目录

[第一节 Python介绍 7](#_Toc454522262)

[第二节 Python解释器 8](#_Toc454522263)

[第三课 命令行模式和Python交互模式 9](#_Toc454522264)

[第四课 使用文本编辑器 11](#_Toc454522265)

[第5课 输入和输出 13](#_Toc454522266)

[第6课 Python代码 16](#_Toc454522267)

[第7课 数据类型和变量 17](#_Toc454522268)

[数据类型 17](#_Toc454522269)

[变量 21](#_Toc454522270)

[常量 23](#_Toc454522271)

[小结 24](#_Toc454522272)

[第8课 字符串和编码 25](#_Toc454522273)

[字符编码 25](#_Toc454522274)

[Python的字符串 28](#_Toc454522275)

[格式化 31](#_Toc454522276)

[小结 33](#_Toc454522277)

[第9课使用list和tuple 33](#_Toc454522278)

[list 33](#_Toc454522279)

[tuple 36](#_Toc454522280)

[第10课 条件判断 39](#_Toc454522281)

[条件判断 39](#_Toc454522282)

[再议 input 41](#_Toc454522283)

[第11课 循环 42](#_Toc454522284)

[循环 42](#_Toc454522285)

[dict 44](#_Toc454522286)

[set 47](#_Toc454522287)

[再议不可变对象 48](#_Toc454522288)

[小结 50](#_Toc454522289)

[第12课 函数 50](#_Toc454522290)

[抽象 51](#_Toc454522291)

[第13课 调用函数 51](#_Toc454522292)

[数据类型转换 53](#_Toc454522293)

[第14课 定义函数 54](#_Toc454522294)

[空函数 54](#_Toc454522295)

[参数检查 55](#_Toc454522296)

[返回多个值 56](#_Toc454522297)

[小结 57](#_Toc454522298)

[第15课 函数的参数 58](#_Toc454522299)

[位置参数 58](#_Toc454522300)

[默认参数 59](#_Toc454522301)

[可变参数 63](#_Toc454522302)

[关键字参数 65](#_Toc454522303)

[命名关键字参数 66](#_Toc454522304)

[参数组合 68](#_Toc454522305)

[小结 69](#_Toc454522306)

[第16课 递归函数 70](#_Toc454522307)

[小结 73](#_Toc454522308)

[第17课 高级特性 73](#_Toc454522309)

[第18课 切片 74](#_Toc454522310)

[小结 77](#_Toc454522311)

[第19课 迭代 77](#_Toc454522312)

[小结 79](#_Toc454522313)

[第20课 列表生成式 79](#_Toc454522314)

[第21课 生成器 82](#_Toc454522315)

[小结 88](#_Toc454522316)

[第22课 迭代器 89](#_Toc454522317)

[小结 90](#_Toc454522318)

[第23课 函数式编程 91](#_Toc454522319)

[第24课 高阶函数 92](#_Toc454522320)

[map/reduce 95](#_Toc454522321)

[filter 98](#_Toc454522322)

[sorted 101](#_Toc454522323)

[第25课 返回函数 102](#_Toc454522324)

[函数作为返回值 102](#_Toc454522325)

[闭包 104](#_Toc454522326)

[小结 106](#_Toc454522327)

[第26课 匿名函数 106](#_Toc454522328)

[小结 107](#_Toc454522329)

[第27课 装饰器 107](#_Toc454522330)

[小结 111](#_Toc454522331)

[附：更多实例 112](#_Toc454522332)

[第28课 偏函数 114](#_Toc454522333)

[小结 116](#_Toc454522334)

[第29课 模块 116](#_Toc454522335)

[附 \_\_init\_\_.py 118](#_Toc454522336)

[\_\_file\_\_与argv[0] 119](#_Toc454522337)

[第30课 使用模块 119](#_Toc454522338)

[作用域 122](#_Toc454522339)

[附 123](#_Toc454522340)

[第31课 安装第三方模块 123](#_Toc454522341)

[模块搜索路径 124](#_Toc454522342)

[第31课 面向对象编程 125](#_Toc454522343)

[小结 127](#_Toc454522344)

[第32课 类和实例 127](#_Toc454522345)

[数据封装 129](#_Toc454522346)

[小结 131](#_Toc454522347)

[附 \_\_init\_\_()方法 131](#_Toc454522348)

[第33课 访问限制 133](#_Toc454522349)

[第34课 继承和多态 135](#_Toc454522350)

[静态语言 vs 动态语言 141](#_Toc454522351)

[小结 141](#_Toc454522352)

[第35课 获取对象信息 142](#_Toc454522353)

[使用type() 142](#_Toc454522354)

[使用isinstance() 143](#_Toc454522355)

[使用dir() 145](#_Toc454522356)

[小结 148](#_Toc454522357)

[第36课 实例属性和类属性 149](#_Toc454522358)

[第37课 面向对象高级编程 151](#_Toc454522359)

[第38课 使用\_\_slots\_\_ 151](#_Toc454522360)

[使用\_\_slots\_\_ 153](#_Toc454522361)

[第39课 使用@property属性\* 154](#_Toc454522362)

[小结 157](#_Toc454522363)

[第40课 多重继承 157](#_Toc454522364)

[MixIn 160](#_Toc454522365)

[小结 161](#_Toc454522366)

[第41课 定制类 161](#_Toc454522367)

[\_\_str\_\_ 161](#_Toc454522368)

[\_\_iter\_\_ 163](#_Toc454522369)

[\_\_getitem\_\_ 164](#_Toc454522370)

[\_\_getattr\_\_ 167](#_Toc454522371)

[\_\_call\_\_ 170](#_Toc454522372)

[小结 172](#_Toc454522373)

[第42课 使用枚举类 172](#_Toc454522374)

[小结 175](#_Toc454522375)

[第43课 使用元类 175](#_Toc454522376)

[type() 175](#_Toc454522377)

[metaclass\* 177](#_Toc454522378)

[小结 183](#_Toc454522379)

[第44课 错误、调试和测试 183](#_Toc454522380)

[第45课 错误处理 184](#_Toc454522381)

[try 185](#_Toc454522382)

[调用堆栈 188](#_Toc454522383)

[记录错误 190](#_Toc454522384)

[抛出错误 192](#_Toc454522385)

[小结 194](#_Toc454522386)

[第46课 调试 194](#_Toc454522387)

[断言 195](#_Toc454522388)

[logging 196](#_Toc454522389)

[pdb 197](#_Toc454522390)

[pdb.set\_trace() 199](#_Toc454522391)

[IDE 200](#_Toc454522392)

[小结 200](#_Toc454522393)

[第47课 单元测试 200](#_Toc454522394)

[运行单元测试 204](#_Toc454522395)

[setUp与tearDown 205](#_Toc454522396)

[小结 205](#_Toc454522397)

[附super() 205](#_Toc454522398)

[第48课 文档测试\* 206](#_Toc454522399)

[第49课 IO编程 211](#_Toc454522400)

[第50课 文件读写 212](#_Toc454522401)

[读文件 213](#_Toc454522402)

[file-like Object 215](#_Toc454522403)

[二进制文件 215](#_Toc454522404)

[字符编码 215](#_Toc454522405)

[写文件 215](#_Toc454522406)

[小结 216](#_Toc454522407)

[附二进制文件与文本文件 216](#_Toc454522408)

[第51课 StringIO和BytesIO 217](#_Toc454522409)

[StringIO 217](#_Toc454522410)

[BytesIO 218](#_Toc454522411)

[小结 219](#_Toc454522412)

[第52课 操作文件和目录 219](#_Toc454522413)

[环境变量 220](#_Toc454522414)

[操作文件和目录 220](#_Toc454522415)

[小结 222](#_Toc454522416)

[第53课 序列化 223](#_Toc454522417)

[JSON 224](#_Toc454522418)

[JSON进阶 225](#_Toc454522419)

[小结 228](#_Toc454522420)

[第54课 进程和线程 228](#_Toc454522421)

[小结 229](#_Toc454522422)

[第55课 多进程 230](#_Toc454522423)

[multiprocessing 231](#_Toc454522424)

[Pool 232](#_Toc454522425)

[子进程 234](#_Toc454522426)

[进程间通信 236](#_Toc454522427)

[小结 238](#_Toc454522428)

[第56课 多线程 238](#_Toc454522429)

[Lock 240](#_Toc454522430)

[多核CPU 243](#_Toc454522431)

[小结 245](#_Toc454522432)

[第57课 ThreadLocal 245](#_Toc454522433)

[小结 248](#_Toc454522434)

[第58课 进程 vs 线程 248](#_Toc454522435)

[线程切换 249](#_Toc454522436)

[计算密集型 vs. IO密集型 250](#_Toc454522437)

[异步IO 250](#_Toc454522438)

[第59课 分布式进程\* 251](#_Toc454522439)

[小结 257](#_Toc454522440)

[第60课 正则表达式 257](#_Toc454522441)

[进阶 258](#_Toc454522442)

[re模块 259](#_Toc454522443)

[切分字符串 260](#_Toc454522444)

[分组 260](#_Toc454522445)

[贪婪匹配 261](#_Toc454522446)

[编译 262](#_Toc454522447)

[小结 262](#_Toc454522448)

[第61课 访问数据库 263](#_Toc454522449)

[NoSQL 266](#_Toc454522450)

[数据库类别 266](#_Toc454522451)

[第62课 使用SQLite 267](#_Toc454522452)

[小结 269](#_Toc454522453)

[第63课 使用mysql 270](#_Toc454522454)

[安装MySQL 270](#_Toc454522455)

[安装MySQL驱动 272](#_Toc454522456)

[小结 273](#_Toc454522457)

[第64课 使用SQLAlchemy\* 273](#_Toc454522458)

[小结 278](#_Toc454522459)

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Python之禅  
赖勇浩翻译

优美胜于丑陋（Python 以编写优美的代码为目标）   
明了胜于晦涩（优美的代码应当是明了的，命名规范，风格相似）   
简洁胜于复杂（优美的代码应当是简洁的，不要有复杂的内部实现）   
复杂胜于凌乱（如果复杂不可避免，那代码间也不能有难懂的关系，要保持接口简洁）   
扁平胜于嵌套（优美的代码应当是扁平的，不能有太多的嵌套）   
间隔胜于紧凑（优美的代码有适当的间隔，不要奢望一行代码解决问题）   
可读性很重要（优美的代码是可读的）   
即便假借特例的实用性之名，也不可违背这些规则（这些规则至高无上）   
不要包容所有错误，除非你确定需要这样做（精准地捕获异常，不写 except:pass 风格的代码）   
当存在多种可能，不要尝试去猜测   
而是尽量找一种，最好是唯一一种明显的解决方案（如果不确定，就用穷举法）   
虽然这并不容易，因为你不是 Python 之父（这里的 Dutch 是指 Guido ）   
做也许好过不做，但不假思索就动手还不如不做（动手之前要细思量）   
如果你无法向人描述你的方案，那肯定不是一个好方案；反之亦然（方案测评标准）   
命名空间是一种绝妙的理念，我们应当多加利用（倡导与号召）

## 第一节 Python介绍

最后说说Python的缺点。

任何编程语言都有缺点，Python也不例外。优点说过了，那Python有哪些缺点呢？

第一个缺点就是运行速度慢，和C程序相比非常慢，因为Python是解释型语言，你的代码在执行时会一行一行地翻译成CPU能理解的机器码，这个翻译过程非常耗时，所以很慢。而C程序是运行前直接编译成CPU能执行的机器码，所以非常快。

但是大量的应用程序不需要这么快的运行速度，因为用户根本感觉不出来。例如开发一个下载MP3的网络应用程序，C程序的运行时间需要0.001秒，而Python程序的运行时间需要0.1秒，慢了100倍，但由于网络更慢，需要等待1秒，你想，用户能感觉到1.001秒和1.1秒的区别吗？这就好比F1赛车和普通的出租车在北京三环路上行驶的道理一样，虽然F1赛车理论时速高达400公里，但由于三环路堵车的时速只有20公里，因此，作为乘客，你感觉的时速永远是20公里。



第二个缺点就是代码不能加密。如果要发布你的Python程序，实际上就是发布源代码，这一点跟C语言不同，C语言不用发布源代码，只需要把编译后的机器码（也就是你在Windows上常见的xxx.exe文件）发布出去。要从机器码反推出C代码是不可能的，所以，凡是编译型的语言，都没有这个问题，而解释型的语言，则必须把源码发布出去。

这个缺点仅限于你要编写的软件需要卖给别人挣钱的时候。好消息是目前的互联网时代，靠卖软件授权的商业模式越来越少了，靠网站和移动应用卖服务的模式越来越多了，后一种模式不需要把源码给别人。

再说了，现在如火如荼的开源运动和互联网自由开放的精神是一致的，互联网上有无数非常优秀的像Linux一样的开源代码，我们千万不要高估自己写的代码真的有非常大的“商业价值”。那些大公司的代码不愿意开放的更重要的原因是代码写得太烂了，一旦开源，就没人敢用他们的产品了。

## 第二节 Python解释器

当我们编写Python代码时，我们得到的是一个包含Python代码的以.py为扩展名的文本文件。要运行代码，就需要Python解释器去执行.py文件。

由于整个Python语言从规范到解释器都是开源的，所以理论上，只要水平够高，任何人都可以编写Python解释器来执行Python代码（当然难度很大）。事实上，确实存在多种Python解释器。

CPython

当我们从[Python官方网站](https://www.python.org/)下载并安装好Python 3.5后，我们就直接获得了一个官方版本的解释器：CPython。这个解释器是用C语言开发的，所以叫CPython。在命令行下运行python就是启动CPython解释器。

CPython是使用最广的Python解释器。教程的所有代码也都在CPython下执行。

IPython

IPython是基于CPython之上的一个交互式解释器，也就是说，IPython只是在交互方式上有所增强，但是执行Python代码的功能和CPython是完全一样的。好比很多国产浏览器虽然外观不同，但内核其实都是调用了IE。

CPython用>>>作为提示符，而IPython用In [序号]:作为提示符。

PyPy

PyPy是另一个Python解释器，它的目标是执行速度。PyPy采用[JIT技术](http://en.wikipedia.org/wiki/Just-in-time_compilation)，对Python代码进行动态编译（注意不是解释），所以可以显著提高Python代码的执行速度。

绝大部分Python代码都可以在PyPy下运行，但是PyPy和CPython有一些是不同的，这就导致相同的Python代码在两种解释器下执行可能会有不同的结果。如果你的代码要放到PyPy下执行，就需要了解[PyPy和CPython的不同点](http://pypy.readthedocs.org/en/latest/cpython_differences.html)。

Jython

Jython是运行在Java平台上的Python解释器，可以直接把Python代码编译成Java字节码执行。

IronPython

IronPython和Jython类似，只不过IronPython是运行在微软.Net平台上的Python解释器，可以直接把Python代码编译成.Net的字节码。

小结

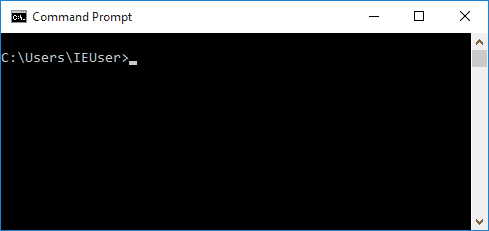
Python的解释器很多，但使用最广泛的还是CPython。如果要和Java或.Net平台交互，最好的办法不是用Jython或IronPython，而是通过网络调用来交互，确保各程序之间的独立性。

本教程的所有代码只确保在CPython 3.5版本下运行。请务必在本地安装CPython（也就是从Python官方网站下载的安装程序）。

## 第三课 命令行模式和Python交互模式

请注意区分命令行模式和Python交互模式。

看到类似C:\>是在Windows提供的命令行模式：



在命令行模式下，可以执行python进入Python交互式环境，也可以执行python hello.py运行一个.py文件。

看到>>>是在Python交互式环境下：



在Python交互式环境下，只能输入Python代码并立刻执行。

此外，在命令行模式运行.py文件和在Python交互式环境下直接运行Python代码有所不同。Python交互式环境会把每一行Python代码的结果自动打印出来，但是，直接运行Python代码却不会。

例如，在Python交互式环境下，输入：

>>> 100 + 200 + 300

600

直接可以看到结果600。

但是，写一个calc.py的文件，内容如下：

100 + 200 + 300

然后在命令行模式下执行：

C:\work>python calc.py

发现什么输出都没有。

这是正常的。想要输出结果，必须自己用print()打印出来。把calc.py改造一下：

**print**(100 + 200 + 300)

再执行，就可以看到结果：

C:\work>python calc.py

600

小结

在Python交互式命令行下，可以直接输入代码，然后执行，并立刻得到结果。

## 第四课 使用文本编辑器

请注意，用哪个都行，但是绝对不能用Word和Windows自带的记事本。Word保存的不是纯文本文件，而记事本会自作聪明地在文件开始的地方加上几个特殊字符（UTF-8 BOM），结果会导致程序运行出现莫名其妙的错误。

安装好文本编辑器后，输入以下代码：

**print**('hello, world')

注意print前面不要有任何空格。然后，选择一个目录，例如C:\work，把文件保存为hello.py，就可以打开命令行窗口，把当前目录切换到hello.py所在目录，就可以运行这个程序了：

C:\work>python hello.py

hello, world

也可以保存为别的名字，比如first.py，但是必须要以.py结尾，其他的都不行。此外，文件名只能是英文字母、数字和下划线的组合。

如果当前目录下没有hello.py这个文件，运行python hello.py就会报错：

C:\Users\IEUser>python hello.py

python: can't open file 'hello.py': [Errno 2] No such file or directory

报错的意思就是，无法打开hello.py这个文件，因为文件不存在。这个时候，就要检查一下当前目录下是否有这个文件了。如果hello.py存放在另外一个目录下，要首先用cd命令切换当前目录。

**直接运行py文件**

有同学问，能不能像.exe文件那样直接运行.py文件呢？在Windows上是不行的，但是，在Mac和Linux上是可以的，方法是在.py文件的第一行加上一个特殊的注释：

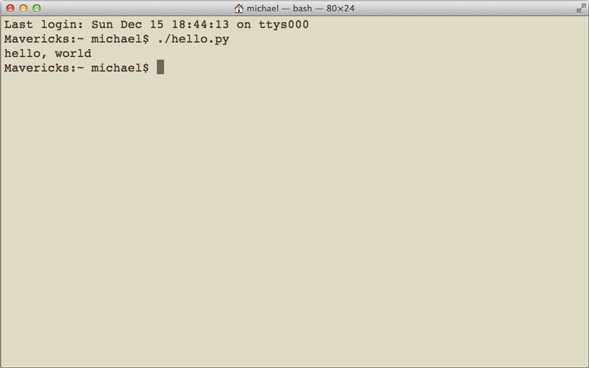
*#!/usr/bin/env python3*

**print**('hello, world')

然后，通过命令给hello.py以执行权限：

$ chmod a+x hello.py

就可以直接运行hello.py了，比如在Mac下运行：



注

#!/usr/bin/Python是告诉操作系统执行这个脚本的时候，调用/usr/bin下的python解释器；  
#!/usr/bin/env python这种用法是为了防止操作系统用户没有将python装在默认的/usr/bin路径里。当系统看到这一行的时候，首先会到env设置里查找python的安装路径，再调用对应路径下的解释器程序完成操作。  
#!/usr/bin/python相当于写死了python路径;  
#!/usr/bin/env python会去环境设置寻找python目录,**推荐这种写法**

