题。

看看例子：

```
>>> d = {}
>>> d['name']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'name'
>>> print d.get('name')
None
>>> d.get('name', 'ethan')   # 'name' 不存在，使用默认值 'ethan'
'ethan'
>>> d
{}
```

setdefault

setdefault 方法用于对字典设定键值。使用形式如下：

```
dict.setdefault(key, default=None)
```

看看例子：

```
>>> d = {}
>>> d.setdefault('name', 'ethan')   # 返回设定的默认值 'ethan'
'ethan'
>>> d                               # d 被更新
{'name': 'ethan'}
>>> d['age'] = 20
>>> d
{'age': 20, 'name': 'ethan'}
>>> d.setdefault('age', 18)         # age 已存在，返回已有的值，不会更新字典
20
>>> d
{'age': 20, 'name': 'ethan'}
```

可以看到，当键不存在的时候，setdefault 返回设定的默认值并且更新字典。当键存在的时候，会返回已有的值，但不会更新字典。

update

update 方法用于将一个字典添加到原字典，如果存在相同的键则会进行覆盖。

看看例子：

```
>>> d = {}
>>> d1 = {'name': 'ethan'}
>>> d.update(d1)                # 将字典 d1 添加到 d
>>> d
{'name': 'ethan'}
>>> d2 = {'age': 20}
>>> d.update(d2)                # 将字典 d2 添加到 d
>>> d
{'age': 20, 'name': 'ethan'}
>>> d3 = {'name': 'michael'}    # 将字典 d3 添加到 d，存在相同的 key，则覆盖
>>> d.update(d3)
```

```
>>> d
{'age': 20, 'name': 'michael'}
```

items/iteritems

items 方法将所有的字典项以列表形式返回，这些列表项的每一项都来自于（键，值）。我们也经常使用这个方法来对字典进行遍历。

看看例子：

```
>>> d = {'name': 'ethan', 'age': 20}
>>> d.items()
[('age', 20), ('name', 'ethan')]
>>> for k, v in d.items():
...     print '%s: %s' % (k, v)
...
age: 20
name: ethan
```

iteritems 的作用大致相同，但会返回一个迭代器对象而不是列表，同样，我们也可以使用这个方法来对字典进行遍历，而且这也是推荐的做法：

```
>>> d = {'name': 'ethan', 'age': 20}
>>> d.iteritems()
<dictionary-itemiterator object at 0x109cf2d60>
>>> for k, v in d.iteritems():
...     print '%s: %s' % (k, v)
...
age: 20
name: ethan
```

keys/iterkeys

keys 方法将字典的键以列表形式返回，iterkeys 则返回针对键的迭代器。

看看例子：

```
>>> d = {'name': 'ethan', 'age': 20}
>>> d.keys()
['age', 'name']
```

```
>>> d.iterkeys()
<dictionary-keyiterator object at 0x1077fad08>
```

values/itervalues

values 方法将字典的值以列表形式返回，itervalues 则返回针对值的迭代器。

看看例子:

```
>>> d = {'name': 'ethan', 'age': 20}
>>> d.values()
[20, 'ethan']
>>> d.itervalues()
<dictionary-valueiterator object at 0x10477dd08>
```

pop

pop 方法用于将某个键值对从字典移除，并返回给定键的值。

看看例子:

```
>>> d = {'name': 'ethan', 'age': 20}
>>> d.pop('name')
'ethan'
>>> d
{'age': 20}
```

popitem

popitem 用于随机移除字典中的某个键值对。

看看例子:

```
>>> d = {'id': 10, 'name': 'ethan', 'age': 20}
>>> d.popitem()
('age', 20)
>>> d
{'id': 10, 'name': 'ethan'}
>>> d.popitem()
('id', 10)
```

```
>>> d
{'name': 'ethan'}
```

对元素为字典的列表排序

事实上，我们很少直接对字典进行排序，而是对元素为字典的列表进行排序。

比如，存在下面的 students 列表，它的元素是字典：

```
students = [
    {'name': 'john', 'score': 'B', 'age': 15},
    {'name': 'jane', 'score': 'A', 'age': 12},
    {'name': 'dave', 'score': 'B', 'age': 10},
    {'name': 'ethan', 'score': 'C', 'age': 20},
    {'name': 'peter', 'score': 'B', 'age': 20},
    {'name': 'mike', 'score': 'C', 'age': 16}
]
```

- 按 score 从小到大排序

```
>>> sorted(students, key=lambda stu: stu['score'])
[{'age': 12, 'name': 'jane', 'score': 'A'},
 {'age': 15, 'name': 'john', 'score': 'B'},
 {'age': 10, 'name': 'dave', 'score': 'B'},
 {'age': 20, 'name': 'peter', 'score': 'B'},
 {'age': 20, 'name': 'ethan', 'score': 'C'},
 {'age': 16, 'name': 'mike', 'score': 'C'}]
```

需要注意的是，这里是按照字母的 ascii 大小排序的，所以 score 从小到大，即从 ‘A’ 到 ‘C’。

- 按 score 从大到小排序

```
>>> sorted(students, key=lambda stu: stu['score'], reverse=True)  # reverse 参数
[{'age': 20, 'name': 'ethan', 'score': 'C'},
 {'age': 16, 'name': 'mike', 'score': 'C'},
 {'age': 15, 'name': 'john', 'score': 'B'},
 {'age': 10, 'name': 'dave', 'score': 'B'},
 {'age': 20, 'name': 'peter', 'score': 'B'},
 {'age': 12, 'name': 'jane', 'score': 'A'}]
```

- 按 score 从小到大，再按 age 从小到大

```
>>> sorted(students, key=lambda stu: (stu['score'], stu['age']))
[{'age': 12, 'name': 'jane', 'score': 'A'},
 {'age': 10, 'name': 'dave', 'score': 'B'},
 {'age': 15, 'name': 'john', 'score': 'B'},
 {'age': 20, 'name': 'peter', 'score': 'B'},
 {'age': 16, 'name': 'mike', 'score': 'C'},
 {'age': 20, 'name': 'ethan', 'score': 'C'}]
```

- 按 score 从小到大，再按 age 从大到小

```
>>> sorted(students, key=lambda stu: (stu['score'], -stu['age']))
[{'age': 12, 'name': 'jane', 'score': 'A'},
 {'age': 20, 'name': 'peter', 'score': 'B'},
 {'age': 15, 'name': 'john', 'score': 'B'},
 {'age': 10, 'name': 'dave', 'score': 'B'},
 {'age': 20, 'name': 'ethan', 'score': 'C'},
 {'age': 16, 'name': 'mike', 'score': 'C'}]
```

参考资料

- 《Python 基础教程》
- python字典和集合 | Alex’s Blog
- Python中实现多属性排序 | 酷 壳 - CoolShell.cn
- HowTo/Sorting - Python Wiki
- How do I sort a list of dictionaries by values of the dictionary in Python? - Stack Overflow

集合

集合（set）和字典（dict）类似，它是一组 key 的集合，但不存储 value。集合的特性就是：key 不能重复。

集合常用操作

创建集合

set 的创建可以使用 `{}` 也可以使用 set 函数：

```
>>> s1 = {'a', 'b', 'c', 'a', 'd', 'b'}   # 使用 {}
>>> s1
set(['a', 'c', 'b', 'd'])
>>>
>>> s2 = set('helloworld')                # 使用 set(), 接收一个字符串
>>> s2
set(['e', 'd', 'h', 'l', 'o', 'r', 'w'])
>>>
>>> s3 = set(['.mp3', '.mp4', '.rmvb', '.mkv', '.mp3'])   # 使用 set(), 接收一个列表
>>> s3
set(['.mp3', '.mkv', '.rmvb', '.mp4'])
```

遍历集合

```
>>> s = {'a', 'b', 'c', 'a', 'd', 'b'}
>>> for e in s:
...     print e
...
a
c
b
d
```

添加元素

`add()` 方法可以将元素添加到 set 中，可以重复添加，但没有效果。

```
>>> s = {'a', 'b', 'c', 'a', 'd', 'b'}
>>> s
set(['a', 'c', 'b', 'd'])
>>> s.add('e')
>>> s
set(['a', 'c', 'b', 'e', 'd'])
>>> s.add('a')
>>> s
set(['a', 'c', 'b', 'e', 'd'])
>>> s.add(4)
>>> s
set(['a', 'c', 'b', 4, 'd', 'e'])
```

删除元素

`remove()` 方法可以删除集合中的元素，但是删除不存在的元素，会抛出 KeyError，可改用 `discard()` 。

看看例子：

```
>>> s = {'a', 'b', 'c', 'a', 'd', 'b'}
>>> s
set(['a', 'c', 'b', 'd'])
>>> s.remove('a')           # 删除元素 'a'
>>> s
set(['c', 'b', 'd'])
>>> s.remove('e')           # 删除不存在的元素，会抛出 KeyError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'e'
>>> s.discard('e')          # 删除不存在的元素，不会抛出 KeyError
```

交集/并集/差集

Python 中的集合也可以看成是数学意义上的无序和无重复元素的集合，因此，我们可以对两个集合作交集、并集等。

看看例子：

```
>>> s1 = {1, 2, 3, 4, 5, 6}
>>> s2 = {3, 6, 9, 10, 12}
>>> s3 = {2, 3, 4}
>>> s1 & s2            # 交集
set([3, 6])
>>> s1 | s2            # 并集
set([1, 2, 3, 4, 5, 6, 9, 10, 12])
>>> s1 - s2            # 差集
set([1, 2, 4, 5])
>>> s3.issubset(s1)    # s3 是否是 s1 的子集
True
>>> s3.issubset(s2)    # s3 是否是 s2 的子集
False
>>> s1.issuperset(s3)  # s1 是否是 s3 的超集
True
>>> s1.issuperset(s2)  # s1 是否是 s2 的超集
False
```

参考资料

- 使用dict和set - 廖雪峰的官方网站
- python字典和集合 | Alex’s Blog
- python - Why is it possible to replace set() with {}? - Stack Overflow

函数

本章讲解函数，包含以下部分：

- 定义函数
- 函数参数

定义函数

在 Python 中，定义函数使用 `def` 语句。一个函数主要由三部分构成：

- 函数名
- 函数参数
- 函数返回值

让我们看一个简单的例子：

```
1.  def hello(name):
2.      return name
3.
4.  >>> r = hello('ethan')
5.  >>> r
6.  'ethan'
```

在上面，我们定义了一个函数。函数名是 `hello`；函数有一个参数，参数名是 name；函数有一个返回值，name。

我们也可以定义一个没有参数和返回值的函数：

```
1.  def greet():                # 没有参数
2.      print 'hello world'    # 没有 return, 会自动 return None
3.
4.  >>> r = greet()
5.  hello world
6.  >>> r == None
```

这里，函数 `greet` 没有参数，它也没有返回值（或者说是 None）。

我们还可以定义返回多个值的函数：

```
1.  >>> def add_one(x, y, z):
2.  ...     return x+1, y+1, z+1        # 有 3 个返回值
3.  ...
4.  >>>
5.  >>> result = add_one(1, 5, 9)
6.  >>> result      # result 实际上是一个 tuple
7.  (2, 6, 10)
8.  >>> type(result)
```

```
9.  <type 'tuple'>
```

小结

- 如果函数没有 `return` 语句，则自动 `return None`。

函数参数

在 Python 中，定义函数和调用函数都很简单，但如何定义函数参数和传递函数参数，则涉及到一些套路了。总的来说，Python 的函数参数主要分为以下几种：

- 必选参数
- 默认参数
- 可变参数
- 关键字参数

必选参数

必选参数可以说是最常见的了，顾名思义，必选参数就是在调用函数的时候要传入数量一致的参数，比如：

```
>>> def add(x, y):          # x, y 是必选参数
...     print x + y
...
>>> add()                   # 啥都没传，不行
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() takes exactly 2 arguments (0 given)
>>> add(1)                  # 只传了一个，也不行
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() takes exactly 2 arguments (1 given)
>>> add(1, 2)               # 数量一致，通过
3
```

默认参数

默认参数是指在定义函数的时候提供一些默认值，如果在调用函数的时候没有传递该参数，则自动使用默认值，否则使用传递时该参数的值。

看看例子就明白了：

```
>>> def add(x, y, z=1):      # x, y 是必选参数，z 是默认参数，默认值是 1
...     print x + y + z
...
```

```
>>> add(1, 2, 3)            # 1+2+3
6
>>> add(1, 2)               # 没有传递 z，自动使用 z=1，即 1+2+1
4
```

可以看到，默认参数使用起来也很简单，但有两点需要注意的是：

- 默认参数要放在所有必选参数的后面
- 默认参数应该使用不可变对象

比如，下面对默认参数的使用是错误的：

```
>>> def add(x=1, y, z):      # x 是默认参数，必须放在所有必选参数的后面
...     return x + y + z
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
>>>
>>> def add(x, y=1, z):      # y 是默认参数，必须放在所有必选参数的后面
...     return x + y + z
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

再来看看为什么默认参数应该使用不可变对象。

我们看一个例子：

```
>>> def add_to_list(L=[]):
...     L.append('END')
...     return L
```

在上面的函数中，L 是一个默认参数，默认值是 `[]`，表示空列表。

我们来看看使用：

```
>>> add_to_list([1, 2, 3])          # 没啥问题
[1, 2, 3, 'END']
>>> add_to_list(['a', 'b', 'c'])    # 没啥问题
['a', 'b', 'c', 'END']
>>> add_to_list()                   # 没有传递参数，使用默认值，也没啥问题
['END']
>>> add_to_list()                   # 没有传递参数，使用默认值，竟出现两个 'END'
```

```
['END', 'END']
>>> add_to_list()                    # 糟糕了，三个 'END'
['END', 'END', 'END']
```

为啥呢？我们在调用函数的时候没有传递参数，那么就默认使用 `L=[]` ，经过处理，L 应该只有一个元素，怎么会出现调用函数两次，L 就有两个元素呢？

原来，L 指向了可变对象 `[]` ，当你调用函数时，L 的内容发生了改变，默认参数的内容也会跟着变，也就是，当你第一次调用时，L 的初始值是 `[]` ，当你第二次调用时，L 的初始值是 `['END']` ，等等。

所以，为了避免不必要的错误，我们应该使用不可变对象作为函数的默认参数。

可变参数

在某些情况下，我们在定义函数的时候，无法预估函数应该制定多少个参数，这时我们就可以使用可变参数了，也就是，函数的参数个数是不确定的。

看看例子：

```
>>> def add(*numbers):
...     sum = 0
...     for i in numbers:
...         sum += i
...     print 'numbers:', numbers
...     return sum
```

在上面的代码中，numbers 就是一个可变参数，参数前面有一个 `*` 号，表示是可变的。在函数内部，参数 numbers 接收到的是一个 tuple。

在调用函数时，我们可以给该函数传递任意个参数，包括 0 个参数：

```
>>> add()           # 传递 0 个参数
numbers: ()
0
>>> add(1)          # 传递 1 个参数
numbers: (1,)
1
>>> add(1, 2)       # 传递 2 个参数
numbers: (1, 2)
3
>>> add(1, 2, 3)    # 传递 3 个参数
```

```
numbers: (1, 2, 3)
6
```

上面的 `*` 表示任意参数，实际上，它还有另外一个用法：用来给函数传递参数。

看看例子：

```
>>> def add(x, y, z):          # 有 3 个必选参数
...     return x + y + z
...
>>> a = [1, 2, 3]
>>> add(a[0], a[1], a[2])     # 这样传递参数很累赘
6
>>> add(*a)                   # 使用 *a，相当于上面的做法
6
>>> b = (4, 5, 6)
>>> add(*b)                   # 对元组一样适用
15
```

再看一个例子：

```
>>> def add(*numbers):        # 函数参数是可变参数
...     sum = 0
...     for i in numbers:
...         sum += i
...     return sum
...
>>> a = [1, 2]
>>> add(*a)                   # 使用 *a 给函数传递参数
3
>>> a = [1, 2, 3, 4]
>>> add(*a)
10
```

关键字参数

可变参数允许你将不定数量的参数传递给函数，而关键字参数则允许你将不定长度的键值对， 作为参数传递给一个函数。

让我们看看例子：

```
>>> def add(**kwargs):
    return kwargs
>>> add()            # 没有参数，kwargs 为空字典
{}
>>> add(x=1)         # x=1 => kwargs={'x': 1}
{'x': 1}
>>> add(x=1, y=2)    # x=1, y=2 => kwargs={'y': 2, 'x': 1}
{'y': 2, 'x': 1}
```

在上面的代码中，kwargs 就是一个关键字参数，它前面有两个 `*` 号。kwargs 可以接收不定长度的键值对，在函数内部，它会表示成一个 dict。

和可变参数类似，我们也可以使用 `**kwargs` 的形式来调用函数，比如：

```
>>> def add(x, y, z):
...     return x + y + z
...
>>> dict1 = {'z': 3, 'x': 1, 'y': 6}
>>> add(dict1['x'], dict1['y'], dict1['z'])    # 这样传参很累赘
10
>>> add(**dict1)        # 使用 **dict1 来传参，等价于上面的做法
10
```

再看一个例子：

```
>>> def sum(**kwargs):                  # 函数参数是关键字参数
...     sum = 0
...     for k, v in kwargs.items():
...         sum += v
...     return sum
>>> sum()                               # 没有参数
0
>>> dict1 = {'x': 1}
>>> sum(**dict1)                        # 相当于 sum(x=1)
1
>>> dict2 = {'x': 2, 'y': 6}
>>> sum(**dict2)                        # 相当于 sum(x=2, y=6)
8
```

参数组合

在实际的使用中，我们经常会同时用到必选参数、默认参数、可变参数和关键字参数或其中的某些。但是，需要注意的是，它们在使用的时候是有顺序的，依次是必选参数、默认参数、可变参数和关键字参数。

比如，定义一个包含上述四种参数的函数：

```
>>> def func(x, y, z=0, *args, **kwargs):
    print 'x =', x
    print 'y =', y
    print 'z =', z
    print 'args =', args
    print 'kwargs =', kwargs
```

在调用函数的时候，Python 会自动按照参数位置和参数名把对应的参数传进去。让我们看看：

```
>>> func(1, 2)                      # 至少提供两个参数，因为 x, y 是必选参数
x = 1
y = 2
z = 0
args = ()
kwargs = {}
>>> func(1, 2, 3)                   # x=1, y=2, z=3
x = 1
y = 2
z = 3
args = ()
kwargs = {}
>>> func(1, 2, 3, 4, 5, 6)          # x=1, y=2, z=3, args=(4, 5, 6), kwargs={}
x = 1
y = 2
z = 3
args = (4, 5, 6)
kwargs = {}
>>> func(1, 2, 4, u=6, v=7)         # args = (), kwargs = {'u': 6, 'v': 7}
x = 1
y = 2
z = 4
args = ()
kwargs = {'u': 6, 'v': 7}
>>> func(1, 2, 3, 4, 5, u=6, v=7)   # args = (4, 5), kwargs = {'u': 6, 'v': 7}
x = 1
y = 2
```

```
z = 3
args = (4, 5)
kwargs = {'u': 6, 'v': 7}
```

我们还可以通过下面的形式来传递参数：

```
>>> a = (1, 2, 3)
>>> b = {'u': 6, 'v': 7}
>>> func(*a, **b)
x = 1
y = 2
z = 3
args = ()
kwargs = {'u': 6, 'v': 7}
```

小结

- 默认参数要放在所有必选参数的后面。
- 应该使用不可变对象作为函数的默认参数。
- `*args` 表示可变参数，`**kwargs` 表示关键字参数。
- 参数组合在使用的时候是有顺序的，依次是必选参数、默认参数、可变参数和关键字参数。
- `*args` 和 `**kwargs` 是 Python 的惯用写法。

参考资料

- args 和 *kwargs · Python进阶
- 函数的参数 - 廖雪峰的官方网站

函数式编程

函数式编程（**functional programming**）是一种编程范式（Programming paradigm），或者说编程模式，比如我们常见的过程式编程是一种编程范式，面向对象编程又是另一种编程范式。

函数式编程的一大特性就是：可以把函数当成变量来使用，比如将函数赋值给其他变量、把函数作为参数传递给其他函数、函数的返回值也可以是一个函数等等。

Python 不是纯函数式编程语言，但它对函数式编程提供了一些支持。本章主要介绍 Python 中的函数式编程，主要包括以下几个方面：

- 高阶函数
- 匿名函数
- map/reduce/filter
- 闭包
- 装饰器
- partial 函数

参考资料

- 什么是函数式编程思维？ - 知乎
- 编程范型 - 维基百科，自由的百科全书
- 函数式编程初探 - 阮一峰的网络日志
- 函数式编程 | 酷 壳 - CoolShell.cn

高阶函数

在函数式编程中，我们可以将函数当作变量一样自由使用。一个函数接收另一个函数作为参数，这种函数称之为高阶函数（**Higher-order Functions**）。

看一个简单的例子：

```
1. def func(g, arr):
2.     return [g(x) for x in arr]
```

上面的代码中，`func` 是一个高阶函数，它接收两个参数，第 1 个参数是函数，第 2 个参数是数组，`func` 的功能是将函数 g 逐个作用于数组 arr 上，并返回一个新的数组，比如，我们可以这样用：

```
1. def double(x):
2.     return 2 * x
3. 
4. def square(x):
5.     return x * x
6. 
7. arr1 = func(double, [1, 2, 3, 4])
8. arr2 = func(square, [1, 2, 3, 4])
```

不难判断出，arr1 是 [2, 4, 6, 8]，arr2 是 [1, 4, 9, 16]。

小结

- 可接收其他函数作为参数的函数称为高阶函数。

map/reduce/filter

map/reduce/filter 是 Python 中较为常用的内建高阶函数，它们为函数式编程提供了不少便利。

map

`map` 函数的使用形式如下：

```
map(function, sequence)
```

解释：对 sequence 中的 item 依次执行 function(item)，并将结果组成一个 List 返回，也就是：

```
[function(item1), function(item2), function(item3), ...]
```

看一些简单的例子。

```
>>> def square(x):
...     return x * x
>>> map(square, [1, 2, 3, 4])
[1, 4, 9, 16]
>>> map(lambda x: x * x, [1, 2, 3, 4])   # 使用 lambda
[1, 4, 9, 16]
>>> map(str, [1, 2, 3, 4])
['1', '2', '3', '4']
>>> map(int, ['1', '2', '3', '4'])
[1, 2, 3, 4]
```

再看一个例子：

```
def double(x):
    return 2 * x

def triple(x):
```

```
    return 3 *x

def square(x):
    return x * x

funcs = [double, triple, square]  # 列表元素是函数对象

# 相当于 [double(4), triple(4), square(4)]
value = list(map(lambda f: f(4), funcs))

print value

# output
[8, 12, 16]
```

上面的代码中，我们加了 list 转换，是为了兼容 Python3，在 Python2 中 map 直接返回列表，Python3 中返回迭代器。

reduce

`reduce` 函数的使用形式如下：

```
reduce(function, sequence[, initial])
```

解释：先将 sequence 的前两个 item 传给 function，即 function(item1, item2)，函数的返回值和 sequence 的下一个 item 再传给 function，即 function(function(item1, item2), item3)，如此迭代，直到 sequence 没有元素，如果有 initial，则作为初始值调用。

也就是说：

```
reduece(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

看一些例子，就能很快理解了。

```
>>> reduce(lambda x, y: x * y, [1, 2, 3, 4])  # 相当于 ((1 * 2) * 3) * 4
24
>>> reduce(lambda x, y: x * y, [1, 2, 3, 4], 5) # ((((5 * 1) * 2) * 3)) * 4
120
>>> reduce(lambda x, y: x / y, [2, 3, 4], 72)  #  (((72 / 2) / 3)) / 4
3
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4], 5)  # ((((5 + 1) + 2) + 3)) + 4
```

```
15
>>> reduce(lambda x, y: x - y, [8, 5, 1], 20)  # ((20 - 8) - 5) - 1
6
>>> f = lambda a, b: a if (a > b) else b   # 两两比较，取最大值
>>> reduce(f, [5, 8, 1, 10])
10
```

filter

`filter` 函数用于过滤元素，它的使用形式如下：

```
filter(function, sequnce)
```

解释：将 function 依次作用于 sequnce 的每个 item，即 function(item)，将返回值为 True 的 item 组成一个 List/String/Tuple（取决于 sequnce 的类型，python3 统一返回迭代器）返回。

看一些例子。

```
>>> even_num = list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5, 6]))
>>> even_num
[2, 4, 6]
>>> odd_num = list(filter(lambda x: x % 2, [1, 2, 3, 4, 5, 6]))
>>> odd_num
[1, 3, 5]
>>> filter(lambda x: x < 'g', 'hijack')
'ac'        # python2
>>> filter(lambda x: x < 'g', 'hijack')
<filter object at 0x1034b4080>   # python3
```

小结

- map/reduce/filter 为函数式编程提供了不少便利，可使代码变得更简洁；
- 注意在 python2 和 python3 中，map/reduce/filter 的返回值类型有所不同，python2 返回的是基本数据类型，而 python3 则返回了迭代器；

参考资料

- Python Tutorial: Lambda Operator, filter, reduce and map

匿名函数

在 Python 中，我们使用 `def` 语句来定义函数，比如：

```
def double(x):
    return 2 * x
```

除了用上面的方式定义函数，Python 还提供了一个关键字 `lambda`，让我们可以创建一个匿名函数，也就是没有名称的函数。它的形式如下：

```
lambda 参数: 表达式
```

关键字 `lambda` 说明它是一个匿名函数，冒号 `:` 前面的变量是该匿名函数的参数，冒号后面是函数的返回值，注意这里不需使用 return 关键字。

我们将上面的 `double` 函数改写成一个匿名函数，如下：

```
lambda x: 2 * x
```

那怎么调用匿名函数呢？可以直接这样使用：

```
>>> (lambda x: 2 * x)(8)
16
```

由于匿名函数本质上是一个函数对象，也可以将其赋值给另一个变量，再由该变量来调用函数，如下：

```
>>> f = lambda x: 2 * x   # 将匿名函数赋给变量 f
>>> f
<function <lambda> at 0x7f835a696578>
>>> f(8)
16
```

使用场景

`lambda` 函数一般适用于创建一些临时性的，小巧的函数。比如上面的 `double` 函数，我们当然可以使用 `def` 来定义，但使用 `lambda` 来创建会显得很简洁，尤其是在高阶函数的使用中。

看一个例子：

```
def func(g, arr):
    return [g(x) for x in arr]
```

现在给一个列表 [1, 2, 3, 4]，利用上面的函数，对列表中的元素加 1，返回一个新的列表，你可能这样用：

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

arr = func(add_one, [1, 2, 3, 4])
```

这样做没什么错，可是 `add_one` 这个函数太简单了，使用 `def` 定义未免有点小题大作，我们改用 `lambda` ：

```
arr = func(lambda x: x + 1, [1, 2, 3, 4])
```

是不是很简洁、易懂？

小结

- 匿名函数本质上是一个函数，没有函数名称，因此使用匿名函数不用担心函数名冲突；
- 匿名函数一般适用于创建一些临时性的，小巧的函数；

闭包

在 Python 中，函数也是一个对象。因此，我们在定义函数时，可以再嵌套定义一个函数，并将该嵌套函数返回，比如：

```
1.  from math import pow
2.
3.  def make_pow(n):
4.      def inner_func(x):      # 嵌套定义了 inner_func
5.          return pow(x, n)    # 注意这里引用了外部函数的 n
6.      return inner_func       # 返回 inner_func
```

上面的代码中，函数 `make_pow` 里面又定义了一个内部函数 `inner_func`，然后将该函数返回。因此，我们可以使用 `make_pow` 来生成另一个函数：

```
1.  >>> pow2 = make_pow(2)  # pow2 是一个函数，参数 2 是一个自由变量
2.  >>> pow2
3.  <function inner_func at 0x10271faa0>
4.  >>> pow2(6)
5.  36.0
```

我们还注意到，内部函数 `inner_func` 引用了外部函数 `make_pow` 的自由变量 `n`，这也就意味着，当函数 `make_pow` 的生命周期结束之后， `n` 这个变量依然会保存在 `inner_func` 中，它被 `inner_func` 所引用。

```
1.  >>> del make_pow          # 删除 make_pow
2.  >>> pow3 = make_pow(3)
3.  Traceback (most recent call last):
4.    File "<stdin>", line 1, in <module>
5.  NameError: name 'make_pow' is not defined
6.  >>> pow2(9)      # pow2 仍可正常调用，自由变量 2 仍保存在 pow2 中
7.  81.0
```

像上面这种情况，一个函数返回了一个内部函数，该内部函数引用了外部函数的相关参数和变量，我们把该返回的内部函数称为闭包（**Closure**）。

在上面的例子中， `inner_func` 就是一个闭包，它引用了自由变量 `n`。

闭包的作用

- 闭包的最大特点就是引用了自由变量，即使生成闭包的环境已经释放，闭包仍然存在。
- 闭包在运行时可以有多个实例，即使传入的参数相同。

```
>>> pow_a = make_pow(2)
>>> pow_b = make_pow(2)
>>> pow_a == pow_b
False
```

- 利用闭包，我们还可以模拟类的实例。

这里构造一个类，用于求一个点到另一个点的距离：

```
from math import sqrt

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def get_distance(self, u, v):
        distance = sqrt((self.x - u) ** 2 + (self.y - v) ** 2)
        return distance

>>> pt = Point(7, 2)        # 创建一个点
>>> pt.get_distance(10, 6)  # 求到另一个点的距离
5.0
```

用闭包来实现：

```
def point(x, y):
    def get_distance(u, v):
        return sqrt((x - u) ** 2 + (y - v) ** 2)

    return get_distance

>>> pt = point(7, 2)
>>> pt(10, 6)
5.0
```

可以看到，结果是一样的，但使用闭包实现比使用类更加简洁。

常见误区

闭包的概念很简单，但实现起来却容易出现一些误区，比如下面的例子：

```
def count():
    funcs = []
    for i in [1, 2, 3]:
        def f():
            return i
        funcs.append(f)
    return funcs
```

在该例子中，我们在每次 `for` 循环中创建了一个函数，并将它存到 `funcs` 中。现在，调用上面的函数，你可能认为返回结果是 1, 2, 3，事实上却不是：

```
>>> f1, f2, f3 = count()
>>> f1()
3
>>> f2()
3
>>> f3()
3
```

为什么呢？原因在于上面的函数 `f` 引用了变量 `i`，但函数 `f` 并非立刻执行，当 `for` 循环结束时，此时变量 `i` 的值是3，`funcs` 里面的函数引用的变量都是 3，最终结果也就全为3。

因此，我们应尽量避免在闭包中引用循环变量，或者后续会发生变化的变量。

那上面这种情况应该怎么解决呢？我们可以再创建一个函数，并将循环变量的值传给该函数，如下：

```
def count():
    funcs = []
    for i in [1, 2, 3]:
        def g(param):
            f = lambda : param     # 这里创建了一个匿名函数
            return f
        funcs.append(g(i))         # 将循环变量的值传给 g
    return funcs

>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
```

```
14. 2
15. >>> f3()
16. 3
```

小结

- 闭包是携带自由变量的函数，即使创建闭包的外部函数的生命周期结束了，闭包所引用的自由变量仍会存在。
- 闭包在运行可以有多个实例。
- 尽量不要在闭包中引用循环变量，或者后续会发生变化的变量。

参考资料

- 返回函数 - 廖雪峰的官方网站
- Why aren’t python nested functions called closures? - Stack Overflow

装饰器

我们知道，在 Python 中，我们可以像使用变量一样使用函数：

- 函数可以被赋值给其他变量
- 函数可以被删除
- 可以在函数里面再定义函数
- 函数可以作为参数传递给另外一个函数
- 函数可以作为另一个函数的返回

简而言之，函数就是一个对象。

对一个简单的函数进行装饰

为了更好地理解装饰器，我们先从一个简单的例子开始，假设有下面的函数：

```
1.  def hello():
2.      return 'hello world'
```

现在我们想增强 `hello()` 函数的功能，希望给返回加上 HTML 标签，比如 `<i>hello world</i>` ，但是有一个要求，不改变原来 `hello()` 函数的定义。这里当然有很多种方法，下面给出一种跟本文相关的方法：

```
1.  def makeitalic(func):
2.      def wrapped():
3.          return "<i>" + func() + "</i>"
4.      return wrapped
```

在上面的代码中，我们定义了一个函数 `makeitalic` ，该函数有一个参数 `func` ，它是一个函数；在 `makeitalic` 函数里面我们又定义了一个内部函数 `wrapped` ，并将该函数作为返回。

现在，我们就可以不改变 `hello()` 函数的定义，给返回加上 HTML 标签了：

```
1.  >>> hello = makeitalic(hello)  # 将 hello 函数传给 makeitalic
2.  >>> hello()
3.  '<i>hello world</i>'
```

在上面，我们将 `hello` 函数传给 `makeitalic` ，再将返回赋给 `hello` ，此时调用 `hello()` 就得到了我们想要的结果。

不过要注意的是，由于我们将 `makeitalic` 的返回赋给了 `hello`，此时 `hello()` 函数仍然存在，但是它的函数名不再是 hello 了，而是 wrapped，正是 `makeitalic` 返回函数的名称，可以验证一下：

```
>>> hello.__name__
'wrapped'
```

对于这个小瑕疵，后文将会给出解决方法。

现在，我们梳理一下上面的例子，为了增强原函数 `hello` 的功能，我们定义了一个函数，它接收原函数作为参数，并返回一个新的函数，完整的代码如下：

```
def makeitalic(func):
    def wrapped():
        return "<i>" + func() + "</i>"
    return wrapped

def hello():
    return 'hello world'

hello = makeitalic(hello)
```

事实上，`makeitalic` 就是一个装饰器（**decorator**），它『装饰』了函数 `hello`，并返回一个函数，将其赋给 `hello`。

一般情况下，我们使用装饰器提供的 `@` 语法糖（Syntactic Sugar），来简化上面的写法：

```
def makeitalic(func):
    def wrapped():
        return "<i>" + func() + "</i>"
    return wrapped

@makeitalic
def hello():
    return 'hello world'
```

像上面的情况，可以动态修改函数（或类）功能的函数就是装饰器。本质上，它是一个高阶函数，以被装饰的函数（比如上面的 **hello**）为参数，并返回一个包装后的函数（比如上面的 **wrapped**）给被装饰函数（**hello**）。

装饰器的使用形式

- 装饰器的一般使用形式如下：

```
@decorator
def func():
    pass
```

等价于下面的形式：

```
def func():
    pass
func = decorator(func)
```

- 装饰器可以定义多个，离函数定义最近的装饰器先被调用，比如：