小结

用文本编辑器写Python程序，然后保存为后缀为.py的文件，就可以用Python直接运行这个程序了。

Python的交互模式和直接运行.py文件有什么区别呢？

直接输入python进入交互模式，相当于启动了Python解释器，但是等待你一行一行地输入源代码，每输入一行就执行一行。

直接运行.py文件相当于启动了Python解释器，然后一次性把.py文件的源代码给执行了，你是没有机会以交互的方式输入源代码的。

用Python开发程序，完全可以一边在文本编辑器里写代码，一边开一个交互式命令窗口，在写代码的过程中，把部分代码粘到命令行去验证，事半功倍！前提是得有个27'的超大显示器！

## 第5课 输入和输出

输出

用print()在括号中加上字符串，就可以向屏幕上输出指定的文字。比如输出'hello, world'，用代码实现如下：

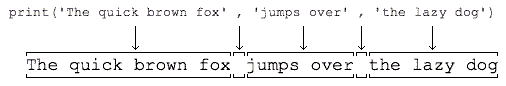
>>> print('hello, world')

print()函数也可以接受多个字符串，用逗号“,”隔开，就可以连成一串输出：

>>> print('The quick brown fox', 'jumps over', 'the lazy dog')

The quick brown fox jumps over the lazy dog

print()会依次打印每个字符串，遇到逗号“,”会输出一个空格，因此，输出的字符串是这样拼起来的：



print()也可以打印整数，或者计算结果：

>>> print(300)

300

>>> print(100 + 200)

300

因此，我们可以把计算100 + 200的结果打印得更漂亮一点：

>>> print('100 + 200 =', 100 + 200)

100 + 200 = 300

注意，对于100 + 200，Python解释器自动计算出结果300，但是，'100 + 200 ='是字符串而非数学公式，Python把它视为字符串，请自行解释上述打印结果。

输入

现在，你已经可以用print()输出你想要的结果了。但是，如果要让用户从电脑输入一些字符怎么办？Python提供了一个input()，可以让用户输入字符串，并存放到一个变量里。比如输入用户的名字：

>>> name = input()

Michael

当你输入name = input()并按下回车后，Python交互式命令行就在等待你的输入了。这时，你可以输入任意字符，然后按回车后完成输入。

输入完成后，不会有任何提示，Python交互式命令行又回到>>>状态了。那我们刚才输入的内容到哪去了？答案是存放到name变量里了。可以直接输入name查看变量内容：

>>> name

'Michael'

**什么是变量？**请回忆初中数学所学的代数基础知识：

设正方形的边长为a，则正方形的面积为a x a。把边长a看做一个变量，我们就可以根据a的值计算正方形的面积，比如：

若a=2，则面积为a x a = 2 x 2 = 4；

若a=3.5，则面积为a x a = 3.5 x 3.5 = 12.25。

在计算机程序中，变量不仅可以为整数或浮点数，还可以是字符串，因此，name作为一个变量就是一个字符串。

要打印出name变量的内容，除了直接写name然后按回车外，还可以用print()函数：（这时输出的变量是不带引号的）

>>> print(name)

Michael

有了输入和输出，我们就可以把上次打印'hello, world'的程序改成有点意义的程序了：

name = input()

**print**('hello,', name)

运行上面的程序，第一行代码会让用户输入任意字符作为自己的名字，然后存入name变量中；第二行代码会根据用户的名字向用户说hello，比如输入Michael：

C:\Workspace> python hello.py

Michael

hello, Michael

但是程序运行的时候，没有任何提示信息告诉用户：“嘿，赶紧输入你的名字”，这样显得很不友好。幸好，input()可以让你显示一个字符串来提示用户，于是我们把代码改成：

name = input('please enter your name: ')

**print**('hello,', name)

再次运行这个程序，你会发现，程序一运行，会首先打印出please enter your name:，这样，用户就可以根据提示，输入名字后，得到hello, xxx的输出：

C:\Workspace> python hello.py

please enter your name: Michael

hello, Michael

每次运行该程序，根据用户输入的不同，输出结果也会不同。

在命令行下，输入和输出就是这么简单。

小结

任何计算机程序都是为了执行一个特定的任务，有了输入，用户才能告诉计算机程序所需的信息，有了输出，程序运行后才能告诉用户任务的结果。

输入是Input，输出是Output，因此，我们把输入输出统称为Input/Output，或者简写为IO。

input()和print()是在命令行下面最基本的输入和输出，但是，用户也可以通过其他更高级的图形界面完成输入和输出，比如，在网页上的一个文本框输入自己的名字，点击“确定”后在网页上看到输出信息。

## 第6课 Python代码

Python是一种计算机编程语言。计算机编程语言和我们日常使用的自然语言有所不同，最大的区别就是，自然语言在不同的语境下有不同的理解，而计算机要根据编程语言执行任务，就必须保证编程语言写出的程序决不能有歧义，所以，任何一种编程语言都有自己的一套语法，编译器或者解释器就是负责把符合语法的程序代码转换成CPU能够执行的机器码，然后执行。Python也不例外。

Python的语法比较简单，采用缩进方式，写出来的代码就像下面的样子：

*# print absolute value of an integer:*

a = 100

**if** a >= 0:

**print**(a)

**else**:

**print**(-a)

以#开头的语句是注释，注释是给人看的，可以是任意内容，解释器会忽略掉注释。其他每一行都是一个语句，当语句以冒号:结尾时，缩进的语句视为代码块。

缩进有利有弊。好处是强迫你写出格式化的代码，但没有规定缩进是几个空格还是Tab。按照约定俗成的管理，应该始终坚持使用4个空格的缩进。

缩进的另一个好处是强迫你写出缩进较少的代码，你会倾向于把一段很长的代码拆分成若干函数，从而得到缩进较少的代码。

缩进的坏处就是“复制－粘贴”功能失效了，这是最坑爹的地方。当你重构代码时，粘贴过去的代码必须重新检查缩进是否正确。此外，IDE很难像格式化Java代码那样格式化Python代码。

最后，请务必注意，Python程序是大小写敏感的，如果写错了大小写，程序会报错。

小结

Python使用缩进来组织代码块，请务必遵守约定俗成的习惯，坚持使用4个空格的缩进。

在文本编辑器中，需要设置把Tab自动转换为4个空格，确保不混用Tab和空格。

## 第7课 数据类型和变量

### 数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在Python中，能够直接处理的数据类型有以下几种：

#### 整数

Python可以处理任意大小的整数，当然包括负整数，在程序中的表示方法和数学上的写法一模一样，例如：1，100，-8080，0，等等。

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用0x前缀和0-9，a-f表示，例如：0xff00，0xa5b4c3d2，等等。

#### 浮点数

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，1.23x109和12.3x108是完全相等的。浮点数可以用数学写法，如1.23，3.14，-9.01，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代，1.23x109就是1.23e9，或者12.3e8，0.000012可以写成1.2e-5，等等。

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的（除法难道也是精确的？是的！），而浮点数运算则可能会有四舍五入的误差。

#### 字符串

字符串是以单引号'或双引号"括起来的任意文本，比如'abc'，"xyz"等等。请注意，''或""本身只是一种表示方式，不是字符串的一部分，因此，字符串'abc'只有a，b，c这3个字符。如果'本身也是一个字符，那就可以用""括起来，比如"I'm OK"包含的字符是I，'，m，空格，O，K这6个字符。

如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如：

'I\'m \"OK\"!'

表示的字符串内容是：

I'm "OK"!

转义字符\可以转义很多字符，比如\n表示换行，\t表示制表符，字符\本身也要转义，所以\\表示的字符就是\，可以在Python的交互式命令行用print()打印字符串看看：

>>> **print**('I\'m ok.')

I'm ok.

>>> print('I\'m learning\nPython.')

I'm learning

Python.

>>> print('\\\n\\')

\

\

如果字符串里面有很多字符都需要转义，就需要加很多\，为了简化，Python还允许用r''表示''内部的字符串默认不转义，可以自己试试：

>>> print('\\\t\\')

\ \

>>> print(r'\\\t\\')

\\\t\\

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用'''...'''的格式表示多行内容，可以自己试试：

>>> print('''line1

... line2

... line3''')

line1

line2

line3

上面是在交互式命令行内输入，注意在输入多行内容时，提示符由>>>变为...，提示你可以接着上一行输入。如果写成程序，就是：

print('''line1

line2

line3''')

多行字符串'''...'''还可以在前面加上r使用，请自行测试。

#### 布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有True、False两种值，要么是True，要么是False，在Python中，可以直接用True、False表示布尔值（请注意大小写），也可以通过布尔运算计算出来：

>>> True

True

>>> False

False

>>> 3 > 2

True

>>> 3 > 5

False

布尔值可以用and、or和not运算。

and运算是与运算，只有所有都为True，and运算结果才是True：

>>> True **and** True

True

>>> True **and** False

False

>>> False **and** False

False

>>> 5 > 3 **and** 3 > 1

True

or运算是或运算，只要其中有一个为True，or运算结果就是True：

>>> True **or** True

True

>>> True **or** False

True

>>> False **or** False

False

>>> 5 > 3 **or** 1 > 3

True

not运算是非运算，它是一个单目运算符，把True变成False，False变成True：

>>> **not** True

False

>>> **not** False

True

>>> **not** 1 > 2

True

布尔值经常用在条件判断中，比如：

**if** age >= 18:

**print**('adult')

**else**:

**print**('teenager')

#### 空值

空值是Python里一个特殊的值，用None表示。None不能理解为0，因为0是有意义的，而None是一个特殊的空值。

此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到。

### 变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在程序中就是用一个变量名表示了，变量名必须是大小写英文、数字和\_的组合，且不能用数字开头，比如：

a = 1

变量a是一个整数。

t\_007 = 'T007'

变量t\_007是一个字符串。

Answer = **True**

变量Answer是一个布尔值True。

在Python中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，例如：

a = 123 *# a是整数*

**print**(a)

a = 'ABC' *# a变为字符串*

**print**(a)

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下（// 表示注释）：

**int** a = 123; *// a是整数类型变量*

a = "ABC"; *// 错误：不能把字符串赋给整型变量*

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

x = 10

x = x + 2

如果从数学上理解x = x + 2那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式x + 2，得到结果12，再赋给变量x。由于x之前的值是10，重新赋值后，x的值变成12。

最后，理解变量在计算机内存中的表示也非常重要。当我们写：

a = 'ABC'

时，Python解释器干了两件事情：

1. 在内存中创建了一个'ABC'的字符串；
2. 在内存中创建了一个名为a的变量，并把它指向'ABC'。

也可以把一个变量a赋值给另一个变量b，这个操作实际上是把变量b指向变量a所指向的数据，例如下面的代码：

a = 'ABC'

b = a

a = 'XYZ'

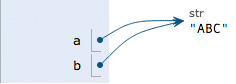
**print**(b)

最后一行打印出变量b的内容到底是'ABC'呢还是'XYZ'？如果从数学意义上理解，就会错误地得出b和a相同，也应该是'XYZ'，但实际上b的值是'ABC'，让我们一行一行地执行代码，就可以看到到底发生了什么事：

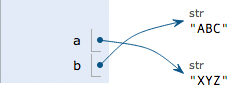
执行a = 'ABC'，解释器创建了字符串'ABC'和变量a，并把a指向'ABC'：

py-var-code-1

执行b = a，解释器创建了变量b，并把b指向a指向的字符串'ABC'：



执行a = 'XYZ'，解释器创建了字符串'XYZ'，并把a的指向改为'XYZ'，但b并没有更改：



所以，最后打印变量b的结果自然是'ABC'了。

### 常量

所谓常量就是不能变的变量，比如常用的数学常数π就是一个常量。在Python中，通常用全部大写的变量名表示常量：

PI = 3.14159265359

但事实上PI仍然是一个变量，Python根本没有任何机制保证PI不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量PI的值，也没人能拦住你。

最后解释一下整数的除法为什么也是精确的。在Python中，有两种除法，一种除法是/：

>>> 10 / 3

3.3333333333333335

/除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数：

>>> 9 / 3

3.0

还有一种除法是//，称为地板除，两个整数的除法仍然是整数：

>>> 10 *// 3*

3

你没有看错，整数的地板除//永远是整数，即使除不尽。要做精确的除法，使用/就可以。

因为//除法只取结果的整数部分，所以Python还提供一个余数运算，可以得到两个整数相除的余数：

>>> 10 % 3

1

无论整数做//除法还是取余数，结果永远是整数，所以，整数运算结果永远是精确的。

小结

Python支持多种数据类型，在计算机内部，可以把任何数据都看成一个“对象”，而变量就是在程序中用来指向这些数据对象的，对变量赋值就是把数据和变量给关联起来。

注意：Python的整数没有大小限制，而某些语言的整数根据其存储长度是有大小限制的，例如Java对32位整数的范围限制在-2147483648-2147483647。

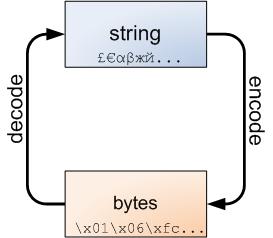
Python的浮点数也没有大小限制，但是超出一定范围就直接表示为inf（无限大）。

## 第8课 字符串和编码

### 字符编码

我们已经讲过了，字符串也是一种数据类型，但是，字符串比较特殊的是还有一个编码问题。

Python 3最重要的新特性大概要算是对文本和二进制数据作了更为清晰的区分。文本总是Unicode，由str类型表示，二进制数据则由bytes类型表示。Python 3不会以任意隐式的方式混用str和bytes，正是这使得两者的区分特别清晰。如下图所示：



因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用8个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是255（二进制11111111=十进制255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是65535，4个字节可以表示的最大整数是4294967295。

由于计算机是美国人发明的，因此，最早只有127个字母被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为ASCII编码，比如大写字母A的编码是65，小写字母z的编码是122。

但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和ASCII编码冲突，所以，中国制定了GB2312编码，用来把中文编进去。

你可以想得到的是，全世界有上百种语言，日本把日文编到Shift\_JIS里，韩国把韩文编到Euc-kr里，各国有各国的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。

因此，Unicode应运而生。Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。

Unicode标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持Unicode。

现在，捋一捋ASCII编码和Unicode编码的区别：ASCII编码是1个字节，而Unicode编码通常是2个字节。

字母A用ASCII编码是十进制的65，二进制的01000001；

字符0用ASCII编码是十进制的48，二进制的00110000，注意字符'0'和整数0是不同的；

汉字中已经超出了ASCII编码的范围，用Unicode编码是十进制的20013，二进制的01001110 00101101。

你可以猜测，如果把ASCII编码的A用Unicode编码，只需要在前面补0就可以，因此，A的Unicode编码是00000000 01000001。

新的问题又出现了：如果统一成Unicode编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

所以，本着节约的精神，又出现了把Unicode编码转化为“可变长编码”的UTF-8编码。UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间：

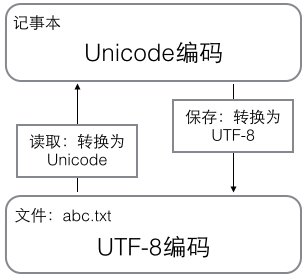
|  |  |  |  |
| --- | --- | --- | --- |
| **字符** | **ASCII** | **Unicode** | **UTF-8** |
| A | 01000001 | 00000000 01000001 | 01000001 |
| 中 | x | 01001110 00101101 | 11100100 10111000 10101101 |

从上面的表格还可以发现，UTF-8编码有一个额外的好处，就是ASCII编码实际上可以被看成是UTF-8编码的一部分，所以，大量只支持ASCII编码的历史遗留软件可以在UTF-8编码下继续工作。

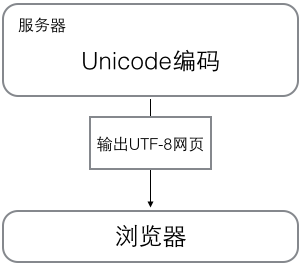
搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。

用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件：



浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器：



所以你看到很多网页的源码上会有类似<meta charset="UTF-8" />的信息，表示该网页正是用的UTF-8编码。

### Python的字符串

搞清楚了令人头疼的字符编码问题后，我们再来研究Python的字符串。

在最新的Python 3版本中，字符串是以Unicode编码的，也就是说，Python的字符串支持多语言，例如：

>>> print('包含中文的str')

包含中文的str

对于单个字符的编码，Python提供了ord()函数获取字符的整数表示，chr()函数把编码转换为对应的字符：

>>> ord('A')

65

>>> ord('中')

20013

>>> chr(66)

'B'

>>> chr(25991)

'文'

如果知道字符的整数编码，还可以用十六进制这么写str：

（\uxxxx这种格式是Unicode写法，表示一个字符，其中xxxx表示一个16进制数字）

>>> '\u4e2d\u6587'

'中文'

两种写法完全是等价的。

由于Python的字符串类型是str，在内存中以Unicode表示，一个字符对应若干个字节。如果要在网络上传输，或者保存到磁盘上，就需要把str变为以字节为单位的bytes。

Python对bytes类型的数据用带b前缀的单引号或双引号表示：

x = b'ABC'

要注意区分'ABC'和b'ABC'，前者是str，后者虽然内容显示得和前者一样，但bytes的每个字符都只占用一个字节。

以Unicode表示的str通过encode()方法可以编码为指定的bytes，例如：

>>> 'ABC'.encode('ascii')

b'ABC'

>>> '中文'.encode('utf-8')

b'\xe4\xb8\xad\xe6\x96\x87'

>>> '中文'.encode('ascii')

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)

纯英文的str可以用ASCII编码为bytes，内容是一样的，含有中文的str可以用UTF-8编码为bytes。含有中文的str无法用ASCII编码，因为中文编码的范围超过了ASCII编码的范围，Python会报错。

在bytes中，无法显示为ASCII字符的字节，用\x##显示。

反过来，如果我们从网络或磁盘上读取了字节流，那么读到的数据就是bytes。要把bytes变为str，就需要用decode()方法：

>>> b'ABC'.decode('ascii')

'ABC'

>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')

'中文'

要计算str包含多少个字符，可以用len()函数：

>>> len('ABC')

3

>>> len('中文')

2

len()函数计算的是str的字符数，如果换成bytes，len()函数就计算字节数：

>>> len(b'ABC')

3

>>> len(b'\xe4\xb8\xad\xe6\x96\x87')

6

>>> len('中文'.encode('utf-8'))

6

可见，1个中文字符经过UTF-8编码后通常会占用3个字节，而1个英文字符只占用1个字节。

在操作字符串时，我们经常遇到str和bytes的互相转换。为了避免乱码问题，应当始终坚持使用UTF-8编码对str和bytes进行转换。

由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

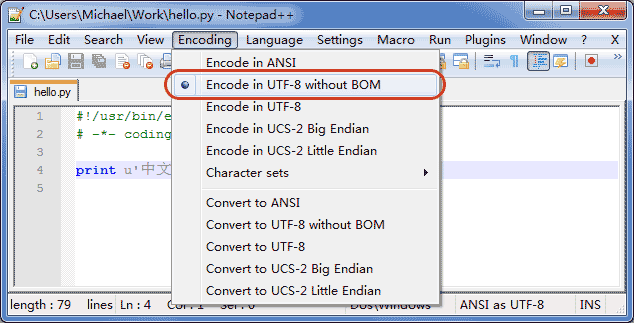
*#!/usr/bin/env python3*

*# -\*- coding: utf-8 -\*-*

第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

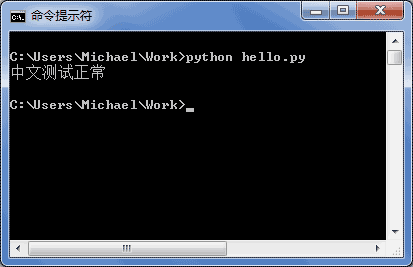
第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

申明了UTF-8编码并不意味着你的.py文件就是UTF-8编码的，必须并且要确保文本编辑器正在使用UTF-8 without BOM编码：



（上面截图用的是Python2.x，因此它如果要显示声明为unicode类型的话，需要在字符串前面加上'u'或者'U'，而在Python3中默认编码即为Unicode）

如果.py文件本身使用UTF-8编码，并且也申明了# -\*- coding: utf-8 -\*-，打开命令提示符测试就可以正常显示中文：



### 格式化

最后一个常见的问题是如何输出格式化的字符串。我们经常会输出类似'亲爱的xxx你好！你xx月的话费是xx，余额是xx'之类的字符串，而xxx的内容都是根据变量变化的，所以，需要一种简便的格式化字符串的方式。

在Python中，采用的格式化方式和C语言是一致的，用%实现，举例如下：

>>> 'Hello, %s' % 'world'

'Hello, world'

>>> 'Hi, %s, you have $%d.' % ('Michael', 1000000)

'Hi, Michael, you have $1000000.'

你可能猜到了，%运算符就是用来格式化字符串的。在字符串内部，%s表示用字符串替换，%d表示用整数替换，有几个%?占位符，后面就跟几个变量或者值，顺序要对应好。如果只有一个%?，括号可以省略。

常见的占位符有：

|  |  |
| --- | --- |
| %d | 整数 |
| %f | 浮点数 |
| %s | 字符串 |
| %x | 十六进制整数 |

其中，格式化整数和浮点数还可以指定是否补0和整数与小数的位数：

>>> '%2d-%02d' % (3, 1)

' 3-01'

>>> '%.2f' % 3.1415926

'3.14'

如果你不太确定应该用什么，%s永远起作用，它会把任何数据类型转换为字符串：

>>> 'Age: %s. Gender: %s' % (25, True)

'Age: 25. Gender: True'

有些时候，字符串里面的%是一个普通字符怎么办？这个时候就需要转义，用%%来表示一个%：

>>> 'growth rate: %d %%' % 7

'growth rate: 7 %'

### 小结

Python 3的字符串使用Unicode，直接支持多语言。

str和bytes互相转换时，需要指定编码。最常用的编码是UTF-8。Python当然也支持其他编码方式，比如把Unicode编码成GB2312：

>>> '中文'.encode('gb2312')

'\xd6\xd0\xce\xc4'

但这种方式纯属自找麻烦，如果没有特殊业务要求，请牢记仅使用UTF-8编码。

格式化字符串的时候，可以用Python的交互式命令行测试，方便快捷。

## 第9课使用list和tuple

list

Python内置的一种数据类型是列表：list。list是一种有序的集合，可以随时添加和删除其中的元素。

比如，列出班里所有同学的名字，就可以用一个list表示：

>>> classmates = ['Michael', 'Bob', 'Tracy']

>>> classmates

['Michael', 'Bob', 'Tracy']

变量classmates就是一个list。用len()函数可以获得list元素的个数：

>>> len(classmates)

3

用索引来访问list中每一个位置的元素，记得索引是从0开始的：

>>> classmates[0]

'Michael'

>>> classmates[1]

'Bob'

>>> classmates[2]

'Tracy'

>>> classmates[3]

Traceback (most recent call last):

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

IndexError: list index out of range

当索引超出了范围时，Python会报一个IndexError错误，所以，要确保索引不要越界，记得最后一个元素的索引是len(classmates) - 1。

如果要取最后一个元素，除了计算索引位置外，还可以用-1做索引，直接获取最后一个元素：

>>> classmates[-1]

'Tracy'

以此类推，可以获取倒数第2个、倒数第3个：

>>> classmates[-2]

'Bob'

>>> classmates[-3]

'Michael'

>>> classmates[-4]

Traceback (most recent call last):

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

IndexError: list index out of range

当然，倒数第4个就越界了。

list是一个可变的有序表，所以，可以往list中追加元素到末尾：

>>> classmates.append('Adam')

>>> classmates

['Michael', 'Bob', 'Tracy', 'Adam']

也可以把元素插入到指定的位置，比如索引号为1的位置：

>>> classmates.insert(1, 'Jack')

>>> classmates

['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']

要删除list末尾的元素，用pop()方法：

>>> classmates.pop()

'Adam'

>>> classmates

['Michael', 'Jack', 'Bob', 'Tracy']

要删除指定位置的元素，用pop(i)方法，其中i是索引位置：

>>> classmates.pop(1)

'Jack'

>>> classmates

['Michael', 'Bob', 'Tracy']

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

>>> classmates[1] = 'Sarah'

>>> classmates

['Michael', 'Sarah', 'Tracy']

list里面的元素的数据类型也可以不同，比如：

>>> L = ['Apple', 123, True]

list元素也可以是另一个list，比如：

>>> s = ['python', 'java', ['asp', 'php'], 'scheme']

>>> len(s)

4

要注意s只有4个元素，其中s[2]又是一个list，如果拆开写就更容易理解了：

>>> p = ['asp', 'php']

>>> s = ['python', 'java', p, 'scheme']

要拿到'php'可以写p[1]或者s[2][1]，因此s可以看成是一个二维数组，类似的还有三维、四维……数组，不过很少用到。

如果一个list中一个元素也没有，就是一个空的list，它的长度为0：

>>> L = []

>>> len(L)

0

tuple

另一种有序列表叫元组：tuple。tuple和list非常类似，但是tuple一旦初始化就不能修改，比如同样是列出同学的名字：

>>> classmates = ('Michael', 'Bob', 'Tracy')

现在，classmates这个tuple不能变了，它也没有append()，insert()这样的方法。其他获取元素的方法和list是一样的，你可以正常地使用classmates[0]，classmates[-1]，但不能赋值成另外的元素。

不可变的tuple有什么意义？因为tuple不可变，所以代码更安全。如果可能，能用tuple代替list就尽量用tuple。

tuple的陷阱：当你定义一个tuple时，在定义的时候，tuple的元素就必须被确定下来，比如：

>>> t = (1, 2)

>>> t

(1, 2)

如果要定义一个空的tuple，可以写成()：

>>> t = ()

>>> t

()

但是，要定义一个只有1个元素的tuple，如果你这么定义：

>>> t = (1)

>>> t

1

定义的不是tuple，是1这个数！这是因为括号()既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是1。

所以，只有1个元素的tuple定义时必须加一个逗号,，来消除歧义：

>>> t = (1,)

>>> t

(1,)

Python在显示只有1个元素的tuple时，也会加一个逗号,，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”tuple：

>>> t = ('a', 'b', ['A', 'B'])

>>> t[2][0] = 'X'

>>> t[2][1] = 'Y'

>>> t

('a', 'b', ['X', 'Y'])

这个tuple定义的时候有3个元素，分别是'a'，'b'和一个list。不是说tuple一旦定义后就不可变了吗？怎么后来又变了？

别急，我们先看看定义的时候tuple包含的3个元素：



当我们把list的元素'A'和'B'修改为'X'和'Y'后，tuple变为：



表面上看，tuple的元素确实变了，但其实变的不是tuple的元素，而是list的元素。tuple一开始指向的list并没有改成别的list，所以，tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向'a'，就不能改成指向'b'，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

理解了“指向不变”后，要创建一个内容也不变的tuple怎么做？那就必须保证tuple的每一个元素本身也不能变。

## 第10课 条件判断

### 条件判断

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，用if语句实现：

age = 20

**if** age >= 18:

**print**('your age is', age)

**print**('adult')

根据Python的缩进规则，如果if语句判断是True，就把缩进的两行print语句执行了，否则，什么也不做。

也可以给if添加一个else语句，意思是，如果if判断是False，不要执行if的内容，去把else执行了：

age = 3

**if** age >= 18:

**print**('your age is', age)

**print**('adult')

**else**:

**print**('your age is', age)

**print**('teenager')

注意不要少写了冒号:。

当然上面的判断是很粗略的，完全可以用elif做更细致的判断：

age = 3

**if** age >= 18:

print('adult')

**elif** age >= 6:

print('teenager')

**else**:

print('kid')

elif是else if的缩写，完全可以有多个elif，所以if语句的完整形式就是：

if <条件判断1>:

<执行1>

elif <条件判断2>:

<执行2>

elif <条件判断3>:

<执行3>

else:

<执行4>

if语句执行有个特点，它是从上往下判断，如果在某个判断上是True，把该判断对应的语句执行后，就忽略掉剩下的elif和else，所以，请测试并解释为什么下面的程序打印的是teenager：

age = 20

**if** age >= 6:

print('teenager')

**elif** age >= 18:

print('adult')

**else**:

print('kid')

if判断条件还可以简写，比如写：

**if** x:

**print**('True')

只要x是非零数值、非空字符串、非空list等，就判断为True，否则为False。

### 再议 input

最后看一个有问题的条件判断。很多同学会用input()读取用户的输入，这样可以自己输入，程序运行得更有意思：

birth = input('birth: ')

**if** birth < 2000:

**print**('00前')

**else**:

**print**('00后')

输入1982，结果报错：

Traceback (most recent call last):

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

TypeError: unorderable types: str() > int()

这是因为input()返回的数据类型是str，str不能直接和整数比较，必须先把str转换成整数。Python提供了int()函数来完成这件事情：

s = input('birth: ')

birth = int(s)

**if** birth < 2000:

**print**('00前')

**else**:

**print**('00后')

再次运行，就可以得到正确地结果。但是，如果输入abc呢？又会得到一个错误信息：

Traceback (most recent **call** **last**):

File "<stdin>", line 1, **in** <**module**>

ValueError: invalid literal **for** **int**() **with** base 10: 'abc'

原来int()函数发现一个字符串并不是合法的数字时就会报错，程序就退出了。

如何检查并捕获程序运行期的错误呢？后面的错误和调试会讲到。

## 第11课 循环

### 循环

要计算1+2+3，我们可以直接写表达式：

>>> 1 + 2 + 3

6

要计算1+2+3+...+10，勉强也能写出来。

但是，要计算1+2+3+...+10000，直接写表达式就不可能了。

为了让计算机能计算成千上万次的重复运算，我们就需要循环语句。

Python的循环有两种，一种是for...in循环，依次把list或tuple中的每个元素迭代出来，看例子：

names = ['Michael', 'Bob', 'Tracy']

**for** name **in** names:

print(name)

执行这段代码，会依次打印names的每一个元素：

Michael

Bob

Tracy

所以for x in ...循环就是把每个元素代入变量x，然后执行缩进块的语句。

再比如我们想计算1-10的整数之和，可以用一个sum变量做累加：

sum = 0

**for** x **in** [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:

sum = sum + x

print(sum)

如果要计算1-100的整数之和，从1写到100有点困难，幸好Python提供一个range()函数，可以生成一个整数序列，再通过list()函数可以转换为list。比如range(5)生成的序列是从0开始小于5的整数：

>>> list(range(5))

[0, 1, 2, 3, 4]

range(101)就可以生成0-100的整数序列，计算如下：

sum = 0

**for** x **in** range(101):

sum = sum + x

print(sum)

请自行运行上述代码，看看结果是不是当年高斯同学心算出的5050。

第二种循环是while循环，只要条件满足，就不断循环，条件不满足时退出循环。比如我们要计算100以内所有奇数之和，可以用while循环实现：

sum = 0

n = 99

**while** n > 0:

sum = sum + n

n = n - 2

**print**(sum)

在循环内部变量n不断自减，直到变为-1时，不再满足while条件，循环退出。

#### 第12课 使用dict和set

dict

Python内置了字典：dict的支持，dict全称dictionary，在其他语言中也称为map，使用键-值（key-value）存储，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用list实现，需要两个list：

names = ['Michael', 'Bob', 'Tracy']

scores = [95, 75, 85]

给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，list越长，耗时越长。

如果用dict实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用Python写一个dict如下：

>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}

>>> d['Michael']

95

为什么dict查找速度这么快？因为dict的实现原理和查字典是一样的。假设字典包含了1万个汉字，我们要查某一个字，一个办法是把字典从第一页往后翻，直到找到我们想要的字为止，这种方法就是在list中查找元素的方法，list越大，查找越慢。

第二种方法是先在字典的索引表里（比如部首表）查这个字对应的页码，然后直接翻到该页，找到这个字。无论找哪个字，这种查找速度都非常快，不会随着字典大小的增加而变慢。

dict就是第二种实现方式，给定一个名字，比如'Michael'，dict在内部就可以直接计算出Michael对应的存放成绩的“页码”，也就是95这个数字存放的内存地址，直接取出来，所以速度非常快。

你可以猜到，这种key-value存储方式，在放进去的时候，必须根据key算出value的存放位置，这样，取的时候才能根据key直接拿到value。

把数据放入dict的方法，除了初始化时指定外，还可以通过key放入：

>>> d['Adam'] = 67

>>> d['Adam']

67

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

>>> d['Jack'] = 90

>>> d['Jack']

90

>>> d['Jack'] = 88

>>> d['Jack']

88

如果key不存在，dict就会报错：

>>> d['Thomas']

Traceback (most recent call last):

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

KeyError: 'Thomas'

要避免key不存在的错误，有两种办法，一是通过in判断key是否存在：

>>> 'Thomas' **in** d

False

二是通过dict提供的get方法，如果key不存在，可以返回None，或者自己指定的value：

>>> d.get('Thomas')

>>> d.get('Thomas', -1)

-1

注意：返回None的时候Python的交互式命令行不显示结果。

要删除一个key，用pop(key)方法，对应的value也会从dict中删除：

>>> d.pop('Bob')

75

>>> d

{'Michael': 95, 'Tracy': 85}

请务必注意，dict内部存放的顺序和key放入的顺序是没有关系的。

和list比较，dict有以下几个特点：

1. 查找和插入的速度极快，不会随着key的增加而变慢；
2. 需要占用大量的内存，内存浪费多。

而list相反：

1. 查找和插入的时间随着元素的增加而增加；
2. 占用空间小，浪费内存很少。

所以，dict是用空间来换取时间的一种方法。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，需要牢记的第一条就是dict的key必须是**不可变对象**。

这是因为dict根据key来计算value的存储位置，如果每次计算相同的key得出的结果不同，那dict内部就完全混乱了。这个通过key计算位置的算法称为哈希算法（Hash）。

要保证hash的正确性，作为key的对象就不能变。在Python中，字符串、整数等都是不可变的，因此，可以放心地作为key。而list是可变的，就不能作为key：

>>> key = [1, 2, 3]

>>> d[key] = 'a list'

Traceback (most recent call last):

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

TypeError: unhashable type: 'list'

set

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。

要创建一个set，需要提供一个list作为输入集合：

>>> s = set([1, 2, 3])

>>> s

{1, 2, 3}

注意，传入的参数[1, 2, 3]是一个list，而显示的{1, 2, 3}只是告诉你这个set内部有1，2，3这3个元素，显示的顺序也不表示set是有序的。。

重复元素在set中自动被过滤：

>>> s = set([1, 1, 2, 2, 3, 3])

>>> s

{1, 2, 3}

通过add(key)方法可以添加元素到set中，可以重复添加，但不会有效果：

>>> s.add(4)

>>> s

{1, 2, 3, 4}

>>> s.add(4)

>>> s

{1, 2, 3, 4}

通过remove(key)方法可以删除元素：

>>> s.remove(4)

>>> s

{1, 2, 3}

set可以看成数学意义上的无序和无重复元素的集合，因此，两个set可以做数学意义上的交集、并集等操作：

>>> s1 = set([1, 2, 3])

>>> s2 = set([2, 3, 4])

>>> s1 & s2

{2, 3}

>>> s1 | s2

{1, 2, 3, 4}

set和dict的唯一区别仅在于没有存储对应的value，但是，set的原理和dict一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证set内部“不会有重复元素”。试试把list放入set，看看是否会报错。

再议不可变对象

上面我们讲了，str是不变对象，而list是可变对象。

对于可变对象，比如list，对list进行操作，list内部的内容是会变化的，比如：

>>> a = ['c', 'b', 'a']

>>> a.sort()

>>> a

['a', 'b', 'c']

而对于不可变对象，比如str，对str进行操作呢：

>>> a = 'abc'

>>> a.replace('a', 'A')

'Abc'

>>> a

'abc'

虽然字符串有个replace()方法，也确实变出了'Abc'，但变量a最后仍是'abc'，应该怎么理解呢？

我们先把代码改成下面这样：

>>> a = 'abc'

>>> b = a.replace('a', 'A')

>>> b

'Abc'

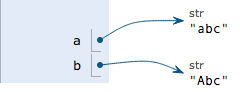
>>> a

'abc'

要始终牢记的是，a是变量，而'abc'才是字符串对象！有些时候，我们经常说，对象a的内容是'abc'，但其实是指，a本身是一个变量，它指向的对象的内容才是'abc'：

a-to-str

当我们调用a.replace('a', 'A')时，实际上调用方法replace是作用在字符串对象'abc'上的，而这个方法虽然名字叫replace，但却没有改变字符串'abc'的内容。相反，replace方法创建了一个新字符串'Abc'并返回，如果我们用变量b指向该新字符串，就容易理解了，变量a仍指向原有的字符串'abc'，但变量b却指向新字符串'Abc'了：



所以，对于不变对象来说，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，这样，就保证了不可变对象本身永远是不可变的。