```
@decorator_one
@decorator_two
def func():
    pass
```

等价于：

```
def func():
    pass

func = decorator_one(decorator_two(func))
```

- 装饰器还可以带参数，比如：

```
@decorator(arg1, arg2)
def func():
    pass
```

等价于：

```
def func():
    pass

func = decorator(arg1, arg2)(func)
```

下面我们再看一些具体的例子，以加深对它的理解。

对带参数的函数进行装饰

前面的例子中，被装饰的函数 `hello()` 是没有带参数的，我们看看被装饰函数带参数的情况。对前面例子中的 `hello()` 函数进行改写，使其带参数，如下：

```
def makeitalic(func):
    def wrapped(*args, **kwargs):
        ret = func(*args, **kwargs)
        return '<i>' + ret + '</i>'
    return wrapped

@makeitalic
def hello(name):
    return 'hello %s' % name

@makeitalic
def hello2(name1, name2):
    return 'hello %s, %s' % (name1, name2)
```

由于函数 `hello` 带参数，因此内嵌包装函数 `wrapped` 也做了一点改变：

- 内嵌包装函数的参数传给了 `func`，即被装饰函数，也就是说内嵌包装函数的参数跟被装饰函数的参数对应，这里使用了 `(*args, **kwargs)`，是为了适应可变参数。

看看使用：

```
>>> hello('python')
'<i>hello python</i>'
>>> hello2('python', 'java')
'<i>hello python, java</i>'
```

带参数的装饰器

上面的例子，我们增强了函数 `hello` 的功能，给它的返回加上了标签 `<i>...</i>`，现在，我们想改用标签 `<b>...</b>` 或 `<p>...</p>`。是不是要像前面一样，再定义一个类似 `makeitalic` 的装饰器呢？其实，我们可以定义一个函数，将标签作为参数，返回一个装饰器，比如：

```
def wrap_in_tag(tag):
    def decorator(func):
        def wrapped(*args, **kwargs):
```

```
            ret = func(*args, **kwargs)
            return '<' + tag + '>' + ret + '</' + tag + '>'
        return wrapped

    return decorator
```

现在，我们可以根据需要生成想要的装饰器了：

```
makebold = wrap_in_tag('b')  # 根据 'b' 返回 makebold 生成器

@makebold
def hello(name):
    return 'hello %s' % name

>>> hello('world')
'<b>hello world</b>'
```

上面的形式也可以写得更加简洁：

```
@wrap_in_tag('b')
def hello(name):
    return 'hello %s' % name
```

这就是带参数的装饰器，其实就是在装饰器外面多了一层包装，根据不同的参数返回不同的装饰器。

多个装饰器

现在，让我们来看看多个装饰器的例子，为了简单起见，下面的例子就不使用带参数的装饰器。

```
def makebold(func):
    def wrapped():
        return '<b>' + func() + '</b>'

    return wrapped

def makeitalic(func):
    def wrapped():
        return '<i>' + func() + '</i>'

    return wrapped

```

```
@makebold
@makeitalic
def hello():
    return 'hello world'
```

上面定义了两个装饰器，对 `hello` 进行装饰，上面的最后几行代码相当于：

```
def hello():
    return 'hello world'

hello = makebold(makeitalic(hello))
```

调用函数 `hello` ：

```
>>> hello()
'<b><i>hello world</i></b>'
```

基于类的装饰器

前面的装饰器都是一个函数，其实也可以基于类定义装饰器，看下面的例子：

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

    def __call__(self, *args, **kwargs):
        return '<b>' + self.func(*args, **kwargs) + '</b>'

@Bold
def hello(name):
    return 'hello %s' % name

>>> hello('world')
'<b>hello world</b>'
```

可以看到，类 `Bold` 有两个方法：

- `__init__()` ：它接收一个函数作为参数，也就是被装饰的函数
- `__call__()` ：让类对象可调用，就像函数调用一样，在调用被装饰函数时被调用

还可以让类装饰器带参数：

```
class Tag(object):
    def __init__(self, tag):
        self.tag = tag

    def __call__(self, func):
        def wrapped(*args, **kwargs):
            return "<{tag}>{res}</{tag}>".format(
                res=func(*args, **kwargs), tag=self.tag
            )
        return wrapped

@Tag('b')
def hello(name):
    return 'hello %s' % name
```

需要注意的是，如果类装饰器有参数，则 `__init__` 接收参数，而 `__call__` 接收 `func` 。

装饰器的副作用

前面提到，使用装饰器有一个瑕疵，就是被装饰的函数，它的函数名称已经不是原来的名称了，回到最开始的例子：

```
def makeitalic(func):
    def wrapped():
        return "<i>" + func() + "</i>"
    return wrapped

@makeitalic
def hello():
    return 'hello world'
```

函数 `hello` 被 `makeitalic` 装饰后，它的函数名称已经改变了：

```
>>> hello.__name__
'wrapped'
```

为了消除这样的副作用，Python 中的 functools 包提供了一个 wraps 的装饰器：

```
from functools import wraps
```

```
2.
3.  def makeitalic(func):
4.      @wraps(func)       # 加上 wraps 装饰器
5.      def wrapped():
6.          return "<i>" + func() + "</i>"
7.      return wrapped
8.
9.  @makeitalic
10. def hello():
11.     return 'hello world'
12.
13. >>> hello.__name__
14. 'hello'
```

小结

- 本质上，装饰器就是一个返回函数的高阶函数。
- 装饰器可以动态地修改一个类或函数的功能，通过在原有的类或者函数上包裹一层修饰类或修饰函数实现。
- 事实上，装饰器就是闭包的一种应用，但它比较特别，接收被装饰函数为参数，并返回一个函数，赋给被装饰函数，闭包则没这种限制。

参考资料

- Python修饰器的函数式编程 - coolshell
- How can I make a chain of function decorators in Python? - Stack Overflow
- Python中的装饰器介绍 – 思诚之道
- python装饰器入门与提高 | 赖明星

partial 函数

Python 提供了一个 functools 的模块，该模块为高阶函数提供支持，partial 就是其中的一个函数，该函数的形式如下：

```
1.  functools.partial(func[,*args][, **kwargs])
```

这里先举个例子，看看它是怎么用的。

假设有如下函数：

```
1.  def multiply(x, y):
2.      return x * y
```

现在，我们想返回某个数的双倍，即：

```
1.  >>> multiply(3, y=2)
2.  6
3.  >>> multiply(4, y=2)
4.  8
5.  >>> multiply(5, y=2)
6.  10
```

上面的调用有点繁琐，每次都要传入 `y=2` ，我们想到可以定义一个新的函数，把 `y=2` 作为默认值，即：

```
1.  def double(x, y=2):
2.      return multiply(x, y)
```

现在，我们可以这样调用了：

```
1.  >>> double(3)
2.  6
3.  >>> double(4)
4.  8
5.  >>> double(5)
6.  10
```

事实上，我们可以不用自己定义 `double` ，利用 `partial` ，我们可以这样：

```
from functools import partial

double = partial(multiply, y=2)
```

`partial` 接收函数 `multiply` 作为参数，固定 `multiply` 的参数 `y=2`，并返回一个新的函数给 `double`，这跟我们自己定义 `double` 函数的效果是一样的。

所以，简单而言，`partial` 函数的功能就是：把一个函数的某些参数给固定住，返回一个新的函数。

需要注意的是，我们上面是固定了 `multiply` 的关键字参数 `y=2`，如果直接使用：

```
double = partial(multiply, 2)
```

则 `2` 是赋给了 `multiply` 最左边的参数 `x`，不信？我们可以验证一下：

```
from functools import partial

def subtraction(x, y):
    return x - y

f = partial(subtraction, 4)  # 4 赋给了 x
>>> f(10)   # 4 - 10
-6
```

小结

- partial 的功能：固定函数参数，返回一个新的函数。
- 当函数参数太多，需要固定某些参数时，可以使用 `functools.partial` 创建一个新的函数。

参考资料

- 偏函数 - 廖雪峰的官方网站

类

Python 是一门面向对象编程（Object Oriented Programming, OOP）的语言，这里的对象可以看做是由数据（或者说特性）以及一系列可以存取、操作这些数据的方法所组成的集合。面向对象编程主要有以下特点：

- 多态（Polymorphism）：不同类（Class）的对象对同一消息会做出不同的响应。
- 封装（Encapsulation）：对外部世界隐藏对象的工作细节。
- 继承（Inheritance）：以已有的类（父类）为基础建立专门的类对象。

在 Python 中，元组、列表和字典等数据类型是对象，函数也是对象。那么，我们能创建自己的对象吗？答案是肯定的。跟其他 OOP 语言类似，我们使用类来自定义对象。

本章主要介绍以下几个方面：

类和实例

类是一个抽象的概念，我们可以把它理解为具有相同属性和方法的一组对象的集合，而实例则是一个具体的对象。我们还是先来看看在 Python 中怎么定义一个类。

这里以动物（Animal）类为例，Python 提供关键字 `class` 来声明一个类：

```
1.  class Animal(object):
2.      pass
```

其中，`Animal` 是类名，通常类名的首字母采用大写（如果有多个单词，则每个单词的首字母大写），后面紧跟着 `(object)`，表示该类是从哪个类继承而来的，所有类最终都会继承自 `object` 类。

类定义好了，接下来我们就可以创建实例了：

```
1.  >>> animal = Animal()  # 创建一个实例对象
2.  >>> animal
3.  <__main__.Animal at 0x1030a44d0>
```

我们在创建实例的时候，还可以传入一些参数，以初始化对象的属性，为此，我们需要添加一个 `__init__` 方法：

```
1.  class Animal(object):
2.      def __init__(self, name):
3.          self.name = name
```

然后，在创建实例的时候，传入参数：

```
1.  >>> animal = Aniaml('dog1')   # 传入参数 'dog1'
2.  >>> animal.name              # 访问对象的 name 属性
3.  'dog1'
```

我们可以把 `__init__` 理解为对象的初始化方法，它的第一个参数永远是 `self`，指向创建的实例本身。定义了 `__init__` 方法，我们在创建实例的时候，就需要传入与 `__init__` 方法匹配的参数。

接下来，我们再来添加一个方法：

```
1.  class Animal(object):
```

```
    def __init__(self, name):
        self.name = name
    def greet(self):
        print 'Hello, I am %s.' % self.name
```

我们添加了方法 `greet`，看看下面的使用：

```
>>> dog1 = Animal('dog1')
>>> dog1.name
'dog1'
>>> dog1.greet()
Hello, I am dog1.
```

现在，让我们做一下总结。我们在 Animal 类定义了两个方法：`__init__` 和 `greet`。`__init__` 是 Python 中的特殊方法（**special method**），它用于对对象进行初始化，类似于 C++ 中的构造函数；`greet` 是我们自定义的方法。

注意到，我们在上面定义的两个方法有一个共同点，就是它们的第一个参数都是 `self`，指向实例本身，也就是说它们是和实例绑定的函数，这也是我们称它们为方法而不是函数的原因。

访问限制

在某些情况下，我们希望限制用户访问对象的属性或方法，也就是希望它是私有的，对外隐蔽。比如，对于上面的例子，我们希望 `name` 属性在外部不能被访问，我们可以在属性或方法的名称前面加上两个下划线，即 `__`，对上面的例子做一点改动：

```
class Animal(object):
    def __init__(self, name):
        self.__name = name
    def greet(self):
        print 'Hello, I am %s.' % self.__name
```

```
>>> dog1 = Animal('dog1')
>>> dog1.__name   # 访问不了
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-206-7f6730db631e> in <module>()
----> 1 dog1.__name

AttributeError: 'Animal' object has no attribute '__name'
>>> dog1.greet()   # 可以访问
```

```
Hello, I am dog1.
```

可以看到，加了 `__` 的 `__name` 是不能访问的，而原来的 `greet` 仍可以正常访问。

需要注意的是，在 Python 中，以双下划线开头，并且以双下划线结尾（即 `__xxx__` ）的变量是特殊变量，特殊变量是可以直接访问的。所以，不要用 `__name__` 这样的变量名。

另外，如果变量名前面只有一个下划线 `_` ，表示不要随意访问这个变量，虽然它可以直接被访问。

获取对象信息

当我们拿到一个对象时，我们往往会考察它的类型和方法等，比如：

```
>>> a = 123
>>> type(a)
int
>>> b = '123'
>>> type(b)
str
```

当我们拿到一个类的对象时，我们用什么去考察它呢？回到前面的例子：

```
class Animal(object):
    def __init__(self, name):
        self.name = name
    def greet(self):
        print 'Hello, I am %s.' % self.name
```

- 第 1 招：使用 `type`

使用 `type(obj)` 来获取对象的相应类型：

```
>>> dog1 = Animal('dog1')
>>> type(dog1)
__main__.Animal
```

- 第 2 招：使用 `isinstance`

使用 `isinstance(obj, type)` 判断对象是否为指定的 type 类型的实例：

```
>>> isinstance(dog1, Animal)
True
```

- 第 3 招：使用 `hasattr/getattr/setattr`
  - 使用 `hasattr(obj, attr)` 判断对象是否具有指定属性/方法；
  - 使用 `getattr(obj, attr[, default])` 获取属性/方法的值，要是没有对应的属性则返回 default 值（前提是设置了 default），否则会抛出 AttributeError 异常；
  - 使用 `setattr(obj, attr, value)` 设定该属性/方法的值，类似于 obj.attr=value；

看下面例子：

```
>>> hasattr(dog1, 'name')
True
>>> hasattr(dog1, 'x')
False
>>> hasattr(dog1, 'greet')
True
>>> getattr(dog1, 'name')
'dog1'
>>> getattr(dog1, 'greet')
<bound method Animal.greet of <__main__.Animal object at 0x10c3564d0>>
>>> getattr(dog1, 'x')
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-241-42f5b7da1012> in <module>()
----> 1 getattr(dog1, 'x')

AttributeError: 'Animal' object has no attribute 'x'
>>> getattr(dog1, 'x', 'xvalue')
'xvalue'
>>> setattr(dog1, 'age', 12)
>>> dog1.age
12
```

- 第 4 招：使用 `dir`

使用 `dir(obj)` 可以获取相应对象的所有属性和方法名的列表：

```
>>> dir(dog1)
['__class__',
 '__delattr__',
 '__dict__',
 '__doc__',
```

```
 6.   '__format__',
 7.   '__getattribute__',
 8.   '__hash__',
 9.   '__init__',
10.   '__module__',
11.   '__new__',
12.   '__reduce__',
13.   '__reduce_ex__',
14.   '__repr__',
15.   '__setattr__',
16.   '__sizeof__',
17.   '__str__',
18.   '__subclasshook__',
19.   '__weakref__',
20.   'age',
21.   'greet',
22.   'name']
```

小结

- 类是具有相同属性和方法的一组对象的集合，实例是一个个具体的对象。
- 方法是与实例绑定的函数。
- 获取对象信息可使用下面方法：
  - `type(obj)`：来获取对象的相应类型；
  - `isinstance(obj, type)`：判断对象是否为指定的 type 类型的实例；
  - `hasattr(obj, attr)`：判断对象是否具有指定属性/方法；
  - `getattr(obj, attr[, default])` 获取属性/方法的值，要是没有对应的属性则返回 default 值（前提是设置了 default），否则会抛出 AttributeError 异常；
  - `setattr(obj, attr, value)`：设定该属性/方法的值，类似于 obj.attr=value；
  - `dir(obj)`：可以获取相应对象的所有属性和方法名的列表：

参考资料

- Python:类和对象object | Hom

继承和多态

在面向对象编程中，当我们已经创建了一个类，而又想再创建一个与之相似的类，比如添加几个方法，或者修改原来的方法，这时我们不必从头开始，可以从原来的类派生出一个新的类，我们把原来的类称为父类或基类，而派生出的类称为子类，子类继承了父类的所有数据和方法。

让我们看一个简单的例子，首先我们定义一个 Animal 类：

```
1.  class Animal(object):
2.      def __init__(self, name):
3.          self.name = name
4.      def greet(self):
5.          print 'Hello, I am %s.' % self.name
```

现在，我们想创建一个 Dog 类，比如：

```
1.  class Dog(object):
2.      def __init__(self, name):
3.          self.name = name
4.      def greet(self):
5.          print 'WangWang.., I am %s. ' % self.name
```

可以看到，Dog 类和 Animal 类几乎是一样的，只是 `greet` 方法不一样，我们完全没必要创建一个新的类，而是从 Animal 类派生出一个新的类：

```
1.  class Dog(Animal):
2.      def greet(self):
3.          print 'WangWang.., I am %s. ' % self.name
```

Dog 类是从 Animal 类继承而来的，Dog 类自动获得了 Animal 类的所有数据和方法，而且还可以对父类的方法进行修改，我们看看使用：

```
1.  >>> animal = Animal('animal')  # 创建 animal 实例
2.  >>> animal.greet()
3.  Hello, I am animal.
4.  >>>
5.  >>> dog = Dog('dog')         # 创建 dog 实例
6.  >>> dog.greet()
7.  WangWang.., I am dog.
```

我们还可以对 Dog 类添加新的方法：

```
class Dog(Animal):
    def greet(self):
        print 'WangWang.., I am %s. ' % self.name
    def run(self):
        print 'I am running.I am running'
```

使用：

```
>>> dog = Dog('dog')
>>> dog.greet()
WangWang.., I am dog.
>>> dog.run()
I am running
```

多态

多态的概念其实不难理解，它是指对不同类型的变量进行相同的操作，它会根据对象（或类）类型的不同而表现出不同的行为。

事实上，我们经常用到多态的性质，比如：

```
>>> 1 + 2
3
>>> 'a' + 'b'
'ab'
```

可以看到，我们对两个整数进行 `+` 操作，会返回它们的和，对两个字符进行相同的 `+` 操作，会返回拼接后的字符串。也就是说，不同类型的对象对同一消息会作出不同的响应。

再看看类的例子：

```
class Animal(object):
    def __init__(self, name):
        self.name = name
    def greet(self):
        print 'Hello, I am %s.' % self.name

class Dog(Animal):
    def greet(self):
```

```
        print 'WangWang.., I am %s.' % self.name

class Cat(Animal):
    def greet(self):
        print 'MiaoMiao.., I am %s' % self.name

def hello(animal):
    animal.greet()
```

看看多态的使用：

```
>>> dog = Dog('dog')
>>> hello(dog)
WangWang.., I am dog.
>>>
>>> cat = Cat('cat')
>>> hello(cat)
MiaoMiao.., I am cat
```

可以看到，`cat` 和 `dog` 是两个不同的对象，对它们调用 `greet` 方法，它们会自动调用实际类型的 `greet` 方法，作出不同的响应。这就是多态的魅力。

小结

- 继承可以拿到父类的所有数据和方法，子类可以重写父类的方法，也可以新增自己特有的方法。
- 有了继承，才有了多态，不同类的对象对同一消息会作出不同的相应。

类方法和静态方法

在讲类方法和静态方法之前，先来看一个简单的例子：

```
1.  class A(object):
2.      def foo(self):
3.          print 'Hello ', self
4.
5.  >>> a = A()
6.  >>> a.foo()
7.  Hello,  <__main__.A object at 0x10c37a450>
```

在上面，我们定义了一个类 A，它有一个方法 foo，然后我们创建了一个对象 a，并调用方法 foo。

类方法

如果我们想通过类来调用方法，而不是通过实例，那应该怎么办呢？

Python 提供了 `classmethod` 装饰器让我们实现上述功能，看下面的例子：

```
1.  class A(object):
2.      bar = 1
3.      @classmethod
4.      def class_foo(cls):
5.          print 'Hello, ', cls
6.          print cls.bar
7.
8.  >>> A.class_foo()   # 直接通过类来调用方法
9.  Hello,  <class '__main__.A'>
10. 1
```

在上面，我们使用了 `classmethod` 装饰方法 `class_foo`，它就变成了一个类方法，`class_foo` 的参数是 cls，代表类本身，当我们使用 `A.class_foo()` 时，cls 就会接收 A 作为参数。另外，被 `classmethod` 装饰的方法由于持有 cls 参数，因此我们可以在方法里面调用类的属性、方法，比如 `cls.bar`。

静态方法

在类中往往有一些方法跟类有关系，但是又不会改变类和实例状态的方法，这种方法是静态方法，我们

使用 `staticmethod` 来装饰，比如下面的例子：

```
class A(object):
    bar = 1
    @staticmethod
    def static_foo():
        print 'Hello, ', A.bar

>>> a = A()
>>> a.static_foo()
Hello, 1
>>> A.static_foo()
Hello, 1
```

可以看到，静态方法没有 `self` 和 `cls` 参数，可以把它看成是一个普通的函数，我们当然可以把它写到类外面，但这是不推荐的，因为这不利于代码的组织和命名空间的整洁。

小结

- 类方法使用 `@classmethod` 装饰器，可以使用类（也可使用实例）来调用方法。
- 静态方法使用 `@staticmethod` 装饰器，它是跟类有关系但在运行时又不需要实例和类参与的方法，可以使用类和实例来调用。

定制类和魔法方法

在 Python 中，我们可以经常看到以双下划线 `__` 包裹起来的方法，比如最常见的 `__init__` ，这些方法被称为魔法方法（**magic method**）或特殊方法（**special method**）。简单地说，这些方法可以给 Python 的类提供特殊功能，方便我们定制一个类，比如 `__init__` 方法可以对实例属性进行初始化。

完整的特殊方法列表可在这里查看，本文介绍部分常用的特殊方法：

- `__new__`
- `__str__` , `__repr__`
- `__iter__`
- `__getitem__` , `__setitem__` , `__delitem__`
- `__getattr__` , `__setattr__` , `__delattr__`
- `__call__`

new

在 Python 中，当我们创建一个类的实例时，类会先调用 `__new__(cls[, ...])` 来创建实例，然后 `__init__` 方法再对该实例（`self`）进行初始化。

关于 `__new__` 和 `__init__` 有几点需要注意：

- `__new__` 是在 `__init__` 之前被调用的；
- `__new__` 是类方法，`__init__` 是实例方法；
- 重载 `__new__` 方法，需要返回类的实例；

一般情况下，我们不需要重载 `__new__` 方法。但在某些情况下，我们想控制实例的创建过程，这时可以通过重载 `__new_` 方法来实现。

让我们看一个例子：

```
1.  class A(object):
2.      _dict = dict()
3.
4.      def __new__(cls):
5.          if 'key' in A._dict:
6.              print "EXISTS"
7.              return A._dict['key']
8.          else:
9.              print "NEW"
```

```
            return object.__new__(cls)

    def __init__(self):
        print "INIT"
        A._dict['key'] = self
```

在上面，我们定义了一个类 `A`，并重载了 `__new__` 方法：当 `key` 在 `A._dict` 中时，直接返回 `A._dict['key']`，否则创建实例。

执行情况：

```
>>> a1 = A()
NEW
INIT
>>> a2 = A()
EXISTS
INIT
>>> a3 = A()
EXISTS
INIT
```

str & repr

先看一个简单的例子：

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

>>> print Foo('ethan')
<__main__.Foo object at 0x10c37aa50>
```

在上面，我们使用 print 打印一个实例对象，但如果我们想打印更多信息呢，比如把 name 也打印出来，这时，我们可以在类中加入 `__str__` 方法，如下：

```
class Foo(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Foo object (name: %s)' % self.name

```

```
>>> print Foo('ethan')      # 使用 print
Foo object (name: ethan)
>>>
>>> str(Foo('ethan'))       # 使用 str
'Foo object (name: ethan)'
>>>
>>> Foo('ethan')             # 直接显示
<__main__.Foo at 0x10c37a490>
```

可以看到，使用 print 和 str 输出的是 `__str__` 方法返回的内容，但如果直接显示则不是，那能不能修改它的输出呢？当然可以，我们只需在类中加入 `__repr__` 方法，比如：

```
class Foo(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Foo object (name: %s)' % self.name
    def __repr__(self):
        return 'Foo object (name: %s)' % self.name

>>> Foo('ethan')
'Foo object (name: ethan)'
```

可以看到，现在直接使用 `Foo('ethan')` 也可以显示我们想要的结果了，然而，我们发现上面的代码中， `__str__` 和 `__repr__` 方法的代码是一样的，能不能精简一点呢，当然可以，如下：

```
class Foo(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Foo object (name: %s)' % self.name
    __repr__ = __str__
```

iter

在某些情况下，我们希望实例对象可被用于 `for...in` 循环，这时我们需要在类中定义 `__iter__` 和 `next` （在 Python3 中是 `__next__` ）方法，其中， `__iter__` 返回一个迭代对象， `next` 返回容器的下一个元素，在没有后续元素时抛出 `StopIteration` 异常。

看一个斐波那契数列的例子：

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1

    def __iter__(self):  # 返回迭代器对象本身
        return self

    def next(self):      # 返回容器下一个元素
        self.a, self.b = self.b, self.a + self.b
        return self.a

>>> fib = Fib()
>>> for i in fib:
...     if i > 10:
...         break
...     print i
...
1
1
2
3
5
8
```

getitem

有时，我们希望可以使用 `obj[n]` 这种方式对实例对象进行取值，比如对斐波那契数列，我们希望可以取出其中的某一项，这时我们需要在类中实现 `__getitem__` 方法，比如下面的例子：

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in xrange(n):
            a, b = b, a + b
        return a

>>> fib = Fib()
>>> fib[0], fib[1], fib[2], fib[3], fib[4], fib[5]
(1, 1, 2, 3, 5, 8)
```

我们还想更进一步，希望支持 `obj[1:3]` 这种切片方法来取值，这时 `__getitem__` 方法传入

的参数可能是一个整数，也可能是一个切片对象 slice，因此，我们需要对传入的参数进行判断，可以使用 `isinstance` 进行判断，改后的代码如下：

```
class Fib(object):
    def __getitem__(self, n):
        if isinstance(n, slice):   # 如果 n 是 slice 对象
            a, b = 1, 1
            start, stop = n.start, n.stop
            L = []
            for i in xrange(stop):
                if i >= start:
                    L.append(a)
                a, b = b, a + b
            return L
        if isinstance(n, int):     # 如果 n 是 int 型
            a, b = 1, 1
            for i in xrange(n):
                a, b = b, a + b
            return a
```

现在，我们试试用切片方法：

```
>>> fib = Fib()
>>> fib[0:3]
[1, 1, 2]
>>> fib[2:6]
[2, 3, 5, 8]
```

上面，我们只是简单地演示了 getitem 的操作，但是它还很不完善，比如没有对负数处理，不支持带 step 参数的切片操作 `obj[1:2:5]` 等等，读者有兴趣的话可以自己实现看看。

`__geitem__` 用于获取值，类似地，`__setitem__` 用于设置值，`__delitem__` 用于删除值，让我们看下面一个例子：

```
class Point(object):
    def __init__(self):
        self.coordinate = {}

    def __str__(self):
        return "point(%s)" % self.coordinate

    def __getitem__(self, key):
```

```
        return self.coordinate.get(key)

    def __setitem__(self, key, value):
        self.coordinate[key] = value

    def __delitem__(self, key):
        del self.coordinate[key]
        print 'delete %s' % key

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

    __repr__ = __str__
```

在上面，我们定义了一个 Point 类，它有一个属性 coordinate（坐标），是一个字典，让我们看看使用：

```
>>> p = Point()
>>> p['x'] = 2    # 对应于 p.__setitem__('x', 2)
>>> p['y'] = 5    # 对应于 p.__setitem__('y', 5)
>>> p             # 对应于 __repr__
point({'y': 5, 'x': 2})
>>> len(p)        # 对应于 p.__len__
2
>>> p['x']        # 对应于 p.__getitem__('x')
2
>>> p['y']        # 对应于 p.__getitem__('y')
5
>>> del p['x']    # 对应于 p.__delitem__('x')
delete x
>>> p
point({'y': 5})
>>> len(p)
1
```

getattr

当我们获取对象的某个属性，如果该属性不存在，会抛出 AttributeError 异常，比如：

```
class Point(object):
    def __init__(self, x=0, y=0):
```

```
        self.x = x
        self.y = y

>>> p = Point(3, 4)
>>> p.x, p.y
(3, 4)
>>> p.z
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-547-6dce4e43e15c> in <module>()
----> 1 p.z

AttributeError: 'Point' object has no attribute 'z'
```

那有没有办法不让它抛出异常呢？当然有，只需在类的定义中加入 `__getattr__` 方法，比如：

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __getattr__(self, attr):
        if attr == 'z':
            return 0

>>> p = Point(3, 4)
>>> p.z
0
```

现在，当我们调用不存在的属性（比如 z）时，解释器就会试图调用 `__getattr__(self, 'z')` 来获取值，但是，上面的实现还有一个问题，当我们调用其他属性，比如 w ，会返回 None，因为 `__getattr__` 默认返回就是 None，只有当 attr 等于 ‘z’ 时才返回 0，如果我们想让 `__getattr__` 只响应几个特定的属性，可以加入异常处理，修改 `__getattr__` 方法，如下：

```
def __getattr__(self, attr):
    if attr == 'z':
        return 0
    raise AttributeError("Point object has no attribute %s" % attr)
```

这里再强调一点，`__getattr__` 只有在属性不存在的情况下才会被调用，对已存在的属性不会调用 `__getattr__` 。

与 `__getattr__` 一起使用的还有 `__setattr__` ， `__delattr__` ，类似 `obj.attr =`

`value`，`del obj.attr`，看下面一个例子：

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __getattr__(self, attr):
        if attr == 'z':
            return 0
        raise AttributeError("Point object has no attribute %s" % attr)

    def __setattr__(self, *args, **kwargs):
        print 'call func set attr (%s, %s)' % (args, kwargs)
        return object.__setattr__(self, *args, **kwargs)

    def __delattr__(self, *args, **kwargs):
        print 'call func del attr (%s, %s)' % (args, kwargs)
        return object.__delattr__(self, *args, **kwargs)

>>> p = Point(3, 4)
call func set attr (('x', 3), {})
call func set attr (('y', 4), {})
>>> p.z
0
>>> p.z = 7
call func set attr (('z', 7), {})
>>> p.z
7
>>> p.w
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: Point object has no attribute w
>>> p.w = 8
call func set attr (('w', 8), {})
>>> p.w
8
>>> del p.w
call func del attr (('w',), {})
>>> p.__dict__
{'y': 4, 'x': 3, 'z': 7}
```

call

我们一般使用 `obj.method()` 来调用对象的方法，那能不能直接在实例本身上调用呢？在 Python 中，只要我们在类中定义 `__call__` 方法，就可以对实例进行调用，比如下面的例子：

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __call__(self, z):
        return self.x + self.y + z
```

使用如下：

```
>>> p = Point(3, 4)
>>> callable(p)     # 使用 callable 判断对象是否能被调用
True
>>> p(6)            # 传入参数，对实例进行调用，对应 p.__call__(6)
13                  # 3+4+6
```

可以看到，对实例进行调用就好像对函数调用一样。

小结

- `__new__` 在 `__init__` 之前被调用，用来创建实例。
- `__str__` 是用 print 和 str 显示的结果，`__repr__` 是直接显示的结果。
- `__getitem__` 用类似 `obj[key]` 的方式对对象进行取值
- `__getattr__` 用于获取不存在的属性 obj.attr
- `__call__` 使得可以对实例进行调用

参考资料

- design patterns - Python’s use of **new** and **init**? - Stack Overflow
- 定制类
- **setitem** implementation in Python for Point(x,y) class - Stack Overflow
- Python对象的特殊属性和方法 | Hom
- 介绍Python的魔术方法 - Magic Method - 旺酱的专栏 - SegmentFault

slots 魔法

在 Python 中，我们在定义类的时候可以定义属性和方法。当我们创建了一个类的实例后，我们还可以给该实例绑定任意新的属性和方法。

看下面一个简单的例子：

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

>>> p = Point(3, 4)
>>> p.z = 5    # 绑定了一个新的属性
>>> p.z
5
>>> p.__dict__
{'x': 3, 'y': 4, 'z': 5}
```

在上面，我们创建了实例 p 之后，给它绑定了一个新的属性 z，这种动态绑定的功能虽然很有用，但它的代价是消耗了更多的内存。

因此，为了不浪费内存，可以使用 `__slots__` 来告诉 Python 只给一个固定集合的属性分配空间，对上面的代码做一点改进，如下：

```
class Point(object):
    __slots__ = ('x', 'y')       # 只允许使用 x 和 y

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

上面，我们给 `__slots__` 设置了一个元组，来限制类能添加的属性。现在，如果我们想绑定一个新的属性，比如 z，就会出错了，如下：

```
>>> p = Point(3, 4)
>>> p.z = 5
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-648-625ed954d865> in <module>()
```

```
----> 1 p.z = 5

AttributeError: 'Point' object has no attribute 'z'
```

使用 `__slots__` 有一点需要注意的是，`__slots__` 设置的属性仅对当前类有效，对继承的子类不起效，除非子类也定义了 `__slots__`，这样，子类允许定义的属性就是自身的 slots 加上父类的 slots。

小结

- **slots 魔法：**限定允许绑定的属性.
- `__slots__` 设置的属性仅对当前类有效，对继承的子类不起效，除非子类也定义了 slots，这样，子类允许定义的属性就是自身的 slots 加上父类的 slots。

参考资料

- slots魔法 · Python进阶

使用 @property

在使用 `@property` 之前，让我们先来看一个简单的例子：

```
1.  class Exam(object):
2.      def __init__(self, score):
3.          self._score = score
4.
5.      def get_score(self):
6.          return self._score
7.
8.      def set_score(self, val):
9.          if val < 0:
10.             self._score = 0
11.         elif val > 100:
12.             self._score = 100
13.         else:
14.             self._score = val
15.
16. >>> e = Exam(60)
17. >>> e.get_score()
18. 60
19. >>> e.set_score(70)
20. >>> e.get_score()
21. 70
```

在上面，我们定义了一个 Exam 类，为了避免直接对 `_score` 属性操作，我们提供了 get_score 和 set_score 方法，这样起到了封装的作用，把一些不想对外公开的属性隐蔽起来，而只是提供方法给用户操作，在方法里面，我们可以检查参数的合理性等。

这样做没什么问题，但是我们有更简单的方式来做这件事，Python 提供了 `property` 装饰器，被装饰的方法，我们可以将其『当作』属性来用，看下面的例子：

```
1.  class Exam(object):
2.      def __init__(self, score):
3.          self._score = score
4.
5.      @property
6.      def score(self):
7.          return self._score
8.
```

```
    @score.setter
    def score(self, val):
        if val < 0:
            self._score = 0
        elif val > 100:
            self._score = 100
        else:
            self._score = val

>>> e = Exam(60)
>>> e.score
60
>>> e.score = 90
>>> e.score
90
>>> e.score = 200
>>> e.score
100
```