小结

使用key-value存储结构的dict在Python中非常有用，选择不可变对象作为key很重要，最常用的key是字符串。

tuple虽然是不变对象，但试试把(1, 2, 3)和(1, [2, 3])放入dict或set中，并解释结果。

### 附 复制、浅拷贝与深拷贝

<http://www.cnblogs.com/wait123/archive/2011/10/10/2206580.html>

这两天又回头看了看python中的赋值、浅拷贝和深拷贝，以前看的时候总觉得他们之间有关系，但又理不清楚，今天总算有了点头绪，所以跟大家分享一下我的理解。

先说说赋值，其实python中的赋值其实是对象的引用（id相同），例如：

foo1=1.0

foo2=foo1

用操作符is判断时，你可以发现结果是true，是因为python是先创建了一个对象1.0，然后这个对象的引用又被赋值给了foo1和foo2（并没有再创建新的对象），但是如果是这样：

foo1=1.0

foo2=1.0

这时你会发现，这其实是创建了两个不同的对象，用内建函数id()可以发现，二者的身份不同；

其实python还有一个特例，例如：

a=1

b=1

你会发现id（a）=id（b），原因是python认为这些小整型是会经常用到的，所以python会缓存一部分小整型和字符串。

因此当创建一个对象，然后把它赋给另一个变量的时候，python并没有拷贝这个对象，而只是拷贝了这个对象的引用（相当于将对象的地址）。当你对一个对象赋值的时候(做为参数传递,或者做为返回值),Python和Java一样,总是传递原始对象的引用,而不是一个副本。

"""传递原始对象的引用,而不是一个副本"""

a = [1,2,3]

b = a

b.append(100)

print b #[1, 2, 3, 100]

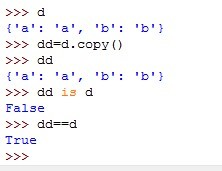
print a #[1, 2, 3, 100]

print id(a) #11530368

print id(b) #11530368

如果你想修改一个对象,而且想让原始的对象不受影响,那你就需要对象拷贝。

**容器类型**的可以通过三种方式实现浅拷贝，浅拷贝也是默认的拷贝类型：（1）完全切片操作；（2）利用工厂函数，比如list()等；（3）使用copy模块中的copy（）函数。**然而对于非容器类型没有拷贝这这一说**。在《python核心编程》一书中说道，“对一个对象进行浅拷贝其实是新创建了一个类型跟原对象一样，其内容是**原来对象元素**的引用，换句话说，这个拷贝的对象本身是新的，但是它的内容不是”。浅拷贝不是赋值，因为赋值的结果是他们的内存地址相同，但是浅拷贝的内存地址却不同，浅拷贝中每个元素的内存地址和原来对象元素的内存地址却是对应相同的。



下面我结合书上的例子以及我自己的想法来说明这句话是什么意思。

>>> jack = ['jack',['age',20]]

>>> tom = jack[:]

>>> anny = list(jack)

>>> jack

['jack', ['age', 20]]

>>> tom

['jack', ['age', 20]]

>>> anny

['jack', ['age', 20]]

>>> print id(jack),id(tom),id(anny)

13457088 18487376 18489136

[复制代码](javascript:void(0);)

可以看到**浅拷贝的内存地址是不同的**。接下来修改上面例子，对姓名和年级进行修改：

[复制代码](javascript:void(0);)

>>> tom[0]='tom'

>>> anny[0]='anny'

>>> print tom

['tom', ['age', 20]]

>>> print anny

['anny', ['age', 20]]

>>> anny[1][1]

20

>>> anny[1][1]= 18

>>> anny[1][1]

18

>>> print jack,tom,anny

['jack', ['age', 18]] ['tom', ['age', 18]] ['anny', ['age', 18]]

[复制代码](javascript:void(0);)

发现，虽然姓名都对号了，但是年龄却都变成了18.这是为什么呢？

我们看看它们元素的id

[复制代码](javascript:void(0);)

>>> [id(x) for x in jack]

[13463040, 13456608]

>>> [id(x) for x in tom]

[13463424, 13456608]

>>> [id(x) for x in anny]

[18501664, 13456608]

发现列表中姓名字符串元素id都不一样，但是年龄列表元素id却都相同。

  这是因为：python中字符串不可以修改，所以在为tom和anny重新命名的时候，会重新创建一个’tom’和’anny’对象，替换旧的’jack’对象。

这就说明了，**浅拷贝(shallow copy)拷贝了对象,但对于对象中的元素,依然使用引用。**

"""浅copy"""

import copy

aa = [1,2,3]

bb = copy.copy(aa)

print id(aa) #11533088

print id(bb) #12014776

bb[0] =100

print bb #[100, 2, 3]

print aa #[1,2,3]

#由于数字不可变，修改的时候会替换旧的对象

print [id(x) for x in bb] #[10247196, 10246388, 10246376]

print [id(y) for y in aa] #[10246400, 10246388, 10246376]

[复制代码](javascript:void(0);)

下面试试对象中可变元素：

lis = [['a'],[1,2],['z',23]]

copyLis = copy.copy(lis)

copyLis[1].append('bar')

print copyLis #[['a'], [1, 2, 'bar'], ['z', 23]]

print lis #[['a'], [1, 2, 'bar'], ['z', 23]]

**如果希望复制一个容器对象,以及它里面的所有元素(包含元素的子元素),使用copy.deepcopy,这个方法会消耗一些时间和空间,不过,如果你需要完全复制,这是唯一的方法.**

"""深copy"""

deepLis = copy.deepcopy(lis)

deepLis[1].append('foo')

print deepLis #[['a'], [1, 2,'foo'], ['z', 23]]

print lis #[['a'], [1, 2], ['z', 23]]

注意：

1、对于非容器类型(如数字、字符串、和其他‘原子’类型的对象)没有被拷贝一说。

2、**如果元组变量只包含原子类型对象，则不能深copy**。

1. 赋值是将一个对象的地址赋值给一个变量，让变量指向该地址（ 旧瓶装旧酒 ）。

2. 浅拷贝是在另一块地址中创建一个新的变量或容器，但是容器内的元素的地址均是源对象的元素的地址的拷贝。也就是说新的容器中的元素全部分别指向了旧的元素（ 新瓶装旧酒 ），但是新的容器的地址是新的。如果容器的某元素是可变对象，那么修改该元素的时候则不会创建新的对象（元素id不变）；如果容器的某元素是不可变对象，那么修改钙元素的时候则会创建新的对象（元素id改变）。

3. 深拷贝是在另一块地址中创建一个新的变量或容器，同时容器内的元素的地址也是新开辟的，仅仅是值相同而已，是完全的副本。也就是说（ 新瓶装新酒 ）。

<http://www.jb51.net/article/61902.htm>

**在python 中is和= = 的区别**

<http://zhidao.baidu.com/link?url=B1wwzdXNL_Su8B4JlTgR2GjNcFjcyu5LS_AFGBG2Jpl08j3hdGBKLfjGmVixjeOFZBkNrMxo0sJzAhGyJGt8Xq>

## 第12课 函数

我们知道圆的面积计算公式为：

S = πr2

当我们知道半径r的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

r1 = 12.34

r2 = 9.08

r3 = 73.1

s1 = 3.14 \* r1 \* r1

s2 = 3.14 \* r2 \* r2

s3 = 3.14 \* r3 \* r3

当代码出现有规律的重复的时候，你就需要当心了，每次写3.14 \* x \* x不仅很麻烦，而且，如果要把3.14改成3.14159265359的时候，得全部替换。

有了函数，我们就不再每次写s = 3.14 \* x \* x，而是写成更有意义的函数调用s = area\_of\_circle(x)，而函数area\_of\_circle本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，Python也不例外。Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

### 抽象

抽象是数学中非常常见的概念。举个例子：

计算数列的和，比如：1 + 2 + 3 + ... + 100，写起来十分不方便，于是数学家发明了求和符号∑，可以把1 + 2 + 3 + ... + 100记作：

100

∑n

n=1

这种抽象记法非常强大，因为我们看到 ∑ 就可以理解成求和，而不是还原成低级的加法运算。

而且，这种抽象记法是可扩展的，比如：

100

∑(n2+1)

n=1

还原成加法运算就变成了：

(1 x 1 + 1) + (2 x 2 + 1) + (3 x 3 + 1) + ... + (100 x 100 + 1)

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。

## 第13课 调用函数

Python内置了很多有用的函数，我们可以直接调用。

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数abs，只有一个参数。可以直接从Python的官方网站查看文档：

<http://docs.python.org/3/library/functions.html#abs>

也可以在交互式命令行通过help(abs)查看abs函数的帮助信息。

调用abs函数：

>>> abs(100)

100

>>> abs(-20)

20

>>> abs(12.34)

12.34

调用函数的时候，如果传入的参数数量不对，会报TypeError的错误，并且Python会明确地告诉你：abs()有且仅有1个参数，但给出了两个：

>>> abs(1, 2)

Traceback (most recent call last):

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

TypeError: abs() takes exactly one argument (2 given)

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报TypeError的错误，并且给出错误信息：str是错误的参数类型：

>>> abs('a')

Traceback (most recent call last):

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

TypeError: bad operand type for abs(): 'str'

而max函数max()可以接收任意多个参数，并返回最大的那个：

>>> max(1, 2)

2

>>> max(2, 3, 1, -5)

3

### 数据类型转换

Python内置的常用函数还包括数据类型转换函数，比如int()函数可以把其他数据类型转换为整数：

>>> int('123')

123

>>> int(12.34)

12

>>> float('12.34')

12.34

>>> str(1.23)

'1.23'

>>> str(100)

'100'

>>> bool(1)

True

>>> bool('')

False

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

>>> a = abs *# 变量a指向abs函数*

>>> a(-1) *# 所以也可以通过a调用abs函数*

1

## 第14课 定义函数

在Python中，定义一个函数要使用def语句，依次写出函数名、括号、括号中的参数和冒号:，然后，在缩进块中编写函数体，函数的返回值用return语句返回。

我们以自定义一个求绝对值的my\_abs函数为例：

**def** **my\_abs**(x):

**if** x >= 0:

**return** x

**else**:

**return** -x

请自行测试并调用my\_abs看看返回结果是否正确。

请注意，函数体内部的语句在执行时，一旦执行到return时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有return语句，函数执行完毕后也会返回结果，只是结果为None。

return None可以简写为return。

在Python交互环境中定义函数时，注意Python会出现...的提示。函数定义结束后需要按两次回车重新回到>>>提示符下。

如果你已经把my\_abs()的函数定义保存为abstest.py文件了，那么，可以在该文件的当前目录下启动Python解释器，用from abstest import my\_abs来导入my\_abs()函数，注意abstest是文件名（不含.py扩展名）。

import的用法在后续[模块](http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014318447437605e90206e261744c08630a836851f5183000)一节中会详细介绍。

### 空函数

如果想定义一个什么事也不做的空函数，可以用pass语句：

**def** **nop**():

**pass**

pass语句什么都不做，那有什么用？实际上pass可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个pass，让代码能运行起来。

pass还可以用在其他语句里，比如：

**if** age >= 18:

**pass**

缺少了pass，代码运行就会有语法错误。

### 参数检查

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出TypeError：

>>> my\_abs(1, 2)

Traceback (most recent call last):

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

TypeError: my\_abs() takes 1 positional argument but 2 were given

但是如果参数类型不对，Python解释器就无法帮我们检查。试试my\_abs和内置函数abs的差别：

>>> my\_abs('A')

Traceback (most recent call last):

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

File "<stdin>", line 2, in my\_abs

TypeError: unorderable types: str() >= int()

>>> abs('A')

Traceback (most recent call last):

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

TypeError: bad operand type for abs(): 'str'

当传入了不恰当的参数时，内置函数abs会检查出参数错误，而我们定义的my\_abs没有参数检查，会导致if语句出错，出错信息和abs不一样。所以，这个函数定义不够完善。

让我们修改一下my\_abs的定义，对参数类型做检查，只允许整数和浮点数类型的参数。数据类型检查可以用内置函数isinstance()实现：

**def** **my\_abs**(x):

**if** **not** isinstance(x, (int, float)):

**raise** TypeError('bad operand type')

**if** x >= 0:

**return** x

**else**:

**return** -x

添加了参数检查后，如果传入错误的参数类型，函数就可以抛出一个错误：

>>> my\_abs('A')

Traceback (most recent call last):

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

File "<stdin>", line 3, in my\_abs

TypeError: bad operand type

错误和异常处理将在后续讲到。

### 返回多个值

函数可以返回多个值吗？答案是肯定的。

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的新的坐标：

**import** math

**def** **move**(x, y, step, angle=0):

nx = x + step \* math.cos(angle)

ny = y - step \* math.sin(angle)

**return** nx, ny

import math语句表示导入math包，并允许后续代码引用math包里的sin、cos等函数。

然后，我们就可以同时获得返回值：

>>> x, y = move(100, 100, 60, math.pi / 6)

>>> print(x, y)

151.96152422706632 70.0

但其实这只是一种假象，Python函数返回的仍然是单一值：

>>> r = move(100, 100, 60, math.pi / 6)

>>> print(r)

(151.96152422706632, 70.0)

原来返回值是一个tuple！但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个tuple，但写起来更方便。

### 小结

定义函数时，需要确定函数名和参数个数；

如果有必要，可以先对参数的数据类型做检查；

函数体内部可以用return随时返回函数结果；

函数执行完毕也没有return语句时，自动return None。

函数可以同时返回多个值，但其实就是一个tuple。

## 第15课 函数的参数

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。

Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

### 位置参数

我们先写一个计算x2的函数：

**def** **power**(x):

**return** x \* x

对于power(x)函数，参数x就是一个位置参数。

当我们调用power函数时，必须传入有且仅有的一个参数x：

>>> power(5)

25

>>> power(15)

225

现在，如果我们要计算x3怎么办？可以再定义一个power3函数，但是如果要计算x4、x5……怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把power(x)修改为power(x, n)，用来计算xn，说干就干：

**def** **power**(x, n):

s = 1

**while** n > 0:

n = n - 1

s = s \* x

**return** s

对于这个修改后的power(x, n)函数，可以计算任意n次方：

>>> power(5, 2)

25

>>> power(5, 3)

125

修改后的power(x, n)函数有两个参数：x和n，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数x和n。

### 默认参数

新的power(x, n)函数定义没有问题，但是，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

>>> power(5)

Traceback (most recent call last):

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

TypeError: power() missing 1 required positional argument: 'n'

Python的错误信息很明确：调用函数power()缺少了一个位置参数n。

这个时候，默认参数就排上用场了。由于我们经常计算x2，所以，完全可以把第二个参数n的默认值设定为2：

**def** **power**(x, n=2):

s = 1

**while** n > 0:

n = n - 1

s = s \* x

**return** s

这样，当我们调用power(5)时，相当于调用power(5, 2)：

>>> power(5)

25

>>> power(5, 2)

25

而对于n > 2的其他情况，就必须明确地传入n，比如power(5, 3)。

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；（答：如果默认参数位于必选参数之前，则调用函数时就必须使用key=Value的形式，而不能使用直接送入Value的形式了，否则会产生歧义）

二是如何设置默认参数。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，我们写个一年级小学生注册的函数，需要传入name和gender两个参数：

**def** **enroll**(name, gender):

print('name:', name)

print('gender:', gender)

这样，调用enroll()函数只需要传入两个参数：

>>> enroll('Sarah', 'F')

name: Sarah

gender: F

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

**def** **enroll**(name, gender, age=6, city='Beijing'):

print('name:', name)

print('gender:', gender)

print('age:', age)

print('city:', city)

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

>>> enroll('Sarah', 'F')

name: Sarah

gender: F

age: 6

city: Beijing

只有与默认参数不符的学生才需要提供额外的信息：

enroll('Bob', 'M', 7)

enroll('Adam', 'M', city='Tianjin')

可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用enroll('Bob', 'M', 7)，意思是，除了name，gender这两个参数外，最后1个参数应用在参数age上，city参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用enroll('Adam', 'M', city='Tianjin')，意思是，city参数用传进去的值，其他默认参数继续使用默认值。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个END再返回：

**def** **add\_end**(L=[]):

L.append('END')

**return** L

当你正常调用时，结果似乎不错：

>>> add\_end([1, 2, 3])

[1, 2, 3, 'END']

>>> add\_end(['x', 'y', 'z'])

['x', 'y', 'z', 'END']

当你使用默认参数调用时，一开始结果也是对的：

>>> add\_end()

['END']

但是，再次调用add\_end()时，结果就不对了：

>>> add\_end()

['END', 'END']

>>> add\_end()

['END', 'END', 'END']

很多初学者很疑惑，默认参数是[]，但是函数似乎每次都“记住了”上次添加了'END'后的list。

原因解释如下：

Python函数在定义的时候，默认参数L的值就被计算出来了，即[]，因为默认参数L也是一个变量，它指向对象[]，每次调用该函数，如果改变了L的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的[]了。

所以，定义默认参数要牢记一点：**默认参数必须指向不变对象**！

要修改上面的例子，我们可以用None这个不变对象来实现：

**def** **add\_end**(L=None):

**if** L **is** None:

L = []

L.append('END')

**return** L

现在，无论调用多少次，都不会有问题：

>>> add\_end()

['END']

>>> add\_end()

['END']

为什么要设计str、None这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

### 可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

我们以数学题为例子，给定一组数字a，b，c……，请计算a2 + b2 + c2 + ……。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把a，b，c……作为一个list或tuple传进来，这样，函数可以定义如下：

**def** **calc**(numbers):

sum = 0

**for** n **in** numbers:

sum = sum + n \* n

**return** sum

但是调用的时候，需要先组装出一个list或tuple：

>>> calc([1, 2, 3])

14

>>> calc((1, 3, 5, 7))

84

如果利用可变参数，调用函数的方式可以简化成这样：

>>> calc(1, 2, 3)

14

>>> calc(1, 3, 5, 7)

84

所以，我们把函数的参数改为可变参数：

**def** **calc**(\*numbers):

sum = 0

**for** n **in** numbers:

sum = sum + n \* n

**return** sum

定义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个\*号。在函数内部，参数numbers接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

>>> calc(1, 2)

5

>>> calc()

0

如果已经有一个list或者tuple，要调用一个可变参数怎么办？可以这样做：

>>> nums = [1, 2, 3]

>>> calc(nums[0], nums[1], nums[2])

14

这种写法当然是可行的，问题是太繁琐，所以Python允许你在list或tuple前面加一个\*号，把list或tuple的元素变成可变参数传进去：

>>> nums = [1, 2, 3]

>>> calc(\*nums)

14

\*nums表示把nums这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

### 关键字参数

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。请看示例：

**def** **person**(name, age, \*\*kw):

print('name:', name, 'age:', age, 'other:', kw)

函数person除了必选参数name和age外，还接受关键字参数kw。在调用该函数时，可以只传入必选参数：

>>> person('Michael', 30)

name: Michael age: 30 other: {}

也可以传入任意个数的关键字参数：

>>> person('Bob', 35, city='Beijing')

name: Bob age: 35 other: {'city': 'Beijing'}

>>> person('Adam', 45, gender='M', job='Engineer')

name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}

关键字参数有什么用？它可以扩展函数的功能。比如，在person函数里，我们保证能接收到name和age这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

和可变参数类似，也可以先组装出一个dict，然后，把该dict转换为关键字参数传进去：

>>> extra = {'city': 'Beijing', 'job': 'Engineer'}

>>> person('Jack', 24, city=extra['city'], job=extra['job'])

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

当然，上面复杂的调用可以用简化的写法：

>>> extra = {'city': 'Beijing', 'job': 'Engineer'}

>>> person('Jack', 24, \*\*extra)

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

\*\*extra表示把extra这个dict的所有key-value用关键字参数传入到函数的\*\*kw参数，kw将获得一个dict，注意kw获得的dict是extra的一份拷贝，对kw的改动不会影响到函数外的extra。

### 命名关键字参数

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过kw检查。

仍以person()函数为例，我们希望检查是否有city和job参数：

**def** **person**(name, age, \*\*kw):

**if** 'city' **in** kw:

*# 有city参数*

**pass**

**if** 'job' **in** kw:

*# 有job参数*

**pass**

print('name:', name, 'age:', age, 'other:', kw)

但是调用者仍可以传入不受限制的关键字参数：

>>> person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123456)

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收city和job作为关键字参数。这种方式定义的函数如下：

**def** **person**(name, age, \*, city, job):

print(name, age, city, job)

和关键字参数\*\*kw不同，命名关键字参数需要一个特殊分隔符\*，\*后面的参数被视为命名关键字参数。

调用方式如下：

>>> person('Jack', 24, city='Beijing', job='Engineer')

Jack 24 Beijing Engineer

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符\*了：

**def** **person**(name, age, \*args, city, job):

print(name, age, args, city, job)

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

>>> person('Jack', 24, 'Beijing', 'Engineer')

Traceback (most recent call last):

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

TypeError: person() takes 2 positional arguments but 4 were given

由于调用时缺少参数名city和job，Python解释器把这4个参数均视为位置参数，但person()函数仅接受2个位置参数。

命名关键字参数可以有缺省值，从而简化调用：

**def** **person**(name, age, \*, city='Beijing', job):

print(name, age, city, job)

由于命名关键字参数city具有默认值，调用时，可不传入city参数：

>>> person('Jack', 24, job='Engineer')

Jack 24 Beijing Engineer

使用命名关键字参数时，要特别注意，如果没有可变参数，就必须加一个\*作为特殊分隔符。如果缺少\*，Python解释器将无法识别位置参数和命名关键字参数：

**def** **person**(name, age, city, job):

*# 缺少 \*，city和job被视为位置参数*

**pass**

### 参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数。

比如定义一个函数，包含上述若干种参数：

**def** **f1**(a, b, c=0, \*args, \*\*kw):

print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

**def** **f2**(a, b, c=0, \*, d, \*\*kw):

print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

>>> f1(1, 2)

a = 1 b = 2 c = 0 args = () kw = {}

>>> f1(1, 2, c=3)

a = 1 b = 2 c = 3 args = () kw = {}

>>> f1(1, 2, 3, 'a', 'b')

a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}

>>> f1(1, 2, 3, 'a', 'b', x=99)

a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}

>>> f2(1, 2, d=99, ext=None)

a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}

最神奇的是通过一个tuple和dict，你也可以调用上述函数：

>>> args = (1, 2, 3, 4)

>>> kw = {'d': 99, 'x': '#'}

>>> f1(\*args, \*\*kw)

a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}

>>> args = (1, 2, 3)

>>> kw = {'d': 88, 'x': '#'}

>>> f2(\*args, \*\*kw)

a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}

所以，对于任意函数，都可以通过类似func(\*args, \*\*kw)的形式调用它，无论它的参数是如何定义的。

### 小结

Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

默认参数一定要用不可变对象，如果是可变对象，程序运行时会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

\*args是可变参数，args接收的是一个tuple；

\*\*kw是关键字参数，kw接收的是一个dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：func(1, 2, 3)，又可以先组装list或tuple，再通过\*args传入：func(\*(1, 2, 3))；

关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装dict，再通过\*\*kw传入：func(\*\*{'a': 1, 'b': 2})。

使用\*args和\*\*kw是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

定义命名的关键字参数在没有可变参数的情况下不要忘了写分隔符\*，否则定义的将是位置参数。

## 第16课 递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘n! = 1 x 2 x 3 x ... x n，用函数fact(n)表示，可以看出：

fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n

所以，fact(n)可以表示为n x fact(n-1)，只有n=1时需要特殊处理。

于是，fact(n)用递归的方式写出来就是：

**def** **fact**(n):

**if** n==1:

**return** 1

**return** n \* fact(n - 1)

上面就是一个递归函数。可以试试：

>>> fact(1)

1

>>> fact(5)

120

>>> fact(100)

93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

如果我们计算fact(5)，可以根据函数定义看到计算过程如下：

===> fact(5)

===> 5 \* fact(4)

===> 5 \* (4 \* fact(3))

===> 5 \* (4 \* (3 \* fact(2)))

===> 5 \* (4 \* (3 \* (2 \* fact(1))))

===> 5 \* (4 \* (3 \* (2 \* 1)))

===> 5 \* (4 \* (3 \* 2))

===> 5 \* (4 \* 6)

===> 5 \* 24

===> 120

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过**栈（stack）**这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试fact(1000)：

>>> fact(1000)

Traceback (most recent call last):

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

File "<stdin>", line 4, in fact

...

File "<stdin>", line 4, in fact

RuntimeError: maximum recursion depth exceeded in comparison

解决递归调用栈溢出的方法是通过**尾递归**优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的fact(n)函数由于return n \* fact(n - 1)引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

**def** **fact**(n):

**return** fact\_iter(n, 1)

**def** **fact\_iter**(num, product):

**if** num == 1:

**return** product

**return** fact\_iter(num - 1, num \* product)

可以看到，return fact\_iter(num - 1, num \* product)仅返回递归函数本身，num - 1和num \* product在函数调用前就会被计算，不影响函数调用。

fact(5)对应的fact\_iter(5, 1)的调用如下：

===> fact\_iter(5, 1)

===> fact\_iter(4, 5)

===> fact\_iter(3, 20)

===> fact\_iter(2, 60)

===> fact\_iter(1, 120)

===> 120

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的fact(n)函数改成尾递归方式，也会导致栈溢出。

### 小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

## 第17课 高级特性

掌握了Python的数据类型、语句和函数，基本上就可以编写出很多有用的程序了。

比如构造一个1, 3, 5, 7, ..., 99的列表，可以通过循环实现：

L = []

n = 1

while n <= 99:

L.append(n)

n = n + 2

取list的前一半的元素，也可以通过循环实现。

但是在Python中，代码不是越多越好，而是越少越好。代码不是越复杂越好，而是越简单越好。

基于这一思想，我们来介绍Python中非常有用的高级特性，1行代码能实现的功能，决不写5行代码。请始终牢记，代码越少，开发效率越高。

## 第18课 切片

取一个list或tuple的部分元素是非常常见的操作。比如，一个list如下：

>>> L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']

取前3个元素，应该怎么做？

笨办法：

>>> [L[0], L[1], L[2]]

['Michael', 'Sarah', 'Tracy']

之所以是笨办法是因为扩展一下，取前N个元素就没辙了。

取前N个元素，也就是索引为0-(N-1)的元素，可以用循环：

>>> r = []

>>> n = 3

>>> **for** i **in** range(n):

... r.append(L[i])

...

>>> r

['Michael', 'Sarah', 'Tracy']

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python提供了切片（Slice）操作符，能大大简化这种操作。

对应上面的问题，取前3个元素，用一行代码就可以完成切片：

>>> L[0:3]

['Michael', 'Sarah', 'Tracy']

L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。

如果第一个索引是0，还可以省略：

>>> L[:3]

['Michael', 'Sarah', 'Tracy']

也可以从索引1开始，取出2个元素出来：

>>> L[1:3]

['Sarah', 'Tracy']

类似的，既然Python支持L[-1]取倒数第一个元素，那么它同样支持倒数切片，试试：

>>> L[-2:]

['Bob', 'Jack']

>>> L[-2:-1]

['Bob']

记住倒数第一个元素的索引是-1。

切片操作十分有用。我们先创建一个0-99的数列：

>>> L = list(range(100))

>>> L

[0, 1, 2, 3, ..., 99]

可以通过切片轻松取出某一段数列。比如前10个数：

>>> L[:10]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

后10个数：

>>> L[-10:]

[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]

前11-20个数：

>>> L[10:20]

[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

前10个数，每两个取一个：

>>> L[:10:2]

[0, 2, 4, 6, 8]

所有数，每5个取一个：

>>> L[::5]

[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]

甚至什么都不写，只写[:]就可以原样复制一个list：

>>> L[:]

[0, 1, 2, 3, ..., 99]

tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只是操作的结果仍是tuple：

>>> (0, 1, 2, 3, 4, 5)[:3]

(0, 1, 2)

字符串'xxx'也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

>>> 'ABCDEFG'[:3]

'ABC'

>>> 'ABCDEFG'[::2]

'ACEG'

在很多编程语言中，针对字符串提供了很多各种截取函数（例如，substring），其实目的就是对字符串切片。Python没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

### 小结

有了切片操作，很多地方循环就不再需要了。Python的切片非常灵活，一行代码就可以实现很多行循环才能完成的操作。

## 第19课 迭代

如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

在Python中，迭代是通过for ... in来完成的，而很多语言比如C或者Java，迭代list是通过下标完成的，比如Java代码：

for (i=0; i<list.length; i++) {

n = list[i];

}

可以看出，Python的for循环抽象程度要高于Java的for循环，因为Python的for循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。

list这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如dict就可以迭代：

>>> d = {'a': 1, 'b': 2, 'c': 3}

>>> **for** key **in** d:

... print(key)

...

a

c

b

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.values()，如果要同时迭代key和value，可以用for k, v in d.items()。

由于字符串也是可迭代对象，因此，也可以作用于for循环：

>>> **for** ch **in** 'ABC':

... print(ch)

...

A

B

C

所以，当我们使用for循环时，只要作用于一个可迭代对象，for循环就可以正常运行，而我们不太关心该对象究竟是list还是其他数据类型。

那么，如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：（collections模块在内置数据类型的基础上，提供了几个额外的数据类型）

>>> **from** collections **import** Iterable

>>> isinstance('abc', Iterable) *# str是否可迭代*

True

>>> isinstance([1,2,3], Iterable) *# list是否可迭代*

True

>>> isinstance(123, Iterable) *# 整数是否可迭代*

False

最后一个小问题，如果要对list实现类似Java那样的下标循环怎么办？Python内置的enumerate函数可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

>>> **for** i, value **in** enumerate(['A', 'B', 'C']):

... print(i, value)

...

0 A

1 B

2 C

上面的for循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码：

>>> **for** x, y **in** [(1, 1), (2, 4), (3, 9)]:

... print(x, y)

...

1 1

2 4

3 9

### 小结

任何可迭代对象都可以作用于for循环，包括我们自定义的数据类型，只要符合迭代条件，就可以使用for循环。

## 第20课 列表生成式

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

举个例子，要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]可以用list(range(1, 11))：

>>> list(range(1, 11))

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

但如果要生成[1x1, 2x2, 3x3, ..., 10x10]怎么做？方法一是循环：

>>> L = []

>>> **for** x **in** range(1, 11):

... L.append(x \* x)

...

>>> L

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的list：

>>> [x \* x **for** x **in** range(1, 11)]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

写列表生成式时，把要生成的元素x \* x放到前面，后面跟for循环，就可以把list创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方：

>>> [x \* x **for** x **in** range(1, 11) **if** x % 2 == 0]

[4, 16, 36, 64, 100]

还可以使用两层循环，可以生成全排列：

>>> [m + n **for** m **in** 'ABC' **for** n **in** 'XYZ']

['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']

三层和三层以上的循环就很少用到了。

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名，可以通过一行代码实现：

>>> **import** os *# 导入os模块，模块的概念后面讲到*

>>> [d **for** d **in** os.listdir('.')] *# os.listdir可以列出文件和目录*

['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop', 'Documents', 'Downloads', 'Library', 'Movies', 'Music', 'Pictures', 'Public', 'VirtualBox VMs', 'Workspace', 'XCode']

for循环其实可以同时使用两个甚至多个变量，比如dict的items()可以同时迭代key和value：

>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }

>>> **for** k, v **in** d.items():

... print(k, '=', v)

...

y = B

x = A

z = C

因此，列表生成式也可以使用两个变量来生成list：

>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }

>>> [k + '=' + v **for** k, v **in** d.items()]

['y=B', 'x=A', 'z=C']

最后把一个list中所有的字符串变成小写：

>>> L = ['Hello', 'World', 'IBM', 'Apple']

>>> [s.lower() **for** s **in** L]

['hello', 'world', 'ibm', 'apple']

## 第21课 生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

>>> L = [x \* x **for** x **in** range(10)]

>>> L

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x \* x **for** x **in** range(10))

>>> g

<generator object <genexpr> at 0x1022ef630>

创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator。

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过next()函数获得generator的下一个返回值：

>>> **next**(g)

0

>>> **next**(g)

1

>>> **next**(g)

4

>>> **next**(g)

9

>>> **next**(g)

16

>>> **next**(g)

25

>>> **next**(g)

36

>>> **next**(g)

49

>>> **next**(g)

64

>>> **next**(g)

81

>>> **next**(g)

Traceback (most recent call last):

File "<stdin>", line 1, **in** <**module**>

StopIteration

我们讲过，generator保存的是算法，每次调用next(g)，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用next(g)实在是太变态了，正确的方法是使用for循环，因为generator也是可迭代对象：

>>> g = (x \* x **for** x **in** range(10))

>>> **for** n **in** g:

... print(n)

...

0

1

4

9

16

25

36

49

64

81

所以，我们创建了一个generator后，基本上永远不会调用next()，而是通过for循环来迭代它，并且不需要关心StopIteration的错误。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

**def** **fib**(max):

n, a, b = 0, 0, 1

**while** n < max:

print(b)

a, b = b, a + b #赋值没有先后顺序，同时进行

n = n + 1

**return** 'done'

上面的函数可以输出斐波那契数列的前N个数：

>>> fib(6)

1

1

2

3

5

8

'done'

仔细观察，可以看出，fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把print(b)改为yield b就可以了：

**def** **fib**(max):

n, a, b = 0, 0, 1

**while** n < max:

**yield** b

a, b = b, a + b

n = n + 1

**return** 'done'

这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

>>> f = fib(6)

>>> f

<generator object fib at 0x104feaaa0>

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

举个简单的例子，定义一个generator，依次返回数字1，3，5：

**def** **odd**():

print('step 1')

**yield** 1

print('step 2')

**yield**(3)

print('step 3')

**yield**(5)

调用该generator时，首先要生成一个generator对象，然后用next()函数不断获得下一个返回值：

>>> o = odd()

>>> **next**(o)

step 1

1

>>> **next**(o)

step 2

3

>>> **next**(o)

step 3

5

>>> **next**(o)

Traceback (most recent call last):

File "<stdin>", line 1, **in** <**module**>

StopIteration

可以看到，odd不是普通函数，而是generator，在执行过程中，遇到yield就中断，下次又继续执行。执行3次yield后，已经没有yield可以执行了，所以，第4次调用next(o)就报错。

回到fib的例子，我们在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代：

>>> **for** n **in** fib(6):

... print(n)

...

1

1

2

3

5

8

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

>>> g = fib(6)

>>> **while** True:

... **try**:

... x = next(g)

... print('g:', x)

... **except** StopIteration **as** e:

... print('Generator return value:', e.value)

... **break**

...

g: 1

g: 1

g: 2

g: 3

g: 5

g: 8

Generator **return** value: done

关于如何捕获错误，后面的错误处理还会详细讲解。

### 小结

generator是非常强大的工具，在Python中，可以简单地把列表生成式改成generator，也可以通过函数实现复杂逻辑的generator。

要理解generator的工作原理，它是在for循环的过程中不断计算出下一个元素，并在适当的条件结束for循环。对于函数改成的generator来说，遇到return语句或者执行到函数体最后一行语句，就是结束generator的指令，for循环随之结束。

请注意区分普通函数和generator函数，普通函数调用直接返回结果：

>>> r = abs(6)

>>> r

6

generator函数的“调用”实际返回一个generator对象：

>>> g = fib(6)

>>> g

<generator object fib at 0x1022ef948>

## 第22课 迭代器

我们已经知道，可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list、tuple、dict、set、str等；

一类是generator，包括生成器和带yield的generator function。

这些可以直接作用于for循环的对象统称为***可迭代对象：Iterable****。*

可以使用isinstance()判断一个对象是否是Iterable对象：

>>> **from** collections **import** Iterable

>>> isinstance([], Iterable)

True

>>> isinstance({}, Iterable)

True

>>> isinstance('abc', Iterable)

True

>>> isinstance((x **for** x **in** range(10)), Iterable)

True

>>> isinstance(100, Iterable)

False

而生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

可以被next()函数调用并不断返回下一个值的对象称为***迭代器：Iterator***。

可以使用isinstance()判断一个对象是否是Iterator对象：

>>> **from** collections **import** Iterator

>>> isinstance((x **for** x **in** range(10)), Iterator)

True

>>> isinstance([], Iterator)

False

>>> isinstance({}, Iterator)

False

>>> isinstance('abc', Iterator)

False

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

>>> isinstance(iter([]), Iterator)

True

>>> isinstance(iter('abc'), Iterator)

True

你可能会问，为什么list、dict、str等数据类型不是Iterator？

这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

### 小结

凡是可作用于for循环的对象都是Iterable类型；

凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；

集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。

Python的for循环本质上就是通过不断调用next()函数实现的，例如：

**for** x **in** [1, 2, 3, 4, 5]:

**pass**

实际上完全等价于：

*# 首先获得Iterator对象:*

it = iter([1, 2, 3, 4, 5])

*# 循环:*

**while** True:

**try**:

*# 获得下一个值:*

x = next(it)

**except** StopIteration:

*# 遇到StopIteration就退出循环*

**break**

## 第23课 函数式编程

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

## 第24课 高阶函数

高阶函数英文叫Higher-order function。什么是高阶函数？我们以实际代码为例子，一步一步深入概念。

**变量可以指向函数**

以Python内置的求绝对值的函数abs()为例，调用该函数用以下代码：

>>> abs(-10)

10

但是，如果只写abs呢？

>>> abs

<built-**in** **function** abs>

可见，abs(-10)是函数调用，而abs是函数本身。

要获得函数调用结果，我们可以把结果赋值给变量：

>>> x = abs(-10)

>>> x

10

但是，如果把函数本身赋值给变量呢？

>>> f = abs

>>> f

<built-**in** **function** abs>

结论：函数本身也可以赋值给变量，即：变量可以指向函数。

如果一个变量指向了一个函数，那么，可否通过该变量来调用这个函数？用代码验证一下：

>>> f = abs

>>> f(-10)

10

成功！说明变量f现在已经指向了abs函数本身。直接调用abs()函数和调用变量f()完全相同。

**函数名也是变量**

那么函数名是什么呢？函数名其实就是指向函数的变量！对于abs()这个函数，完全可以把函数名abs看成变量，它指向一个可以计算绝对值的函数！

如果把abs指向其他对象，会有什么情况发生？

>>> abs = 10

>>> abs(-10)

Traceback (most recent **call** **last**):

File "<stdin>", line 1, **in** <**module**>

TypeError: 'int' object **is** **not** callable

把abs指向10后，就无法通过abs(-10)调用该函数了！因为abs这个变量已经不指向求绝对值函数而是指向一个整数10！

当然实际代码绝对不能这么写，这里是为了说明函数名也是变量。要恢复abs函数，请重启Python交互环境。

注：由于abs函数实际上是定义在\_\_builtins\_\_模块中的，所以要让修改abs变量的指向在其它模块也生效，要用\_\_builtins\_\_.abs = 10。

**传入函数**

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

**def** **add**(x, y, f):

**return** f(x) + f(y)

当我们调用add(-5, 6, abs)时，参数x，y和f分别接收-5，6和abs，根据函数定义，我们可以推导计算过程为：

x = -5

y = 6

f = abs

f(x) + f(y) ==> abs(-5) + abs(6) ==> 11

**return** 11

用代码验证一下：

>>> add(-5, 6, abs)

11

编写高阶函数，就是让函数的参数能够接收别的函数。

小结

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

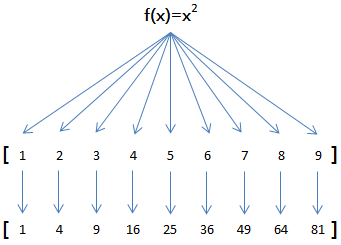
### map/reduce

Python内建了map()和reduce()函数。

如果你读过Google的那篇大名鼎鼎的论文“[MapReduce: Simplified Data Processing on Large Clusters](http://research.google.com/archive/mapreduce.html)”，你就能大概明白map/reduce的概念。

我们先看map。map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

举例说明，比如我们有一个函数f(x)=x2，要把这个函数作用在一个list [1, 2, 3, 4, 5, 6, 7, 8, 9]上，就可以用map()实现如下：



现在，我们用Python代码实现：

>>> **def** **f**(x):

... **return** x \* x

...