在上面，我们给方法 score 加上了 `@property` ，于是我们可以把 score 当成一个属性来用，此时，又会创建一个新的装饰器 `score.setter` ，它可以把被装饰的方法变成属性来赋值。

另外，我们也不一定要使用 `score.setter` 这个装饰器，这时 score 就变成一个只读属性了：

```
class Exam(object):
    def __init__(self, score):
        self._score = score

    @property
    def score(self):
        return self._score

>>> e = Exam(60)
>>> e.score
60
>>> e.score = 200  # score 是只读属性，不能设置值
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-676-b0515304f6e0> in <module>()
----> 1 e.score = 200

AttributeError: can't set attribute
```

小结

- `@property` 把方法『变成』了属性。

你不知道的 super

在类的继承中，如果重定义某个方法，该方法会覆盖父类的同名方法，但有时，我们希望能同时实现父类的功能，这时，我们就需要调用父类的方法了，可通过使用 `super` 来实现，比如：

```
class Animal(object):
    def __init__(self, name):
        self.name = name
    def greet(self):
        print 'Hello, I am %s.' % self.name

class Dog(Animal):
    def greet(self):
        super(Dog, self).greet()   # Python3 可使用 super().greet()
        print 'WangWang...'
```

在上面，Animal 是父类，Dog 是子类，我们在 Dog 类重定义了 `greet` 方法，为了能同时实现父类的功能，我们又调用了父类的方法，看下面的使用：

```
>>> dog = Dog('dog')
>>> dog.greet()
Hello, I am dog.
WangWang..
```

`super` 的一个最常见用法可以说是在子类中调用父类的初始化方法了，比如：

```
class Base(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

class A(Base):
    def __init__(self, a, b, c):
        super(A, self).__init__(a, b)  # Python3 可使用 super().__init__(a, b)
        self.c = c
```

深入 super()

看了上面的使用，你可能会觉得 `super` 的使用很简单，无非就是获取了父类，并调用父类的方

法。其实，在上面的情况下，super 获得的类刚好是父类，但在其他情况就不一定了，**super** 其实和父类没有实质性的关联。

让我们看一个稍微复杂的例子，涉及到多重继承，代码如下：

```
class Base(object):
    def __init__(self):
        print "enter Base"
        print "leave Base"

class A(Base):
    def __init__(self):
        print "enter A"
        super(A, self).__init__()
        print "leave A"

class B(Base):
    def __init__(self):
        print "enter B"
        super(B, self).__init__()
        print "leave B"

class C(A, B):
    def __init__(self):
        print "enter C"
        super(C, self).__init__()
        print "leave C"
```

其中，Base 是父类，A, B 继承自 Base, C 继承自 A, B，它们的继承关系是一个典型的『菱形继承』，如下：

```
       Base
       /  \
      /    \
     A      B
      \    /
       \  /
        C
```

现在，让我们看一下使用：

```
>>> c = C()
```

```
enter C
enter A
enter B
enter Base
leave Base
leave B
leave A
leave C
```

如果你认为 `super` 代表『调用父类的方法』，那你很可能会疑惑为什么 enter A 的下一句不是 enter Base 而是 enter B。原因是，**`super`** 和父类没有实质性的关联，现在让我们搞清 `super` 是怎么运作的。

MRO 列表

事实上，对于你定义的每一个类，Python 会计算出一个方法解析顺序（**Method Resolution Order, MRO**）列表，它代表了类继承的顺序，我们可以使用下面的方式获得某个类的 MRO 列表：

```
>>> C.mro()   # or C.__mro__ or C().__class__.mro()
[__main__.C, __main__.A, __main__.B, __main__.Base, object]
```

那这个 MRO 列表的顺序是怎么定的呢，它是通过一个 C3 线性化算法来实现的，这里我们就不去深究这个算法了，感兴趣的读者可以自己去了解一下，总的来说，一个类的 MRO 列表就是合并所有父类的 MRO 列表，并遵循以下三条原则：

- 子类永远在父类前面
- 如果有多个父类，会根据它们在列表中的顺序被检查
- 如果对下一个类存在两个合法的选择，选择第一个父类

super 原理

`super` 的工作原理如下：

```
def super(cls, inst):
    mro = inst.__class__.mro()
    return mro[mro.index(cls) + 1]
```

其中，cls 代表类，inst 代表实例，上面的代码做了两件事：

- 获取 inst 的 MRO 列表
- 查找 cls 在当前 MRO 列表中的 index，并返回它的下一个类，即 mro[index + 1]

当你使用 `super(cls, inst)` 时，Python 会在 inst 的 MRO 列表上搜索 cls 的下一个类。

现在，让我们回到前面的例子。

首先看类 C 的 `__init__` 方法：

```
1.  super(C, self).__init__()
```

这里的 self 是当前 C 的实例，self.**class**.mro() 结果是：

```
1.  [__main__.C, __main__.A, __main__.B, __main__.Base, object]
```

可以看到，C 的下一个类是 A，于是，跳到了 A 的 `__init__`，这时会打印出 enter A，并执行下面一行代码：

```
1.  super(A, self).__init__()
```

注意，这里的 self 也是当前 C 的实例，MRO 列表跟上面是一样的，搜索 A 在 MRO 中的下一个类，发现是 B，于是，跳到了 B 的 `__init__`，这时会打印出 enter B，而不是 enter Base。

整个过程还是比较清晰的，关键是要理解 super 的工作方式，而不是想当然地认为 super 调用了父类的方法。

小结

- 事实上，`super` 和父类没有实质性的关联。
- `super(cls, inst)` 获得的是 cls 在 inst 的 MRO 列表中的下一个类。

参考资料

- 调用父类方法 — python3-cookbook 2.0.0 文档
- 理解 Python super - laike9m’s blog
- python super() - 漩涡鸣人 - 博客园
- Python:super函数 | Hom
- Python’s super() considered super! | Deep Thoughts by Raymond Hettinger
- Python super() inheritance and needed arguments - Stack Overflow

陌生的 metaclass

Python 中的元类（**metaclass**）是一个深度魔法，平时我们可能比较少接触到元类，本文将通过一些简单的例子来理解这个魔法。

类也是对象

在 Python 中，一切皆对象。字符串，列表，字典，函数是对象，类也是一个对象，因此你可以：

- 把类赋值给一个变量
- 把类作为函数参数进行传递
- 把类作为函数的返回值
- 在运行时动态地创建类

看一个简单的例子：

```
1.  class Foo(object):
2.      foo = True
3.
4.  class Bar(object):
5.      bar = True
6.
7.  def echo(cls):
8.      print cls
9.
10. def select(name):
11.     if name == 'foo':
12.         return Foo        # 返回值是一个类
13.     if name == 'bar':
14.         return Bar
15.
16. >>> echo(Foo)             # 把类作为参数传递给函数 echo
17. <class '__main__.Foo'>
18. >>> cls = select('foo')   # 函数 select 的返回值是一个类，把它赋给变量 cls
19. >>> cls
20. __main__.Foo
```

熟悉又陌生的 type

在日常使用中，我们经常使用 `object` 来派生一个类，事实上，在这种情况下，Python 解释器会调用 `type` 来创建类。

这里，出现了 `type` ，没错，是你知道的 `type` ，我们经常使用它来判断一个对象的类型，比如：

```
class Foo(object):
    Foo = True

>>> type(10)
<type 'int'>
>>> type('hello')
<type 'str'>
>>> type(Foo())
<class '__main__.Foo'>
>>> type(Foo)
<type 'type'>
```

事实上， **`type`** 除了可以返回对象的类型，它还可以被用来动态地创建类（对象）。下面，我们看几个例子，来消化一下这句话。

使用 `type` 来创建类（对象）的方式如下：

> type(类名, 父类的元组（针对继承的情况，可以为空），包含属性和方法的字典（名称和值））

最简单的情况

假设有下面的类：

```
class Foo(object):
    pass
```

现在，我们不使用 `class` 关键字来定义，而使用 `type` ，如下：

```
Foo = type('Foo', (object, ), {})    # 使用 type 创建了一个类对象
```

上面两种方式是等价的。我们看到， `type` 接收三个参数：

- 第 1 个参数是字符串 ‘Foo’，表示类名
- 第 2 个参数是元组 (object,)，表示所有的父类
- 第 3 个参数是字典，这里是一个空字典，表示没有定义属性和方法

在上面，我们使用 `type()` 创建了一个名为 Foo 的类，然后把它赋给了变量 Foo，我们当然可以把它赋给其他变量，但是，此刻没必要给自己找麻烦。

接着，我们看看使用：

```
>>> print Foo
<class '__main__.Foo'>
>>> print Foo()
<__main__.Foo object at 0x10c34f250>
```

有属性和方法的情况

假设有下面的类：

```
class Foo(object):
    foo = True
    def greet(self):
        print 'hello world'
        print self.foo
```

用 `type` 来创建这个类，如下：

```
def greet(self):
    print 'hello world'
    print self.foo

Foo = type('Foo', (object, ), {'foo': True, 'greet': greet})
```

上面两种方式的效果是一样的，看下使用：

```
>>> f = Foo()
>>> f.foo
True
>>> f.greet
<bound method Foo.greet of <__main__.Foo object at 0x10c34f890>>
>>> f.greet()
hello world
True
```

继承的情况

再来看看继承的情况，假设有如下的父类：

```
class Base(object):
    pass
```

我们用 Base 派生一个 Foo 类，如下：

```
class Foo(Base):
    foo = True
```

改用 `type` 来创建，如下：

```
Foo = type('Foo', (Base, ), {'foo': True})
```

什么是元类（metaclass）

元类（**metaclass**）是用来创建类（对象）的可调用对象。这里的可调用对象可以是函数或者类等。但一般情况下，我们使用类作为元类。对于实例对象、类和元类，我们可以用下面的图来描述：

```
类是实例对象的模板，元类是类的模板

+----------+             +----------+             +----------+
|          |             |          |             |          |
|          | instance of |          | instance of |          |
| instance +------------>+  class   +------------>+ metaclass|
|          |             |          |             |          |
|          |             |          |             |          |
+----------+             +----------+             +----------+
```

我们在前面使用了 `type` 来创建类（对象），事实上，**`type`** 就是一个元类。

那么，元类到底有什么用呢？要你何用…

元类的主要目的是为了控制类的创建行为。我们还是先来看看一些例子，以消化这句话。

元类的使用

先从一个简单的例子开始，假设有下面的类：

```
class Foo(object):
```

```
    name = 'foo'
    def bar(self):
        print 'bar'
```

现在我们想给这个类的方法和属性名称前面加上 `my_` 前缀，即 name 变成 my_name，bar 变成 my_bar，另外，我们还想加一个 echo 方法。当然，有很多种做法，这里展示用元类的做法。

1.首先，定义一个元类，按照默认习惯，类名以 Metaclass 结尾，代码如下：

```
class PrefixMetaclass(type):
    def __new__(cls, name, bases, attrs):
        # 给所有属性和方法前面加上前缀 my_
        _attrs = (('my_' + name, value) for name, value in attrs.items())

        _attrs = dict((name, value) for name, value in _attrs)  # 转化为字典
        _attrs['echo'] = lambda self, phrase: phrase  # 增加了一个 echo 方法

        return type.__new__(cls, name, bases, _attrs)  # 返回创建后的类
```

上面的代码有几个需要注意的点：

- PrefixMetaClass 从 `type` 继承，这是因为 PrefixMetaclass 是用来创建类的
- `__new__` 是在 `__init__` 之前被调用的特殊方法，它用来创建对象并返回创建后的对象，对它的参数解释如下：
    - cls: 当前准备创建的类
    - name: 类的名字
    - bases: 类的父类集合
    - attrs: 类的属性和方法，是一个字典

2.接着，我们需要指示 Foo 使用 PrefixMetaclass 来定制类。

在 Python2 中，我们只需在 Foo 中加一个 `__metaclass__` 的属性，如下：

```
class Foo(object):
    __metaclass__ = PrefixMetaclass
    name = 'foo'
    def bar(self):
        print 'bar'
```

在 Python3 中，这样做：

```
class Foo(metaclass=PrefixMetaclass):
```

```
    name = 'foo'
    def bar(self):
        print 'bar'
```

现在，让我们看看使用：

```
>>> f = Foo()
>>> f.name    # name 属性已经被改变
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-774-4511c8475833> in <module>()
----> 1 f.name

AttributeError: 'Foo' object has no attribute 'name'
>>>
>>> f.my_name
'foo'
>>> f.my_bar()
bar
>>> f.echo('hello')
'hello'
```

可以看到，Foo 原来的属性 name 已经变成了 my_name，而方法 bar 也变成了 my_bar，这就是元类的魔法。

再来看一个继承的例子，下面是完整的代码：

```
class PrefixMetaclass(type):
    def __new__(cls, name, bases, attrs):
        # 给所有属性和方法前面加上前缀 my_
        _attrs = (('my_' + name, value) for name, value in attrs.items())

        _attrs = dict((name, value) for name, value in _attrs)  # 转化为字典
        _attrs['echo'] = lambda self, phrase: phrase  # 增加了一个 echo 方法

        return type.__new__(cls, name, bases, _attrs)

class Foo(object):
    __metaclass__ = PrefixMetaclass   # 注意跟 Python3 的写法有所区别
    name = 'foo'
    def bar(self):
        print 'bar'
```

```
16.
17. class Bar(Foo):
18.     prop = 'bar'
```

其中，PrefixMetaclass 和 Foo 跟前面的定义是一样的，只是新增了 Bar，它继承自 Foo。先让我们看看使用：

```
1.  >>> b = Bar()
2.  >>> b.prop     # 发现没这个属性
3.  ---------------------------------------------------------------------------
4.  AttributeError                            Traceback (most recent call last)
5.  <ipython-input-778-825e0b6563ea> in <module>()
6.  ----> 1 b.prop
7.
8.  AttributeError: 'Bar' object has no attribute 'prop'
9.  >>> b.my_prop
10. 'bar'
11. >>> b.my_name
12. 'foo'
13. >>> b.my_bar()
14. bar
15. >>> b.echo('hello')
16. 'hello'
```

我们发现，Bar 没有 prop 这个属性，但是有 my_prop 这个属性，这是为什么呢？

原来，当我们定义 `class Bar(Foo)` 时，Python 会首先在当前类，即 Bar 中寻找 `__metaclass__`，如果没有找到，就会在父类 Foo 中寻找 `__metaclass__`，如果找不到，就继续在 Foo 的父类寻找，如此继续下去，如果在任何父类都找不到 `__metaclass__`，就会到模块层次中寻找，如果还是找不到，就会用 type 来创建这个类。

这里，我们在 Foo 找到了 `__metaclass__`，Python 会使用 PrefixMetaclass 来创建 Bar，也就是说，元类会隐式地继承到子类，虽然没有显示地在子类使用 `__metaclass__`，这也解释了为什么 Bar 的 prop 属性被动态修改成了 my_prop。

写到这里，不知道你理解元类了没？希望理解了，如果没理解，就多看几遍吧~

小结

- 在 Python 中，类也是一个对象。
- 类创建实例，元类创建类。
- 元类主要做了三件事：

    - 拦截类的创建
    - 修改类的定义
    - 返回修改后的类
- 当你创建类时，解释器会调用元类来生成它，定义一个继承自 `object` 的普通类意味着调用 `type` 来创建它。

参考资料

- oop - What is a metaclass in Python? - Stack Overflow
- 深刻理解Python中的元类(metaclass) - 伯乐在线
- 使用元类 - 廖雪峰的官方网站
- Python基础：元类
- 在Python中使用class decorator和metaclass

高级特性

本章主要介绍：

- 迭代器
- 生成器
- 上下文管理器

迭代器（Iterator）

迭代和可迭代

迭代器这个概念在很多语言中（比如 C++，Java）都是存在的，但是不同语言实现迭代器的方式各不相同。在 **Python** 中，迭代器是指遵循迭代器协议（**iterator protocol**）的对象。至于什么是迭代器协议，稍后自然会说明。为了更好地理解迭代器，我先介绍和迭代器相关的两个概念：

- 迭代（Iteration）
- 可迭代对象（Iterable）

你可能会觉得这是在玩文字游戏，但这确实是要搞清楚的。

> 当我们用一个循环（比如 for 循环）来遍历容器（比如列表，元组）中的元素时，这种遍历的过程就叫迭代。

在 Python 中，我们使用 `for...in...` 进行迭代。比如，遍历一个 list：

```
numbers = [1, 2, 3, 4]
for num in numbers:
    print num
```

像上面这种可以使用 `for` 循环进行迭代的对象，就是可迭代对象，它的定义如下：

> 含有 `__iter__()` 方法或 `__getitem__()` 方法的对象称之为可迭代对象。

我们可以使用 Python 内置的 `hasattr()` 函数来判断一个对象是不是可迭代的：

```
>>> hasattr((), '__iter__')
True
>>> hasattr([], '__iter__')
True
>>> hasattr({}, '__iter__')
True
>>> hasattr(123, '__iter__')
False
>>> hasattr('abc', '__iter__')
False
>>> hasattr('abc', '__getitem__')
True
```

另外，我们也可使用 `isinstance()` 进行判断：

```
>>> from collections import Iterable

>>> isinstance((), Iterable)        # 元组
True
>>> isinstance([], Iterable)        # 列表
True
>>> isinstance({}, Iterable)        # 字典
True
>>> isinstance('abc', Iterable)     # 字符串
True
>>> isinstance(100, Iterable)       # 数字
False
```

可见，我们熟知的字典（dict）、元组（tuple）、集合（set）和字符串对象都是可迭代的。

迭代器

现在，让我们看看什么是迭代器（Iterator）。上文说过，迭代器是指遵循迭代器协议（**iterator protocol**）的对象。从这句话我们可以知道，迭代器是一个对象，但比较特别，它需要遵循迭代器协议，那什么是迭代器协议呢？

> 迭代器协议（***iterator protocol***）是指要实现对象的 `__iter()__` 和 `next()` 方法（注意：Python3 要实现 `__next__()` 方法），其中，`__iter()__` 方法返回迭代器对象本身，`next()` 方法返回容器的下一个元素，在没有后续元素时抛出 `StopIteration` 异常。

接下来讲讲迭代器的例子，有什么常见的迭代器呢？列表是迭代器吗？字典是迭代器吗？我们使用 `hasattr()` 进行判断：

```
>>> hasattr((1, 2, 3), '__iter__')
True
>>> hasattr((1, 2, 3), 'next')  # 有 __iter__ 方法但是没有 next 方法，不是迭代器
False
>>>
>>> hasattr([1, 2, 3], '__iter__')
True
>>> hasattr([1, 2, 3], 'next')
False
>>>
>>> hasattr({'a': 1, 'b': 2}, '__iter__')
True
>>> hasattr({'a': 1, 'b': 2}, 'next')
```

```
False
```

同样，我们也可以使用 `isinstance()` 进行判断：

```
>>> from collections import Iterator
>>>
>>> isinstance((), Iterator)
False
>>> isinstance([], Iterator)
False
>>> isinstance({}, Iterator)
False
>>> isinstance('', Iterator)
False
>>> isinstance(123, Iterator)
False
```

可见，虽然元组、列表和字典等对象是可迭代的，但它们却不是迭代器！对于这些可迭代对象，可以使用 Python 内置的 `iter()` 函数获得它们的迭代器对象，看下面的使用：

```
>>> from collections import Iterator
>>> isinstance(iter([1, 2, 3]), Iterator)  # 使用 iter() 函数，获得迭代器对象
True
>>> isinstance(iter('abc'), Iterator)
True
>>>
>>> my_str = 'abc'
>>> next(my_str)      # my_str 不是迭代器，不能使用 next()，因此出错
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-5f369cd8082f> in <module>()
----> 1 next(my_str)

TypeError: str object is not an iterator
>>>
>>> my_iter = iter(my_str)   # 获得迭代器对象
>>> isinstance(my_iter, Iterator)
True
>>> next(my_iter)   # 可使用内置的 next() 函数获得下一个元素
'a'
```

事实上，Python 的 `for` 循环就是先通过内置函数 `iter()` 获得一个迭代器，然后再不断调

用 `next()` 函数实现的，比如：

```
for x in [1, 2, 3]:
    print i
```

等价于

```
# 获得 Iterator 对象
it = iter([1, 2, 3])

# 循环
while True:
    try:
        # 获得下一个值
        x = next(it)
        print x
    except StopIteration:
        # 没有后续元素，退出循环
        break
```

斐波那契数列迭代器

现在，让我们来自定义一个迭代器：斐波那契（Fibonacci）数列迭代器。根据迭代器的定义，我们需要实现 `__iter()__` 和 `next()` 方法（在 Python3 中是 `__next__()` 方法）。先看代码：

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

from collections import Iterator

class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1

    # 返回迭代器对象本身
    def __iter__(self):
        return self

    # 返回容器下一个元素
    def next(self):
        self.a, self.b = self.b, self.a + self.b
```

```
16.         return self.a
17.
18. def main():
19.     fib = Fib()    # fib 是一个迭代器
20.     print 'isinstance(fib, Iterator): ', isinstance(fib, Iterator)
21.
22.     for i in fib:
23.         if i > 10:
24.             break
25.         print i
26.
27. if __name__ == '__main__':
28.     main()
```

在上面的代码中，我们定义了一个 Fib 类，用于生成 Fibonacci 数列。在类的实现中，我们定义了 `__iter__` 方法，它返回对象本身，这个方法会在遍历时被 Python 内置的 `iter()` 函数调用，返回一个迭代器。类中的 `next()` 方法用于返回容器的下一个元素，当使用 `for` 循环进行遍历的时候，就会使用 Python 内置的 `next()` 函数调用对象的 `next` 方法（在 Python3 中是 `__next__` 方法）对迭代器进行遍历。

运行上面的代码，可得到如下结果：

```
1. isinstance(fib, Iterator):  True
2. 1
3. 1
4. 2
5. 3
6. 5
7. 8
```

小结

- 元组、列表、字典和字符串对象是可迭代的，但不是迭代器，不过我们可以通过 `iter()` 函数获得一个迭代器对象；
- Python 的 `for` 循环实质上是先通过内置函数 `iter()` 获得一个迭代器，然后再不断调用 `next()` 函数实现的；
- 自定义迭代器需要实现对象的 `__iter()__` 和 `next()` 方法（注意：Python3 要实现 `__next__()` 方法），其中，`__iter()__` 方法返回迭代器对象本身，`next()` 方法返回容器的下一个元素，在没有后续元素时抛出 `StopIteration` 异常。

参考资料

- Callback or Iterator in Python
- 迭代器 - 廖雪峰的官方网站

生成器

生成器（generator）也是一种迭代器，在每次迭代时返回一个值，直到抛出 `StopIteration` 异常。它有两种构造方式：

- 生成器表达式

和列表推导式的定义类似，生成器表达式使用 `()` 而不是 `[]`，比如：

```
numbers = (x for x in range(5))   # 注意是(), 而不是[]
for num in numbers:
    print num
```

- 生成器函数

含有 `yield` 关键字的函数，调用该函数时会返回一个生成器。

本文主要讲生成器函数。

生成器函数

先来看一个简单的例子：

```
>>> def generator_function():
...     print 'hello 1'
...     yield 1
...     print 'hello 2'
...     yield 2
...     print 'hello 3'
>>>
>>> g = generator_function()  # 函数没有立即执行，而是返回了一个生成器，当然也是一个迭代器
>>> g.next()  # 当使用 next() 方法，或使用 next(g) 的时候开始执行，遇到 yield 暂停
hello 1
1
>>> g.next()    # 从原来暂停的地方继续执行
hello 2
2
>>> g.next()    # 从原来暂停的地方继续执行，没有 yield，抛出异常
hello 3
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
StopIteration
```

可以看到，上面的函数没有使用 `return` 语句返回值，而是使用了 `yield` 『生出』一个值。一个带有 `yield` 的函数就是一个生成器函数，当我们使用 `yield` 时，它帮我们自动创建了 `__iter__()` 和 `next()` 方法，而且在没有数据时，也会抛出 `StopIteration` 异常，也就是我们不费吹灰之力就获得了一个迭代器，非常简洁和高效。

带有 `yield` 的函数执行过程比较特别：

- 调用该函数的时候不会立即执行代码，而是返回了一个生成器对象；
- 当使用 `next()` （在 for 循环中会自动调用 `next()`）作用于返回的生成器对象时，函数开始执行，在遇到 `yield` 的时候会『暂停』，并返回当前的迭代值；
- 当再次使用 `next()` 的时候，函数会从原来『暂停』的地方继续执行，直到遇到 `yield` 语句，如果没有 `yield` 语句，则抛出异常；

整个过程看起来就是不断地 `执行->中断->执行->中断` 的过程。一开始，调用生成器函数的时候，函数不会立即执行，而是返回一个生成器对象；然后，当我们使用 `next()` 作用于它的时候，它开始执行，遇到 `yield` 语句的时候，执行被中断，并返回当前的迭代值，要注意的是，此刻会记住中断的位置和所有的变量值，也就是执行时的上下文环境被保留起来；当再次使用 `next()` 的时候，从原来中断的地方继续执行，直至遇到 `yield`，如果没有 `yield`，则抛出异常。

简而言之，就是 **`next`** 使函数执行，**`yield`** 使函数暂停。

例子

看一个 Fibonacci 数列的例子，如果使用自定义迭代器的方法，是这样的：

```
>>> class Fib(object):
...     def __init__(self):
...         self.a, self.b = 0, 1
...     def __iter__(self):
...         return self
...     def next(self):
...         self.a, self.b = self.b, self.a + self.b
...         return self.a
...
>>> f = Fib()
>>> for item in f:
...     if item > 10:
...         break
...     print item
```

```
...
1
1
2
3
5
8
```

而使用生成器的方法，是这样的：

```
>>> def fib():
...     a, b = 0, 1
...     while True:
...         a, b = b, a + b
...         yield a
...
>>> f = fib()
>>> for item in f:
...     if item > 10:
...         break
...     print item
...
1
1
2
3
5
8
```

可以看到，使用生成器的方法非常简洁，不用自定义 `__iter__()` 和 `next()` 方法。

另外，在处理大文件的时候，我们可能无法一次性将其载入内存，这时可以通过构造固定长度的缓冲区，来不断读取文件内容。有了 `yield` ，我们就不用自己实现读文件的迭代器了，比如下面的实现：

```
def read_in_chunks(file_object, chunk_size=1024):
    """Lazy function (generator) to read a file piece by piece.
    Default chunk size: 1k."""
    while True:
        data = file_object.read(chunk_size)
        if not data:
            break
```

```
        yield data

f = open('really_big_file.dat')
for piece in read_in_chunks(f):
    process_data(piece)
```

进阶使用

我们除了能对生成器进行迭代使它返回值外，还能：

- 使用 `send()` 方法给它发送消息；
- 使用 `throw()` 方法给它发送异常；
- 使用 `close()` 方法关闭生成器；

send() 方法

看一个简单的例子：

```
>>> def generator_function():
...     value1 = yield 0
...     print 'value1 is ', value1
...     value2 = yield 1
...     print 'value2 is ', value2
...     value3 = yield 2
...     print 'value3 is ', value3
...
>>> g = generator_function()
>>> g.next()   # 调用 next() 方法开始执行，返回 0
0
>>> g.send(2)
value1 is  2
1
>>> g.send(3)
value2 is  3
2
>>> g.send(4)
value3 is  4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

在上面的代码中，我们先调用 `next()` 方法，使函数开始执行，代码执行到 `yield 0` 的时候暂停，返回了 0；接着，我们执行了 `send()` 方法，它会恢复生成器的运行，并将发送的值赋给上次中断时 yield 表达式的执行结果，也就是 value1，这时控制台打印出 value1 的值，并继续执行，直到遇到 yield 后暂停，此时返回 1；类似地，再次执行 `send()` 方法，将值赋给 value2。

简单地说，`send()` 方法就是 `next()` 的功能，加上传值给 `yield`。

throw() 方法

除了可以给生成器传值，我们还可以给它传异常，比如：

```
>>> def generator_function():
...     try:
...         yield 'Normal'
...     except ValueError:
...         yield 'Error'
...     finally:
...         print 'Finally'
...
>>> g = generator_function()
>>> g.next()
'Normal'
>>> g.throw(ValueError)
'Error'
>>> g.next()
Finally
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

可以看到，`throw()` 方法向生成器函数传递了 `ValueError` 异常，此时代码进入 except ValueError 语句，遇到 yield ‘Error’，暂停并返回 Error 字符串。

简单的说，`throw()` 就是 `next()` 的功能，加上传异常给 `yield`。

close() 方法

我们可以使用 `close()` 方法来关闭一个生成器。生成器被关闭后，再次调用 next() 方法，不管能否遇到 yield 关键字，都会抛出 StopIteration 异常，比如：

```
>>> def generator_function():
...     yield 1
...     yield 2
...     yield 3
...
>>> g = generator_function()
>>>
>>> g.next()
1
>>> g.close()  # 关闭生成器
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

小结

- yield 把函数变成了一个生成器。
- 生成器函数的执行过程看起来就是不断地 `执行->中断->执行->中断` 的过程。
    - 一开始，调用生成器函数的时候，函数不会立即执行，而是返回一个生成器对象；
    - 然后，当我们使用 `next()` 作用于它的时候，它开始执行，遇到 `yield` 语句的时候，执行被中断，并返回当前的迭代值，要注意的是，此刻会记住中断的位置和所有的数据，也就是执行时的上下文环境被保留起来；
    - 当再次使用 `next()` 的时候，从原来中断的地方继续执行，直至遇到 `yield`，如果没有 `yield`，则抛出异常。

参考资料

- Python yield 使用浅析
- 谈谈Python的生成器 – 思诚之道
- Function vs Generator in Python
- Lazy Method for Reading Big File in Python? - Stack Overflow

上下文管理器

什么是上下文？其实我们可以简单地把它理解成环境。从一篇文章中抽出一句话，让你来理解，我们会说这是断章取义。为什么？因为我们压根就没考虑到这句话的上下文是什么。编程中的上下文也与此类似，比如『进程上下文』，指的是一个进程在执行的时候，CPU 的所有寄存器中的值、进程的状态以及堆栈上的内容等，当系统需要切换到其他进程时，系统会保留当前进程的上下文，也就是运行时的环境，以便再次执行该进程。

迭代器有迭代器协议（Iterator Protocol），上下文管理器（Context manager）也有上下文管理协议（Context Management Protocol）。

- 上下文管理器协议，是指要实现对象的 `__enter__()` 和 `__exit__()` 方法。
- 上下文管理器也就是支持上下文管理器协议的对象，也就是实现了 `__enter__()` 和 `__exit__()` 方法。

这里先构造一个简单的上下文管理器的例子，以理解 `__enter__()` 和 `__exit__()` 方法。

```
1.  from math import sqrt, pow
2.
3.  class Point(object):
4.      def __init__(self, x, y):
5.          print 'initialize x and y'
6.          self.x, self.y = x, y
7.
8.      def __enter__(self):
9.          print "Entering context"
10.         return self
11.
12.     def __exit__(self, type, value, traceback):
13.         print "Exiting context"
14.
15.     def get_distance(self):
16.         distance = sqrt(pow(self.x, 2) + pow(self.y, 2))
17.         return distance
```

上面的代码定义了一个 `Point` 类，并实现了 `__enter__()` 和 `__exit__()` 方法，我们还定义了 `get_distance` 方法，用于返回点到原点的距离。

通常，我们使用 `with` 语句调用上下文管理器：

```
1.  with Point(3, 4) as pt:
```

```
    print 'distance: ', pt.get_distance()

# output
initialize x and y    # 调用了 __init__ 方法
Entering context      # 调用了 __enter__ 方法
distance:  5.0        # 调用了 get_distance 方法
Exiting context       # 调用了 __exit__ 方法
```

上面的 `with` 语句执行过程如下：

- Point(3, 4) 生成了一个上下文管理器；
- 调用上下文管理器的 `__enter__()` 方法，并将 `__enter__()` 方法的返回值赋给 as 字句中的变量 pt；
- 执行语句体（指 with 语句包裹起来的代码块）内容，输出 distance；
- 不管执行过程中是否发生异常，都执行上下文管理器的 `__exit__()` 方法。`__exit__()` 方法负责执行『清理』工作，如释放资源，关闭文件等。如果执行过程没有出现异常，或者语句体中执行了语句 break/continue/return，则以 None 作为参数调用 `__exit__(None, None, None)`；如果执行过程中出现异常，则使用 `sys.exc_info` 得到的异常信息为参数调用 `__exit__(exc_type, exc_value, exc_traceback)`；
- 出现异常时，如果 `__exit__(type, value, traceback)` 返回 False 或 None，则会重新抛出异常，让 with 之外的语句逻辑来处理异常；如果返回 True，则忽略异常，不再对异常进行处理；

上面的 with 语句执行过程没有出现异常，我们再来看出现异常的情形：

```
with Point(3, 4) as pt:
    pt.get_length()        # 访问了对象不存在的方法

# output
initialize x and y
Entering context
Exiting context
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-216-ab4a0e6b6b4a> in <module>()
      1 with Point(3, 4) as pt:
----> 2     pt.get_length()

AttributeError: 'Point' object has no attribute 'get_length'
```

在我们的例子中，`__exit__` 方法返回的是 None（如果没有 return 语句那么方法会返回 None）。因此，with 语句抛出了那个异常。我们对 `__exit__` 方法做一些改动，让它返回

True。

```
from math import sqrt, pow

class Point(object):
    def __init__(self, x, y):
        print 'initialize x and y'
        self.x, self.y = x, y

    def __enter__(self):
        print "Entering context"
        return self

    def __exit__(self, type, value, traceback):
        print "Exception has been handled"
        print "Exiting context"
        return True

    def get_distance(self):
        distance = sqrt(pow(self.x, 2) + pow(self.y,2 ))
        return distance

with Point(3, 4) as pt:
    pt.get_length()      # 访问了对象不存在的方法

# output
initialize x and y
Entering context
Exception has been handled
Exiting context
```

可以看到，由于 `__exit__` 方法返回了 True，因此没有异常会被 with 语句抛出。

内建对象使用 with 语句

除了自定义上下文管理器，Python 中也提供了一些内置对象，可直接用于 with 语句中，比如最常见的文件操作。

传统的文件操作经常使用 `try/finally` 的方式，比如：

```
file = open('somefile', 'r')
try:
```

```
    for line in file:
        print line
finally:
    file.close()     # 确保关闭文件
```

将上面的代码改用 with 语句:

```
with open('somefile', 'r') as file:
    for line in file:
        print line
```