>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> list(r)

[1, 4, 9, 16, 25, 36, 49, 64, 81]

map()传入的第一个参数是f，即函数对象本身。由于结果r是一个Iterator，Iterator是惰性序列，因此通过list()函数让它把整个序列都计算出来并返回一个list。

你可能会想，不需要map()函数，写一个循环，也可以计算出结果：

L = []

**for** n **in** [1, 2, 3, 4, 5, 6, 7, 8, 9]:

L.append(f(n))

print(L)

的确可以，但是，从上面的循环代码，能一眼看明白“把f(x)作用在list的每一个元素并把结果生成一个新的list”吗？

所以，map()作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x2，还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串：

>>> list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

['1', '2', '3', '4', '5', '6', '7', '8', '9']

只需要一行代码。

再看reduce的用法。reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，其效果就是：

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

比方说对一个序列求和，就可以用reduce实现：

>>> **from** functools **import** reduce

>>> **def** **add**(x, y):

... **return** x + y

...

>>> reduce(add, [1, 3, 5, 7, 9])

25

当然求和运算可以直接用Python内建函数sum()，没必要动用reduce。

但是如果要把序列[1, 3, 5, 7, 9]变换成整数13579，reduce就可以派上用场：

>>> **from** functools **import** reduce

>>> **def** **fn**(x, y):

... **return** x \* 10 + y

...

>>> reduce(fn, [1, 3, 5, 7, 9])

13579

这个例子本身没多大用处，但是，如果考虑到字符串str也是一个序列，对上面的例子稍加改动，配合map()，我们就可以写出把str转换为int的函数：

>>> **from** functools **import** reduce

>>> **def** **fn**(x, y):

... **return** x \* 10 + y

...

>>> **def** **char2num**(s):

... **return** {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[s]

...

>>> reduce(fn, map(char2num, '13579'))

13579

整理成一个str2int的函数就是：

**from** functools **import** reduce

**def** **str2int**(s):

**def** **fn**(x, y):

**return** x \* 10 + y

**def** **char2num**(s):

**return** {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[s]

**return** reduce(fn, map(char2num, s))

还可以用lambda函数进一步简化成：

**from** functools **import** reduce

**def** **char2num**(s):

**return** {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[s]

**def** **str2int**(s):

**return** reduce(**lambda** x, y: x \* 10 + y, map(char2num, s))

也就是说，假设Python没有提供int()函数，你完全可以自己写一个把字符串转化为整数的函数，而且只需要几行代码！

lambda函数的用法在后面介绍。

### filter

Python内建的filter()函数用于过滤序列。

和map()类似，filter()也接收一个函数和一个序列。和map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

**def** **is\_odd**(n):

**return** n % 2 == 1

list(filter(is\_odd, [1, 2, 4, 5, 6, 9, 10, 15]))

*# 结果: [1, 5, 9, 15]*

把一个序列中的空字符串删掉，可以这么写：

**def** **not\_empty**(s):

**return** s **and** s.strip()

list(filter(not\_empty, ['A', '', 'B', None, 'C', ' ']))

*# 结果: ['A', 'B', 'C']*

可见用filter()这个高阶函数，关键在于正确实现一个“筛选”函数。

注意到filter()函数返回的是一个Iterator，也就是一个惰性序列，所以要强迫filter()完成计算结果，需要用list()函数获得所有结果并返回list。

**用filter求素数**

计算[素数](http://baike.baidu.com/view/10626.htm)的一个方法是[埃氏筛法](http://baike.baidu.com/view/3784258.htm)，它的算法理解起来非常简单：

首先，列出从2开始的所有自然数， 一个序列：

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...

取序列的第一个数2，它一定是素数，然后用2把序列的2的倍数筛掉：

3, , 5, , 7, , 9, , 11, , 13, , 15, , 17, , 19, , ...

取新序列的第一个数3，它一定是素数，然后用3把序列的3的倍数筛掉：

5, , 7, , , , 11, , 13, , , , 17, , 19, , ...

取新序列的第一个数5，然后用5把序列的5的倍数筛掉：

7, , , , 11, , 13, , , , 17, , 19, , ...

不断筛下去，就可以得到所有的素数。

用Python来实现这个算法，可以先构造一个从3开始的奇数序列：

**def** **\_odd\_iter**():

n = 1

**while** True:

n = n + 2

**yield** n

注意这是一个生成器，并且是一个无限序列。

然后定义一个筛选函数：

**def** **\_not\_divisible**(n):

**return** **lambda** x: x % n > 0

最后，定义一个生成器，不断返回下一个素数：

**def** **primes**():

**yield** 2

it = \_odd\_iter() *# 初始序列*

**while** True:

n = next(it) *# 返回序列的第一个数*

**yield** n

it = filter(\_not\_divisible(n), it) *# 构造新序列*

这个生成器先返回第一个素数2，然后，利用filter()不断产生筛选后的新的序列。

由于primes()也是一个无限序列，所以调用时需要设置一个退出循环的条件：

*# 打印1000以内的素数:*

**for** n **in** primes():

**if** n < 1000:

print(n)

**else**:

**break**

注意到Iterator是惰性计算的序列，所以我们可以用Python表示“全体自然数”，“全体素数”这样的序列，而代码非常简洁。

### sorted

**排序算法**

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个dict呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。

Python内置的sorted()函数就可以对list进行排序：

>>> sorted([36, 5, -12, 9, -21])

[-21, -12, 5, 9, 36]

此外，sorted()函数也是一个高阶函数，它还可以接收一个key函数来实现自定义的排序，例如按绝对值大小排序：

>>> sorted([36, 5, -12, 9, -21], key=abs)

[5, 9, -12, -21, 36]

key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序。对比原始的list和经过key=abs处理过的list：

list = [36, 5, -12, 9, -21]

keys = [36, 5, 12, 9, 21]

然后sorted()函数按照keys进行排序，并按照对应关系返回list相应的元素：

keys排序结果 => [5, 9, 12, 21, 36]

| | | | |

最终结果 => [5, 9, -12, -21, 36]

我们再看一个字符串排序的例子：

>>> sorted(['bob', 'about', 'Zoo', 'Credit'])

['Credit', 'Zoo', 'about', 'bob']

默认情况下，对字符串排序，是按照ASCII的大小比较的，由于'Z' < 'a'，结果，大写字母Z会排在小写字母a的前面。

现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能用一个key函数把字符串映射为忽略大小写排序即可。忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

这样，我们给sorted传入key函数，即可实现忽略大小写的排序：

>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)

['about', 'bob', 'Credit', 'Zoo']

要进行反向排序，不必改动key函数，可以传入第三个参数reverse=True：

>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, reverse=True)

['Zoo', 'Credit', 'bob', 'about']

从上述例子可以看出，高阶函数的抽象能力是非常强大的，而且，核心代码可以保持得非常简洁。

**小结**

sorted()也是一个高阶函数。用sorted()排序的关键在于实现一个映射函数。

## 第25课 返回函数

### 函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

**def** **calc\_sum**(\*args):

ax = 0

**for** n **in** args:

ax = ax + n

**return** ax

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数：

**def** **lazy\_sum**(\*args):

**def** **sum**():

ax = 0

**for** n **in** args:

ax = ax + n

**return** ax

**return** sum

当我们调用lazy\_sum()时，返回的并不是求和结果，而是求和函数：

>>> f = lazy\_sum(1, 3, 5, 7, 9)

>>> f

<**function** lazy\_sum.<locals>.sum at 0x101c6ed90>

调用函数f时，才真正计算求和的结果：

>>> f()

25

在这个例子中，我们在函数lazy\_sum中又定义了函数sum，并且，内部函数sum可以引用外部函数lazy\_sum的参数和局部变量，当lazy\_sum返回函数sum时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用lazy\_sum()时，每次调用都会返回一个新的函数，即使传入相同的参数：

>>> f1 = lazy\_sum(1, 3, 5, 7, 9)

>>> f2 = lazy\_sum(1, 3, 5, 7, 9)

>>> f1==f2

False

f1()和f2()的调用结果互不影响。

### 闭包

注意到返回的函数在其定义内部引用了局部变量args，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了f()才执行。我们来看一个例子：

**def** **count**():

fs = []

**for** i **in** range(1, 4):

**def** **f**():

**return** i\*i

fs.append(f)

**return** fs

f1, f2, f3 = count()#自注：这种赋值方式等号后面的必须是Iterable

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果是：

>>> f1()

9

>>> f2()

9

>>> f3()

9

全部都是9！原因就在于返回的函数引用了变量i，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量i已经变成了3，因此最终结果为9。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

**def** **count**():

**def** **f**(j):

**def** **g**():

**return** j\*j

**return** g

fs = []

**for** i **in** range(1, 4):

fs.append(f(i)) *# f(i)立刻被执行，因此i的当前值被传入f()*

**return** fs

再看看结果：

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

>>> f1()

1

>>> f2()

4

>>> f3()

9

缺点是代码较长，可利用lambda函数缩短代码。

### 小结

一个函数可以返回一个计算结果，也可以返回一个函数。

返回一个函数时，牢记该函数并未执行，返回函数中不要引用任何可能会变化的变量。

### 附 闭包

有很多不同的人都对闭包过进行了定义：

   1、是引用了**自由变量**的**函数**。这个函数通常被定义在另一个外部函数中，并且引用了外部函数中的变量。 — <<[wikipedia](http://zh.wikipedia.org/zh-cn/%E9%97%AD%E5%8C%85_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6))>>

   2、是一个可调用的**对象**，它记录了一些**信息**，这些信息来自于创建它的**作用域**。- <<Java编程思想>>

   3、是一个匿名的**代码块**，可以接受参数，并返回一个返回值，也可以引用和使用在它周围的，可见域中定义的**变量**。- [Groovy](http://docs.codehaus.org/pages/viewpage.action?pageId=167477326) ['ɡru:vi]

   4、是一个表达式，它具有自由变量及邦定这些变量的**上下文环境**。

   5、闭包允许你将一些行为**封装**，将它像一个**对象**一样传来递去，而且它依然能够访问到原来第一次声明时的**上下文**。

   6、是指拥有多个**变量**和绑定了这些变量的环境的**表达式**(通常是一个函数)，因而这些变量也是该表达式的一部分。

   7、闭包是可以包含**自由(未绑定)变量**的**代码块**；这些变量不是在这个代码块或者任何全局上下文中定义的，而是在定义代码块的环境中定义。

   取第4和5的解释。闭包是一个整体，包含了绑定变量和自由变量，绑定变量在函数对象被调用时，以参数的形式传入。而内部的自由变量则会持久化，在多次调用时，该变量的值会累积上一次的结果。

   下面这个例子就是用来说明这两个概念：

|  |  |
| --- | --- |
| 1  2  3  4 | def create\_multipliers():      return [lambda x : i \* x for i in range(5)]  for multiplier in create\_multipliers():      print multiplier(2) |

   这里定义了一个lambda表达式，create\_multipliers()返回的是一个函数对象序列。表面上看，函数中的每一个对象中的i值是不一样的，但是当实际输入x的值进行函数调用时，i的值都是一样的，都是最后一次i的值。因为延迟绑定的特性，即直到函数调用时，函数内部的执行内容才被确定，而此时i值在产生序列时，最终被赋值为4了，因此每一个函数调用中的上下文的i都是4，而不是预期的递增变化的。闭包和延迟绑定一起导致了这里的输出结果不是预期的0、2、4、6、8序列。

比较直观的解释：<https://www.zhihu.com/question/29483144>

## 第26课 匿名函数

当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，对匿名函数提供了有限支持。还是以map()函数为例，计算f(x)=x2时，除了定义一个f(x)的函数外，还可以直接传入匿名函数：

>>> list(map(lambda x: x \* x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

[1, 4, 9, 16, 25, 36, 49, 64, 81]

通过对比可以看出，匿名函数lambda x: x \* x实际上就是：

**def** **f**(x):

**return** x \* x

关键字lambda表示匿名函数，冒号前面的x表示函数参数，可以无参数。

匿名函数有个限制，就是只能有一个表达式，不用写return，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

>>> f = lambda x: x \* x

>>> f

<**function** <lambda> at 0x101c6ef28>

>>> f(5)

25

同样，也可以把匿名函数作为返回值返回，比如：

**def** **build**(x, y):

**return** **lambda**: x \* x + y \* y

### 小结

Python对匿名函数的支持有限，只有一些简单的情况下可以使用匿名函数。

## 第27课 装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

>>> **def** **now**():

... print('2015-3-25')

...

>>> f = now

>>> f()

2015-3-25

函数对象有一个\_\_name\_\_属性，可以拿到函数的名字：

>>> now.\_\_name\_\_

'now'

>>> f.\_\_name\_\_

'now'

现在，假设我们要增强now()函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改now()函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的decorator，可以定义如下：

**def** **log**(func):

**def** **wrapper**(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

**return** func(\*args, \*\*kw)

**return** wrapper

观察上面的log，因为它是一个decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的@语法，把decorator置于函数的定义处：

@log

**def** **now**():

print('2015-3-25')

调用now()函数，不仅会运行now()函数本身，还会在运行now()函数前打印一行日志：

>>> now()

**call** now():

2015-3-25

把@log放到now()函数的定义处，相当于执行了语句：

now = log(now)

由于log()是一个decorator，返回一个函数，所以，原来的now()函数仍然存在，只是现在同名的now变量指向了新的函数，于是调用now()将执行新函数，即在log()函数中返回的wrapper()函数。

wrapper()函数的参数定义是(\*args, \*\*kw)，因此，wrapper()函数可以接受任意参数的调用。在wrapper()函数内，首先打印日志，再紧接着调用原始函数。

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

**def** **log**(text):

**def** **decorator**(func):

**def** **wrapper**(\*args, \*\*kw):

print('%s %s():' % (text, func.\_\_name\_\_))

**return** func(\*args, \*\*kw)

**return** wrapper

**return** decorator

这个3层嵌套的decorator用法如下：

@log('execute')

**def** **now**():

print('2015-3-25')

执行结果如下：

>>> now()

execute now():

2015-3-25

和两层嵌套的decorator相比，3层嵌套的效果是这样的：

>>> now = log('execute')(now)

我们来剖析上面的语句，首先执行log('execute')，返回的是decorator函数，再调用返回的函数，参数是now函数，返回值最终是wrapper函数。

以上两种decorator的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有\_\_name\_\_等属性，但你去看经过decorator装饰之后的函数，它们的\_\_name\_\_已经从原来的'now'变成了'wrapper'：

>>> now.\_\_name\_\_

'wrapper'

因为返回的那个wrapper()函数名字就是'wrapper'，所以，需要把原始函数的\_\_name\_\_等属性复制到wrapper()函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写wrapper.\_\_name\_\_ = func.\_\_name\_\_这样的代码，Python内置的functools.wraps就是干这个事的，所以，一个完整的decorator的写法如下：

**import** functools

**def** **log**(func):

@functools.wraps(func)

**def** **wrapper**(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

**return** func(\*args, \*\*kw)

**return** wrapper

或者针对带参数的decorator：

**import** functools

**def** **log**(text):

**def** **decorator**(func):

@functools.wraps(func)

**def** **wrapper**(\*args, \*\*kw):

print('%s %s():' % (text, func.\_\_name\_\_))

**return** func(\*args, \*\*kw)

**return** wrapper

**return** decorator

import functools是导入functools模块。模块的概念稍候讲解。现在，只需记住在定义wrapper()的前面加上@functools.wraps(func)即可。

### 小结

在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持OOP的decorator外，直接从语法层次支持decorator。Python的decorator可以用函数实现，也可以用类实现。

decorator可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

请编写一个decorator，能在函数调用的前后打印出'begin call'和'end call'的日志。

再思考一下能否写出一个@log的decorator，使它既支持：

@log

**def** **f**():

**pass**

又支持：

@log('execute')

**def** **f**():

**pass**

### 附：更多实例

<http://www.cnblogs.com/vamei/archive/2013/02/16/2820212.html>

<http://blog.csdn.net/lainegates/article/details/8166764>

装饰器(decorator)是一种高级Python语法。装饰器可以对一个函数、方法或者类进行加工。在Python中，我们有多种方法对函数和类进行加工，比如在[Python闭包](http://www.cnblogs.com/vamei/archive/2012/12/15/2772451.html)中，我们见到函数对象作为某一个函数的返回结果。相对于其它方式，装饰器语法简单，代码可读性高。因此，装饰器在Python项目中有广泛的应用。

装饰器最早在Python 2.5中出现，它最初被用于加工函数和方法这样的可调用对象(callable object，这样的对象定义有\_\_call\_\_方法)。在Python 2.6以及之后的Python版本中，装饰器被进一步用于加工类。

### 装饰函数和方法

我们先定义两个简单的数学函数，一个用来计算平方和，一个用来计算平方差：

# get square sum

def square\_sum(a, b):

return a\*\*2 + b\*\*2

# get square diff

def square\_diff(a, b):

return a\*\*2 - b\*\*2

print(square\_sum(3, 4))

print(square\_diff(3, 4))