可以看到，通过使用 with，代码变得很简洁，而且即使处理过程发生异常，with 语句也会确保我们的文件被关闭。

contextlib 模块

除了在类中定义 `__enter__` 和 `__exit__` 方法来实现上下文管理器，我们还可以通过生成器函数（也就是带有 yield 的函数）结合装饰器来实现上下文管理器，Python 中自带的 contextlib 模块就是做这个的。

contextlib 模块提供了三个对象：装饰器 contextmanager、函数 nested 和上下文管理器 closing。其中，contextmanager 是一个装饰器，用于装饰生成器函数，并返回一个上下文管理器。需要注意的是，被装饰的生成器函数只能产生一个值，否则会产生 RuntimeError 异常。

下面我们看一个简单的例子:

```
from contextlib import contextmanager

@contextmanager
def point(x, y):
    print 'before yield'
    yield x * x + y * y
    print 'after yield'

with point(3, 4) as value:
    print 'value is: %s' % value

# output
before yield
value is: 25
after yield
```

可以看到，yield 产生的值赋给了 as 子句中的 value 变量。

另外，需要强调的是，虽然通过使用 contextmanager 装饰器，我们可以不必再编写 `__enter__` 和 `__exit__` 方法，但是『获取』和『清理』资源的操作仍需要我们自己编写：『获取』资源的操作定义在 yield 语句之前，『释放』资源的操作定义在 yield 语句之后。

小结

- 上下文管理器是支持上下文管理协议的对象，也就是实现了 `__enter__` 和 `__exit__` 方法。
- 通常，我们使用 `with` 语句调用上下文管理器。with 语句尤其适用于对资源进行访问的场景，确保执行过程中出现异常情况时也可以对资源进行回收，比如自动关闭文件等。
- `__enter__` 方法在 with 语句体执行前调用，with 语句将该方法的返回值赋给 as 字句中的变量，如果有 as 字句的话。
- `__exit__` 方法在退出`运行时上下文`时被调用，它负责执行『清理』工作，比如关闭文件，释放资源等。如果退出时没有发生异常，则 `__exit__` 的三个参数，即 type, value 和 traceback 都为 None。如果发生异常，返回 True 表示不处理异常，否则会在退出该方法后重新抛出异常以由 with 语句之外的代码逻辑进行处理。

参考资料

- 编程中什么是「Context(上下文)」？ - 知乎
- 浅谈 Python 的 with 语句
- 上下文管理器 · Python进阶

文件和目录

本章主要介绍文件和目录操作，包含以下部分：

- 读写文本文件
- 读写二进制文件
- os 模块

读写文本文件

读写文件是最常见的 IO 操作。通常，我们使用 `input` 从控制台读取输入，使用 `print` 将内容输出到控制台。实际上，我们也经常从文件读取输入，将内容写到文件。

读文件

在 Python 中，读文件主要分为三个步骤：

- 打开文件
- 读取内容
- 关闭文件

一般使用形式如下：

```
try:
    f = open('/path/to/file', 'r')    # 打开文件
    data = f.read()                   # 读取文件内容
finally:
    if f:
        f.close()                     # 确保文件被关闭
```

注意到，我们在代码中加了 `try...finally`，这是因为，如果打开和读取文件时出现错误，文件就没有被关闭。为了确保在任何情况下，文件都能被关闭，我们加了 `try...finally`。

上面的代码中，'r' 模式表示读模式，`open` 函数的常用模式主要有：

'r'	读模式
'w'	写模式
'a'	追加模式
'b'	二进制模式（可添加到其他模式中使用）
'+'	读/写模式（可添加到其他模式中使用）

上面的读文件做法很繁琐，我们可以使用 Python 的 `with` 语句来帮我们自动调用 `close` 方法：

```
with open('/path/to/file', 'r') as f:
    data = f.read()
```

可以看到，这种方式很简洁，而且还能在出现异常的情况下自动关闭文件。

通常而言，读取文件有以下几种方式：

- 一次性读取所有内容，使用 `read()` 或 `readlines()`；
- 按字节读取，使用 `read(size)`；
- 按行读取，使用 `readline()`；

读取所有内容

读取所有内容可以使用 `read()` 或 `readlines()`。我们在上面已经介绍过 `read()` 了，现在，让我们看看 `readlines()`。

`readlines()` 方法会把文件读入一个字符串列表，在列表中每个字符串就是一行。

假设有一个文件 data.txt，它的文件内容如下（数字之间的间隔符是'\t'）：

```
10  1   9   9
6   3   2   8
20  10  3   23
1   4   1   10
10  8   6   3
10  2   1   6
```

我们使用 `readlines()` 将文件读入一个字符串列表：

```
with open('data.txt', 'r') as f:
    lines = f.readlines()
    line_num = len(lines)
    print lines
    print line_num
```

执行结果：

```
['10\t1\t9\t9\n', '6\t3\t2\t8\n', '20\t10\t3\t23\n', '1\t4\t1\t10\n',
'10\t8\t6\t3\n', '10\t2\t1\t6']
6
```

可以看到，列表中的每个元素都是一个字符串，刚好对应文件中的每一行。

按字节读取

如果文件较小，一次性读取所有内容确实比较方便。但是，如果文件很大，比如有 100G，那就不能一

次性读取所有内容了。这时，我们构造一个固定长度的缓冲区，来不断读取文件内容。

看看例子：

```
with open('path/to/file', 'r') as f:
    while True:
        piece = f.read(1024)        # 每次读取 1024 个字节（即 1 KB）的内容
        if not piece:
            break
        print piece
```

在上面，我们使用 `f.read(1024)` 来每次读取 1024 个字节（1KB） 的文件内容，将其存到 piece，再对 piece 进行处理。

事实上，我们还可以结合 yield 来使用：

```
def read_in_chunks(file_object, chunk_size=1024):
    """Lazy function (generator) to read a file piece by piece.
    Default chunk size: 1k."""
    while True:
        data = file_object.read(chunk_size)
        if not data:
            break
        yield data

with open('path/to/file', 'r') as f:
    for piece in read_in_chunks(f):
        print piece
```

逐行读取

在某些情况下，我们希望逐行读取文件，这时可以使用 `readline()` 方法。

看看例子：

```
with open('data.txt', 'r') as f:
    while True:
        line = f.readline()     # 逐行读取
        if not line:
            break
        print line,             # 这里加了 ',' 是为了避免 print 自动换行
```

执行结果：

```
10  1   9   9
6   3   2   8
20  10  3   23
1   4   1   10
10  8   6   3
10  2   1   6
```

文件迭代器

在 Python 中，文件对象是可迭代的，这意味着我们可以直接在 for 循环中使用它们，而且是逐行迭代的，也就是说，效果和 `readline()` 是一样的，而且更简洁。

看看例子：

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

在上面的代码中，f 就是一个文件迭代器，因此我们可以直接使用 `for line in f` ，它是逐行迭代的。

看看执行结果：

```
10  1   9   9
6   3   2   8
20  10  3   23
1   4   1   10
10  8   6   3
10  2   1   6
```

再看一个例子：

```
with open(file_path, 'r') as f:
    lines = list(f)
    print lines
```

执行结果：

```
    ['10\t1\t9\t9\n', '6\t3\t2\t8\n', '20\t10\t3\t23\n', '1\t4\t1\t10\n',
1.  '10\t8\t6\t3\n', '10\t2\t1\t6']
```

可以看到，我们可以对文件迭代器执行和普通迭代器相同的操作，比如上面使用 `list(open(filename))` 将 f 转为一个字符串列表，这样所达到的效果和使用 `readlines` 是一样的。

写文件

写文件使用 `write` 方法，如下：

```
1.  with open('/Users/ethan/data2.txt', 'w') as f:
2.      f.write('one\n')
3.      f.write('two')
```

- 如果上述文件已存在，则会清空原内容并覆盖掉；
- 如果上述路径是正确的（比如存在 /Users/ethan 的路径），但是文件不存在（data2.txt 不存在），则会新建一个文件，并写入上述内容；
- 如果上述路径是不正确的（比如将路径写成 /Users/eth ），这时会抛出 IOError；

如果我们想往已存在的文件追加内容，可以使用 ‘a’ 模式，如下：

```
1.  with open('/Users/ethan/data2.txt', 'a') as f:
2.      f.write('three\n')
3.      f.write('four')
```

小结

- 推荐使用 with 语句操作文件 IO。
- 如果文件较大，可以按字节读取或按行读取。
- 使用文件迭代器进行逐行迭代。

参考资料

- 《Python 基础教程》
- 读写文本数据 — python3-cookbook 2.0.0 文档

读写二进制文件

Python 不仅支持文本文件的读写，也支持二进制文件的读写，比如图片，声音文件等。

读取二进制文件

读取二进制文件使用 ‘rb’ 模式。

这里以图片为例：

```
1. with open('test.png', 'rb') as f:
2.     image_data = f.read()    # image_data 是字节字符串格式的，而不是文本字符串
```

这里需要注意的是，在读取二进制数据时，返回的数据是字节字符串格式的，而不是文本字符串。一般情况下，我们可能会对它进行编码，比如 base64 编码，可以这样做：

```
1. import base64
2.
3. with open('test.png', 'rb') as f:
4.     image_data = f.read()
5.     base64_data = base64.b64encode(image_data)    # 使用 base64 编码
6.     print base64_data
```

下面是执行结果的一部分：

```
1. iVBORw0KGgoAAAANSUhEUgAAAQAAAAEACAYAAABccqhmAAAACGFjVEw
```

写入二进制文件

写入二进制文件使用 ‘wb’ 模式。

以图片为例：

```
1. with open('test.png', 'rb') as f:
2.     image_data = f.read()
3.
4. with open('/Users/ethan/test2.png', 'wb') as f:
5.     f.write(image_data)
```

小结

- 读取二进制文件使用 ‘rb’ 模式。
- 写入二进制文件使用 ‘wb’ 模式。

参考资料

- 读写字节数据 — python3-cookbook 2.0.0 文档

os 模块

Python 的 os 模块封装了常见的文件和目录操作，本文只列出部分常用的方法，更多的方法可以查看官方文档。

下面是部分常见的用法：

方法	说明
os.mkdir	创建目录
os.rmdir	删除目录
os.rename	重命名
os.remove	删除文件
os.getcwd	获取当前工作路径
os.walk	遍历目录
os.path.join	连接目录与文件名
os.path.split	分割文件名与目录
os.path.abspath	获取绝对路径
os.path.dirname	获取路径
os.path.basename	获取文件名或文件夹名
os.path.splitext	分离文件名与扩展名
os.path.isfile	判断给出的路径是否是一个文件
os.path.isdir	判断给出的路径是否是一个目录

例子

后文的例子以下面的目录结构为参考，工作目录为 `/Users/ethan/coding/python` 。

```
Users/ethan
└── coding
    └── python
        ├── hello.py    - 文件
        └── web         - 目录
```

看看例子：

- os.path.abspath: 获取文件或目录的绝对路径

```
$ pwd
```

```
/Users/ethan/coding/python
$ python
>>> import os                              # 记得导入 os 模块
>>> os.path.abspath('hello.py')
'/Users/ethan/coding/python/hello.py'
>>> os.path.abspath('web')
'/Users/ethan/coding/python/web'
>>> os.path.abspath('.')                   # 当前目录的绝对路径
'/Users/ethan/coding/python'
```

- os.path.dirname: 获取文件或文件夹的路径

```
>>> os.path.dirname('/Users/ethan/coding/python/hello.py')
'/Users/ethan/coding/python'
>>> os.path.dirname('/Users/ethan/coding/python/')
'/Users/ethan/coding/python'
>>> os.path.dirname('/Users/ethan/coding/python')
'/Users/ethan/coding'
```

- os.path.basename: 获取文件名或文件夹名

```
>>> os.path.basename('/Users/ethan/coding/python/hello.py')
'hello.py'
>>> os.path.basename('/Users/ethan/coding/python/')
''
>>> os.path.basename('/Users/ethan/coding/python')
'python'
```

- os.path.splitext: 分离文件名与扩展名

```
>>> os.path.splitext('/Users/ethan/coding/python/hello.py')
('/Users/ethan/coding/python/hello', '.py')
>>> os.path.splitext('/Users/ethan/coding/python')
('/Users/ethan/coding/python', '')
>>> os.path.splitext('/Users/ethan/coding/python/')
('/Users/ethan/coding/python/', '')
```

- os.path.split: 分离目录与文件名

```
>>> os.path.split('/Users/ethan/coding/python/hello.py')
('/Users/ethan/coding/python', 'hello.py')
```

```
>>> os.path.split('/Users/ethan/coding/python/')
('/Users/ethan/coding/python', '')
>>> os.path.split('/Users/ethan/coding/python')
('/Users/ethan/coding', 'python')
```

- os.path.isfile/os.path.isdir

```
>>> os.path.isfile('/Users/ethan/coding/python/hello.py')
True
>>> os.path.isdir('/Users/ethan/coding/python/')
True
>>> os.path.isdir('/Users/ethan/coding/python')
True
>>> os.path.isdir('/Users/ethan/coding/python/hello.py')
False
```

- os.walk

os.walk 是遍历目录常用的模块，它返回一个包含 3 个元素的元组：(dirpath, dirnames, filenames)。dirpath 是以 string 字符串形式返回该目录下所有的绝对路径；dirnames 是以列表 list 形式返回每一个绝对路径下的文件夹名字；filenames 是以列表 list 形式返回该路径下所有文件名字。

```
>>> for root, dirs, files in os.walk('/Users/ethan/coding'):
...     print root
...     print dirs
...     print files
...
/Users/ethan/coding
['python']
[]
/Users/ethan/coding/python
['web2']
['hello.py']
/Users/ethan/coding/python/web2
[]
[]
```

参考资料

- 关于python文件操作 - Rollen Holt

- 操作文件和目录 - 廖雪峰的官方网站
- Python os.walk的用法与举例

进程、线程和协程

操作系统的设计，可以归结为三点：

- 以多进程形式，允许多个任务同时运行；
- 以多线程形式，允许将单个任务分成多个子任务运行；
- 提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

本章主要介绍在 Python 中如何进行进程和线程编程等，主要有以下几个方面：

- 进程
- 线程
- ThreadLocal
- 协程

参考资料

- 进程和线程 - 廖雪峰的官方网站
- 进程与线程的一个简单解释 - 阮一峰的网络日志

进程

进程（**process**）是正在运行的程序的实例，但一个程序可能会产生多个进程。比如，打开 Chrome 浏览器程序，它可能会产生多个进程，主程序需要一个进程，一个网页标签需要一个进程，一个插件也需要一个进程，等等。

每个进程都有自己的地址空间，内存，数据栈以及其他记录其运行状态的辅助数据，不同的进程只能使用消息队列、共享内存等进程间通讯（IPC）方法进行通信，而不能直接共享信息。

fork()

在介绍 Python 的进程编程之前，让我们先看看 Unix/Linux 中的 `fork` 函数。在 Unix/Linux 系统中，`fork` 函数被用于创建进程。这个函数很特殊，对于普通的函数，调用它一次，返回一次，但是调用 `fork` 一次，它返回两次。事实上，`fork` 函数创建了新的进程，我们把它称为子进程，子进程几乎是当前进程（即父进程）的一个拷贝：它会复制父进程的代码段，堆栈段和数据段。

对于父进程，`fork` 函数返回了子进程的进程号 pid，对于子进程，`fork` 函数则返回 `0`，这也是 `fork` 函数返回两次的原因，根据返回值，我们可以判断进程是父进程还是子进程。

下面我们看一段 C 代码，它展示了 fork 的基本使用：

```
#include <unistd.h>
#include <stdio.h>

int main(int argc, char const *argv[])
{
    int pid;
    pid = fork();    // 使用 fork 函数

    if (pid < 0) {
        printf("Fail to create process\n");
    }
    else if (pid == 0) {
        printf("I am child process (%d) and my parent is (%d)\n", getpid(),
getppid());
    }
    else {
        printf("I (%d) just created a child process (%d)\n", getpid(), pid);
    }
    return 0;
```

```
}
```

其中，`getpid` 用于获取当前进程号，`getppid` 用于获取父进程号。

事实上，Python 的 os 模块包含了普遍的操作系统功能，该模块也提供了 `fork` 函数，把上面的代码改成用 Python 来实现，如下：

```
import os

pid = os.fork()

if pid < 0:
    print 'Fail to create process'
elif pid == 0:
    print 'I am child process (%s) and my parent is (%s).' % (os.getpid(),
os.getppid())
else:
    print 'I (%s) just created a child process (%s).' % (os.getpid(), pid)
```

运行上面的代码，产生如下输出：

```
I (86645) just created a child process (86646).
I am child process (86646) and my parent is (86645).
```

需要注意的是，虽然子进程复制了父进程的代码段和数据段等，但是一旦子进程开始运行，子进程和父进程就是相互独立的，它们之间不再共享任何数据。

多进程

Python 提供了一个 multiprocessing 模块，利用它，我们可以来编写跨平台的多进程程序，但需要注意的是 multiprocessing 在 Windows 和 Linux 平台的不一致性：一样的代码在 Windows 和 Linux 下运行的结果可能不同。因为 Windows 的进程模型和 Linux 不一样，Windows 下没有 fork。

我们先来看一个简单的例子，该例子演示了在主进程中启动一个子进程，并等待其结束，代码如下：

```
import os
from multiprocessing import Process

# 子进程要执行的代码
def child_proc(name):
```

```
    print 'Run child process %s (%s)...' % (name, os.getpid())

if __name__ == '__main__':
    print 'Parent process %s.' % os.getpid()
    p = Process(target=child_proc, args=('test',))
    print 'Process will start.'
    p.start()
    p.join()
    print 'Process end.'
```

在上面的代码中，我们从 multiprocessing 模块引入了 Process，Process 是一个用于创建进程对象的类，其中，target 指定了进程要执行的函数，args 指定了参数。在创建了进程实例 p 之后，我们调用 start 方法开始执行该子进程，接着，我们又调用了 join 方法，该方法用于阻塞子进程以外的所有进程（这里指父进程），当子进程执行完毕后，父进程才会继续执行，它通常用于进程间的同步。

可以看到，用上面这种方式来创建进程比直接使用 `fork` 更简单易懂。现在，让我们看下输出结果：

```
Parent process 7170.
Process will start.
Run child process test (10075)...
Process end.
```

multiprocessing 与平台有关

```
import random
import os
from multiprocessing import Process

num = random.randint(0, 100)

def show_num():
    print("pid:{}, num is {}".format(os.getpid(), num))

if __name__ == "__main__":
    print("pid:{}, num is {}".format(os.getpid(), num))
    p = Process(target=show_num)
    p.start()
    p.join()
```

在 Windows 下运行以上代码，输出的结果如下（你得到不一样的结果也是对的）：

```
pid:6504, num is 25
pid:6880, num is 6
```

我们发现，num 的值是不一样的！

在 Linux 下运行以上代码，可以看到 num 的值是一样的：

```
pid:11747, num is 13
pid:11748, num is 13
```

使用进程池创建多个进程

在上面，我们只是创建了一个进程，如果要创建多个进程呢？Python 提供了进程池的方式，让我们批量创建子进程，让我们看一个简单的示例：

```
import os, time
from multiprocessing import Pool

def foo(x):
    print 'Run task %s (pid:%s)...' % (x, os.getpid())
    time.sleep(2)
    print 'Task %s result is: %s' % (x, x * x)

if __name__ == '__main__':
    print 'Parent process %s.' % os.getpid()
    p = Pool(4)         # 设置进程数
    for i in range(5):
        p.apply_async(foo, args=(i,))    # 设置每个进程要执行的函数和参数
    print 'Waiting for all subprocesses done...'
    p.close()
    p.join()
    print 'All subprocesses done.'
```

在上面的代码中，Pool 用于生成进程池，对 Pool 对象调用 apply_async 方法可以使每个进程异步执行任务，也就说不用等上一个任务执行完才执行下一个任务，close 方法用于关闭进程池，确保没有新的进程加入，join 方法会等待所有子进程执行完毕。

看看执行结果：

```
Parent process 7170.
Run task 1 (pid:10320)...
Run task 0 (pid:10319)...
Run task 3 (pid:10322)...
Run task 2 (pid:10321)...
Waiting for all subprocesses done...
Task 1 result is: 1
Task 0 result is: 0
Run task 4 (pid:10320)...
Task 3 result is: 9
Task 2 result is: 4
Task 4 result is: 16
All subprocesses done.
```

进程间通信

进程间的通信可以通过管道（Pipe），队列（Queue）等多种方式来实现。Python 的 multiprocessing 模块封装了底层的实现机制，让我们可以很容易地实现进程间的通信。

下面以队列（Queue）为例，在父进程中创建两个子进程，一个往队列写数据，一个从对列读数据，代码如下：

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

from multiprocessing import Process, Queue

# 向队列中写入数据
def write_task(q):
    try:
        n = 1
        while n < 5:
            print "write, %d" % n
            q.put(n)
            time.sleep(1)
            n += 1
    except BaseException:
        print "write_task error"
    finally:
        print "write_task end"

# 从队列读取数据
```

```
def read_task(q):
    try:
        n = 1
        while n < 5:
            print "read, %d" % q.get()
            time.sleep(1)
            n += 1
    except BaseException:
        print "read_task error"
    finally:
        print "read_task end"

if __name__ == "__main__":
    q = Queue()  # 父进程创建Queue, 并传给各个子进程

    pw = Process(target=write_task, args=(q,))
    pr = Process(target=read_task, args=(q,))

    pw.start()   # 启动子进程 pw, 写入
    pr.start()   # 启动子进程 pr, 读取
    pw.join()    # 等待 pw 结束
    pr.join()    # 等待 pr 结束
    print "DONE"
```

执行结果如下：

```
write, 1
read, 1
write, 2
read, 2
write, 3
read, 3
write, 4
read, 4
write_task end
read_task end
DONE
```

小结

- 进程是正在运行的程序的实例。

- 由于每个进程都有各自的内存空间，数据栈等，所以只能使用进程间通讯（Inter-Process Communication, IPC），而不能直接共享信息。
- Python 的 multiprocessing 模块封装了底层的实现机制，让我们可以更简单地编写多进程程序。

参考资料

- 多进程 - 廖雪峰的官方网站
- Linux下Fork与Exec使用 - hicjiajia - 博客园
- Python 中的进程、线程、协程、同步、异步、回调 - 七牛云存储 - SegmentFault
- 编程中的进程、线程、协程、同步、异步、回调 · 浮生半日闲
- python中多进程以及多线程编程的总结 - Codefly
- multithreading - Python multiprocessing.Pool: when to use apply, apply_async or map? - Stack Overflow

线程

线程（**thread**）是进程（**process**）中的一个实体，一个进程至少包含一个线程。比如，对于视频播放器，显示视频用一个线程，播放音频用另一个线程。如果我们把进程看成一个容器，则线程是此容器的工作单位。

进程和线程的区别主要有：

- 进程之间是相互独立的，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，但互不影响；而同一个进程的多个线程是内存共享的，所有变量都由所有线程共享；
- 由于进程间是独立的，因此一个进程的崩溃不会影响到其他进程；而线程是包含在进程之内的，线程的崩溃就会引发进程的崩溃，继而导致同一进程内的其他线程也奔溃；

多线程

在 Python 中，进行多线程编程的模块有两个：thread 和 threading。其中，thread 是低级模块，threading 是高级模块，对 thread 进行了封装，一般来说，我们只需使用 threading 这个模块。

下面，我们看一个简单的例子：

```
1.  from threading import Thread, current_thread
2.
3.  def thread_test(name):
4.      print 'thread %s is running...' % current_thread().name
5.      print 'hello', name
6.      print 'thread %s ended.' % current_thread().name
7.
8.  if __name__ == "__main__":
9.      print 'thread %s is running...' % current_thread().name
10.     print 'hello world!'
11.     t = Thread(target=thread_test, args=("test",), name="TestThread")
12.     t.start()
13.     t.join()
14.     print 'thread %s ended.' % current_thread().name
```

可以看到，创建一个新的线程，就是把一个函数和函数参数传给 Thread 实例，然后调用 start 方法开始执行。代码中的 current_thread 用于返回当前线程的实例。

执行结果如下：

```
thread MainThread is running...
hello world!
thread TestThread is running...
hello test
thread TestThread ended.
thread MainThread ended.
```

锁

由于同一个进程之间的线程是内存共享的，所以当多个线程对同一个变量进行修改的时候，就会得到意想不到的结果。

让我们先看一个简单的例子：

```
from threading import Thread, current_thread

num = 0

def calc():
    global num
    print 'thread %s is running...' % current_thread().name
    for _ in xrange(10000):
        num += 1
    print 'thread %s ended.' % current_thread().name

if __name__ == '__main__':
    print 'thread %s is running...' % current_thread().name

    threads = []
    for i in range(5):
        threads.append(Thread(target=calc))
        threads[i].start()
    for i in range(5):
        threads[i].join()

    print 'global num: %d' % num
    print 'thread %s ended.' % current_thread().name
```

在上面的代码中，我们创建了 5 个线程，每个线程对全局变量 num 进行 10000 次的 加 1 操作，这里之所以要循环 10000 次，是为了延长单个线程的执行时间，使线程执行时能出现中断切换的情况。现在问题来了，当这 5 个线程执行完毕时，全局变量的值是多少呢？是 50000 吗？

让我们看下执行结果：

```
thread MainThread is running...
thread Thread-34 is running...
thread Thread-34 ended.
thread Thread-35 is running...
thread Thread-36 is running...
thread Thread-37 is running...
thread Thread-38 is running...
thread Thread-35 ended.
thread Thread-38 ended.
thread Thread-36 ended.
thread Thread-37 ended.
global num: 30668
thread MainThread ended.
```

我们发现 num 的值是 30668，事实上，num 的值是不确定的，你再运行一遍，会发现结果变了。

原因是因为 `num += 1` 不是一个原子操作，也就是说它在执行时被分成若干步：

- 计算 num + 1，存入临时变量 tmp 中；
- 将 tmp 的值赋给 num.

由于线程是交替运行的，线程在执行时可能中断，就会导致其他线程读到一个脏值。

为了保证计算的准确性，我们就需要给 `num += 1` 这个操作加上 `锁` 。当某个线程开始执行这个操作时，由于该线程获得了锁，因此其他线程不能同时执行该操作，只能等待，直到锁被释放，这样就可以避免修改的冲突。创建一个锁可以通过 `threading.Lock()` 来实现，代码如下：

```
from threading import Thread, current_thread, Lock

num = 0
lock = Lock()

def calc():
    global num
    print 'thread %s is running...' % current_thread().name
    for _ in xrange(10000):
        lock.acquire()    # 获取锁
        num += 1
        lock.release()    # 释放锁
    print 'thread %s ended.' % current_thread().name

```

```
15. if __name__ == '__main__':
16.     print 'thread %s is running...' % current_thread().name
17. 
18.     threads = []
19.     for i in range(5):
20.         threads.append(Thread(target=calc))
21.         threads[i].start()
22.     for i in range(5):
23.         threads[i].join()
24. 
25.     print 'global num: %d' % num
26.     print 'thread %s ended.' % current_thread().name
```

让我们看下执行结果：

```
1. thread MainThread is running...
2. thread Thread-44 is running...
3. thread Thread-45 is running...
4. thread Thread-46 is running...
5. thread Thread-47 is running...
6. thread Thread-48 is running...
7. thread Thread-45 ended.
8. thread Thread-47 ended.
9. thread Thread-48 ended.
10. thread Thread-46 ended.
11. thread Thread-44 ended.
12. global num: 50000
13. thread MainThread ended.
```

GIL 锁

讲到 Python 中的多线程，就不得不面对 `GIL` 锁，`GIL` 锁的存在导致 Python 不能有效地使用多线程实现多核任务，因为在同一时间，只能有一个线程在运行。

`GIL` 全称是 Global Interpreter Lock，译为全局解释锁。早期的 Python 为了支持多线程，引入了 GIL 锁，用于解决多线程之间数据共享和同步的问题。但这种实现方式后来被发现是非常低效的，当大家试图去除 GIL 的时候，却发现大量库代码已重度依赖 GIL，由于各种各样的历史原因，GIL 锁就一直保留到现在。

小结

- 一个程序至少有一个进程,一个进程至少有一个线程。
- 进程是操作系统分配资源（比如内存）的最基本单元，线程是操作系统能够进行调度和分派的最基本单元。
- 在 Python 中，进行多线程编程的模块有两个：thread 和 threading。其中，thread 是低级模块，threading 是高级模块，对 thread 进行了封装，一般来说，我们只需使用 threading 这个模块。
- 在执行多线程操作时，注意加锁。

参考资料

- 多线程 - 廖雪峰的官方网站
- Python的GIL是什么鬼，多线程性能究竟如何 • cenalulu’s Tech Blog
- python中多进程以及多线程编程的总结 - Codefly

ThreadLocal

我们知道，同一进程的多个线程之间是内存共享的，这意味着，当一个线程对全局变量做了修改，将会影响到其他所有线程，这是很危险的。为了避免多个线程同时修改全局变量，我们就需要对全局变量的修改加锁。

除了对全局变量的修改进行加锁，你可能也想到了可以使用线程自己的局部变量，因为局部变量只有线程自己能看见，对同一进程的其他线程是不可访问的。确实如此，让我们先看一个例子：

```
1.  from threading import Thread, current_thread
2.
3.  def echo(num):
4.      print current_thread().name, num
5.
6.  def calc():
7.      print 'thread %s is running...' % current_thread().name
8.      local_num = 0
9.      for _ in xrange(10000):
10.         local_num += 1
11.     echo(local_num)
12.     print 'thread %s ended.' % current_thread().name
13.
14. if __name__ == '__main__':
15.     print 'thread %s is running...' % current_thread().name
16.
17.     threads = []
18.     for i in range(5):
19.         threads.append(Thread(target=calc))
20.         threads[i].start()
21.     for i in range(5):
22.         threads[i].join()
23.
24.     print 'thread %s ended.' % current_thread().name
```

在上面的代码中，我们创建了 5 个线程，每个线程都对自己的局部变量 local_num 进行 10000 次的加 1 操作。由于对线程局部变量的修改不会影响到其他线程，因此，我们可以看到，每个线程结束时打印的 local_num 的值都为 10000，执行结果如下：

```
1.  thread MainThread is running...
2.  thread Thread-4 is running...
```

```
Thread-4 10000
thread Thread-4 ended.
thread Thread-5 is running...
Thread-5 10000
thread Thread-5 ended.
thread Thread-6 is running...
Thread-6 10000
thread Thread-6 ended.
thread Thread-7 is running...
Thread-7 10000
thread Thread-7 ended.
thread Thread-8 is running...
Thread-8 10000
thread Thread-8 ended.
thread MainThread ended.
```

上面这种线程使用自己的局部变量的方法虽然可以避免多线程对同一变量的访问冲突，但还是有一些问题。在实际的开发中，我们会调用很多函数，每个函数又有很多个局部变量，这时每个函数都这么传参数显然是不可取的。

为了解决这个问题，一个比较容易想到的做法就是创建一个全局字典，以线程的 ID 作为 key，线程的局部数据作为 value，这样就可以消除函数传参的问题，代码如下：

```
from threading import Thread, current_thread

global_dict = {}

def echo():
    num = global_dict[current_thread()]    # 线程根据自己的 ID 获取数据
    print current_thread().name, num

def calc():
    print 'thread %s is running...' % current_thread().name

    global_dict[current_thread()] = 0
    for _ in xrange(10000):
        global_dict[current_thread()] += 1
    echo()

    print 'thread %s ended.' % current_thread().name

if __name__ == '__main__':
```

```
    print 'thread %s is running...' % current_thread().name

    threads = []
    for i in range(5):
        threads.append(Thread(target=calc))
        threads[i].start()
    for i in range(5):
        threads[i].join()

    print 'thread %s ended.' % current_thread().name
```

看下执行结果：

```
thread MainThread is running...
thread Thread-64 is running...
thread Thread-65 is running...
thread Thread-66 is running...
thread Thread-67 is running...
thread Thread-68 is running...
Thread-67 10000
thread Thread-67 ended.
Thread-65 10000
thread Thread-65 ended.
Thread-68 10000
thread Thread-68 ended.
Thread-66 10000
thread Thread-66 ended.
Thread-64 10000
thread Thread-64 ended.
thread MainThread ended.
```

上面的做法虽然消除了函数传参的问题，但是还是有些不完美，为了获取线程的局部数据，我们需要先获取线程 ID，另外，global_dict 是个全局变量，所有线程都可以对它进行修改，还是有些危险。

那到底如何是好？

事实上，Python 提供了 ThreadLocal 对象，它真正做到了线程之间的数据隔离，而且不用查找 dict，代码如下：

```
from threading import Thread, current_thread, local

global_data = local()
```

```
4.
5.  def echo():
6.      num = global_data.num
7.      print current_thread().name, num
8.
9.  def calc():
10.     print 'thread %s is running...' % current_thread().name
11.
12.     global_data.num = 0
13.     for _ in xrange(10000):
14.         global_data.num += 1
15.     echo()
16.
17.     print 'thread %s ended.' % current_thread().name
18.
19. if __name__ == '__main__':
20.     print 'thread %s is running...' % current_thread().name
21.
22.     threads = []
23.     for i in range(5):
24.         threads.append(Thread(target=calc))
25.         threads[i].start()
26.     for i in range(5):
27.         threads[i].join()
28.
29.     print 'thread %s ended.' % current_thread().name
```

在上面的代码中，global_data 就是 ThreadLocal 对象，你可以把它当作一个全局变量，但它的每个属性，比如 `global_data.num` 都是线程的局部变量，没有访问冲突的问题。

让我们看下执行结果：

```
1.  thread MainThread is running...
2.  thread Thread-94 is running...
3.  thread Thread-95 is running...
4.  thread Thread-96 is running...
5.  thread Thread-97 is running...
6.  thread Thread-98 is running...
7.  Thread-96 10000
8.  thread Thread-96 ended.
9.  Thread-97 10000
10. thread Thread-97 ended.
```

```
Thread-95 10000
thread Thread-95 ended.
Thread-98 10000
thread Thread-98 ended.
Thread-94 10000
thread Thread-94 ended.
thread MainThread ended.
```

小结

- 使用 ThreadLocal 对象来线程绑定自己独有的数据。

参考资料

- ThreadLocal - 廖雪峰的官方网站
- 深入理解Python中的ThreadLocal变量（上） | Just For Fun
- Python线程同步机制 | Python见闻志

协程

与子程序（或者说函数）一样，协程（**coroutine**）也是一种程序组件。Donald Knuth 曾说，子程序是协程的特例。

一个子程序就是一次函数调用，它只有一个入口，一次返回，调用顺序是明确的。但协程的调用和子程序则大不一样，协程允许有多个入口对程序进行中断、继续执行等操作。

Python2 可以通过 yield 来实现基本的协程，但不够强大，第三方库 gevent 对协程提供了强大的支持。另外，Python3.5 提供了 async/await 语法来实现对协程的支持。本文只讨论通过 yield 来实现协程。

对于经典的生产者-消费者模型，如果用多线程来实现，我们就需要一个线程写消息，一个线程读消息，而且需要锁机制来避免对共享资源的访问冲突。

相比多线程，协程的一大特点就是它在一个线程内执行，既避免了多线程之间切换带来的开销，也避免了对共享资源的访问冲突。

下面，让我们看看怎么用 yield 来实现简单的生产者-消费者模型。