在拥有了基本的数学功能之后，我们可能想为函数增加其它的功能，比如打印输入。我们可以改写函数来实现这一点：

# modify: print input

# get square sum

def square\_sum(a, b):

print("intput:", a, b)

return a\*\*2 + b\*\*2

# get square diff

def square\_diff(a, b):

print("input", a, b)

return a\*\*2 - b\*\*2

print(square\_sum(3, 4))

print(square\_diff(3, 4))

我们修改了函数的定义，为函数增加了功能。

现在，我们使用装饰器来实现上述修改：

def decorator(F):

def new\_F(a, b):

print("input", a, b)

return F(a, b)

return new\_F

# get square sum

@decorator

def square\_sum(a, b):

return a\*\*2 + b\*\*2

# get square diff

@decorator

def square\_diff(a, b):

return a\*\*2 - b\*\*2

print(square\_sum(3, 4))

print(square\_diff(3, 4))

装饰器可以用def的形式定义，如上面代码中的decorator。装饰器接收一个可调用对象作为输入参数，并返回一个新的可调用对象。装饰器新建了一个可调用对象，也就是上面的new\_F。new\_F中，我们增加了打印的功能，并通过调用F(a, b)来实现原有函数的功能。

定义好装饰器后，我们就可以通过@语法使用了。在函数square\_sum和square\_diff定义之前调用@decorator，我们实际上将square\_sum或square\_diff传递给decorator，并将decorator返回的新的可调用对象赋给原来的函数名(square\_sum或square\_diff)。 所以，当我们调用square\_sum(3, 4)的时候，就相当于：

square\_sum = decorator(square\_sum)

square\_sum(3, 4)

我们知道，Python中的变量名和对象是分离的。变量名可以指向任意一个对象。从本质上，装饰器起到的就是这样一个重新指向变量名的作用(name binding)，让同一个变量名指向一个新返回的可调用对象，从而达到修改可调用对象的目的。

与加工函数类似，我们可以使用装饰器加工类的方法。

如果我们有其他的类似函数，我们可以继续调用decorator来修饰函数，而不用重复修改函数或者增加新的封装。这样，我们就提高了程序的可重复利用性，并增加了程序的可读性。

### 含参的装饰器

在上面的装饰器调用中，比如@decorator，该装饰器默认它后面的函数是唯一的参数。装饰器的语法允许我们调用decorator时，提供其它参数，比如@decorator(a)。这样，就为装饰器的编写和使用提供了更大的灵活性。

# a new wrapper layer

def pre\_str(pre=''):

# old decorator

def decorator(F):

def new\_F(a, b):

print(pre + "input", a, b)

return F(a, b)

return new\_F

return decorator

# get square sum

@pre\_str('^\_^')

def square\_sum(a, b):

return a\*\*2 + b\*\*2

# get square diff

@pre\_str('T\_T')

def square\_diff(a, b):

return a\*\*2 - b\*\*2

print(square\_sum(3, 4))

print(square\_diff(3, 4))

上面的pre\_str是允许参数的装饰器。它实际上是对原有装饰器的一个函数封装，并返回一个装饰器。我们可以将它理解为一个含有环境参量的[闭包](http://www.cnblogs.com/vamei/archive/2012/12/15/2772451.html)。当我们使用@pre\_str('^\_^')调用的时候，Python能够发现这一层的封装，并把参数传递到装饰器的环境中。该调用相当于:

square\_sum = pre\_str('^\_^') (square\_sum)

### 装饰类

在上面的例子中，装饰器接收一个函数，并返回一个函数，从而起到加工函数的效果。在Python 2.6以后，装饰器被拓展到类。一个装饰器可以接收一个类，并返回一个类，从而起到加工类的效果。

def decorator(aClass):

class newClass:

def \_\_init\_\_(self, age):

self.total\_display = 0

self.wrapped = aClass(age)

def display(self):

self.total\_display += 1

print("total display", self.total\_display)

self.wrapped.display()

return newClass

@decorator

class Bird:

def \_\_init\_\_(self, age):

self.age = age

def display(self):

print("My age is",self.age)

eagleLord = Bird(5)

for i in range(3):

eagleLord.display()

在decorator中，我们返回了一个新类newClass。在新类中，我们记录了原来类生成的对象（self.wrapped），并附加了新的属性total\_display，用于记录调用display的次数。我们也同时更改了display方法。

通过修改，我们的Bird类可以显示调用display的次数了。

### 总结

装饰器的核心作用是name binding。这种语法是Python多编程范式的又一个体现。大部分Python用户都不怎么需要定义装饰器，但有可能会使用装饰器。鉴于装饰器在Python项目中的广泛使用，了解这一语法是非常有益的。

**实例（1）：**

def spamrun(fn):  
    def sayspam(\*args):  
        print "spam,spam,spam"  
    return sayspam

@spamrun  
def useful(a,b):  
    print a\*\*2+b\*\*2  
     
useful(3,4)

**结果：**

spam，spam，spam

**实例（2）：**

def spamrun(fn):  
        print "spam,spam,spam"

@spamrun  
def useful(a,b):  
    print a\*\*2+b\*\*2

**结果：**

spam，spam，spam

**实例（3）：**

def spamrun(fn):  
    def sayspam(\*args):  
        print "spam,spam,spam"  
    return sayspam

@spamrun  
def useful(a,b):  
    print a\*\*2+b\*\*2  
     
useful(3,4)

**结果：**

spam，spam，spam

**实例（4）：**

def addspam(fn):  
    def new(\*args):  
        print "spam,spam,spam"  
        return fn(\*args)  
    return new

@addspam  
def useful(a,b):  
    print a\*\*2+b\*\*2  
useful(4,3)

**结果：**

spam,spam,spam  
25

**追加**

**实例**

def decorator(fn):  
    def test(\*args):  
        print "My god!"\*3  
        return fn(\*args)  
    return test

@decorator  
def other(a,b):  
    print a\*\*2+b\*\*2

if \_\_name\_\_=="\_\_main\_\_":  
    other(4,3)  
    other(3,4)  
     
结果：

My god!My god!My god!  
25  
My god!My god!My god!  
25  
注释掉//print return fn(\*args)

结果是：

My god!My god!My god!  
My god!My god!My god!  
  
要想使other函数能正常运行，必须加返回值，@decorator是一个statement,会将other函数当作参数传入来执行test方法

## 第28课 偏函数

Python的functools模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下：

int()函数可以把字符串转换为整数，当仅传入字符串时，int()函数默认按十进制转换：

>>> **int**('12345')

12345

但int()函数还提供额外的base参数，默认值为10。如果传入base参数，就可以做N进制的转换：

>>> **int**('12345', base=8)

5349

>>> **int**('12345', 16)

74565

假设要转换大量的二进制字符串，每次都传入int(x, base=2)非常麻烦，于是，我们想到，可以定义一个int2()的函数，默认把base=2传进去：

**def** **int2**(x, base=2):

**return** int(x, base)

这样，我们转换二进制就非常方便了：

>>> int2('1000000')

64

>>> int2('1010101')

85

functools.partial就是帮助我们创建一个偏函数的，不需要我们自己定义int2()，可以直接使用下面的代码创建一个新的函数int2：

>>> **import** functools

>>> int2 = functools.partial(int, base=2)

>>> int2('1000000')

64

>>> int2('1010101')

85

所以，简单总结functools.partial的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的int2函数，仅仅是把base参数重新设定默认值为2，但也可以在函数调用时传入其他值：

>>> int2('1000000', base=10)

1000000

最后，创建偏函数时，实际上可以接收函数对象、\*args和\*\*kw这3个参数，当传入：

int2 = functools.partial(int, base=2)

实际上固定了int()函数的关键字参数base，也就是：

int2('10010')

相当于：

kw = { 'base': 2 }

**int**('10010', \*\*kw)

当传入：

max2 = functools.partial(max, 10)

实际上会把10作为\*args的一部分自动加到左边，也就是：

max2(5, 6, 7)

相当于：

args = (10, 5, 6, 7)

max(\*args)

结果为10。

### 小结

当函数的参数个数太多，需要简化时，使用functools.partial可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。

## 第29课 模块

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Python中，一个.py文件就称之为一个模块（Module）。

使用模块有什么好处？

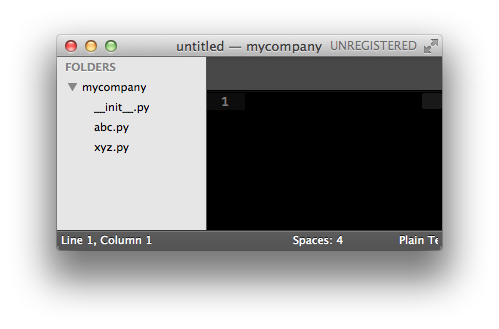
最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。点[这里](http://docs.python.org/3/library/functions.html)查看Python的所有内置函数。

你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

举个例子，一个abc.py的文件就是一个名字叫abc的模块，一个xyz.py的文件就是一个名字叫xyz的模块。

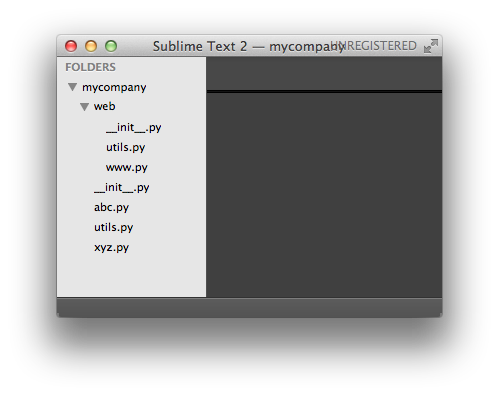
现在，假设我们的abc和xyz这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如mycompany，按照如下目录存放：



引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，abc.py模块的名字就变成了mycompany.abc，类似的，xyz.py的模块名变成了mycompany.xyz。

请注意，每一个包目录下面都会有一个\_\_init\_\_.py的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包。\_\_init\_\_.py可以是空文件，也可以有Python代码，因为\_\_init\_\_.py本身就是一个模块，而它的模块名就是mycompany。

类似的，可以有多级目录，组成多级层次的包结构。比如如下的目录结构：



文件www.py的模块名就是mycompany.web.www，两个文件utils.py的模块名分别是mycompany.utils和mycompany.web.utils。

自己创建模块时要注意命名，不能和Python自带的模块名称冲突。例如，系统自带了sys模块，自己的模块就不可命名为sys.py，否则将无法导入系统自带的sys模块。

mycompany.web也是一个模块，请指出该模块对应的.py文件。

### 附 \_\_init\_\_.py

python的每个模块的包中，都有一个\_\_init\_\_.py文件，有了这个文件，我们才能导入这个目录下的module。  
那么，\_\_init\_\_.py还有什么别的功能呢？  
其实，\_\_init\_\_.py里面还是可以有内容的，我们在导入一个包时，实际上导入了它的\_\_init\_\_.py文件。  
我们可以再\_\_init\_\_.py文件中再导入其他的包，或者模块。  
[python]  
import readers   
import writers   
import commands   
import users   
import meta   
import auth   
import admin

这样，当我们导入这个包的时候，\_\_init\_\_.py文件自动运行。帮我们导入了这么多个模块，我们就不需要将所有的import语句写在一个文件里了，也可以减少代码量。  
不需要一个个去导入module了。  
\_\_init\_\_.py 中还有一个重要的变量，叫做 \_\_all\_\_。我们有时会使出一招“全部导入”，也就是这样：  
from PackageName import \*  
这时 import 就会把注册在包 \_\_init\_\_.py 文件中 \_\_all\_\_ 列表中的子模块和子包导入到当前作用域中来。比如：  
#文件 \_\_init\_\_.py

\_\_all\_\_ = ["Module1", "Module2", "subPackage1", "subPackage2"]

如：在一个包里有foo.py、\_\_init\_\_.py

#\_\_init\_\_.py

import os

import datetime

#foo.py

from \_\_init\_\_ import \*

print datetime.datetime.now()

则输出：2013-07-11 11:34:41.250000

### \_\_file\_\_与argv[0]

在python下，获取当前执行主脚本的方法有两个：sys.argv[0]和\_\_file\_\_。

获取主执行文件路径的最佳方法是用sys.argv[0]，它可能是一个相对路径，所以再取一下abspath是保险的做法，像这样：

**import** os,sys

dirname, filename = os.path.split(os.path.abspath(sys.argv[0]))

**print** "running from", dirname

**print** "file is", filename

\_\_file\_\_ 是用来获得模块所在的路径的，这可能得到的是一个相对路径，比如在脚本test.py中写入：

#!/usr/bin/env python  
print \_\_file\_\_

* 按相对路径./test.py来执行，则打印得到的是相对路径，
* 按绝对路径执行则得到的是绝对路径。
* 而按用户目录来执行（~/practice/test.py），则得到的也是绝对路径（~被展开）
* 所以为了得到绝对路径，我们需要 os.path.realpath(\_\_file\_\_)。

而在Python控制台下，直接使用print \_\_file\_\_是会导致  name ‘\_\_file\_\_’ is not defined错误的，因为这时没有在任何一个脚本下执行，自然没有 \_\_file\_\_的定义了。

## 第30课 使用模块

Python本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。

我们以内建的sys模块为例，编写一个hello的模块：

*#!/usr/bin/env python3*

*# -\*- coding: utf-8 -\*-*

' a test module '

\_\_author\_\_ = 'Michael Liao'

**import** sys

**def** **test**():

args = sys.argv

**if** len(args)==1:

print('Hello, world!')

**elif** len(args)==2:

print('Hello, %s!' % args[1])

**else**:

print('Too many arguments!')

**if** \_\_name\_\_=='\_\_main\_\_':

test()

第1行和第2行是标准注释，第1行注释可以让这个hello.py文件直接在Unix/Linux/Mac上运行，第2行注释表示.py文件本身使用标准UTF-8编码；

第4行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

第6行使用\_\_author\_\_变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

以上就是Python模块的标准文件模板，当然也可以全部删掉不写，但是，按标准办事肯定没错。

后面开始就是真正的代码部分。

你可能注意到了，使用sys模块的第一步，就是导入该模块：

**import** sys

导入sys模块后，我们就有了变量sys指向该模块，利用sys这个变量，就可以访问sys模块的所有功能。

sys模块有一个argv变量，用list存储了命令行的所有参数。argv至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行python3 hello.py获得的sys.argv就是['hello.py']；

运行python3 hello.py Michael获得的sys.argv就是['hello.py', 'Michael]。

最后，注意到这两行代码：

**if** \_\_name\_\_=='\_\_main\_\_':

test()

当我们在命令行运行hello模块文件时，Python解释器把一个特殊变量\_\_name\_\_置为\_\_main\_\_，而如果在其他地方导入该hello模块时，if判断将失败，因此，这种if测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

我们可以用命令行运行hello.py看看效果：

$ python3 hello.py

Hello, world!

$ python hello.py Michael

Hello, Michael!

如果启动Python交互环境，再导入hello模块：

$ python3

Python 3**.4.3** (v3**.4.3**:9b73f1c3e601, Feb 23 2015, 02:52:03)

[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>> import hello

>>>

导入时，没有打印Hello, word!，因为没有执行test()函数。

调用hello.test()时，才能打印出Hello, word!：

>>> hello.test()

Hello, world!

### 作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过\_前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：abc，x123，PI等；

类似\_\_xxx\_\_这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的\_\_author\_\_，\_\_name\_\_就是特殊变量，hello模块定义的文档注释也可以用特殊变量\_\_doc\_\_访问，我们自己的变量一般不要用这种变量名；

类似\_xxx和\_\_xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc等；

之所以我们说，private函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

private函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

**def** **\_private\_1**(name):

**return** 'Hello, %s' % name

**def** **\_private\_2**(name):

**return** 'Hi, %s' % name

**def** **greeting**(name):

**if** len(name) > 3:

**return** \_private\_1(name)

**else**:

**return** \_private\_2(name)

我们在模块里公开greeting()函数，而把内部逻辑用private函数隐藏起来了，这样，调用greeting()函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

### 附

from .YYY import XXX 这里 “ . ” 的意思是你XXX当前文件夹里的初始化文件也就是XXX所在的目录下的\_\_init\_\_.py文件。而这里面也是你XXX注册的地方。你可以打开文件目录看一下。  
  
" . "的意思有点类似命令行里面，cd ..是返回上一层目录的意思，同样，from ..YYY import XXX就是从上级目录里面导入XXX

## 第31课 安装第三方模块

在Python中，安装第三方模块，是通过包管理工具pip完成的。

如果你正在使用Mac或Linux，安装pip本身这个步骤就可以跳过了。

如果你正在使用Windows，请参考[安装Python](http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014316090478912dab2a3a9e8f4ed49d28854b292f85bb000)一节的内容，确保安装时勾选了pip和Add python.exe to Path。

在命令提示符窗口下尝试运行pip，如果Windows提示未找到命令，可以重新运行安装程序添加pip。

注意：Mac或Linux上有可能并存Python 3.x和Python 2.x，因此对应的pip命令是pip3。

现在，让我们来安装一个第三方库——Python Imaging Library，这是Python下非常强大的处理图像的工具库。不过，PIL目前只支持到Python 2.7，并且有年头没有更新了，因此，基于PIL的Pillow项目开发非常活跃，并且支持最新的Python 3。

一般来说，第三方库都会在Python官方的[pypi.python.org](https://pypi.python.org/)网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者pypi上搜索，比如Pillow的名称叫[Pillow](https://pypi.python.org/pypi/Pillow/)，因此，安装Pillow的命令就是：

pip install Pillow

耐心等待下载并安装后，就可以使用Pillow了。

有了Pillow，处理图片易如反掌。随便找个图片生成缩略图：

>>> **from** PIL **import** Image

>>> im = Image.open('test.png')

>>> print(im.format, im.size, im.mode)

PNG (400, 300) RGB

>>> im.thumbnail((200, 100))

>>> im.save('thumb.jpg', 'JPEG')

其他常用的第三方库还有MySQL的驱动：mysql-connector-python，用于科学计算的NumPy库：numpy，用于生成文本的模板工具Jinja2，等等。

### 模块搜索路径

当我们试图加载一个模块时，Python会在指定的路径下搜索对应的.py文件，如果找不到，就会报错：

>>> import mymodule

Traceback (most recent **call** **last**):

File "<stdin>", line 1, **in** <**module**>

ImportError: **No** **module** named mymodule

默认情况下，Python解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在sys模块的path变量中：

>>> **import** sys

>>> sys.path

['', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python34.zip', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/plat-darwin', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-packages']

如果我们要添加自己的搜索目录，有两种方法：

一是直接修改sys.path，添加要搜索的目录：

>>> **import** sys

>>> sys.path.append('/Users/michael/my\_py\_scripts')

这种方法是在运行时修改，运行结束后失效。

第二种方法是设置环境变量PYTHONPATH，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置Path环境变量类似。注意只需要添加你自己的搜索路径，Python自己本身的搜索路径不受影响。

## 第31课 面向对象编程

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个dict表示：

std1 = { 'name': 'Michael', 'score': 98 }

std2 = { 'name': 'Bob', 'score': 81 }

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

**def** **print\_score**(std):

print('%s: %s' % (std['name'], std['score']))

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是Student这种数据类型应该被视为一个对象，这个对象拥有name和score这两个属性（Property）。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个print\_score消息，让对象自己把自己的数据打印出来。

**class Student(object):**

**def** **\_\_init\_\_**(self, name, score):

self.name = name

self.score = score

**def** **print\_score**(self):

print('%s: %s' % (self.name, self.score))

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

bart = Student('Bart Simpson', 59)

lisa = Student('Lisa Simpson', 87)

bart.print\_score()

lisa.print\_score()

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student。

所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。

面向对象的抽象程度又比函数要高，因为一个Class既包含数据，又包含操作数据的方法。

### 小结

数据封装、继承和多态是面向对象的三大特点，我们后面会详细讲解。

## 第32课 类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，比如Student类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以Student类为例，在Python中，定义类是通过class关键字：

**class Student(object):**

**pass**

class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

定义好了Student类，就可以根据Student类创建出Student的实例，创建实例是通过类名+()实现的：

>>> bart = Student()

>>> bart

<\_\_main\_\_.Student object at 0x10a67a590>

>>> Student

<class '\_\_main\_\_.Student'>

可以看到，变量bart指向的就是一个Student的实例，后面的0x10a67a590是内存地址，每个object的地址都不一样，而Student本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例bart绑定一个name属性：

>>> bart.name = 'Bart Simpson'

>>> bart.name

'Bart Simpson'

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方法，在创建实例的时候，就把name，score等属性绑上去：

**class Student(object):**

**def** **\_\_init\_\_**(self, name, score):

self.name = name

self.score = score

注意到\_\_init\_\_方法的第一个参数永远是self，表示创建的实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。

有了\_\_init\_\_方法，在创建实例的时候，就不能传入空的参数了，必须传入与\_\_init\_\_方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去：

>>> bart = Student('Bart Simpson', 59)

>>> bart.name

'Bart Simpson'

>>> bart.score

59

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

### 数据封装

面向对象编程的一个重要特点就是数据封装。在上面的Student类中，每个实例就拥有各自的name和score这些数据。我们可以通过函数来访问这些数据，比如打印一个学生的成绩：

>>> **def** **print\_score**(std):

... print('%s: %s' % (std.name, std.score))

...