```
1.  import time
2.
3.  def consumer():
4.      message = ''
5.      while True:
6.          n = yield message     # yield 使函数中断
7.          if not n:
8.              return
9.          print '[CONSUMER] Consuming %s...' % n
10.         time.sleep(2)
11.         message = '200 OK'
12.
13. def produce(c):
14.     c.next()           # 启动生成器
15.     n = 0
16.     while n < 5:
17.         n = n + 1
18.         print '[PRODUCER] Producing %s...' % n
19.         r = c.send(n)  # 通过 send 切换到 consumer 执行
20.         print '[PRODUCER] Consumer return: %s' % r
21.     c.close()
22.
```

```
23. if __name__ == '__main__':
24.     c = consumer()
25.     produce(c)
```

在上面的代码中，消费者 `consumer` 是一个生成器函数，我们把它作为参数传给 `produce`，其中，next 方法用于启动生成器，send 方法用于发送消息给 `consumer`，并切换到 `consumer` 执行。`consumer` 通过 yield 获取到消息，然后进行处理，又通过 yield 返回消息给 `produce`，并转到 `produce` 执行，如此反复。执行结果如下：

```
1. [PRODUCER] Producing 1...
2. [CONSUMER] Consuming 1...
3. [PRODUCER] Consumer return: 200 OK
4. [PRODUCER] Producing 2...
5. [CONSUMER] Consuming 2...
6. [PRODUCER] Consumer return: 200 OK
7. [PRODUCER] Producing 3...
8. [CONSUMER] Consuming 3...
9. [PRODUCER] Consumer return: 200 OK
10. [PRODUCER] Producing 4...
11. [CONSUMER] Consuming 4...
12. [PRODUCER] Consumer return: 200 OK
13. [PRODUCER] Producing 5...
14. [CONSUMER] Consuming 5...
15. [PRODUCER] Consumer return: 200 OK
```

小结

- 子程序就是协程的一种特例
- 协程的特点在于是一个线程内执行，没有线程之间切换的开销
- 协程只有一个线程，不需多线程的锁机制
- 协程的切换由用户自己管理和调度
- 通过创建协程将异步编程同步化

参考资料

- 协程 - 维基百科，自由的百科全书
- Python 线程与协程 - Yu’s
- 谈谈Python的生成器 – 思诚之道
- 协程 - 廖雪峰的官方网站

异常处理

我们在编写程序的时候，经常需要对异常情况做处理。比如，当一个数试图除以 0 时，我们需要捕获这个异常情况并做处理。你可能会使用类似 if/else 的条件语句来对异常情况做判断，比如，判断除法的分母是否为零，如果为零，则打印错误信息。

这在某些简单的情况下是可以的，但是，在大多数时候，我们应该使用 Python 的异常处理机制。这主要有两方面的好处：

- 一方面，你可以选择忽略某些不重要的异常事件，或在需要的时候自己引发异常；
- 另一方面，异常处理不会搞乱原来的代码逻辑，但如果使用一大堆 if/else 语句，不仅会没效率和不灵活，而且会让代码相当难读；

异常对象

Python 用异常对象（**exception object**）来表示异常情况。当程序在运行过程中遇到错误时，会引发异常。如果异常对象未被处理或捕捉，程序就会用所谓的回溯（Traceback，一种错误信息）终止执行。

比如：

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

上面的 `ZeroDivisionError` 就是一个异常类，相应的异常对象就是该类的实例。Python 中所有的异常类都是从 `BaseException` 类派生的，常见的异常类型可以在这里查看。

使用 try/except 捕捉异常

在编写程序的时候，如果我们知道某段代码可能会导致某种异常，而又不希望程序以堆栈跟踪的形式终止，这时我们可以根据需要添加 `try/except` 或者 `try/finally` 语句（或者它们的组合）进行处理。一般来说，有以下使用形式：

```
try...except...
try...except...else...
try...except...else...finally...
try...except...except...else...finally...
```

```
5. try...finally...
```

基本形式

捕捉异常的基本形式是 `try...except` 。

让我们看看第一个例子：

```
1. try:
2.     x = input('Enter x: ')
3.     y = input('Enter y: ')
4.     print x / y
5. except ZeroDivisionError as e:
6.     print 'Error:',e
7. 
8. print 'hello world'
```

当 y = 0 时，看看执行结果：

```
1. Enter x: 3
2. Enter y: 0
3. Error: integer division or modulo by zero
4. hello world
```

可以看到，我们的程序正确捕获了 `除以零` 的异常，而且程序没有以堆栈跟踪的形式终止，而是继续执行后面的代码，打印出 ‘hello world’。

多个 except 子句

有时，我们的程序可能会出现多个异常，这时可以用多个 except 子句来处理这种情况。

让我们继续看第一个例子，如果 y 输入的是一个非数字的值，就会产生另外一个异常：

```
1. Enter x: 2
2. Enter y: 'a'        # y 的输入是一个字符
3. ---------------------------------------------------------------------------
4. TypeError                                 Traceback (most recent call last)
5. <ipython-input-209-d4666cfaefb4> in <module>()
6.       2     x = input('Enter x: ')
7.       3     y = input('Enter y: ')
8. ----> 4     print x / y
```

```
      5 except ZeroDivisionError as e:
      6     print e

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

可以看到，当 y 输入一个字符 ‘a’ 之后，程序产生了一个 `TypeError` 异常，并且终止，这是因为我们的 except 子句只是捕获了 `ZeroDivisionError` 异常，为了能捕获 `TypeError` 异常，我们可以再加一个 except 子句，完整代码如下：

```
try:
    x = input('Enter x: ')
    y = input('Enter y: ')
    print x / y
except ZeroDivisionError as e:      # 处理 ZeroDivisionError 异常
    print 'ZeroDivisionError:',e
except TypeError as e:              # 处理 TypeError 异常
    print 'TypeError:',e

print 'hello world'
```

当 y 输入 ‘a’ 时，看看执行结果：

```
Enter x: 3
Enter y: 'a'
TypeError: unsupported operand type(s) for /: 'int' and 'str'
hello world
```

捕捉未知异常

事实上，在编写程序的时候，我们很难预料到程序的所有异常情况。比如，对于第一个例子，我们可以预料到一个 `ZeroDivisionError` 异常，如果细心一点，也会预料到一个 `TypeError` 异常，可是，还是有一些其他情况我们没有考虑到，比如在输入 x 的时候，我们直接按回车键，这时又会引发一个异常，程序也会随之挂掉：

```
Enter x:      # 这里输入回车键
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<string>", line 0

    ^
```

```
7.  SyntaxError: unexpected EOF while parsing
```

那么，我们应该怎么在程序捕获某些难以预料的异常呢？我们在上文说过，Python 中所有的异常类都是从 `BaseException` 类派生的，也就是说，`ZeroDivisionError`、`SyntaxError` 等都是它的子类，因此，对于某些难以预料的异常，我们就可以使用 `BaseException` 来捕获，在大部分情况下，我们也可以使用 `Exception` 来捕获，因为 `Exception` 是大部分异常的父类，可以到这里查看所有异常类的继承关系。

因此，对于第一个例子，我们可以把程序做一些修改，使其更加健壮：

```
1.  try:
2.      x = input('Enter x: ')
3.      y = input('Enter y: ')
4.      print x / y
5.  except ZeroDivisionError as e:      # 捕获 ZeroDivisionError 异常
6.      print 'ZeroDivisionError:',e
7.  except TypeError as e:              # 捕获 TypeError 异常
8.      print 'TypeError:',e
9.  except BaseException as e:          # 捕获其他异常
10.     print 'BaseException:',e
11.
12. print 'hello world'
```

注意到，我们把 `BaseException` 写在了最后一个 except 子句。如果你把它写在了第一个 except 子句，由于 `BaseException` 是所有异常的父类，那么程序的所有异常都会被第一个 except 子句捕获。

else 子句

我们可以在 except 子句后面加一个 else 子句。当没有异常发生时，会自动执行 else 子句。

对第一个例子，加入 else 子句：

```
1.  try:
2.      x = input('Enter x: ')
3.      y = input('Enter y: ')
4.      print x / y
5.  except ZeroDivisionError as e:
6.      print 'ZeroDivisionError:',e
7.  except TypeError as e:
8.      print 'TypeError:',e
```

```
except BaseException as e:
    print 'BaseException:',e
else:
    print 'no error!'

print 'hello world'
```

看看执行结果：

```
Enter x: 6
Enter y: 2
3
no error!
hello world
```

finally 子句

finally 子句不管有没有出现异常都会被执行。

看看例子：

```
try:
    x = 1/0
    print x
finally:
    print 'DONE'
```

执行结果：

```
DONE
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
```

再看一个例子：

```
try:
    x = 1/0
    print x
except ZeroDivisionError as e:
    print 'ZeroDivisionError:',e
```

```
finally:
    print 'DONE'
```

执行结果：

```
ZeroDivisionError: integer division or modulo by zero
DONE
```

使用 raise 手动引发异常

有时，我们使用 except 捕获了异常，又想把异常抛出去，这时可以使用 raise 语句。

看看例子：

```
try:
    x = input('Enter x: ')
    y = input('Enter y: ')
    print x / y
except ZeroDivisionError as e:
    print 'ZeroDivisionError:',e
except TypeError as e:
    print 'TypeError:',e
except BaseException as e:
    print 'BaseException:',e
    raise                           # 使用 raise 抛出异常
else:
    print 'no error!'

print 'hello world'
```

运行上面代码，当 x 输入一个回车键时，错误会被打印出来，并被抛出：

```
Enter x:    # 这里输入回车键
BaseException: unexpected EOF while parsing (<string>, line 0)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<string>", line 0

    ^
SyntaxError: unexpected EOF while parsing
```

上面的 raise 语句是不带参数的，它会把当前错误原样抛出。事实上，我们也创建自己的异常类，并抛出自定义的异常。

创建自定义的异常类需要从 **Exception** 类继承，可以间接继承或直接继承，也就是可以继承其他的内建异常类。比如：

```
# 自定义异常类
class SomeError(Exception):
    pass

try:
    x = input('Enter x: ')
    y = input('Enter y: ')
    print x / y
except ZeroDivisionError as e:
    print 'ZeroDivisionError:',e
except TypeError as e:
    print 'TypeError:',e
except BaseException as e:
    print 'BaseException:',e
    raise SomeError('invalid value')    # 抛出自定义的异常
else:
    print 'no error!'

print 'hello world'
```

运行上面代码，当 x 输入一个回车键时，错误被打印出来，并抛出我们自定义的异常：

```
Enter x:
BaseException: unexpected EOF while parsing (<string>, line 0)
---------------------------------------------------------------------------
SomeError                                 Traceback (most recent call last)
<ipython-input-20-66060b472f91> in <module>()
     12 except BaseException as e:
     13     print 'BaseException:',e
---> 14     raise SomeError('invalid value')
     15 else:
     16     print 'no error!'

SomeError: invalid value
```

小结

- Python 中所有的异常类都是从 `BaseException` 类派生的。
- 通过 try/except 来捕捉异常，可以使用多个 except 子句来分别处理不同的异常。
- else 子句在主 try 块没有引发异常的情况下被执行。
- finally 子句不管是否发生异常都会被执行。
- 通过继承 Exception 类可以创建自己的异常类。

参考资料

- 《Python 基础教程》
- Python 异常处理

单元测试

软件系统的开发是一个很复杂的过程，随着系统复杂性的提高，代码中隐藏的 bug 也可能变得越来越多。为了保证软件的质量，测试是一个必不可少的部分，甚至还有测试驱动开发（**Test-driven development, TDD**）的理念，也就是先测试再编码。

在计算机编程中，单元测试（**Unit Testing**）又称为模块测试，是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作，所谓的单元是指一个函数，一个模块，一个类等。

在 Python 中，我们可以使用 `unittest` 模块编写单元测试。

下面以官方文档的例子进行介绍，该例子对字符串的一些方法进行测试：

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

import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')     # 判断两个值是否相等

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())           # 判断值是否为 True
        self.assertFalse('Foo'.isupper())          # 判断值是否为 False

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):          # 检测异常
            s.split(2)
```

在上面，我们定义了一个 TestStringMethods 类，它从 `unittest.TestCase` 继承。注意到，我们的方法名都是以 `test` 开头，表明该方法是测试方法，不以 `test` 开头的方法测试的时候不会被执行。

在方法里面，我们使用了 `断言（assert）` 判断程序运行的结果是否和预期相符。其中：

- `assertEqual` 用于判断两个值是否相等；
- `assertTrue/assertFalse` 用于判断表达式的值是 True 还是 False；

- `assertRaises` 用于检测异常；

断言方法主要有三种类型：

- 检测两个值的大小关系：相等，大于，小于等
- 检查逻辑表达式的值：True/Flase
- 检查异常

下面列举了部分常用的断言方法：

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

现在，让我们来运行上面的单元测试，将上面的代码保存为文件 `mytest.py` ，通过 `-m unittest` 参数运行单元测试：

```
$ python -m unittest mytest
test_isupper (mytest.TestStringMethods) ... ok
test_split (mytest.TestStringMethods) ... ok
test_upper (mytest.TestStringMethods) ... ok
```

执行结果：

```
...
----------------------------------------------------------------------
```

```
Ran 3 tests in 0.000s

OK
```

上面的结果表明测试通过，我们也可以加 `-v` 参数得到更加详细的测试结果：

```
$ python -m unittest -v mytest
test_isupper (mytest.TestStringMethods) ... ok
test_split (mytest.TestStringMethods) ... ok
test_upper (mytest.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

上面这种运行单元测试的方法是我们推荐的做法，当然，你也可以在代码的最后添加两行：

```
if __name__ == '__main__':
    unittest.main()
```

然后再直接运行：

```
$ python mytest.py
```

setUp 和 tearDown

在某些情况下，我们需要在每个测试方法执行前和执行后做一些相同的操作，比如我们想在每个测试方法执行前连接数据库，执行后断开数据库连接，为了避免在每个测试方法中编写同样的代码，我们可以使用 setUp 和 tearDown 方法，比如：

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

import unittest

class TestStringMethods(unittest.TestCase):

    def setUp(self):                                    # 在每个测试方法执行前被调用
        print 'setUp, Hello'

```

```
    def tearDown(self):                             # 在每个测试方法执行后被调用
        print 'tearDown, Bye!'

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')      # 判断两个值是否相等

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())            # 判断值是否为 True
        self.assertFalse('Foo'.isupper())           # 判断值是否为 False

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):          # 检测异常
            s.split(2)
```

看看执行结果：

```
$ python -m unittest -v mytest
test_isupper (mytest.TestStringMethods) ... setUp, Hello
tearDown, Bye!
ok
test_split (mytest.TestStringMethods) ... setUp, Hello
tearDown, Bye!
ok
test_upper (mytest.TestStringMethods) ... setUp, Hello
tearDown, Bye!
ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

小结

- 通过从 `unittest.TestCase` 继承来编写测试类。
- 使用断言方法判断程序运行的结果是否和预期相符。
- `setUp` 在每个测试方法执行前被调用，`tearDown` 在每个测试方法执行后被调用。

参考资料

- unittest — Unit testing framework — Python 2.7.12 documentation
- python单元测试unittest | Lucia Garden
- 单元测试 - 廖雪峰的官方网站
- 单元测试 - 维基百科，自由的百科全书
- “单元测试要做多细？” | 酷 壳 - CoolShell.cn

正则表达式

正则表达式（regular expression）是可以匹配文本片段的模式。最简单的正则表达式就是普通字符串，可以匹配其自身。比如，正则表达式 'hello' 可以匹配字符串 'hello'。

要注意的是，正则表达式并不是一个程序，而是用于处理字符串的一种模式，如果你想用它来处理字符串，就必须使用支持正则表达式的工具，比如 Linux 中的 awk, sed, grep，或者编程语言 Perl, Python, Java 等。

正则表达式有多种不同的风格，下表（改编自 huxi）列出了适用于 Python 或 Perl 等编程语言的部分元字符以及说明：

实例

- 匹配 python.org 的正则表达式：

```
python\.org
```

注：如果使用 `python.org` 来匹配，由于 `.` 可以匹配任意一个字符（换行符除外），因此，它也会匹配到类似 pythonmorg 的字符串，为了匹配点号，我们需要加 `\` 来转义。

- 匹配 010-85692930 的正则表达式：

```
\d{3}\-\d{8}
```

注：`\d` 表示匹配数字，`\d{3}` 表示匹配 3 个数字，`\-` 表示匹配 `-`。

- 匹配由数字、26个英文字母或下划线组成的字符串的正则表达式：

```
^\w+$
```

或

```
^[0-9a-zA-Z_]+$
```

- 匹配 13、15、18 开头的手机号的正则表达式：

```
^(13[0-9]|15[0|1|2|3|5|6|7|8|9]|18[0-9])\d{8}$
```

- 匹配金额，精确到 2 位小数

```
^[0-9]+(.[0-9]{2})?$
```

- 匹配中文的正则表达式：

```
^[\u4e00-\u9fa5]{0,}$
```

注：中文的 unicode 编码范围主要在 `\u4e00-\u9fa5` 。

参考资料

- 正则表达式 - 维基百科，自由的百科全书
- Python正则表达式指南
- 知道这20个正则表达式，能让你少写1,000行代码 - 简书
- regexr

re 模块

在 Python 中，我们可以使用内置的 re 模块来使用正则表达式。

有一点需要特别注意的是，正则表达式使用 `\` 对特殊字符进行转义，比如，为了匹配字符串‘python.org’，我们需要使用正则表达式 `'python\.org'`，而 Python 的字符串本身也用 `\` 转义，所以上面的正则表达式在 Python 中应该写成 `'python\\.org'`，这会很容易陷入 `\` 的困扰中，因此，我们建议使用 Python 的原始字符串，只需加一个 r 前缀，上面的正则表达式可以写成：

```
1.  r'python\.org'
```

re 模块提供了不少有用的函数，用以匹配字符串，比如：

- compile 函数
- match 函数
- search 函数
- findall 函数
- finditer 函数
- split 函数
- sub 函数
- subn 函数

re 模块的一般使用步骤如下：

- 使用 compile 函数将正则表达式的字符串形式编译为一个 Pattern 对象
- 通过 Pattern 对象提供的一系列方法对文本进行匹配查找，获得匹配结果（一个 Match 对象）
- 最后使用 Match 对象提供的属性和方法获得信息，根据需要进行其他的操作

compile 函数

compile 函数用于编译正则表达式，生成一个 **Pattern** 对象，它的一般使用形式如下：

```
1.  re.compile(pattern[, flag])
```

其中，pattern 是一个字符串形式的正则表达式，flag 是一个可选参数，表示匹配模式，比如忽略大小写，多行模式等。

下面，让我们看看例子。

```
import re

# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
```

在上面，我们已将一个正则表达式编译成 Pattern 对象，接下来，我们就可以利用 pattern 的一系列方法对文本进行匹配查找了。Pattern 对象的一些常用方法主要有：

- match 方法
- search 方法
- findall 方法
- finditer 方法
- split 方法
- sub 方法
- subn 方法

match 方法

match 方法用于查找字符串的头部（也可以指定起始位置），它是一次匹配，只要找到了一个匹配的结果就返回，而不是查找所有匹配的结果。它的一般使用形式如下：

```
match(string[, pos[, endpos]])
```

其中，string 是待匹配的字符串，pos 和 endpos 是可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。因此，当你不指定 **pos** 和 **endpos** 时，**match** 方法默认匹配字符串的头部。

当匹配成功时，返回一个 Match 对象，如果没有匹配上，则返回 None。

看看例子。

```
>>> import re
>>> pattern = re.compile(r'\d+')                    # 用于匹配至少一个数字
>>> m = pattern.match('one12twothree34four')        # 查找头部，没有匹配
>>> print m
None
>>> m = pattern.match('one12twothree34four', 2, 10) # 从'e'的位置开始匹配，没有匹配
>>> print m
None
>>> m = pattern.match('one12twothree34four', 3, 10) # 从'1'的位置开始匹配，正好匹配
>>> print m                                         # 返回一个 Match 对象
```

```
<_sre.SRE_Match object at 0x10a42aac0>
>>> m.group(0)   # 可省略 0
'12'
>>> m.start(0)   # 可省略 0
3
>>> m.end(0)     # 可省略 0
5
>>> m.span(0)    # 可省略 0
(3, 5)
```

在上面，当匹配成功时返回一个 Match 对象，其中：

- `group([group1, …])` 方法用于获得一个或多个分组匹配的字符串，当要获得整个匹配的子串时，可直接使用 `group()` 或 `group(0)`；
- `start([group])` 方法用于获取分组匹配的子串在整个字符串中的起始位置（子串第一个字符的索引），参数默认值为 0；
- `end([group])` 方法用于获取分组匹配的子串在整个字符串中的结束位置（子串最后一个字符的索引+1），参数默认值为 0；
- `span([group])` 方法返回 `(start(group), end(group))`。

再看看一个例子：

```
>>> import re
>>> pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I)   # re.I 表示忽略大小写
>>> m = pattern.match('Hello World Wide Web')
>>> print m                               # 匹配成功，返回一个 Match 对象
<_sre.SRE_Match object at 0x10bea83e8>
>>> m.group(0)                            # 返回匹配成功的整个子串
'Hello World'
>>> m.span(0)                             # 返回匹配成功的整个子串的索引
(0, 11)
>>> m.group(1)                            # 返回第一个分组匹配成功的子串
'Hello'
>>> m.span(1)                             # 返回第一个分组匹配成功的子串的索引
(0, 5)
>>> m.group(2)                            # 返回第二个分组匹配成功的子串
'World'
>>> m.span(2)                             # 返回第二个分组匹配成功的子串
(6, 11)
>>> m.groups()                            # 等价于 (m.group(1), m.group(2), ...)
('Hello', 'World')
>>> m.group(3)                            # 不存在第三个分组
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group
```

search 方法

search 方法用于查找字符串的任何位置，它也是一次匹配，只要找到了一个匹配的结果就返回，而不是查找所有匹配的结果，它的一般使用形式如下：

```
search(string[, pos[, endpos]])
```

其中，string 是待匹配的字符串，pos 和 endpos 是可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。

当匹配成功时，返回一个 Match 对象，如果没有匹配上，则返回 None。

让我们看看例子：

```
>>> import re
>>> pattern = re.compile('\d+')
>>> m = pattern.search('one12twothree34four')  # 这里如果使用 match 方法则不匹配
>>> m
<_sre.SRE_Match object at 0x10cc03ac0>
>>> m.group()
'12'
>>> m = pattern.search('one12twothree34four', 10, 30)  # 指定字符串区间
>>> m
<_sre.SRE_Match object at 0x10cc03b28>
>>> m.group()
'34'
>>> m.span()
(13, 15)
```

再来看一个例子：

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

import re

# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
```

```
7.
8.  # 使用 search() 查找匹配的子串，不存在匹配的子串时将返回 None
9.  # 这里使用 match() 无法成功匹配
10. m = pattern.search('hello 123456 789')
11.
12. if m:
13.     # 使用 Match 获得分组信息
14.     print 'matching string:',m.group()
15.     print 'position:',m.span()
```

执行结果：

```
1.  matching string: 123456
2.  position: (6, 12)
```

findall 方法

上面的 match 和 search 方法都是一次匹配，只要找到了一个匹配的结果就返回。然而，在大多数时候，我们需要搜索整个字符串，获得所有匹配的结果。

findall 方法的使用形式如下：

```
1.  findall(string[, pos[, endpos]])
```

其中，string 是待匹配的字符串，pos 和 endpos 是可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。

findall 以列表形式返回全部能匹配的子串，如果没有匹配，则返回一个空列表。

看看例子：

```
1.  import re
2.
3.  pattern = re.compile(r'\d+')   # 查找数字
4.  result1 = pattern.findall('hello 123456 789')
5.  result2 = pattern.findall('one1two2three3four4', 0, 10)
6.
7.  print result1
8.  print result2
```

执行结果：

```
['123456', '789']
['1', '2']
```

finditer 方法

finditer 方法的行为跟 findall 的行为类似，也是搜索整个字符串，获得所有匹配的结果。但它返回一个顺序访问每一个匹配结果（Match 对象）的迭代器。

看看例子：

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

import re

pattern = re.compile(r'\d+')

result_iter1 = pattern.finditer('hello 123456 789')
result_iter2 = pattern.finditer('one1two2three3four4', 0, 10)

print type(result_iter1)
print type(result_iter2)

print 'result1...'
for m1 in result_iter1:   # m1 是 Match 对象
    print 'matching string: {}, position: {}'.format(m1.group(), m1.span())

print 'result2...'
for m2 in result_iter2:
    print 'matching string: {}, position: {}'.format(m2.group(), m2.span())
```

执行结果：

```
<type 'callable-iterator'>
<type 'callable-iterator'>
result1...
matching string: 123456, position: (6, 12)
matching string: 789, position: (13, 16)
result2...
matching string: 1, position: (3, 4)
matching string: 2, position: (7, 8)
```

split 方法

split 方法按照能够匹配的子串将字符串分割后返回列表，它的使用形式如下：

```
split(string[, maxsplit])
```

其中，maxsplit 用于指定最大分割次数，不指定将全部分割。

看看例子：

```
import re

p = re.compile(r'[\s\,\;]+')
print p.split('a,b;; c   d')
```

执行结果：

```
['a', 'b', 'c', 'd']
```

sub 方法

sub 方法用于替换。它的使用形式如下：

```
sub(repl, string[, count])
```

其中，repl 可以是字符串也可以是一个函数：

- 如果 repl 是字符串，则会使用 repl 去替换字符串每一个匹配的子串，并返回替换后的字符串，另外，repl 还可以使用 `\id` 的形式来引用分组，但不能使用编号 0；
- 如果 repl 是函数，这个方法应当只接受一个参数（Match 对象），并返回一个字符串用于替换（返回的字符串中不能再引用分组）。

count 用于指定最多替换次数，不指定时全部替换。

看看例子：

```
import re

p = re.compile(r'(\w+) (\w+)')
s = 'hello 123, hello 456'

```

```
def func(m):
    return 'hi' + ' ' + m.group(2)

print p.sub(r'hello world', s)  # 使用 'hello world' 替换 'hello 123' 和 'hello 456'
print p.sub(r'\2 \1', s)        # 引用分组
print p.sub(func, s)
print p.sub(func, s, 1)         # 最多替换一次
```

执行结果：

```
hello world, hello world
123 hello, 456 hello
hi 123, hi 456
hi 123, hello 456
```

subn 方法

subn 方法跟 sub 方法的行为类似，也用于替换。它的使用形式如下：

```
subn(repl, string[, count])
```

它返回一个元组：

```
(sub(repl, string[, count]), 替换次数)
```

元组有两个元素，第一个元素是使用 sub 方法的结果，第二个元素返回原字符串被替换的次数。

看看例子：

```
import re

p = re.compile(r'(\w+) (\w+)')
s = 'hello 123, hello 456'

def func(m):
    return 'hi' + ' ' + m.group(2)

print p.subn(r'hello world', s)
print p.subn(r'\2 \1', s)
print p.subn(func, s)
```

```
print p.subn(func, s, 1)
```

执行结果：

```
('hello world, hello world', 2)
('123 hello, 456 hello', 2)
('hi 123, hi 456', 2)
('hi 123, hello 456', 1)
```

其他函数

事实上，使用 compile 函数生成的 Pattern 对象的一系列方法跟 re 模块的多数函数是对应的，但在使用上有细微差别。

match 函数

match 函数的使用形式如下：

```
re.match(pattern, string[, flags]):
```

其中，pattern 是正则表达式的字符串形式，比如 `\d+` ， `[a-z]+` 。

而 Pattern 对象的 match 方法使用形式是：

```
match(string[, pos[, endpos]])
```

可以看到，match 函数不能指定字符串的区间，它只能搜索头部，看看例子：

```
import re

m1 = re.match(r'\d+', 'One12twothree34four')
if m1:
    print 'matching string:',m1.group()
else:
    print 'm1 is:',m1

m2 = re.match(r'\d+', '12twothree34four')
if m2:
    print 'matching string:', m2.group()
else:
```

```
13.     print 'm2 is:',m2
```

执行结果：

```
1.  m1 is: None
2.  matching string: 12
```

search 函数

search 函数的使用形式如下：

```
1.  re.search(pattern, string[, flags])
```

search 函数不能指定字符串的搜索区间，用法跟 Pattern 对象的 search 方法类似。

findall 函数

findall 函数的使用形式如下：

```
1.  re.findall(pattern, string[, flags])
```

findall 函数不能指定字符串的搜索区间，用法跟 Pattern 对象的 findall 方法类似。

看看例子：

```
1.  import re
2.
3.  print re.findall(r'\d+', 'hello 12345 789')
4.
5.  # 输出
6.  ['12345', '789']
```

finditer 函数

finditer 函数的使用方法跟 Pattern 的 finditer 方法类似，形式如下：

```
1.  re.finditer(pattern, string[, flags])
```

split 函数

split 函数的使用形式如下：

```
re.split(pattern, string[, maxsplit])
```

sub 函数

sub 函数的使用形式如下：

```
re.sub(pattern, repl, string[, count])
```

subn 函数

subn 函数的使用形式如下：

```
re.subn(pattern, repl, string[, count])
```

到底用哪种方式

从上文可以看到，使用 re 模块有两种方式：

- 使用 re.compile 函数生成一个 Pattern 对象，然后使用 Pattern 对象的一系列方法对文本进行匹配查找；
- 直接使用 re.match, re.search 和 re.findall 等函数直接对文本匹配查找；

下面，我们用一个例子展示这两种方法。

先看第 1 种用法：

```
import re

# 将正则表达式先编译成 Pattern 对象
pattern = re.compile(r'\d+')

print pattern.match('123, 123')
print pattern.search('234, 234')
print pattern.findall('345, 345')
```

再看第 2 种用法：

```
1. import re
2.
3. print re.match(r'\d+', '123, 123')
4. print re.search(r'\d+', '234, 234')
5. print re.findall(r'\d+', '345, 345')
```

如果一个正则表达式需要用到多次（比如上面的 `\d+` ），在多种场合经常需要被用到，出于效率的考虑，我们应该预先编译该正则表达式，生成一个 Pattern 对象，再使用该对象的一系列方法对需要匹配的文件进行匹配；而如果直接使用 re.match, re.search 等函数，每次传入一个正则表达式，它都会被编译一次，效率就会大打折扣。

因此，我们推荐使用第 1 种用法。

匹配中文

在某些情况下，我们想匹配文本中的汉字，有一点需要注意的是，中文的 unicode 编码范围 主要在 `[\u4e00-\u9fa5]` ，这里说主要是因为这个范围并不完整，比如没有包括全角（中文）标点，不过，在大部分情况下，应该是够用的。

假设现在想把字符串 `title = u'你好, hello, 世界'` 中的中文提取出来，可以这么做：

```
1. # -*- coding: utf-8 -*-
2.
3. import re
4.
5. title = u'你好, hello, 世界'
6. pattern = re.compile(ur'[\u4e00-\u9fa5]+')
7. result = pattern.findall(title)
8.
9. print result
```

注意到，我们在正则表达式前面加上了两个前缀 `ur` ，其中 `r` 表示使用原始字符串， `u` 表示是 unicode 字符串。

执行结果：

```
1. [u'\u4f60\u597d', u'\u4e16\u754c']
```

贪婪匹配

在 Python 中，正则匹配默认是贪婪匹配（在少数语言中可能是非贪婪），也就是匹配尽可能多的字符。

比如，我们想找出字符串中的所有 `div` 块：

```
1. import re
2.
3. content = 'aa<div>test1</div>bb<div>test2</div>cc'
4. pattern = re.compile(r'<div>.*</div>')
5. result = pattern.findall(content)
6.
7. print result
```

执行结果：

```
1. ['<div>test1</div>bb<div>test2</div>']
```

由于正则匹配是贪婪匹配，也就是尽可能多的匹配，因此，在成功匹配到第一个 `</div>` 时，它还会向右尝试匹配，查看是否还有更长的可以成功匹配的子串。

如果我们想非贪婪匹配，可以加一个 `?` ，如下：

```
1. import re
2.
3. content = 'aa<div>test1</div>bb<div>test2</div>cc'
4. pattern = re.compile(r'<div>.*?</div>')    # 加上 ?
5. result = pattern.findall(content)
6.
7. print result
```

结果：

```
1. ['<div>test1</div>', '<div>test2</div>']
```

小结

- re 模块的一般使用步骤如下：
    - 使用 compile 函数将正则表达式的字符串形式编译为一个 Pattern 对象；
    - 通过 Pattern 对象提供的一系列方法对文本进行匹配查找，获得匹配结果（一个 Match 对象）；

    - 最后使用 `Match` 对象提供的属性和方法获得信息，根据需要进行其他的操作；
- `Python` 的正则匹配默认是贪婪匹配。

参考资料

- Python正则表达式指南
- 正则表达式 - 廖雪峰的官方网站

HTTP 服务

本章主要介绍：

- HTTP 协议
- Requests 库的使用

HTTP 协议简介

HTTP (HyperText Transfer Protocol, 超文本传输协议)是互联网上应用最为广泛的一种网络协议，它是基于 TCP 的应用层协议，简单地说就是客户端和服务器进行通信的一种规则，它的模式非常简单，就是客户端发起请求，服务器响应请求，如下图所示：

HTTP 最早于 1991 年发布，是 0.9 版，不过目前该版本已不再用。HTTP 目前在使用的版本主要有：

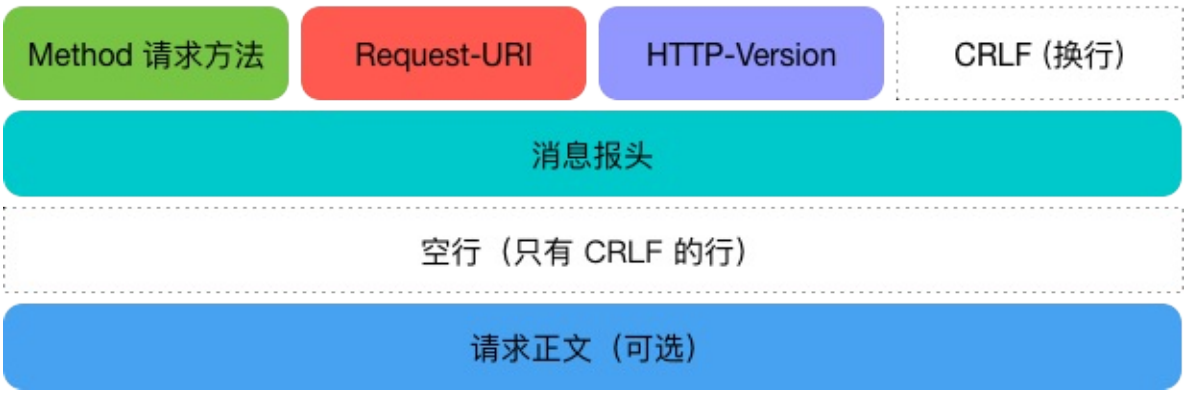
- HTTP/1.0，于 1996 年 5 月发布，引入了多种功能，至今仍在使用当中。
- HTTP/1.1，于 1997 年 1 月发布，持久连接被默认采用，是目前最流行的版本。
- HTTP/2 ，于 2015 年 5 月发布，引入了服务器推送等多种功能，是目前最新的版本。

HTTP 请求

HTTP 请求由三部分组成：

- 请求行：包含请求方法、请求地址和 HTTP 协议版本
- 消息报头：包含一系列的键值对
- 请求正文（可选）：注意和消息报头之间有一个空行

如图所示：



下面是一个 HTTP GET 请求的例子：

```
1. GET / HTTP/1.1
2. Host: httpbin.org
3. Connection: keep-alive
4. Cache-Control: max-age=0
5. Upgrade-Insecure-Requests: 1
```

```
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6,zh-TW;q=0.4
Cookie: _ga=GA1.2.475070272.1480418329; _gat=1
```

上面的第一行就是一个请求行：

```
GET / HTTP/1.1
```

其中，`GET` 是请求方法，表示从服务器获取资源；`/` 是一个请求地址；`HTTP/1.1` 表明 HTTP 的版本是 1.1。

请求行后面的一系列键值对就是消息报头：

```
Host: httpbin.org
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6,zh-TW;q=0.4
Cookie: _ga=GA1.2.475034215.1480418329; _gat=1
```

其中：

- Host 是请求报头域，用于指定被请求资源的 Internet 主机和端口号，它通常从 HTTP URL 中提取出来；
- Connection 表示连接状态，keep-alive 表示该连接是持久连接（persistent connection），即 TCP 连接默认不关闭，可以被多个请求复用，如果客户端和服务器发现对方有一段时间没有活动，就可以主动关闭连接；
- Cache-Control 用于指定缓存指令，它的值有 no-cache, no-store, max-age 等，`max-age=秒` 表示资源在本地缓存多少秒；
- User-Agent 用于标识请求者的一些信息，比如浏览器类型和版本，操作系统等；
- Accept 用于指定客户端希望接受哪些类型的信息，比如 text/html, image/gif 等；
- Accept-Encoding 用于指定可接受的内容编码；
- Accept-Language 用于指定可接受的自然语言；

- Cookie 用于维护状态，可做用户认证，服务器检验等，它是浏览器储存在用户电脑上的文本片段；

HTTP 请求方法

HTTP 通过不同的请求方法以多种方式来操作指定的资源，常用的请求方法如下表：

方法	描述
GET	从服务器获取指定（请求地址）的资源的信息，它通常只用于读取数据，就像数据库查询一样，不会对资源进行修改。
POST	向指定资源提交数据（比如提交表单，上传文件），请求服务器进行处理。数据被包含在请求正文中，这个请求可能会创建新的资源或更新现有的资源。
PUT	通过指定资源的唯一标识（在服务器上的具体存放位置），请求服务器创建或更新资源。
DELETE	请求服务器删除指定资源。
HEAD	与 GET 方法类似，从服务器获取资源信息，和 GET 方法不同的是，HEAD 不含有呈现数据，仅仅是 HTTP 头信息。HEAD 的好处在于，使用这个方法可以在不必传输全部内容的情况下，就可以获得资源的元信息（或元数据）。
OPTIONS	该方法可使服务器传回资源所支持的所有 HTTP 请求方法。

HTTP 响应

HTTP 响应与 HTTP 请求相似，由三部分组成：

- 状态行：包含 HTTP 协议版本、状态码和状态描述，以空格分隔
- 响应头：即消息报头，包含一系列的键值对
- 响应正文：返回内容，注意和响应头之间有一个空行

如图所示：

下面是一个 HTTP GET 请求的响应结果：

```
1. HTTP/1.1 200 OK
2. Server: nginx
```

```
Date: Tue, 29 Nov 2016 13:08:38 GMT
Content-Type: application/json
Content-Length: 203
Connection: close
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

{
  "args": {},
  "headers": {
    "Host": "httpbin.org",
    "User-Agent": "Paw/2.3.1 (Macintosh; OS X/10.11.3) GCDHTTPRequest"
  },
  "origin": "13.75.42.240",
  "url": "https://httpbin.org/get"
}
```

上面的第一行就是一个状态行：

```
HTTP/1.1 200 OK
```

其中，`200` 是状态码，表示客户端请求成功，`OK` 是相应的状态描述。

状态码是一个三位的数字，常见的状态码有以下几类：

- 1XX 消息 – 请求已被服务接收，继续处理
- 2XX 成功 – 请求已成功被服务器接收、理解、并接受
  - 200 OK
  - 201 Created 已创建
  - 202 Accepted 接收
  - 203 Non-Authoritative Information 非认证信息
  - 204 No Content 无内容
- 3XX 重定向 – 需要后续操作才能完成这一请求
  - 301 Moved Permanently 请求永久重定向
  - 302 Moved Temporarily 请求临时重定向
  - 304 Not Modified 文件未修改，可以直接使用缓存的文件
  - 305 Use Proxy 使用代理
- 4XX 请求错误 – 请求含有词法错误或者无法被执行
  - 400 Bad Request 由于客户端请求有语法错误，不能被服务器所理解
  - 401 Unauthorized 请求未经授权。这个状态代码必须和WWW-Authenticate报头域一起使用

    - 403 Forbidden 服务器收到请求，但是拒绝提供服务。服务器通常会在响应正文中给出不提供服务的原因
    - 404 Not Found 请求的资源不存在，例如，输入了错误的URL
- 5XX 服务器错误 — 服务器在处理某个正确请求时发生错误
    - 500 Internal Server Error 服务器发生不可预期的错误，导致无法完成客户端的请求
    - 503 Service Unavailable 服务器当前不能够处理客户端的请求，在一段时间之后，服务器可能会恢复正常
    - 504 Gateway Time-out 网关超时

状态行后面的一系列键值对就是消息报头，即响应头：

```
Server: nginx
Date: Tue, 29 Nov 2016 13:08:38 GMT
Content-Type: application/json
Content-Length: 203
Connection: close
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
```

其中：

- Server 包含了服务器用来处理请求的软件信息，跟请求报头域 User-Agent 相对应；
- Content-Type 用于指定发送给接收者（比如浏览器）的响应正文的媒体类型，比如 text/html, text/css, image/png, image/jpeg, video/mp4, application/pdf, application/json 等；
- Content-Length 指明本次回应的数据长度；

再议 POST 和 PUT

注意到，POST 和 PUT 都可用于创建或更新资源，然而，它们之间还是有比较大的区别：

- POST 所对应的 URI 并非创建的资源本身，而是资源的接收者，资源本身的存放位置由服务器决定；而 PUT 所对应的 URI 是要创建或更新的资源本身，它指明了具体的存放位置

比如，往某个站点添加一篇文章，如果使用 **POST** 来创建资源，可类似这样：

```
POST /articles HTTP/1.1

{
    "author": "ethan",
```

```
    "title": "hello world",
    "content": "hello world"
}
```

在上面，POST 对应的 URI 是 `/articles`，它是资源的接收者，而非资源的标识，如果资源被成功创建，服务器可以返回 `201 Created` 状态以及新建资源的位置，比如：

```
HTTP/1.1 201 Created
Location: /articles/abcdef123
```

我们如果知道新建资源的标识符，可以使用 **PUT** 来创建资源，比如：

```
PUT /articles/abcdef234 HTTP/1.1

{
    "author": "peter",
    "title": "hello world",
    "content": "hello world"
}
```

在上面，PUT 对应的 URI 是 `/articles/abcdef234`，它指明了资源的存放位置，如果资源被成功创建，服务器可以返回 `201 Created` 状态以及新建资源的位置，比如：

```
HTTP/1.1 201 Created
Location: /articles/abcdef234
```

- 使用 PUT 更新某一资源，需要更新资源的全部属性；而使用 POST，可以更新全部或一部分值

比如使用 PUT 更新地址为 `/articles/abcdef234` 的文章的标题，我们需要发送所有值：

```
PUT /articles/abcdef234 HTTP/1.1

{
    "author": "peter",
    "title": "hello python",
    "content": "hello world"
}
```

而使用 POST，可以更新某个域的值：

```
POST /articles/abcdef234 HTTP/1.1
```

```
2.
3.  {
4.      "title": "hello python"
5.  }
```

- POST 是不幂等的，PUT 是幂等的，这是一个很重要的区别

> HTTP 方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用，注意这里是副作用，而不是返回结果。

GET 方法用于获取资源，不会改变资源的状态，不论调用一次还是多次都没有副作用，因此它是幂等的；**DELETE** 方法用于删除资源，有副作用，但调用一次或多次都是删除同个资源，产生的副作用是相同的，因此也是幂等的；**POST** 是不幂等的，因为两次相同的 POST 请求会在服务器创建两份资源，它们具有不同的 URI；**PUT** 是幂等的，对同一 URI 进行多次 PUT 的副作用和一次 PUT 是相同的。

HTTP 特点

- 客户端/服务器模式
- 简单快速：客户端向服务器请求服务时，通过传送请求方法、请求地址和数据体（可选）即可
- 灵活：允许传输任意类型的数据对象，通过 Content-Type 标识
- 无状态：对事物处理没记忆能力

小结

- HTTP 是在网络上传输 HTML 的协议，用于浏览器和服务器的通信，默认使用 80 端口。
- URL 地址用于定位资源，HTTP 中的 GET, POST, PUT, DELETE 用于操作资源，比如查询，增加，更新等。
- GET, PUT, DELETE 是幂等的，POST 是不幂等的。
- POST VS PUT
    - 使用 PUT 创建资源需要提供资源的唯一标识（具体存放位置），POST 不需要，POST 的数据存放位置由服务器自己决定
    - 使用 PUT 更新某一资源，需要更新资源的全部属性；而使用 POST，可以更新全部或一部分值
    - POST 是不幂等的，PUT 是幂等的，这是一个很重要的区别
- GET 可提交的数据量受到 URL 长度的限制，HTTP 协议规范没有对 URL 长度进行限制，这个限制是特定的浏览器及服务器对它的限制。
- 理论上讲，POST 是没有大小限制的，HTTP 协议规范也没有进行大小限制，出于安全考虑，服务器软件在实现时会做一定限制。

参考资料

- 超文本传输协议 - 维基百科，自由的百科全书
- HTTP 协议入门 - 阮一峰的网络日志
- HTTP幂等性概念和应用 | 酷 壳 - CoolShell.cn
- Http协议详解 - 简书
- When to use PUT or POST - The RESTful cookbook
- To PUT or POST?
- 全面解读HTTP Cookie
- HTTP cookies 详解 | bubkoo
- HTTP 接口设计指北

Requests 库的使用

Python 的标准库 urllib 提供了大部分 HTTP 功能，但使用起来较繁琐。通常，我们会使用另外一个优秀的第三方库：Requests，它的标语是：**Requests: HTTP for Humans**。

Requests 提供了很多功能特性，几乎涵盖了当今 Web 服务的需求，比如：

- 浏览器式的 SSL 验证
- 身份认证
- Keep-Alive & 连接池
- 带持久 Cookie 的会话
- 流下载
- 文件分块上传

下面，我们将从以下几个方面介绍 Requests 库：

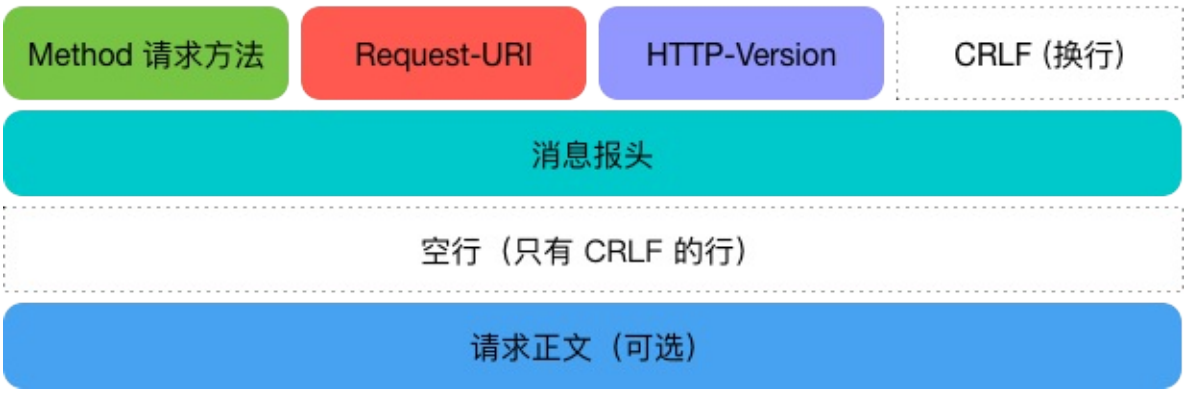
- HTTP 请求
- HTTP 响应
- cookie
- 会话对象
- 代理
- 身份认证

HTTP 请求

我们知道，一个 HTTP 请求由三部分构成：

- 请求行：包含请求方法（比如 GET, POST）、请求地址和 HTTP 协议版本
- 请求头：包含一系列的键值对
- 请求正文（可选）

如图所示：



Requests 提供了几乎所有 HTTP 动词的功能：GET、OPTIONS、HEAD、POST、PUT、PATCH、DELETE，另外，它提供了 `headers` 参数让我们根据需求定制请求头。

使用 Requests 发送一个请求很方便，比如：

```
import requests

r = requests.get("http://httpbin.org/get")
r = requests.post("http://httpbin.org/post")
r = requests.put("http://httpbin.org/put")
r = requests.delete("http://httpbin.org/delete")
r = requests.head("http://httpbin.org/get")
r = requests.options("http://httpbin.org/get")
```

下面，我们重点讲一下 GET 请求，POST 请求和定制请求头。

GET 请求

使用 Requests 发送 GET 请求非常简单，如下：

```
import requests

r = requests.get("http://httpbin.org/get")
```

在有些情况下，URL 会带参数，比如 `https://segmentfault.com/blogs?page=2`，这个 URL 有一个参数 page，值为 2。Requests 提供了 `params` 关键字参数，允许我们以一个字典来提供这些参数，比如：

```
import requests

payload = {'page': '1', 'per_page': '10'}
r = requests.get("http://httpbin.org/get", params=payload)
```

通过打印该 URL，我们可以看到 URL 已被正确编码：

```
>>> print r.url
http://httpbin.org/get?per_page=10&page=1
```

需要注意的是字典里值为 None 的键不会被添加到 URL 的查询字符串中。

POST 请求

使用 Requests 发送 POST 请求也很简单，如下：

```
1. import requests
2.
3. r = requests.post("http://httpbin.org/post")
```

通常，我们在发送 POST 请求时还会附上数据，比如发送编码为表单形式的数据或编码为 JSON 形式的数据，这时，我们可以使用 Requests 提供的 `data` 参数。

- 发送编码为表单形式的数据

通过给 `data` 参数传递一个 `dict` ，我们的数据字典在发出请求时会被自动编码为表单形式，比如：

```
1. import requests
2.
3. payload = {'page': 1, 'per_page': 10}
4. r = requests.post("http://httpbin.org/post", data=payload)
```

看看返回的内容（省略了部分数据）：

```
1. >>> print r.text
2. {
3.   ...
4.   "form": {
5.     "page": "1",
6.     "per_page": "10"
7.   },
8.   ...
9. }
```

- 发送编码为 JSON 形式的数据

如果给 `data` 参数传递一个 `string` ，我们的数据会被直接发布出去，比如：

```
1. import json
2. import requests
3.
4. payload = {'page': 1, 'per_page': 10}
5. r = requests.post("http://httpbin.org/post", data=json.dumps(payload))
```

看看返回：

```
>>> print r.text
{
  "args": {},
  "data": "{\"per_page\": 10, \"page\": 1}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "27",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.9.1"
  },
  "json": {
    "page": 1,
    "per_page": 10
  },
  "origin": "13.75.42.240",
  "url": "http://httpbin.org/post"
}
```

在上面，我们自行对 `dict` 进行了编码，这种方式等价于使用 `json` 参数，而给它传递 `dict` ，如下：

```
import requests

payload = {'page': 1, 'per_page': 10}
r = requests.post("http://httpbin.org/post", json=payload)
```

这种做法跟上面的做法是等价的，数据在发出时会被自动编码。

请求头

有时，我们需要为请求添加 HTTP 头部，我们可以通过传递一个 `dict` 给 `headers` 参数来实现。比如：

```
import requests

```

```
3. url = 'http://httpbin.org/post'
4. payload = {'page': 1, 'per_page': 10}
5. headers = {'User-Agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'}
6.
7. r = requests.post("http://httpbin.org/post", json=payload, headers=headers)
```

发送到服务器的请求的头部可以通过 `r.request.headers` 访问：

```
1. >>> print r.request.headers
   {'Content-Length': '27', 'Accept-Encoding': 'gzip, deflate', 'Accept': '*/*',
   'User-Agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)', 'Connection':
2. 'keep-alive', 'Content-Type': 'application/json'}
```

服务器返回给我们的响应头部信息可以通过 `r.headers` 访问：

```
1. >>> print r.headers
   {'Content-Length': '462', 'Server': 'nginx', 'Connection': 'close', 'Access-
   Control-Allow-Credentials': 'true', 'Date': 'Mon, 05 Dec 2016 15:41:05 GMT',
2. 'Access-Control-Allow-Origin': '*', 'Content-Type': 'application/json'}
```

HTTP 响应

HTTP 响应与 HTTP 请求相似，由三部分组成：

- 状态行：包含 HTTP 协议版本、状态码和状态描述，以空格分隔
- 响应头：包含一系列的键值对
- 响应正文

如图所示：

当我们使用 `requests.*` 发送请求时，Requests 做了两件事：

- 构建一个 Request 对象，该对象会根据请求方法或相关参数发起 HTTP 请求

- 一旦服务器返回响应，就会产生一个 Response 对象，该响应对象包含服务器返回的所有信息，也包含你原来创建的 Request 对象

对于响应状态码，我们可以访问响应对象的 `status_code` 属性：

```
1. import requests
2.
3. r = requests.get("http://httpbin.org/get")
4. print r.status_code
5.
6. # 输出
7. 200
```

对于响应正文，我们可以通过多种方式读取，比如：

- 普通响应，使用 `r.text` 获取
- JSON 响应，使用 `r.json()` 获取
- 二进制响应，使用 `r.content` 获取
- 原始响应，使用 `r.raw` 获取

普通响应

我们可以使用 `r.text` 来读取 unicode 形式的响应，看看例子：

```
1. import requests
2.
3. r = requests.get("https://github.com/timeline.json")
4. print r.text
5. print r.encoding
6.
7. # 输出
   {"message":"Hello there, wayfaring stranger. If you’re reading this then you prob
   didn’t see our blog post a couple of years back announcing that this API would go
   away: http://git.io/17AROg Fear not, you should be able to get what you need from
   shiny new Events API
   instead.","documentation_url":"https://developer.github.com/v3/activity/events/#l
8. public-events"}
9. utf-8
```

Requests 会自动解码来自服务器的内容，大多数 unicode 字符集都能被正确解码。

JSON 响应

对于 JSON 响应的内容，我们可以使用 `json()` 方法把返回的数据解析成 Python 对象。

看看例子：

```
import requests

r = requests.get("https://github.com/timeline.json")

if r.status_code == 200:
    print r.headers.get('content-type')
    print r.json()

# 输出
application/json; charset=utf-8
{u'documentation_url': u'https://developer.github.com/v3/activity/events/#list-public-events', u'message': u'Hello there, wayfaring stranger. If you\u2019re reading this then you probably didn\u2019t see our blog post a couple of years back announcing that this API would go away: http://git.io/17AROg Fear not, you should be able to get what you need from the shiny new Events API instead.'}
```

如果 JSON 解码失败，`r.json()` 就会抛出异常，比如：

```
import requests

r = requests.get("https://www.baidu.com")

if r.status_code == 200:
    print r.headers.get('content-type')
    print r.json()

# 输出
text/html
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-9216431f0e2d> in <module>()
      1 if r.status_code == 200:
      2     print r.headers.get('content-type')
----> 3     print r.json()
      4
....
```

```
....
ValueError: No JSON object could be decoded
```

二进制响应

我们也可以以字节的方式访问响应正文，访问 `content` 属性可以获取二进制数据，比如用返回的二进制数据创建一张图片：

```
import requests

url = 'https://github.com/reactjs/redux/blob/master/logo/logo.png?raw=true'
r = requests.get(url)
image_data = r.content   # 获取二进制数据

with open('/Users/Ethan/Downloads/redux.png', 'wb') as fout:
    fout.write(image_data)
```

原始响应

在少数情况下，我们可能想获取来自服务器的原始套接字响应，这可以通过访问响应对象的 `raw` 属性来实现，但要确保在初始请求中设置了 `stream=True` ，比如：

```
import requests

url = 'https://github.com/reactjs/redux/blob/master/logo/logo.png?raw=true'
r = requests.get(url, stream=True)
print r.raw
r.raw.read(10)

# 输出
<requests.packages.urllib3.response.HTTPResponse object at 0x1113b0a90>
'\x89PNG\r\n\x1a\n\x00\x00'
```

重定向

默认情况下，除了 HEAD，Requests 会自动处理所有重定向。我们可以使用响应对象的 `history` 属性来追踪重定向，**Response.history** 是一个 Response 对象的列表，这个对象列表按照从最老到最近的请求进行排序。

比如，点击某些网站的链接，它会将页面重定向到其他网站：

```
>>> import requests

>>> headers = {'User-Agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'}
>>> r = requests.get('https://toutiao.io/k/c32y51', headers=headers)

>>> r.status_code
200

>>> r.url   # 发生了重定向，响应对象的 url，跟请求对象不一样
u'http://www.jianshu.com/p/490441391db6?
hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io'

>>> r.history
[<Response [302]>]

>>> r.history[0].text
u'<html><body>You are being <a href="http://www.jianshu.com/p/490441391db6?
hmsr=toutiao.io&amp;utm_medium=toutiao.io&amp;utm_source=toutiao.io">redirected</a>
</body></html>'
```

可以看到，我们访问网址 `https://toutiao.io/k/c32y51` 被重定向到了下面的链接：

```
http://www.jianshu.com/p/490441391db6?hmsr=toutiao.io&utm_medium=toutiao.io&utm_s
```

我们还看到 `r.history` 包含了一个 Response 对象列表，我们可以用它来追踪重定向。

如果请求方法是 GET、POST、PUT、OPTIONS、PATCH 或 DELETE，我们可以通过 `all_redirects` 参数禁止重定向：

```
>>> import requests

>>> headers = {'User-Agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'}
>>> r = requests.get('https://toutiao.io/k/c32y51', headers=headers,
allow_redirects=False)
>>> r.url    # 禁止重定向，响应对象的 url 跟请求对象一致
u'https://toutiao.io/k/c32y51'
>>> r.history
[]
>>> r.text
```

```
    u'<html><body>You are being <a href="http://www.jianshu.com/p/490441391db6?
    hmsr=toutiao.io&amp;utm_medium=toutiao.io&amp;utm_source=toutiao.io">redirected</
10. </body></html>'
```

Cookie

- 如果某个响应包含一些 cookie，我们可以直接访问它们，比如：

```
1. >>> import requests
2.
3. >>> url = 'http://exmaple.com/some/cookie/setting/url'
4. >>> r = requests.get(url)
5.
6. >>> r.cookies['some_key']
7. 'some_value'
```

- 发送 cookies 到服务器，可以使用 `cookies` 参数：

```
1. >>> import requests
2.
3. >>> url = 'http://httpbin.org/cookies'
4. >>> cookies = dict(key1='value1')
5.
6. >>> r = requests.get(url, cookies=cookies)
7. >>> r.text
8. u'{\n  "cookies": {\n    "key1": "value1"\n  }\n}\n'
9. >>> print r.text
10. {
11.   "cookies": {
12.     "key1": "value1"
13.   }
14. }
```

会话对象

我们知道，HTTP 协议是无状态的，这意味着每个请求都是独立的，如果后续的处理需要用到前面的信息，则数据需要重传，为了解决这个问题，我们可以使用 Cookie 或 Session 来存储某些特定的信息，比如用户名、密码等，这样，当用户在不同 Web 页面跳转或再次登录网站时，就不用重新输入用户名和密码（当然，如果 Cookie 被删除和 Session 过期则需要重新输入）。

Requests 提供了会话对象让我们能够跨请求保持某些参数，也可以在同一个 Session 实例发出的所有请求之间保持 Cookie。

下面，我们看看会话对象的使用。

下面是一个跨请求保持 Cookie 的例子：

```
>>> import requests
>>> s = requests.Session()
>>> s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
<Response [200]>
>>> r = s.get("http://httpbin.org/cookies")
>>> print r.text
{
  "cookies": {
    "sessioncookie": "123456789"
  }
}
```

会话还可用来为请求方法提供缺省数据，通过设置会话对象的属性来实现：

```
import requests

s = requests.Session()
s.auth = ('user', 'pass')
s.headers.update({'x-test': 'true'})

# x-test 和 x-test2 都会被发送
s.get('http://httpbin.org/headers', headers={'x-test2': 'true'})
```

代理

Requests 支持基本的 HTTP 代理 和 SOCKS 代理（2.10.0 新增功能）。

HTTP 代理

如果需要使用 HTTP 代理，我们可以为任意请求方法提供 `proxies` 参数，如下：

```
import requests

proxies = {
```

```
  "http": "http://10.10.1.10:3128",
  "https": "http://10.10.1.10:1080",
}

requests.get("http://example.org", proxies=proxies)
```

我们也可以通过设置环境变量 `HTTP_PROXY` 和 `HTTPS_PROXY` 来配置代理：

```
$ export HTTP_PROXY="http://10.10.1.10:3128"
$ export HTTPS_PROXY="http://10.10.1.10:1080"

$ python
>>> import requests
>>> requests.get("http://example.org")
```

SOCKS 代理

Requests 自 2.10.0 版起，开始支持 SOCKS 协议的代理，如果要使用，我们还需安装一个第三方库：

```
$ pip install requests[socks]
```

SOCKS 代理的使用和 HTTP 代理类似：

```
import requests

proxies = {
  "http": "socks5://user:pass@host:port",
  "https": "socks5://user:pass@host:port",
}

requests.get("http://example.org", proxies=proxies)
```

身份认证

大部分 Web 服务都需要身份认证，并且有多种不同的认证类型，比如：

- 基本身份认证
- 摘要式身份认证
- OAuth 认证

下面介绍一下基本身份认证和 OAuth 认证。

基本身份认证

基本身份认证（HTTP Basic Auth）是最简单的一种身份认证，一般需要身份认证的 Web 服务也都接受 HTTP Basic Auth，Requests 提供了非常简单的形式让我们使用 HTTP Basic Auth:

```
>>> from requests.auth import HTTPBasicAuth
>>> requests.get('https://api.github.com/user', auth=HTTPBasicAuth('user',
'pass'))
```

由于 HTTP Basic Auth 非常常见，Requests 提供了一种简写的形式:

```
requests.get('https://api.github.com/user', auth=('user', 'pass'))
```

OAuth 2 认证

OAuth 是一种常见的 Web API 认证方式，目前的版本是 2.0。Requests 并不直接支持 OAuth 认证，而是要配合另外一个库一起使用，该库是 requests-oauthlib。

下面以 GitHub 为例，介绍一下 OAuth 2 认证。

```
>>> # Credentials you get from registering a new application
>>> client_id = '<the id you get from github>'
>>> client_secret = '<the secret you get from github>'

>>> # OAuth endpoints given in the GitHub API documentation
>>> authorization_base_url = 'https://github.com/login/oauth/authorize'
>>> token_url = 'https://github.com/login/oauth/access_token'

>>> from requests_oauthlib import OAuth2Session
>>> github = OAuth2Session(client_id)

>>> # Redirect user to GitHub for authorization
>>> authorization_url, state = github.authorization_url(authorization_base_url)
>>> print 'Please go here and authorize,', authorization_url

>>> # Get the authorization verifier code from the callback url
>>> redirect_response = raw_input('Paste the full redirect URL here:')

>>> # Fetch the access token
```

```
>>> github.fetch_token(token_url, client_secret=client_secret,
>>>         authorization_response=redirect_response)

>>> # Fetch a protected resource, i.e. user profile
>>> r = github.get('https://api.github.com/user')
>>> print r.content
```

更多关于 OAuth 工作流程的信息，可以参考 OAuth 官方网站，关于 `requests-oauthlib` 库的使用，可以参考官方文档。

小结

- 任何时候调用 requests.*() 你都在做两件主要的事情。其一，你在构建一个 Request 对象， 该对象将被发送到某个服务器请求或查询一些资源。其二，一旦 requests 得到一个从服务器返回的响应就会产生一个 Response 对象。该响应对象包含服务器返回的所有信息，也包含你原来创建的 Request 对象。

参考资料

- Requests: Requests 2.10.0 文档
- 如何理解HTTP协议的“无连接，无状态”特点？
- 理解OAuth 2.0 - 阮一峰的网络日志
- 作为客户端与HTTP服务交互 — python3-cookbook 2.0.0 文档

标准模块

前面我们介绍了 os 模块和 re 模块，本章再介绍 Python 常用的一些标准模块：

- argparse
- base64
- collections
- datetime
- hashlib
- hmac

其中：

- argparse 是用于创建命令行的库；
- base64 是用于 base64 编码和解码的库；
- collections 模块提供了 5 个高性能的数据类型，如 `Counter`，`OrderedDict` 等；
- datetime 是用于处理日期时间的模块；
- hashlib 模块提供了常见的摘要算法，比如 MD5，SHA1 等；
- hmac 模块提供了 HMAC 哈希算法；

argparse

argparse 是 Python 内置的一个用于命令项选项与参数解析的模块，通过在程序中定义好我们需要的参数，argparse 将会从 sys.argv 中解析出这些参数，并自动生成帮助和使用信息。当然，Python 也有第三方的库可用于命令行解析，而且功能也更加强大，比如 docopt，Click。

argparse 使用

简单示例

我们先来看一个简单示例。主要有三个步骤：

- 创建 ArgumentParser() 对象
- 调用 add_argument() 方法添加参数
- 使用 parse_args() 解析添加的参数

```
1. # -*- coding: utf-8 -*-
2.
3. import argparse
4.
5. parser = argparse.ArgumentParser()
6. parser.add_argument('integer', type=int, help='display an integer')
7. args = parser.parse_args()
8.
9. print args.integer
```

将上面的代码保存为文件 `argparse_usage.py` ，在终端运行，结果如下：

```
1. $ python argparse_usage.py
2. usage: argparse_usage.py [-h] integer
3. argparse_usage.py: error: too few arguments
4.
5. $ python argparse_usage.py abcd
6. usage: argparse_usage.py [-h] integer
7. argparse_usage.py: error: argument integer: invalid int value: 'abcd'
8.
9. $ python argparse_usage.py -h
10. usage: argparse_usage.py [-h] integer
11.
```

```
positional arguments:
  integer     display an integer

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

$ python argparse_usage.py 10
10
```

定位参数

上面的示例，其实就展示了定位参数的使用，我们再来看一个例子 - 计算一个数的平方：

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

import argparse

parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
type=int)
args = parser.parse_args()
print args.square**2
```

将上面的代码保存为文件 `argparse_usage.py` ，在终端运行，结果如下：

```
$ python argparse_usage.py 9
81
```

可选参数

现在看下可选参数的用法，所谓可选参数，也就是命令行参数是可选的，废话少说，看下面例子：

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

import argparse

parser = argparse.ArgumentParser()

parser.add_argument("--square", help="display a square of a given number",
type=int)
```

```
    parser.add_argument("--cubic", help="display a cubic of a given number",
8.  type=int)
9.
10. args = parser.parse_args()
11.
12. if args.square:
13.     print args.square**2
14.
15. if args.cubic:
16.     print args.cubic**3
```

将上面的代码保存为文件 `argparse_usage.py` ，在终端运行，结果如下：

```
1.  $ python argparse_usage.py --h
2.  usage: argparse_usage.py [-h] [--square SQUARE] [--cubic CUBIC]
3.
4.  optional arguments:
5.    -h, --help       show this help message and exit
6.    --square SQUARE  display a square of a given number
7.    --cubic CUBIC    display a cubic of a given number
8.
9.  $ python argparse_usage.py --square 8
10. 64
11.
12. $ python argparse_usage.py --cubic 8
13. 512
14.
15. $ python argparse_usage.py 8
16. usage: argparse_usage.py [-h] [--square SQUARE] [--cubic CUBIC]
17. argparse_usage.py: error: unrecognized arguments: 8
18.
19. $ python argparse_usage.py  # 没有输出
```

混合使用

定位参数和选项参数可以混合使用，看下面一个例子，给一个整数序列，输出它们的和或最大值（默认）：

```
1.  import argparse
2.
3.  parser = argparse.ArgumentParser(description='Process some integers.')
```

```
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print args.accumulate(args.integers)
```

结果：

```
$ python argparse_usage.py
usage: argparse_usage.py [-h] [--sum] N [N ...]
argparse_usage.py: error: too few arguments
$ python argparse_usage.py 1 2 3 4
4
$ python argparse_usage.py 1 2 3 4 --sum
10
```

add_argument() 方法

add_argument() 方法定义如何解析命令行参数：

```
ArgumentParser.add_argument(name or flags...[, action][, nargs][, const][,
default][, type][, choices][, required][, help][, metavar][, dest])
```

每个参数解释如下：

- name or flags - 选项字符串的名字或者列表，例如 foo 或者 -f, –foo。
- action - 命令行遇到参数时的动作，默认值是 store。
    - store_const，表示赋值为const；
    - append，将遇到的值存储成列表，也就是如果参数重复则会保存多个值；
    - append_const，将参数规范中定义的一个值保存到一个列表；
    - count，存储遇到的次数；此外，也可以继承 argparse.Action 自定义参数解析；
- nargs - 应该读取的命令行参数个数，可以是具体的数字，或者是?号，当不指定值时对于 Positional argument 使用 default，对于 Optional argument 使用 const；或者是 * 号，表示 0 或多个参数；或者是 + 号表示 1 或多个参数。
- const - action 和 nargs 所需要的常量值。
- default - 不指定参数时的默认值。
- type - 命令行参数应该被转换成的类型。

- choices - 参数可允许的值的一个容器。
- required - 可选参数是否可以省略 (仅针对可选参数)。
- help - 参数的帮助信息，当指定为 `argparse.SUPPRESS` 时表示不显示该参数的帮助信息.
- metavar - 在 usage 说明中的参数名称，对于必选参数默认就是参数名称，对于可选参数默认是全大写的参数名称.
- dest - 解析后的参数名称，默认情况下，对于可选参数选取最长的名称，中划线转换为下划线.

参考资料

- Argparse Tutorial — Python 2.7.12 documentation
- Argparse – Command line option and argument parsing
- Argparse — Parser for command-line options, arguments and sub-commands
- Python 中的命令行解析工具介绍

Base64

Base64，简单地讲，就是用 **64** 个字符来表示二进制数据的方法。这 64 个字符包含小写字母 a-z、大写字母 A-Z、数字 0-9 以及符号”+”、”/“，其实还有一个 “=” 作为后缀用途，所以实际上有 65 个字符。

本文主要介绍如何使用 Python 进行 Base64 编码和解码，关于 Base64 编码转换的规则可以参考 Base64 笔记。

Python 内置了一个用于 Base64 编解码的库： `base64` ：

- 编码使用 `base64.b64encode()`
- 解码使用 `base64.b64decode()`

下面，我们介绍文本和图片的 Base64 编解码。

对文本进行 Base64 编码和解码

```
>>> import base64
>>> str = 'hello world'
>>>
>>> base64_str = base64.b64encode(str)       # 编码
>>> print base64_str
aGVsbG8gd29ybGQ=
>>>
>>> ori_str = base64.b64decode(base64_str)   # 解码
>>> print ori_str
hello world
```

对图片进行 Base64 编码和解码

```
def convert_image():
    # 原始图片 ==> base64 编码
    with open('/path/to/alpha.png', 'r') as fin:
        image_data = fin.read()
        base64_data = base64.b64encode(image_data)

        fout = open('/path/to/base64_content.txt', 'w')
        fout.write(base64_data)
```

```
        fout.close()

    # base64 编码 ==> 原始图片
    with open('/path/to/base64_content.txt', 'r') as fin:
        base64_data = fin.read()
        ori_image_data = base64.b64decode(base64_data)

        fout = open('/path/to/beta.png', 'wb'):
        fout.write(ori_image_data)
        fout.close()
```

小结

- Base64 可以将任意二进制数据编码到文本字符串，常用于在 URL、Cookie 和网页中传输少量二进制数据。

参考资料

- Base64 - 维基百科，自由的百科全书
- Base64笔记 - 阮一峰的网络日志

collections

我们知道，Python 的数据类型有 list, tuple, dict, str 等，**collections** 模块提供了额外 5 个高性能的数据类型：

- `Counter`：计数器
- `OrderedDict`：有序字典
- `defaultdict`：带有默认值的字典
- `namedtuple`：生成可以通过属性访问元素内容的 tuple 子类
- `deque`：双端队列，能够在队列两端添加或删除元素

Counter

`Counter` 是一个简单的计数器，可用于统计字符串、列表等的元素个数。

看看例子：

```
>>> from collections import Counter
>>>
>>> s = 'aaaabbbccd'
>>> c = Counter(s)          # 创建了一个 Counter 对象
>>> c
Counter({'a': 4, 'b': 3, 'c': 2, 'd': 1})
>>> isinstance(c, dict)    # c 其实也是一个字典对象
True
>>> c.get('a')
4
>>> c.most_common(2)       # 获取出现次数最多的前两个元素
[('a', 4), ('b', 3)]
```

在上面，我们使用 `Counter()` 创建了一个 `Counter` 对象 `c`，`Counter` 其实是 dict 的一个子类，我们可以使用 `get` 方法来获取某个元素的个数。`Counter` 对象有一个 `most_common` 方法，允许我们获取出现次数最多的前几个元素。

另外，两个 `Counter` 对象还可以做运算：

```
>>> from collections import Counter
>>>
>>> s1 = 'aaaabbbccd'
>>> c1 = Counter(s1)
```

```
>>> c1
Counter({'a': 4, 'b': 3, 'c': 2, 'd': 1})
>>>
>>> s2 = 'aaabbef'
>>> c2 = Counter(s2)
>>> c2
Counter({'a': 3, 'b': 2, 'e': 1, 'f': 1})
>>>
>>> c1 + c2       # 两个计数结果相加
Counter({'a': 7, 'b': 5, 'c': 2, 'e': 1, 'd': 1, 'f': 1})
>>> c1 - c2       # c2 相对于 c1 的差集
Counter({'c': 2, 'a': 1, 'b': 1, 'd': 1})
>>> c1 & c2       # c1 和 c2 的交集
Counter({'a': 3, 'b': 2})
>>> c1 | c2       # c1 和 c2 的并集
Counter({'a': 4, 'b': 3, 'c': 2, 'e': 1, 'd': 1, 'f': 1})
```

OrderedDict

Python 中的 dict 是无序的:

```
>>> dict([('a', 10), ('b', 20), ('c', 15)])
{'a': 10, 'c': 15, 'b': 20}
```

有时，我们希望保持 key 的顺序，这时可以用 OrderedDict:

```
>>> from collections import OrderedDict
>>> OrderedDict([('a', 10), ('b', 20), ('c', 15)])
OrderedDict([('a', 10), ('b', 20), ('c', 15)])
```

defaultdict

在 Python 中使用 dict 时，如果访问了不存在的 key，会抛出 KeyError 异常，因此，在访问之前，我们经常需要对 key 作判断，比如:

```
>>> d = dict()
>>> s = 'aaabbc'
>>> for char in s:
...     if char in d:
...         d[char] += 1
```

```
...     else:
...         d[char] = 1
...
>>> d
{'a': 3, 'c': 1, 'b': 2}
```

使用 defaultdict，我们可以给字典中的 key 提供一个默认值。访问 defaultdict 中的 key，如果 key 存在，就返回 key 对应的 value，如果 key 不存在，就返回默认值。

```
>>> from collections import defaultdict
>>> d = defaultdict(int)   # 默认的 value 值是 0
>>> s = 'aaabbc'
>>> for char in s:
...     d[char] += 1
...
>>> d
defaultdict(<type 'int'>, {'a': 3, 'c': 1, 'b': 2})
>>> d.get('a')
3
>>> d['z']
0
```

使用 defaultdict 时，我们可以传入一个工厂方法来指定默认值，比如传入 int，表示默认值是 0，传入 list，表示默认是 `[]`：

```
>>> from collections import defaultdict
>>>
>>> d1 = defaultdict(int)
>>> d1['a']
0
>>> d2 = defaultdict(list)
>>> d2['b']
[]
>>> d3 = defaultdict(str)
>>> d3['a']
''
```

我们还可以自定义默认值，通过 `lambda` 函数来实现：

```
>>> from collections import defaultdict
>>>
```

```
>>> d = defaultdict(lambda: 10)
>>> d['a']
10
```

namedtuple

我们经常用 tuple（元组）来表示一个不可变对象，比如用一个 `(姓名, 学号, 年龄)` 的元组来表示一个学生：

```
>>> stu = ('ethan', '001', 20)
>>> stu[0]
'ethan'
```

这里使用 tuple 没什么问题，但可读性比较差，我们必须清楚索引代表的含义，比如索引 0 表示姓名，索引 1 表示学号。如果用类来定义，就可以通过设置属性 name, id, age 来表示，但就有些小题大作了。

我们可以通过 namedtuple 为元组的每个索引设置名称，然后通过「属性名」来访问：

```
>>> from collections import namedtuple
>>> Student = namedtuple('Student', ['name', 'id', 'age'])  # 定义了一个 Student
元组
>>>
>>> stu = Student('ethan', '001', 20)
>>> stu.name
'ethan'
>>> stu.id
'001'
```

deque

deque 是双端队列，允许我们在队列两端添加或删除元素。

```
>>> from collections import deque

>>> q = deque(['a', 'b', 'c', 'd'])
>>> q.append('e')          # 添加到尾部
>>> q
deque(['a', 'b', 'c', 'd', 'e'])
>>> q.appendleft('o')      # 添加到头部
```

```
>>> q
deque(['o', 'a', 'b', 'c', 'd', 'e'])
>>> q.pop()                 # 从尾部弹出元素
'e'
>>> q
deque(['o', 'a', 'b', 'c', 'd'])
>>> q.popleft()             # 从头部弹出元素
'o'
>>> q
deque(['a', 'b', 'c', 'd'])
>>> q.extend('ef')          # 在尾部 extend 元素
>>> q
deque(['a', 'b', 'c', 'd', 'e', 'f'])
>>> q.extendleft('uv')      # 在头部 extend 元素，注意顺序
>>> q
deque(['v', 'u', 'a', 'b', 'c', 'd', 'e', 'f'])
>>>
>>> q.rotate(2)             # 将尾部的两个元素移动到头部
>>> q
deque(['e', 'f', 'v', 'u', 'a', 'b', 'c', 'd'])
>>> q.rotate(-2)            # 将头部的两个元素移动到尾部
>>> q
deque(['v', 'u', 'a', 'b', 'c', 'd', 'e', 'f'])
```

其中，`rotate` 方法用于旋转，如果旋转参数 n 大于 0，表示将队列右端的 n 个元素移动到左端，否则相反。

参考资料

- collections — High-performance container datatypes — Python 2.7.13 documentation

itertools

我们知道，迭代器的特点是：惰性求值（Lazy evaluation），即只有当迭代至某个值时，它才会被计算，这个特点使得迭代器特别适合于遍历大文件或无限集合等，因为我们不用一次性将它们存储在内存中。

Python 内置的 itertools 模块包含了一系列用来产生不同类型迭代器的函数或类，这些函数的返回都是一个迭代器，我们可以通过 for 循环来遍历取值，也可以使用 `next()` 来取值。

itertools 模块提供的迭代器函数有以下几种类型：

- 无限迭代器：生成一个无限序列，比如自然数序列 `1, 2, 3, 4, ...`；
- 有限迭代器：接收一个或多个序列（sequence）作为参数，进行组合、分组和过滤等；
- 组合生成器：序列的排列、组合，求序列的笛卡儿积等；

无限迭代器

itertools 模块提供了三个函数（事实上，它们是类）用于生成一个无限序列迭代器：

- count(firstval=0, step=1)

  创建一个从 firstval（默认值为 0）开始，以 step（默认值为 1）为步长的的无限整数迭代器

- cycle(iterable)

  对 iterable 中的元素反复执行循环，返回迭代器

- repeat(object [,times]

  反复生成 object，如果给定 times，则重复次数为 times，否则为无限

下面，让我们看看一些例子。

count

`count()` 接收两个参数，第一个参数指定开始值，默认为 0，第二个参数指定步长，默认为 1：

```
1.  >>> import itertools
2.  >>>
3.  >>> nums = itertools.count()
4.  >>> for i in nums:
```

```
...     if i > 6:
...         break
...     print i
...
0
1
2
3
4
5
6
>>> nums = itertools.count(10, 2)    # 指定开始值和步长
>>> for i in nums:
...     if i > 20:
...         break
...     print i
...
10
12
14
16
18
20
```

cycle

`cycle()` 用于对 iterable 中的元素反复执行循环:

```
>>> import itertools
>>>
>>> cycle_strings = itertools.cycle('ABC')
>>> i = 1
>>> for string in cycle_strings:
...     if i == 10:
...         break
...     print i, string
...     i += 1
...
1 A
2 B
3 C
```

```
4 A
5 B
6 C
7 A
8 B
9 C
```

repeat

`repeat()` 用于反复生成一个 object:

```
>>> import itertools
>>>
>>> for item in itertools.repeat('hello world', 3):
...     print item
...
hello world
hello world
hello world
>>>
>>> for item in itertools.repeat([1, 2, 3, 4], 3):
...     print item
...
[1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3, 4]
```

有限迭代器

itertools 模块提供了多个函数（类），接收一个或多个迭代对象作为参数，对它们进行组合、分组和过滤等:

- chain()
- compress()
- dropwhile()
- groupby()
- ifilter()
- ifilterfalse()
- islice()
- imap()

- starmap()
- tee()
- takewhile()
- izip()
- izip_longest()

chain

`chain` 的使用形式如下：

```
chain(iterable1, iterable2, iterable3, ...)
```

`chain` 接收多个可迭代对象作为参数，将它们『连接』起来，作为一个新的迭代器返回。

```
>>> from itertools import chain
>>>
>>> for item in chain([1, 2, 3], ['a', 'b', 'c']):
...     print item
...
1
2
3
a
b
c
```

`chain` 还有一个常见的用法：

```
chain.from_iterable(iterable)
```

接收一个可迭代对象作为参数，返回一个迭代器：

```
>>> from itertools import chain
>>>
>>> string = chain.from_iterable('ABCD')
>>> string.next()
'A'
```

compress

`compress` 的使用形式如下：

```
compress(data, selectors)
```

`compress` 可用于对数据进行筛选，当 selectors 的某个元素为 true 时，则保留 data 对应位置的元素，否则去除：

```
>>> from itertools import compress
>>>
>>> list(compress('ABCDEF', [1, 1, 0, 1, 0, 1]))
['A', 'B', 'D', 'F']
>>> list(compress('ABCDEF', [1, 1, 0, 1]))
['A', 'B', 'D']
>>> list(compress('ABCDEF', [True, False, True]))
['A', 'C']
```

dropwhile

`dropwhile` 的使用形式如下：

```
dropwhile(predicate, iterable)
```

其中，predicate 是函数，iterable 是可迭代对象。对于 iterable 中的元素，如果 predicate(item) 为 true，则丢弃该元素，否则返回该项及所有后续项。

```
>>> from itertools import dropwhile
>>>
>>> list(dropwhile(lambda x: x < 5, [1, 3, 6, 2, 1]))
[6, 2, 1]
>>>
>>> list(dropwhile(lambda x: x > 3, [2, 1, 6, 5, 4]))
[2, 1, 6, 5, 4]
```

groupby

`groupby` 用于对序列进行分组，它的使用形式如下：

```
groupby(iterable[, keyfunc])
```

其中，iterable 是一个可迭代对象，keyfunc 是分组函数，用于对 iterable 的连续项进行分组，如果不指定，则默认对 iterable 中的连续相同项进行分组，返回一个 `(key, sub-iterator)` 的迭代器。

```
>>> from itertools import groupby
>>>
>>> for key, value_iter in groupby('aaabbbaaccd'):
...     print key, ':', list(value_iter)
...
a : ['a', 'a', 'a']
b : ['b', 'b', 'b']
a : ['a', 'a']
c : ['c', 'c']
d : ['d']
>>>
>>> data = ['a', 'bb', 'ccc', 'dd', 'eee', 'f']
>>> for key, value_iter in groupby(data, len):    # 使用 len 函数作为分组函数
...     print key, ':', list(value_iter)
...
1 : ['a']
2 : ['bb']
3 : ['ccc']
2 : ['dd']
3 : ['eee']
1 : ['f']
>>>
>>> data = ['a', 'bb', 'cc', 'ddd', 'eee', 'f']
>>> for key, value_iter in groupby(data, len):
...     print key, ':', list(value_iter)
...
1 : ['a']
2 : ['bb', 'cc']
3 : ['ddd', 'eee']
1 : ['f']
```

ifilter

`ifilter` 的使用形式如下：

```
ifilter(function or None, sequence)
```

将 iterable 中 function(item) 为 True 的元素组成一个迭代器返回，如果 function 是 None，则返回 iterable 中所有计算为 True 的项。

```
>>> from itertools import ifilter
>>>
>>> list(ifilter(lambda x: x < 6, range(10)))
[0, 1, 2, 3, 4, 5]
>>>
>>> list(ifilter(None, [0, 1, 2, 0, 3, 4]))
[1, 2, 3, 4]
```

ifilterfalse

`ifilterfalse` 的使用形式和 `ifilter` 类似，它将 iterable 中 function(item) 为 False 的元素组成一个迭代器返回，如果 function 是 None，则返回 iterable 中所有计算为 False 的项。

```
>>> from itertools import ifilterfalse
>>>
>>> list(ifilterfalse(lambda x: x < 6, range(10)))
[6, 7, 8, 9]
>>>
>>> list(ifilter(None, [0, 1, 2, 0, 3, 4]))
[0, 0]
```

islice

`islice` 是切片选择，它的使用形式如下：

```
islice(iterable, [start,] stop [, step])
```

其中，iterable 是可迭代对象，start 是开始索引，stop 是结束索引，step 是步长，start 和 step 可选。

```
>>> from itertools import count, islice
>>>
>>> list(islice([10, 6, 2, 8, 1, 3, 9], 5))
[10, 6, 2, 8, 1]
>>>
>>> list(islice(count(), 6))
```

```
[0, 1, 2, 3, 4, 5]
>>>
>>> list(islice(count(), 3, 10))
[3, 4, 5, 6, 7, 8, 9]
>>> list(islice(count(), 3, 10 ,2))
[3, 5, 7, 9]
```

imap

`imap` 类似 `map` 操作，它的使用形式如下：

```
imap(func, iter1, iter2, iter3, ...)
```

`imap` 返回一个迭代器，元素为 `func(i1, i2, i3, ...)` ， `i1` ， `i2` 等分别来源于 `iter` ， `iter2` 。

```
>>> from itertools import imap
>>>
>>> imap(str, [1, 2, 3, 4])
<itertools.imap object at 0x10556d050>
>>>
>>> list(imap(str, [1, 2, 3, 4]))
['1', '2', '3', '4']
>>>
>>> list(imap(pow, [2, 3, 10], [4, 2, 3]))
[16, 9, 1000]
```

tee

`tee` 的使用形式如下：

```
tee(iterable [,n])
```

`tee` 用于从 iterable 创建 n 个独立的迭代器，以元组的形式返回，n 的默认值是 2。

```
>>> from itertools import tee
>>>
>>> tee('abcd')   # n 默认为 2，创建两个独立的迭代器
(<itertools.tee object at 0x1049957e8>, <itertools.tee object at 0x104995878>)
>>>
```

```
>>> iter1, iter2 = tee('abcde')
>>> list(iter1)
['a', 'b', 'c', 'd', 'e']
>>> list(iter2)
['a', 'b', 'c', 'd', 'e']
>>>
>>> tee('abc', 3)  # 创建三个独立的迭代器
(<itertools.tee object at 0x104995998>, <itertools.tee object at 0x1049959e0>,
<itertools.tee object at 0x104995a28>)
```

takewhile

`takewhile` 的使用形式如下：

```
takewhile(predicate, iterable)
```

其中，predicate 是函数，iterable 是可迭代对象。对于 iterable 中的元素，如果 predicate(item) 为 true，则保留该元素，只要 predicate(item) 为 false，则立即停止迭代。

```
>>> from itertools import takewhile
>>>
>>> list(takewhile(lambda x: x < 5, [1, 3, 6, 2, 1]))
[1, 3]
>>> list(takewhile(lambda x: x > 3, [2, 1, 6, 5, 4]))
[]
```

izip

`izip` 用于将多个可迭代对象对应位置的元素作为一个元组，将所有元组『组成』一个迭代器，并返回。它的使用形式如下：

```
izip(iter1, iter2, ..., iterN)
```

如果某个可迭代对象不再生成值，则迭代停止。

```
>>> from itertools import izip
>>>
>>> for item in izip('ABCD', 'xy'):
...     print item
```

```
...
('A', 'x')
('B', 'y')
>>> for item in izip([1, 2, 3], ['a', 'b', 'c', 'd', 'e']):
...     print item
...
(1, 'a')
(2, 'b')
(3, 'c')
```

izip_longest

`izip_longest` 跟 `izip` 类似，但迭代过程会持续到所有可迭代对象的元素都被迭代完。它的形式如下：

```
izip_longest(iter1, iter2, ..., iterN, [fillvalue=None])
```

如果有指定 fillvalue，则会用其填充缺失的值，否则为 None。

```
>>> from itertools import izip_longest
>>>
>>> for item in izip_longest('ABCD', 'xy'):
...     print item
...
('A', 'x')
('B', 'y')
('C', None)
('D', None)
>>>
>>> for item in izip_longest('ABCD', 'xy', fillvalue='-'):
...     print item
...
('A', 'x')
('B', 'y')
('C', '-')
('D', '-')
```

组合生成器

itertools 模块还提供了多个组合生成器函数，用于求序列的排列、组合等：

- product
- permutations
- combinations
- combinations_with_replacement

product

`product` 用于求多个可迭代对象的笛卡尔积，它跟嵌套的 for 循环等价。它的一般使用形式如下：

```
product(iter1, iter2, ... iterN, [repeat=1])
```

其中，repeat 是一个关键字参数，用于指定重复生成序列的次数，

```
>>> from itertools import product
>>>
>>> for item in product('ABCD', 'xy'):
...     print item
...
('A', 'x')
('A', 'y')
('B', 'x')
('B', 'y')
('C', 'x')
('C', 'y')
('D', 'x')
('D', 'y')
>>>
>>> list(product('ab', range(3)))
[('a', 0), ('a', 1), ('a', 2), ('b', 0), ('b', 1), ('b', 2)]
>>>
>>> list(product((0,1), (0,1), (0,1)))
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0),
(1, 1, 1)]
>>>
>>> list(product('ABC', repeat=2))
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'), ('C',
'A'), ('C', 'B'), ('C', 'C')]
>>>
```

permutations

`permutations` 用于生成一个排列，它的一般使用形式如下：

```
permutations(iterable[, r])
```

其中，r 指定生成排列的元素的长度，如果不指定，则默认为可迭代对象的元素长度。

```
>>> from itertools import permutations
>>>
>>> permutations('ABC', 2)
<itertools.permutations object at 0x1074d9c50>
>>>
>>> list(permutations('ABC', 2))
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
>>>
>>> list(permutations('ABC'))
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'), ('C', 'A',
'B'), ('C', 'B', 'A')]
>>>
```

combinations

`combinations` 用于求序列的组合，它的使用形式如下：

```
combinations(iterable, r)
```

其中，r 指定生成组合的元素的长度。

```
>>> from itertools import combinations
>>>
>>> list(combinations('ABC', 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

combinations_with_replacement

`combinations_with_replacement` 和 `combinations` 类似，但它生成的组合包含自身元素。

```
>>> from itertools import combinations_with_replacement
```

```
>>>
>>> list(combinations_with_replacement('ABC', 2))
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
```

小结

- itertools 模块提供了很多用于产生多种类型迭代器的函数，它们的返回值不是 list，而是迭代器。

参考链接

- itertools — Functions creating iterators for efficient looping
- itertools – Iterator functions for efficient looping - Python Module of the Week

datetime

Python 提供了两个标准库用于处理跟时间相关的问题，一个是 `time`，另一个是 `datetime`，`datetime` 对 `time` 进行了封装，提供了更多实用的函数。本文介绍 `datetime` 库的简单使用。

当前时间

获取当前时间可以使用 `now()` 或 `utcnow()` 方法，其中，`now()` 用于获取当地时间，而 `utcnow()` 用于获取 UTC 时间。

```
>>> from datetime import datetime

>>> datetime.now()     # 返回一个 datetime 对象，这里是当地时间
datetime.datetime(2016, 12, 10, 11, 32, 43, 806970)

>>> datetime.utcnow()  # 返回一个 datetime 对象，这里是 UTC 时间
datetime.datetime(2016, 12, 10, 3, 32, 49, 999423)

>>> datetime.now().year, datetime.now().month, datetime.now().day     # 年月日
(2016, 12, 10)

>>> datetime.now().hour, datetime.now().minute, datetime.now().second  # 时分秒
(11, 35, 37)
```

时间格式化

有时，我们需要对时间做格式化处理，可以使用 `strftime()` 或 `strptime()` 方法，其中，`strftime` 用于对 `datetime` 对象进行格式化，`strptime` 用于对字符串对象进行格式化。

```
>>> from datetime import datetime

# 获取当前当地时间
>>> now = datetime.now()
>>> now
datetime.datetime(2016, 12, 10, 11, 46, 24, 432168)

# 对 datetime 对象进行格式化，转为字符串格式
```

```
>>> now_str = now.strftime('%Y-%m-%d %H:%M:%S.%f')
>>> now_str
'2016-12-10 11:46:24.432168'

# 对字符串对象进行格式化，转为 datetime 对象
>>> datetime.strptime(now_str, '%Y-%m-%d %H:%M:%S.%f')
datetime.datetime(2016, 12, 10, 11, 46, 24, 432168)
```

时间戳

Unix 时间戳根据精度的不同，有 10 位（秒级），13 位（毫秒级），16 位（微妙级）和 19 位（纳秒级）。

要注意的是，由于每个时区都有自己的本地时间（北京在东八区），因此也产生了世界标准时间（UTC，Universal Time Coordinated）。所以，在将一个时间戳转换为普通时间（比如 2016-01-01 12:00:00）时，要注意是要本地时区的时间还是世界时间等。

- 获取当前当地时间戳

```
>>> import time
>>> from datetime import datetime

# 获取当前当地时间，返回一个 datetime 对象
>>> now = datetime.now()
>>> now
datetime.datetime(2016, 12, 9, 11, 56, 47, 632778)

# 13 位的毫秒时间戳
>>> long(time.mktime(now.timetuple()) * 1000.0 + now.microsecond / 1000.0)
1481255807632L

# 10 位的时间戳
>>> int(time.mktime(now.timetuple()))
1481255807
```

- 获取当前 UTC 时间戳

```
>>> import calendar
>>> from datetime import datetime

# 获取当前的 UTC 时间，返回 datetime 对象
```

```
>>> utc_now = datetime.utcnow()
>>> utc_now
datetime.datetime(2016, 12, 9, 4, 0, 53, 356641)

# 13 位的时间戳
>>> long(calendar.timegm(utc_now.timetuple()) * 1000.0 + utc_now.microsecond / 1000.0)
1481256053356L

# 10 位的时间戳
>>> calendar.timegm(utc_now.timetuple())
1481256053
```

- 将时间戳转为字符串形式

```
>>> from datetime import datetime

# 13 位的毫秒时间戳
>>> timestamp = 1456402864242

# 根据时间戳构建当地时间
>>> datetime.fromtimestamp(timestamp / 1000.0).strftime('%Y-%m-%d %H:%M:%S.%f')
'2016-02-25 20:21:04.242000'

# 根据时间戳构建 UTC 时间
>>> datetime.utcfromtimestamp(timestamp / 1000.0).strftime('%Y-%m-%d %H:%M:%S.%f')
'2016-02-25 12:21:04.242000'

# 10 位的时间戳
>>> timestamp = 1456402864

# 根据时间戳构建当地时间
>>> datetime.fromtimestamp(timestamp).strftime('%Y-%m-%d %H:%M:%S')
'2016-02-25 20:21:04'

# 根据时间戳构建 UTC 时间
>>> datetime.utcfromtimestamp(timestamp).strftime('%Y-%m-%d %H:%M:%S')
'2016-02-25 12:21:04'
```

- 将时间戳转为 datetime 形式

```
>>> from datetime import datetime

# 13 位的毫秒时间戳
>>> timestamp = 1456402864242

# 根据时间戳构建当地时间
>>> datetime.fromtimestamp(timestamp / 1000.0)
datetime.datetime(2016, 2, 25, 20, 21, 4, 242000)

# 根据时间戳构建 UTC 时间
>>> datetime.utcfromtimestamp(timestamp / 1000.0)
datetime.datetime(2016, 2, 25, 12, 21, 4, 242000)

# 10 位的时间戳
>>> timestamp = 1456402864

# 根据时间戳构建当地时间
>>> datetime.fromtimestamp(timestamp)
datetime.datetime(2016, 2, 25, 20, 21, 4)

# 根据时间戳构建 UTC 时间
>>> datetime.utcfromtimestamp(timestamp)
datetime.datetime(2016, 2, 25, 12, 21, 4)
```

参考资料

- python-datetime-time-conversions

hashlib

Python 内置的 hashlib 模块提供了常见的摘要算法（或称哈希算法，散列算法），如 MD5，SHA1，SHA256 等。摘要算法的基本原理是：将数据（如一段文字）运算变为另一固定长度值。

MD5（Message-Digest Algorithm 5，消息摘要算法），是一种被广泛使用的密码散列函数，可以产生出一个 128 位（16 字节）的散列值（hash value），用于确保信息传输完整一致。

SHA1（Secure Hash Algorithm，安全哈希算法）是 SHA 家族的其中一个算法，它经常被用作数字签名。

MD5

hashlib 模块提供了 `md5` 函数，我们可以很方便地使用它：

```
>>> import hashlib
>>>
>>> m = hashlib.md5('md5 test in Python!')
>>> m.digest()
'\xad\xc0\x99\x01\x12\xc7&\xb5~\xb0\xaf \x974\x11\xab'
>>> m.hexdigest()   # 使用一个 32 位的 16 进制字符串表示
'adc0990112c726b57eb0af20973411ab'
```

上面，我们是直接把数据传入 `md5()` 函数，我们也可以通过一次或多次使用 `update` 来实现：

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update('md5 test ')
>>> m.update('in Python!')
>>> m.hexdigest()
'adc0990112c726b57eb0af20973411ab'
```

SHA1

SHA1 的使用和 MD5 的使用类似：

```
>>> import hashlib
>>>
```

```
>>> sha1 = hashlib.sha1()
>>> sha1.update('md5 test ')
>>> sha1.update('in Python!')
>>> sha1.hexdigest()
'698a8b18d5f99a140520475c342af455183c58a3'
```

小结

- MD5 以前经常用来做用户密码的存储。2004年，它被证明无法防止碰撞，因此无法适用于安全性认证，但仍广泛应用于普通数据的错误检查领域。如果你需要储存用户密码，你应该去了解一下诸如 PBKDF2 或 bcrypt 之类的算法。而 SHA1 则经常用作数字签名。

参考资料

- MD5 - 维基百科，自由的百科全书
- SHA家族 - 维基百科，自由的百科全书

hmac

HMAC 是用于消息认证的加密哈希算法，全称是 keyed-Hash Message Authentication Code。**HMAC** 利用哈希算法，以一个密钥和一个消息作为输入，生成一个加密串作为输出。HMAC 可以有效防止类似 MD5 的彩虹表等攻击，比如将常见密码的 MD5 值存入数据库，可能被反向破解。

Python 的 `hmac` 模块提供了 HMAC 算法，它的使用形式是：

```
1. hmac.new(key[, msg[, digestmod]])
```

其中，key 是一个密钥；msg 是消息，可选，如果给出 msg，则调用方法 `update(msg)`；digestmod 是 HMAC 对象使用的摘要构造函数或模块，默认为 `hashlib.md5` 构造函数。

HMAC 对象常用的方法有：

- HMAC.update(msg)

用字符串 msg 更新 HMAC 对象，重复的调用等同于一次调用所有参数的组合，即：

```
1. m.update(a);
2. m.update(b);
```

相当于

```
1. m.update(a+b)
```

- HMAC.digest()

返回目前传递给 `update()` 方法的字符串的摘要。此字符串长度将与给构造函数的摘要的 digest_size 相同。它可能包含非 ASCII 字符，包括 NULL 字节。

- HMAC.hexdigest()

类似 digest()，但是返回的摘要的字符串的长度翻倍，且只包含十六进制数字。

现在，让我们看一个简单的例子：

```
1. >>> from datetime import datetime
2. >>> import hashlib
3. >>> import hmac
4.
5. >>> key = 'you-never-know'
```

```
>>> msg = datetime.utcnow().strftime('%Y-%m-%d')

>>> m = hmac.new(key, msg, hashlib.sha1)
>>> signature = m.hexdigest()
>>> signature
'fdb2087a66a2f00afbc1884738467ba089782779'
```

参考资料

- hmac — 用于消息认证的加密哈希算法

第三方模块

在前面，我们介绍了一个优秀的第三方库 — requests，本章再介绍两个第三方库：

- celery
- click

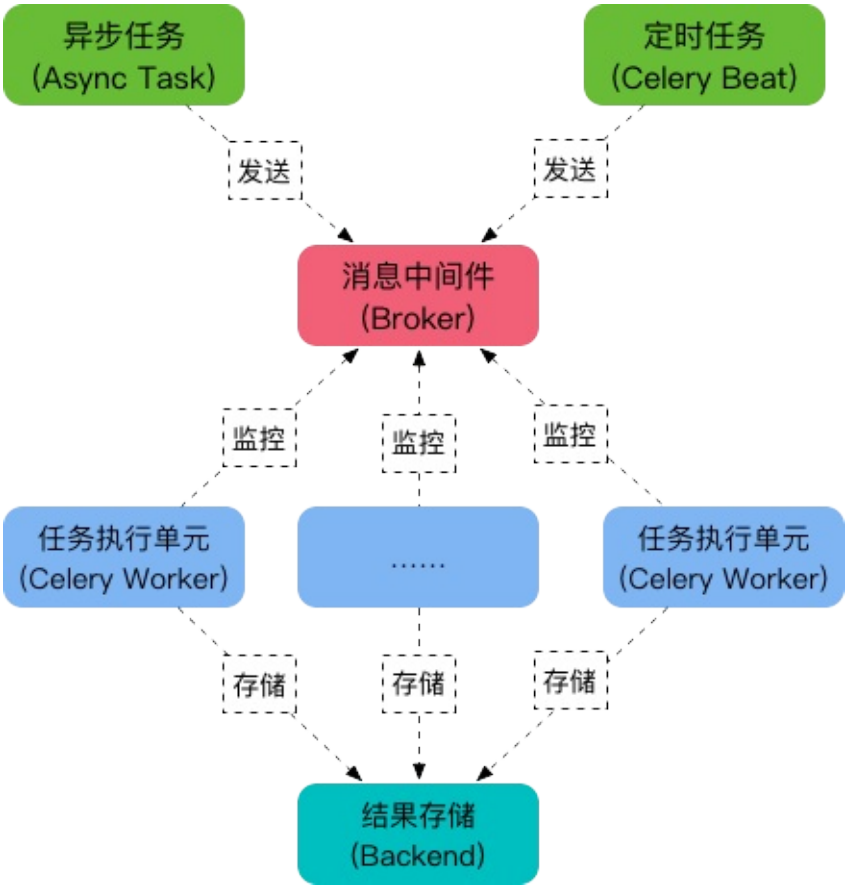
其中：

- celery 是一个强大的分布式任务队列，通常用于实现异步任务；
- click 是快速创建命令行的神器；

Celery

在程序的运行过程中，我们经常会碰到一些耗时耗资源的操作，为了避免它们阻塞主程序的运行，我们经常会采用多线程或异步任务。比如，在 Web 开发中，对新用户的注册，我们通常会给他发一封激活邮件，而发邮件是个 IO 阻塞式任务，如果直接把它放到应用当中，就需要等邮件发出去之后才能进行下一步操作，此时用户只能等待再等待。更好的方式是在业务逻辑中触发一个发邮件的异步任务，而主程序可以继续往下运行。

Celery 是一个强大的分布式任务队列，它可以让任务的执行完全脱离主程序，甚至可以被分配到其他主机上运行。我们通常使用它来实现异步任务（async task）和定时任务（crontab）。它的架构组成如下图：



可以看到，Celery 主要包含以下几个模块：

- 任务模块

  包含异步任务和定时任务。其中，异步任务通常在业务逻辑中被触发并发往任务队列，而定时任务由 **Celery Beat** 进程周期性地将任务发往任务队列。

- 消息中间件 Broker

  Broker，即为任务调度队列，接收任务生产者发来的消息（即任务），将任务存入队列。

Celery 本身不提供队列服务，官方推荐使用 RabbitMQ 和 Redis 等。

- 任务执行单元 Worker

  Worker 是执行任务的处理单元，它实时监控消息队列，获取队列中调度的任务，并执行它。

- 任务结果存储 Backend

  Backend 用于存储任务的执行结果，以供查询。同消息中间件一样，存储也可使用 RabbitMQ, Redis 和 MongoDB 等。

异步任务

使用 Celery 实现异步任务主要包含三个步骤:

1. 创建一个 Celery 实例
2. 启动 Celery Worker
3. 应用程序调用异步任务

快速入门

为了简单起见，对于 Broker 和 Backend，这里都使用 redis。在运行下面的例子之前，请确保 redis 已正确安装，并开启 redis 服务，当然，celery 也是要安装的。可以使用下面的命令来安装 celery 及相关依赖:

```
$ pip install 'celery[redis]'
```

创建 Celery 实例

将下面的代码保存为文件 `tasks.py` :

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

import time
from celery import Celery

broker = 'redis://127.0.0.1:6379'
backend = 'redis://127.0.0.1:6379/0'

app = Celery('my_task', broker=broker, backend=backend)

@app.task
```

```
12. def add(x, y):
13.     time.sleep(5)     # 模拟耗时操作
14.     return x + y
```

上面的代码做了几件事：

- 创建了一个 Celery 实例 app，名称为 `my_task`；
- 指定消息中间件用 redis，URL 为 `redis://127.0.0.1:6379`；
- 指定存储用 redis，URL 为 `redis://127.0.0.1:6379/0`；
- 创建了一个 Celery 任务 `add`，当函数被 `@app.task` 装饰后，就成为可被 Celery 调度的任务；

启动 Celery Worker

在当前目录，使用如下方式启动 Celery Worker：

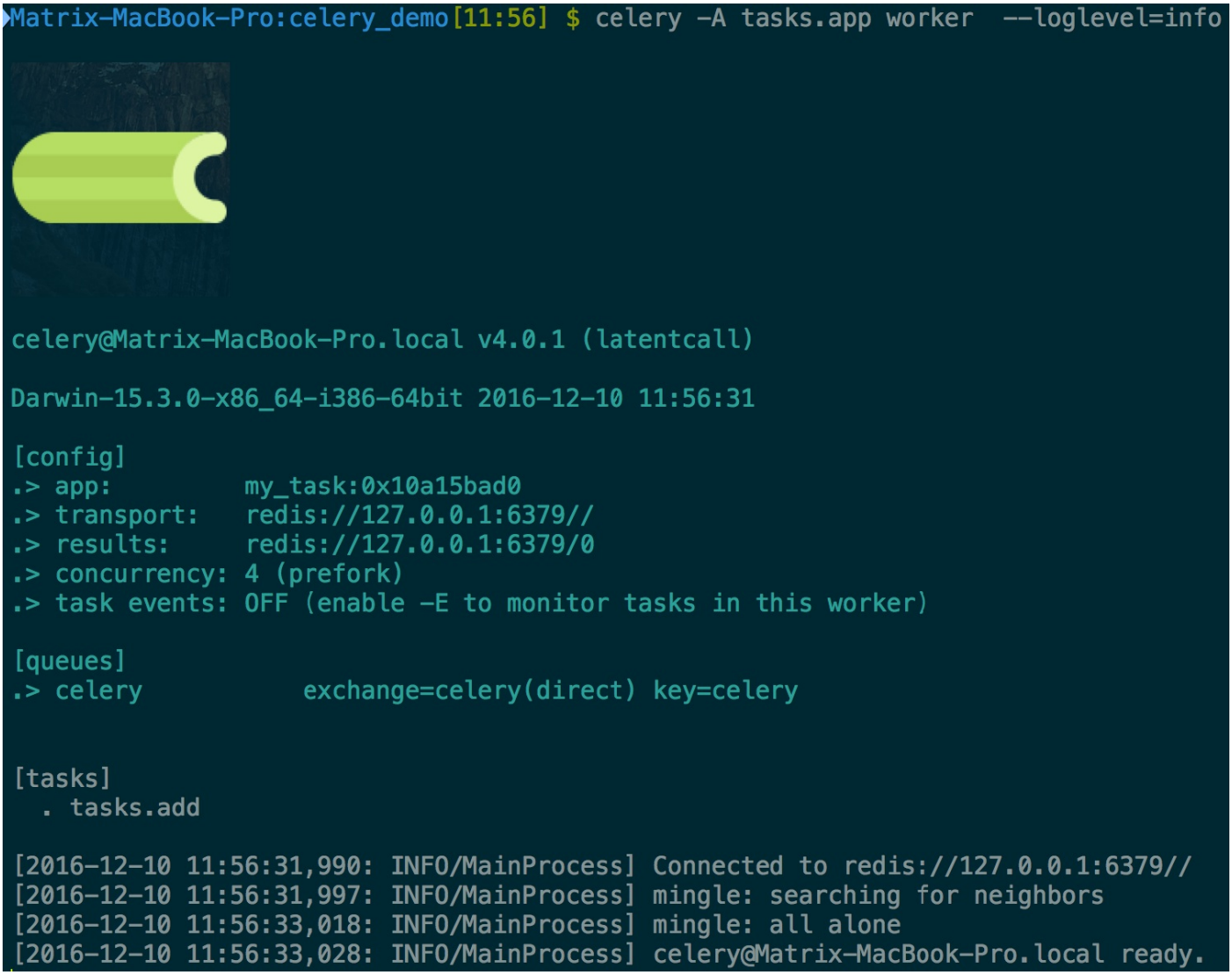
```
1. $ celery worker -A tasks --loglevel=info
```

其中：

- 参数 `-A` 指定了 Celery 实例的位置，本例是在 `tasks.py` 中，Celery 会自动在该文件中寻找 Celery 对象实例，当然，我们也可以自己指定，在本例，使用 `-A tasks.app`；
- 参数 `--loglevel` 指定了日志级别，默认为 warning，也可以使用 `-l info` 来表示；

在生产环境中，我们通常会使用 Supervisor 来控制 Celery Worker 进程。

启动成功后，控制台会显示如下输出：



调用任务

现在，我们可以在应用程序中使用 `delay()` 或 `apply_async()` 方法来调用任务。

在当前目录打开 Python 控制台，输入以下代码：

```
1. >>> from tasks import add
2. >>> add.delay(2, 8)
3. <AsyncResult: 2272ddce-8be5-493f-b5ff-35a0d9fe600f>
```

在上面，我们从 `tasks.py` 文件中导入了 `add` 任务对象，然后使用 `delay()` 方法将任务发送到消息中间件（Broker），Celery Worker 进程监控到该任务后，就会进行执行。我们将窗口切换到 Worker 的启动窗口，会看到多了两条日志：

```
    [2016-12-10 12:00:50,376: INFO/MainProcess] Received task: tasks.add[2272ddce-
1.  8be5-493f-b5ff-35a0d9fe600f]
    [2016-12-10 12:00:55,385: INFO/PoolWorker-4] Task tasks.add[2272ddce-8be5-493f-
2.  b5ff-35a0d9fe600f] succeeded in 5.00642602402s: 10
```

这说明任务已经被调度并执行成功。

另外，我们如果想获取执行后的结果，可以这样做：

```
>>> result = add.delay(2, 6)
>>> result.ready()   # 使用 ready() 判断任务是否执行完毕
False
>>> result.ready()
False
>>> result.ready()
True
>>> result.get()     # 使用 get() 获取任务结果
8
```

在上面，我们是在 Python 的环境中调用任务。事实上，我们通常在应用程序中调用任务。比如，将下面的代码保存为 `client.py` ：

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

from tasks import add

# 异步任务
add.delay(2, 8)

print 'hello world'
```

运行命令 `$ python client.py` ，可以看到，虽然任务函数 `add` 需要等待 5 秒才返回执行结果，但由于它是一个异步任务，不会阻塞当前的主程序，因此主程序会往下执行 `print` 语句，打印出结果。

使用配置

在上面的例子中，我们直接把 Broker 和 Backend 的配置写在了程序当中，更好的做法是将配置项统一写入到一个配置文件中，通常我们将该文件命名为 `celeryconfig.py` 。Celery 的配置比较多，可以在官方文档查询每个配置项的含义。

下面，我们再看一个例子。项目结构如下：

```
celery_demo                 # 项目根目录
    ├── celery_app          # 存放 celery 相关文件
    │   ├── __init__.py
```

```
    |   ├── celeryconfig.py    # 配置文件
    |   ├── task1.py           # 任务文件 1
    |   └── task2.py           # 任务文件 2
    └── client.py              # 应用程序
```

`__init__.py` 代码如下：

```python
# -*- coding: utf-8 -*-

from celery import Celery

app = Celery('demo')                                    # 创建 Celery 实例
app.config_from_object('celery_app.celeryconfig')   # 通过 Celery 实例加载配置模块
```

`celeryconfig.py` 代码如下：

```python
BROKER_URL = 'redis://127.0.0.1:6379'                # 指定 Broker
CELERY_RESULT_BACKEND = 'redis://127.0.0.1:6379/0'  # 指定 Backend

CELERY_TIMEZONE='Asia/Shanghai'                      # 指定时区，默认是 UTC
# CELERY_TIMEZONE='UTC'

CELERY_IMPORTS = (                                   # 指定导入的任务模块
    'celery_app.task1',
    'celery_app.task2'
)
```

`task1.py` 代码如下：

```python
import time
from celery_app import app

@app.task
def add(x, y):
    time.sleep(2)
    return x + y
```

`task2.py` 代码如下：

```python
import time
from celery_app import app
```

```
3.
4.  @app.task
5.  def multiply(x, y):
6.      time.sleep(2)
7.      return x * y
```

`client.py` 代码如下：

```
1.  # -*- coding: utf-8 -*-
2.
3.  from celery_app import task1
4.  from celery_app import task2
5.
6.  task1.add.apply_async(args=[2, 8])        # 也可用 task1.add.delay(2, 8)
7.  task2.multiply.apply_async(args=[3, 7])   # 也可用 task2.multiply.delay(3, 7)
8.
9.  print 'hello world'
```

现在，让我们启动 Celery Worker 进程，在项目的根目录下执行下面命令：

```
1.  celery_demo $ celery -A celery_app worker --loglevel=info
```

接着，运行 `$ python client.py`，它会发送两个异步任务到 Broker，在 Worker 的窗口我们可以看到如下输出：

```
    [2016-12-10 13:51:58,939: INFO/MainProcess] Received task:
1.  celery_app.task1.add[9ccffad0-aca4-4875-84ce-0ccfce5a83aa]
    [2016-12-10 13:51:58,941: INFO/MainProcess] Received task:
2.  celery_app.task2.multiply[64b1f889-c892-4333-bd1d-ac667e677a8a]
    [2016-12-10 13:52:00,948: INFO/PoolWorker-3] Task
    celery_app.task1.add[9ccffad0-aca4-4875-84ce-0ccfce5a83aa] succeeded in
3.  2.00600231002s: 10
    [2016-12-10 13:52:00,949: INFO/PoolWorker-4] Task
    celery_app.task2.multiply[64b1f889-c892-4333-bd1d-ac667e677a8a] succeeded in
4.  2.00601326401s: 21
```

delay 和 apply_async

在前面的例子中，我们使用 `delay()` 或 `apply_async()` 方法来调用任务。事实上，`delay` 方法封装了 `apply_async`，如下：

```
def delay(self, *partial_args, **partial_kwargs):
    """Shortcut to :meth:`apply_async` using star arguments."""
    return self.apply_async(partial_args, partial_kwargs)
```

也就是说，`delay` 是使用 `apply_async` 的快捷方式。`apply_async` 支持更多的参数，它的一般形式如下：

```
apply_async(args=(), kwargs={}, route_name=None, **options)
```

apply_async 常用的参数如下：

- countdown：指定多少秒后执行任务

```
task1.apply_async(args=(2, 3), countdown=5)    # 5 秒后执行任务
```

- eta (estimated time of arrival)：指定任务被调度的具体时间，参数类型是 datetime

```
from datetime import datetime, timedelta

# 当前 UTC 时间再加 10 秒后执行任务
task1.multiply.apply_async(args=[3, 7], eta=datetime.utcnow() +
timedelta(seconds=10))
```

- expires：任务过期时间，参数类型可以是 int，也可以是 datetime

```
task1.multiply.apply_async(args=[3, 7], expires=10)    # 10 秒后过期
```

更多的参数列表可以在官方文档中查看。

定时任务

Celery 除了可以执行异步任务，也支持执行周期性任务（**Periodic Tasks**），或者说定时任务。Celery Beat 进程通过读取配置文件的内容，周期性地将定时任务发往任务队列。

让我们看看例子，项目结构如下：

```
celery_demo                    # 项目根目录
    ├── celery_app             # 存放 celery 相关文件
        ├── __init__.py
        ├── celeryconfig.py    # 配置文件
```

```
        ├── task1.py            # 任务文件
        └── task2.py            # 任务文件
```

`__init__.py` 代码如下:

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

from celery import Celery

app = Celery('demo')
app.config_from_object('celery_app.celeryconfig')
```

`celeryconfig.py` 代码如下:

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

from datetime import timedelta
from celery.schedules import crontab

# Broker and Backend
BROKER_URL = 'redis://127.0.0.1:6379'
CELERY_RESULT_BACKEND = 'redis://127.0.0.1:6379/0'

# Timezone
CELERY_TIMEZONE='Asia/Shanghai'    # 指定时区，不指定默认为 'UTC'
# CELERY_TIMEZONE='UTC'

# import
CELERY_IMPORTS = (
    'celery_app.task1',
    'celery_app.task2'
)

# schedules
CELERYBEAT_SCHEDULE = {
    'add-every-30-seconds': {
         'task': 'celery_app.task1.add',
         'schedule': timedelta(seconds=30),       # 每 30 秒执行一次
         'args': (5, 8)                           # 任务函数参数
    },
    'multiply-at-some-time': {
        'task': 'celery_app.task2.multiply',
```

```
        'schedule': crontab(hour=9, minute=50),   # 每天早上 9 点 50 分执行一次
        'args': (3, 7)                            # 任务函数参数
    }
}
```

`task1.py` 代码如下：

```
import time
from celery_app import app

@app.task
def add(x, y):
    time.sleep(2)
    return x + y
```

`task2.py` 代码如下：

```
import time
from celery_app import app

@app.task
def multiply(x, y):
    time.sleep(2)
    return x * y
```

现在，让我们启动 Celery Worker 进程，在项目的根目录下执行下面命令：

```
celery_demo $ celery -A celery_app worker --loglevel=info
```

接着，启动 Celery Beat 进程，定时将任务发送到 Broker，在项目根目录下执行下面命令：

```
celery_demo $ celery beat -A celery_app
celery beat v4.0.1 (latentcall) is starting.
__    -    ... __   -        _
LocalTime -> 2016-12-11 09:48:16
Configuration ->
    . broker -> redis://127.0.0.1:6379//
    . loader -> celery.loaders.app.AppLoader
    . scheduler -> celery.beat.PersistentScheduler
    . db -> celerybeat-schedule
    . logfile -> [stderr]@%WARNING
```

```
11.      . maxinterval -> 5.00 minutes (300s)
```

之后，在 Worker 窗口我们可以看到，任务 `task1` 每 30 秒执行一次，而 `task2` 每天早上 9 点 50 分执行一次。

在上面，我们用两个命令启动了 Worker 进程和 Beat 进程，我们也可以将它们放在一个命令中：

```
1.  $ celery -B -A celery_app worker --loglevel=info
```

Celery 周期性任务也有多个配置项，可参考官方文档。

参考资料

- Celery - Distributed Task Queue — Celery 4.0.1 documentation
- 使用Celery - Python之美
- 分布式任务队列Celery的介绍 – 思诚之道
- 异步任务神器 Celery 简明笔记

Click

Click 是 Flask 的开发团队 Pallets 的另一款开源项目，它是用于快速创建命令行的第三方模块。我们知道，Python 内置了一个 Argparse 的标准库用于创建命令行，但使用起来有些繁琐，Click 相比于 Argparse，就好比 requests 相比于 urllib。

快速使用

Click 的使用大致有两个步骤：

1. 使用 `@click.command()` 装饰一个函数，使之成为命令行接口；
2. 使用 `@click.option()` 等装饰函数，为其添加命令行选项等。

它的一种典型使用形式如下：

```
import click

@click.command()
@click.option('--param', default=default_value, help='description')
def func(param):
    pass
```

下面，让我们看一下官方文档的入门例子：

```
import click

@click.command()
@click.option('--count', default=1, help='Number of greetings.')
@click.option('--name', prompt='Your name', help='The person to greet.')
def hello(count, name):
    """Simple program that greets NAME for a total of COUNT times."""
    for x in range(count):
        click.echo('Hello %s!' % name)

if __name__ == '__main__':
    hello()
```

在上面的例子中，函数 `hello` 有两个参数：count 和 name，它们的值从命令行中获取。

- `@click.command()` 使函数 `hello` 成为命令行接口；

- `@click.option` 的第一个参数指定了命令行选项的名称，不难猜到，count 的默认值是 1，name 的值从输入获取；
- 使用 `click.echo` 进行输出是为了获得更好的兼容性，因为 `print` 在 Python2 和 Python3 的用法有些差别。

看看执行情况：

```
$ python hello.py
Your name: Ethan            # 这里会显示 'Your name: '(对应代码中的 prompt)，接受用户输入
Hello Ethan!

$ python hello.py --help    # click 帮我们自动生成了 `--help` 用法
Usage: hello.py [OPTIONS]

  Simple program that greets NAME for a total of COUNT times.

Options:
  --count INTEGER  Number of greetings.
  --name TEXT      The person to greet.
  --help           Show this message and exit.

$ python hello.py --count 3 --name Ethan    # 指定 count 和 name 的值
Hello Ethan!
Hello Ethan!
Hello Ethan!

$ python hello.py --count=3 --name=Ethan    # 也可以使用 `=`，和上面等价
Hello Ethan!
Hello Ethan!
Hello Ethan!

$ python hello.py --name=Ethan              # 没有指定 count，默认值是 1
Hello Ethan!
```

click.option

option 最基本的用法就是通过指定命令行选项的名称，从命令行读取参数值，再将其传递给函数。在上面的例子，我们看到，除了设置命令行选项的名称，我们还会指定默认值，help 说明等，option 常用的设置参数如下：

- default: 设置命令行参数的默认值
- help: 参数说明
- type: 参数类型，可以是 string, int, float 等
- prompt: 当在命令行中没有输入相应的参数时，会根据 prompt 提示用户输入
- nargs: 指定命令行参数接收的值的个数

下面，我们再看看相关的例子。

指定 type

我们可以使用 `type` 来指定参数类型：

```
1. import click
2.
3. @click.command()
4. @click.option('--rate', type=float, help='rate')   # 指定 rate 是 float 类型
5. def show(rate):
6.     click.echo('rate: %s' % rate)
7.
8. if __name__ == '__main__':
9.     show()
```

执行情况：

```
1. $ python click_type.py --rate 1
2. rate: 1.0
3. $ python click_type.py --rate 0.66
4. rate: 0.66
```

可选值

在某些情况下，一个参数的值只能是某些可选的值，如果用户输入了其他值，我们应该提示用户输入正确的值。在这种情况下，我们可以通过 `click.Choice()` 来限定：

```
1. import click
2.
3. @click.command()
4. @click.option('--gender', type=click.Choice(['man', 'woman']))    # 限定值
5. def choose(gender):
6.     click.echo('gender: %s' % gender)
```

```
7.
8.  if __name__ == '__main__':
9.      choose()
```

执行情况：

```
1.  $ python click_choice.py --gender boy
2.  Usage: click_choice.py [OPTIONS]
3.
    Error: Invalid value for "--gender": invalid choice: boy. (choose from man,
4.  woman)
5.
6.  $ python click_choice.py --gender man
7.  gender: man
```

多值参数

有时，一个参数需要接收多个值。**option** 支持设置固定长度的参数值，通过 `nargs` 指定。

看看例子就明白了：

```
1.  import click
2.
3.  @click.command()
4.  @click.option('--center', nargs=2, type=float, help='center of the circle')
5.  @click.option('--radius', type=float, help='radius of the circle')
6.  def circle(center, radius):
7.      click.echo('center: %s, radius: %s' % (center, radius))
8.
9.  if __name__ == '__main__':
10.     circle()
```

在上面的例子中，option 指定了两个参数：center 和 radius，其中，center 表示二维平面上一个圆的圆心坐标，接收两个值，以元组的形式将值传递给函数，而 radius 表示圆的半径。

执行情况：

```
1.  $ python click_multi_values.py --center 3 4 --radius 10
2.  center: (3.0, 4.0), radius: 10.0
3.
4.  $ python click_multi_values.py --center 3 4 5 --radius 10
```

```
Usage: click_multi_values.py [OPTIONS]

Error: Got unexpected extra argument (5)
```

输入密码

有时，在输入密码的时候，我们希望能隐藏显示。option 提供了两个参数来设置密码的输入：hide_input 和 confirmation_promt，其中，hide_input 用于隐藏输入，confirmation_promt 用于重复输入。

看看例子：

```
import click

@click.command()
@click.option('--password', prompt=True, hide_input=True,
confirmation_prompt=True)
def input_password(password):
    click.echo('password: %s' % password)


if __name__ == '__main__':
    input_password()
```

执行情况：

```
$ python click_password.py
Password:                       # 不会显示密码
Repeat for confirmation:        # 重复一遍
password: 666666
```

由于上面的写法有点繁琐，click 也提供了一种快捷的方式，通过使用 `@click.password_option()` ，上面的代码可以简写成：

```
import click

@click.command()
@click.password_option()
def input_password(password):
    click.echo('password: %s' % password)

```

```
if __name__ == '__main__':
    input_password()
```

改变命令行程序的执行

有些参数会改变命令行程序的执行，比如在终端输入 `python` 是进入 python 控制台，而输入 `python --version` 是打印 python 版本。Click 提供 eager 标识对参数名进行标识，如果输入该参数，则会拦截既定的命令行执行流程，跳转去执行一个回调函数。

让我们看看例子：

```
import click

def print_version(ctx, param, value):
    if not value or ctx.resilient_parsing:
        return
    click.echo('Version 1.0')
    ctx.exit()

@click.command()
@click.option('--version', is_flag=True, callback=print_version,
              expose_value=False, is_eager=True)
@click.option('--name', default='Ethan', help='name')
def hello(name):
    click.echo('Hello %s!' % name)

if __name__ == '__main__':
    hello()
```

其中：

- `is_eager=True` 表明该命令行选项优先级高于其他选项；
- `expose_value=False` 表示如果没有输入该命令行选项，会执行既定的命令行流程；
- `callback` 指定了输入该命令行选项时，要跳转执行的函数；

执行情况：

```
$ python click_eager.py
Hello Ethan!

$ python click_eager.py --version                    # 拦截既定的命令行执行流程
Version 1.0
```

```
6.
7.  $ python click_eager.py --name Michael
8.  Hello Michael!
9.
10. $ python click_eager.py --version --name Ethan      # 忽略 name 选项
11. Version 1.0
```

click.argument

我们除了使用 `@click.option` 来添加可选参数，还会经常使用 `@click.argument` 来添加固定参数。它的使用和 option 类似，但支持的功能比 option 少。

入门使用

下面是一个简单的例子：

```
1.  import click
2.
3.  @click.command()
4.  @click.argument('coordinates')
5.  def show(coordinates):
6.      click.echo('coordinates: %s' % coordinates)
7.
8.  if __name__ == '__main__':
9.      show()
```

看看执行情况：

```
1.  $ python click_argument.py                    # 错误，缺少参数 coordinates
2.  Usage: click_argument.py [OPTIONS] COORDINATES
3.
4.  Error: Missing argument "coordinates".
5.
    $ python click_argument.py --help             # argument 指定的参数在 help 中没有
6.  显示
7.  Usage: click_argument.py [OPTIONS] COORDINATES
8.
9.  Options:
10.   --help  Show this message and exit.
11.
12. $ python click_argument.py --coordinates 10    # 错误用法，这是 option 参数的用法
```

```
Error: no such option: --coordinates

$ python click_argument.py 10                    # 正确，直接输入值即可
coordinates: 10
```

多个 argument

我们再来看看多个 argument 的例子：

```
import click

@click.command()
@click.argument('x')
@click.argument('y')
@click.argument('z')
def show(x, y, z):
    click.echo('x: %s, y: %s, z:%s' % (x, y, z))

if __name__ == '__main__':
    show()
```

执行情况：

```
$ python click_argument.py 10 20 30
x: 10, y: 20, z:30

$ python click_argument.py 10
Usage: click_argument.py [OPTIONS] X Y Z

Error: Missing argument "y".

$ python click_argument.py 10 20
Usage: click_argument.py [OPTIONS] X Y Z

Error: Missing argument "z".

$ python click_argument.py 10 20 30 40
Usage: click_argument.py [OPTIONS] X Y Z

Error: Got unexpected extra argument (40)
```

不定参数

argument 还有另外一种常见的用法，就是接收不定量的参数，让我们看看例子：

```
1.  import click
2.
3.  @click.command()
4.  @click.argument('src', nargs=-1)
5.  @click.argument('dst', nargs=1)
6.  def move(src, dst):
7.      click.echo('move %s to %s' % (src, dst))
8.
9.  if __name__ == '__main__':
10.     move()
```

其中，`nargs=-1` 表明参数 `src` 接收不定量的参数值，参数值会以 tuple 的形式传入函数。如果 `nargs` 大于等于 1，表示接收 `nargs` 个参数值，上面的例子中，`dst` 接收一个参数值。

让我们看看执行情况：

```
1.  $ python click_argument.py file1 trash    # src=('file1',)  dst='trash'
2.  move (u'file1',) to trash
3.
    $ python click_argument.py file1 file2 file3 trash   # src=('file1', 'file2',
4.  'file3')  dst='trash'
5.  move (u'file1', u'file2', u'file3') to trash
```

彩色输出

在前面的例子中，我们使用 `click.echo` 进行输出，如果配合 colorama 这个模块，我们可以使用 `click.secho` 进行彩色输出，在使用之前，使用 pip 安装 colorama：

```
1.  $ pip install colorama
```

看看例子：

```
1.  import click
2.
3.  @click.command()
```

```
@click.option('--name', help='The person to greet.')
def hello(name):
    click.secho('Hello %s!' % name, fg='red', underline=True)
    click.secho('Hello %s!' % name, fg='yellow', bg='black')

if __name__ == '__main__':
    hello()
```

其中：

- `fg` 表示前景颜色（即字体颜色），可选值有：BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, WHITE 等；
- `bg` 表示背景颜色，可选值有：BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, WHITE 等；
- `underline` 表示下划线，可选的样式还有：`dim=True`，`bold=True` 等；

小结

- 使用 `click.command()` 装饰一个函数，使其成为命令行接口。
- 使用 `click.option()` 添加可选参数，支持设置固定长度的参数值。
- 使用 `click.argument()` 添加固定参数，支持设置不定长度的参数值。

参考资料

- Click Documentation (6.0)
- Python Click 学习笔记 | I sudo X
- click模块 - cdwanze

结束语

到这里，虽然本书结束了，但对于 Python 的学习和实践还远远没结束，后面我也会持续更新本书。虽然 Python 的语法相比 C++ 等语言比较简洁，但想熟练运用，仍需在实际的项目中多多实践，而不只是停留在简单的概念学习中。

这里主要推荐 Python 相关的一些学习资源，同时也列出本书的主要参考资料。

资源推荐

这里列出了 Python 相关的一些资源，欢迎读者补充。

- vinta/awesome-python: A curated list of awesome Python frameworks, libraries, software and resources

  包含了 Python 框架、Python 库和软件的 awesome 列表。

- aosabook/500lines: 500 Lines or Less

  Python 神书，里面有若干个项目，每个项目都是由业内大神所写，每个项目代码在 500 行左右。

- Python Module of the Week - PyMOTW 2

  自 2007 年以来，Doug Hellmann 在他的博客上发表了颇受关注的「Python Module of the Week」系列，计划每周介绍一个 Python 标准库的使用。上面的链接是介绍 Python2 中的标准库，同样也有 Python3 的：Python 3 Module of the Week — PyMOTW 3。

- Transforming Code into Beautiful, Idiomatic Python

  写出简洁的、优雅的 Python 代码。

- jobbole/awesome-python-cn: Python资源大全中文版

  Python 资源大全，包含：Web 框架、网络爬虫、模板引擎和数据库等，由伯乐在线更新。

- Pycoder’s Weekly | A Weekly Python E-Mail Newsletter

  优秀的免费邮件 Python 新闻周刊。

- Python 初学者的最佳学习资源

  伯乐在线翻译的 Python 学习资源。

- Full Stack Python

  Python 资源汇总，从基础入门到各种 Web 开发框架，再到高级的 ORM，Docker 等等。

- The Hitchhiker’s Guide to Python!

  Requests 作者 kennethreitz 的一本开源书籍，介绍 Python 的最佳实践。

- Welcome to Python for you and me

介绍 Python 的基本语法，特点等。

- District Data Labs - How to Develop Quality Python Code

  开发高质量的 Python 代码。

- A “Best of the Best Practices” (BOBP) guide to developing in Python.

  Python 最佳实践。

参考资料

本书的参考资料主要有:

- 《Python 基础教程》
- 《Python 核心编程》
- 廖雪峰 python 教程
- Python进阶 · GitBook
- python3-cookbook 2.0.0 文档

单例模式

单例模式（**Singleton Pattern**）是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。当你希望在整个系统中，某个类只能出现一个实例时，单例对象就能派上用场。

比如，某个服务器程序的配置信息存放在一个文件中，客户端通过一个 AppConfig 的类来读取配置文件的信息。如果在程序运行期间，有很多地方都需要使用配置文件的内容，也就是说，很多地方都需要创建 AppConfig 对象的实例，这就导致系统中存在多个 AppConfig 的实例对象，而这样会严重浪费内存资源，尤其是在配置文件内容很多的情况下。事实上，类似 AppConfig 这样的类，我们希望在程序运行期间只存在一个实例对象。

在 Python 中，我们可以用多种方法来实现单例模式：

- 使用模块
- 使用 `__new__`
- 使用装饰器（decorator）
- 使用元类（metaclass）

使用模块

其实，**Python** 的模块就是天然的单例模式，因为模块在第一次导入时，会生成 `.pyc` 文件，当第二次导入时，就会直接加载 `.pyc` 文件，而不会再次执行模块代码。因此，我们只需把相关的函数和数据定义在一个模块中，就可以获得一个单例对象了。如果我们真的想要一个单例类，可以考虑这样做：

```
1. # mysingleton.py
2. class My_Singleton(object):
3.     def foo(self):
4.         pass
5.
6. my_singleton = My_Singleton()
```

将上面的代码保存在文件 `mysingleton.py` 中，然后这样使用：

```
1. from mysingleton import my_singleton
2.
3. my_singleton.foo()
```

使用 `__new__`

为了使类只能出现一个实例，我们可以使用 `__new__` 来控制实例的创建过程，代码如下：

```
class Singleton(object):
    _instance = None
    def __new__(cls, *args, **kw):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls, *args, **kw)
        return cls._instance

class MyClass(Singleton):
    a = 1
```

在上面的代码中，我们将类的实例和一个类变量 `_instance` 关联起来，如果 `cls._instance` 为 None 则创建实例，否则直接返回 `cls._instance` 。

执行情况如下：

```
>>> one = MyClass()
>>> two = MyClass()
>>> one == two
True
>>> one is two
True
>>> id(one), id(two)
(4303862608, 4303862608)
```

使用装饰器

我们知道，装饰器（decorator）可以动态地修改一个类或函数的功能。这里，我们也可以使用装饰器来装饰某个类，使其只能生成一个实例，代码如下：

```
from functools import wraps

def singleton(cls):
    instances = {}
    @wraps(cls)
    def getinstance(*args, **kw):
        if cls not in instances:
            instances[cls] = cls(*args, **kw)
        return instances[cls]
    return getinstance
```

```
11.
12. @singleton
13. class MyClass(object):
14.     a = 1
```

在上面，我们定义了一个装饰 `singleton` ，它返回了一个内部函数 `getinstance` ，该函数会判断某个类是否在字典 `instances` 中，如果不存在，则会将 `cls` 作为 key， `cls(*args, **kw)` 作为 value 存到 `instances` 中，否则，直接返回 `instances[cls]` 。

使用 metaclass

元类（metaclass）可以控制类的创建过程，它主要做三件事：

- 拦截类的创建
- 修改类的定义
- 返回修改后的类

使用元类实现单例模式的代码如下：

```
1. class Singleton(type):
2.     _instances = {}
3.     def __call__(cls, *args, **kwargs):
4.         if cls not in cls._instances:
               cls._instances[cls] = super(Singleton, cls).__call__(*args,
5. **kwargs)
6.         return cls._instances[cls]
7.
8. # Python2
9. class MyClass(object):
10.     __metaclass__ = Singleton
11.
12. # Python3
13. # class MyClass(metaclass=Singleton):
14. #     pass
```

小结

- Python 的模块是天然的单例模式，这在大部分情况下应该是够用的，当然，我们也可以使用装饰器、元类等方法

参考资料

- Creating a singleton in Python - Stack Overflow
- 深入浅出单实例Singleton设计模式 | 酷 壳
- design patterns - Python’s use of **new** and **init**? - Stack Overflow