>>> print\_score(bart)

Bart Simpson: 59

但是，既然Student实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在Student类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法：

**class Student(object):**

**def** **\_\_init\_\_**(self, name, score):

self.name = name

self.score = score

**def** **print\_score**(self):

print('%s: %s' % (self.name, self.score))

要定义一个方法，除了第一个参数是self外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了self不用传递，其他参数正常传入：

>>> bart.print\_score()

Bart Simpson: 59

这样一来，我们从外部看Student类，就只需要知道，创建实例需要给出name和score，而如何打印，都是在Student类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给Student类增加新的方法，比如get\_grade：

**class Student(object):**

...

**def** **get\_grade**(self):

**if** self.score >= 90:

**return** 'A'

**elif** self.score >= 60:

**return** 'B'

**else**:

**return** 'C'

同样的，get\_grade方法可以直接在实例变量上调用，不需要知道内部实现细节：

>>> bart.get\_grade()

'C'

### 小结

类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；

方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；

通过在实例上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。

和静态语言不同，Python允许对实例变量绑定任何数据，也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同：

>>> bart = Student('Bart Simpson', 59)

>>> lisa = Student('Lisa Simpson', 87)

>>> bart.age = 8

>>> bart.age

8

>>> lisa.age

Traceback (most recent call last):

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

AttributeError: 'Student' object has no attribute 'age'

### 附 \_\_init\_\_()方法

**注意1**、\_\_init\_\_并不相当于C#中的构造函数，执行它的时候，**实例已构造**出来了。

|  |  |
| --- | --- |
| 1  2  3  4  5 | class A(object):      def \_\_init\_\_(self,name):          self.name=name      def getName(self):          return 'A '+self.name |

当我们执行

|  |  |
| --- | --- |
| 1 | a=A('hello') |

时，可以理解为

|  |  |
| --- | --- |
| 1  2 | a=object.\_\_new\_\_(A)  A.\_\_init\_\_(a,'hello') |

即\_\_init\_\_作用是初始化已实例化后的对象。

**注意2**、子类可以**不重写**\_\_init\_\_，实例化子类时，**会自动调用**超类中已定义的\_\_init\_\_

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class B(A):      def getName(self):          return 'B '+self.name    if \_\_name\_\_=='\_\_main\_\_':      b=B('hello')      print b.getName() |

但如果**重写了**\_\_init\_\_，实例化子类时，则**不会隐式的再去调用**超类中已定义的\_\_init\_\_

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class C(A):      def \_\_init\_\_(self):          pass      def getName(self):          return 'C '+self.name    if \_\_name\_\_=='\_\_main\_\_':      c=C()      print c.getName() |

则会报"AttributeError: 'C' object has no attribute 'name'”错误，所以**如果重写了\_\_init\_\_，为了能使用或扩展超类中的行为，最好显式的调用超类的\_\_init\_\_方法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class C(A):      def \_\_init\_\_(self,name):          super(C,self).\_\_init\_\_(name)      def getName(self):          return 'C '+self.name    if \_\_name\_\_=='\_\_main\_\_':      c=C('hello')      print c.getName() |

### 附 self

<http://python.jobbole.com/81921/#article-comment>

## 第33课 访问限制

在Class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的name、score属性：

>>> bart = Student('Bart Simpson', 98)

>>> bart.score

98

>>> bart.score = 59

>>> bart.score

59

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在Python中，实例的变量名如果以\_\_开头（不以\_\_结尾），就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把Student类改一改：

**class Student(object):**

**def** **\_\_init\_\_**(self, name, score):

self.\_\_name = name

self.\_\_score = score

**def** **print\_score**(self):

print('%s: %s' % (self.\_\_name, self.\_\_score))

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量.\_\_name和实例变量.\_\_score了：

>>> bart = Student('Bart Simpson', 98)

>>> bart.\_\_name

Traceback (most recent call last):

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

AttributeError: 'Student' object has no attribute '\_\_name'

这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

但是如果外部代码要获取name和score怎么办？可以给Student类增加get\_name和get\_score这样的方法：

**class Student(object):**

...

**def** **get\_name**(self):

**return** self.\_\_name

**def** **get\_score**(self):

**return** self.\_\_score

如果又要允许外部代码修改score怎么办？可以再给Student类增加set\_score方法：

**class Student(object):**

...

**def** **set\_score**(self, score):

self.\_\_score = score

你也许会问，原先那种直接通过bart.score = 59也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数：

**class Student(object):**

...

**def** **set\_score**(self, score):

**if** 0 <= score <= 100:

self.\_\_score = score

**else**:

**raise** ValueError('bad score')

需要注意的是，在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。

有些时候，你会看到以一个下划线开头的实例变量名，比如\_name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问\_\_name是因为Python解释器对外把\_\_name变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name来访问\_\_name变量：

>>> bart.\_Student\_\_name

'Bart Simpson'

但是强烈建议你不要这么干，因为不同版本的Python解释器可能会把\_\_name改成不同的变量名。

总的来说就是，Python本身没有任何机制阻止你干坏事，一切全靠自觉。

## 第34课 继承和多态

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

比如，我们已经编写了一个名为Animal的class，有一个run()方法可以直接打印：

**class Animal(object):**

**def** **run**(self):

print('Animal is running...')

当我们需要编写Dog和Cat类时，就可以直接从Animal类继承：

**class Dog(Animal):**

**pass**

**class Cat(Animal):**

**pass**

对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类。Cat和Dog类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于Animial实现了run()方法，因此，Dog和Cat作为它的子类，什么事也没干，就自动拥有了run()方法：

dog = Dog()

dog.run()

cat = Cat()

cat.run()

运行结果如下：

Animal **is** running...

Animal **is** running...

当然，也可以对子类增加一些方法，比如Dog类：

**class Dog(Animal):**

**def** **run**(self):

print('Dog is running...')

**def** **eat**(self):

print('Eating meat...')

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是Dog还是Cat，它们run()的时候，显示的都是Animal is running...，符合逻辑的做法是分别显示Dog is running...和Cat is running...，因此，对Dog和Cat类改进如下：

**class Dog(Animal):**

**def** **run**(self):

print('Dog is running...')

**class Cat(Animal):**

**def** **run**(self):

print('Cat is running...')

再次运行，结果如下：

Dog **is** running...

Cat **is** running...

当子类和父类都存在相同的run()方法时，我们说，子类的run()覆盖了父类的run()，在代码运行的时候，总是会调用子类的run()。这样，我们就获得了继承的另一个好处：多态。

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样：

a = list() # a是list类型

b = Animal() # b是Animal类型

c = Dog() # c是Dog类型

判断一个变量是否是某个类型可以用isinstance()判断：

>>> isinstance(a, list)

True

>>> isinstance(b, Animal)

True

>>> isinstance(c, Dog)

True

看来a、b、c确实对应着list、Animal、Dog这3种类型。

但是等等，试试：

>>> isinstance(c, Animal)

True

看来c不仅仅是Dog，c还是Animal！

不过仔细想想，这是有道理的，因为Dog是从Animal继承下来的，当我们创建了一个Dog的实例c时，我们认为c的数据类型是Dog没错，但c同时也是Animal也没错，Dog本来就是Animal的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

>>> b = Animal()

>>> isinstance(b, Dog)

False

Dog可以看成Animal，但Animal不可以看成Dog。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个Animal类型的变量：

**def** **run\_twice**(animal):

animal.run()

animal.run()

当我们传入Animal的实例时，run\_twice()就打印出：

>>> run\_twice(Animal())

Animal **is** running...

Animal **is** running...

当我们传入Dog的实例时，run\_twice()就打印出：

>>> run\_twice(Dog())

Dog **is** running...

Dog **is** running...

当我们传入Cat的实例时，run\_twice()就打印出：

>>> run\_twice(Cat())

Cat **is** running...

Cat **is** running...

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个Tortoise类型，也从Animal派生：

**class Tortoise(Animal):**

**def** **run**(self):

print('Tortoise is running slowly...')

当我们调用run\_twice()时，传入Tortoise的实例：

>>> run\_twice(Tortoise())

Tortoise **is** running slowly...

Tortoise **is** running slowly...

你会发现，新增一个Animal的子类，不必对run\_twice()做任何修改，实际上，任何依赖Animal作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

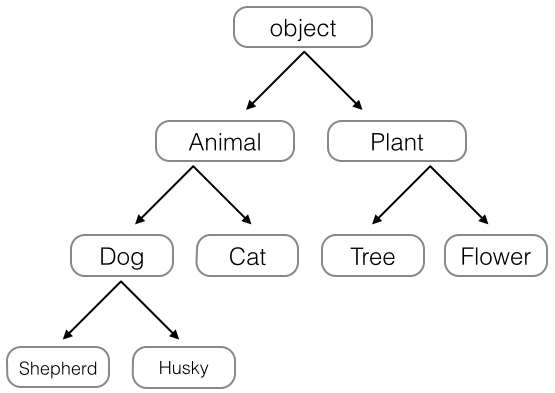
多态的好处就是，当我们需要传入Dog、Cat、Tortoise……时，我们只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise……都是Animal类型，然后，按照Animal类型进行操作即可。由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思：

对于一个变量，我们只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal、Dog、Cat还是Tortoise对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增Animal子类；

对修改封闭：不需要修改依赖Animal类型的run\_twice()等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



### 静态语言 vs 动态语言

对于静态语言（例如Java）来说，如果需要传入Animal类型，则传入的对象必须是Animal类型或者它的子类，否则，将无法调用run()方法。

对于Python这样的动态语言来说，则不一定需要传入Animal类型。我们只需要保证传入的对象有一个run()方法就可以了：

**class Timer(object):**

**def** **run**(self):

print('Start...')

这就是动态语言的“鸭子类型”，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

Python的“file-like object“就是一种鸭子类型。对真正的文件对象，它有一个read()方法，返回其内容。但是，许多对象，只要有read()方法，都被视为“file-like object“。许多函数接收的参数就是“file-like object“，你不一定要传入真正的文件对象，完全可以传入任何实现了read()方法的对象。

### 小结

继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写。

动态语言的鸭子类型特点决定了继承不像静态语言那样是必须的（只要需要传入的对象有特定的方法）。

多态：

<http://zhidao.baidu.com/link?url=7kvLCjR4hR4kD7FT6rHIuFikdzi71tgQhgwt5CIRzSfFk0ynw-Jtu6JKV8p2vhkb8nQzz3VoenKkNYwQh9jbHa>

## 第35课 获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

### 使用type()

首先，我们来判断对象类型，使用type()函数：

基本类型都可以用type()判断：

>>> type(123)

<**class 'int'>**

**>>> type('str')**

**<class 'str'>**

**>>> type(None)**

**<type(None) 'NoneType'>**

如果一个变量指向函数或者类，也可以用type()判断：

>>> type(abs)

<**class 'builtin\_function\_or\_method'>**

**>>> type(a)**

**<class '\_\_main\_\_.Animal'>**

但是type()函数返回的是什么类型呢？它返回对应的Class类型。如果我们要在if语句中判断，就需要比较两个变量的type类型是否相同：

>>> type(123)==type(456)

True

>>> type(123)==int

True

>>> type('abc')==type('123')

True

>>> type('abc')==str

True

>>> type('abc')==type(123)

False

判断基本数据类型可以直接写int，str等，但如果要判断一个对象是否是函数怎么办？可以使用types模块中定义的常量：

>>> **import** types

>>> **def** **fn**():

... **pass**

...

>>> type(fn)==types.FunctionType

True

>>> type(abs)==types.BuiltinFunctionType

True

>>> type(**lambda** x: x)==types.LambdaType

True

>>> type((x **for** x **in** range(10)))==types.GeneratorType

True

### 使用isinstance()

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用isinstance()函数。

我们回顾上次的例子，如果继承关系是：

object -> Animal -> Dog -> Husky

那么，isinstance()就可以告诉我们，一个对象是否是某种类型。先创建3种类型的对象：

>>> a = Animal()

>>> d = Dog()

>>> h = Husky()

然后，判断：

>>> isinstance(h, Husky)

True

没有问题，因为h变量指向的就是Husky对象。

再判断：

>>> isinstance(h, Dog)

True

h虽然自身是Husky类型，但由于Husky是从Dog继承下来的，所以，h也还是Dog类型。换句话说，isinstance()判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

因此，我们可以确信，h还是Animal类型：

>>> isinstance(h, Animal)

True

同理，实际类型是Dog的d也是Animal类型：

>>> isinstance(d, Dog) **and** isinstance(d, Animal)

True

但是，d不是Husky类型：

>>> isinstance(d, Husky)

False

能用type()判断的基本类型也可以用isinstance()判断：

>>> isinstance('a', str)

True

>>> isinstance(123, int)

True

>>> isinstance(b'a', bytes)

True

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是list或者tuple：

>>> isinstance([1, 2, 3], (**list**, tuple))

**True**

>>> isinstance((1, 2, 3), (**list**, tuple))

**True**

### 使用dir()

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：

>>> dir('ABC')

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', '**replace**', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

类似\_\_xxx\_\_的属性和方法在Python中都是有特殊用途的，比如\_\_len\_\_方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的\_\_len\_\_()方法，所以，下面的代码是等价的：

>>> len('ABC')

3

>>> 'ABC'.\_\_len\_\_()

3

我们自己写的类，如果也想用len(myObj)的话，就自己写一个\_\_len\_\_()方法：

>>> **class MyDog(object):**

... **def** **\_\_len\_\_**(self):

... **return** 100

...

>>> dog = MyDog()

>>> len(dog)

100

剩下的都是普通属性或方法，比如lower()返回小写的字符串：

>>> 'ABC'.lower()

'abc'

仅仅把属性和方法列出来是不够的，配合getattr()、setattr()以及hasattr()，我们可以直接操作一个对象的状态：

>>> **class MyObject(object):**

... **def** **\_\_init\_\_**(self):

... self.x = 9

... **def** **power**(self):

... **return** self.x \* self.x

...

>>> obj = MyObject()

紧接着，可以测试该对象的属性：

>>> hasattr(obj, 'x') *# 有属性'x'吗？*

True

>>> obj.x

9

>>> hasattr(obj, 'y') *# 有属性'y'吗？*

False

>>> setattr(obj, 'y', 19) *# 设置一个属性'y'*

>>> hasattr(obj, 'y') *# 有属性'y'吗？*

True

>>> getattr(obj, 'y') *# 获取属性'y'*

19

>>> obj.y *# 获取属性'y'*

19

如果试图获取不存在的属性，会抛出AttributeError的错误：

>>> getattr(obj, 'z') # 获取属性'z'

Traceback (most recent call last):

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

AttributeError: 'MyObject' object has no attribute 'z'

可以传入一个default参数，如果属性不存在，就返回默认值：

>>> getattr(obj, 'z', 404) *# 获取属性'z'，如果不存在，返回默认值404*

404

也可以获得对象的方法：

>>> hasattr(obj, 'power') *# 有属性'power'吗？*

True

>>> getattr(obj, 'power') *# 获取属性'power'*

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x10077a6a0>>

>>> fn = getattr(obj, 'power') *# 获取属性'power'并赋值到变量fn*

>>> fn *# fn指向obj.power*

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x10077a6a0>>

>>> fn() *# 调用fn()与调用obj.power()是一样的*

81

### 小结

通过内置的一系列函数，我们可以对任意一个Python对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以直接写：

sum = obj.x + obj.y

就不要写：

sum = getattr(obj, 'x') + getattr(obj, 'y')

一个正确的用法的例子如下：

**def** **readImage**(fp):

**if** hasattr(fp, 'read'):

**return** readData(fp)

**return** None

假设我们希望从文件流fp中读取图像，我们首先要判断该fp对象是否存在read方法，如果存在，则该对象是一个流，如果不存在，则无法读取。hasattr()就派上了用场。

请注意，在Python这类动态语言中，根据鸭子类型，有read()方法，不代表该fp对象就是一个文件流，它也可能是网络流，也可能是内存中的一个字节流，但只要read()方法返回的是有效的图像数据，就不影响读取图像的功能。

## 第36课 实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过self变量：

**class Student(object):**

**def** **\_\_init\_\_**(self, name):

self.name = name

s = Student('Bob')

s.score = 90

但是，如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有：

**class Student(object):**

name = 'Student'

当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下：

>>> **class Student(object):**

... name = 'Student'

...

>>> s = Student() *# 创建实例s*

>>> print(s.name) *# 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性*

Student

>>> print(Student.name) *# 打印类的name属性*

Student

>>> s.name = 'Michael' *# 给实例绑定name属性*

>>> print(s.name) *# 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性*

Michael

>>> print(Student.name) *# 但是类属性并未消失，用Student.name仍然可以访问*

Student

>>> **del** s.name *# 如果删除实例的name属性*

>>> print(s.name) *# 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了*

Student

从上面的例子可以看出，在编写程序的时候，千万不要把实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

## 第37课 面向对象高级编程

数据封装、继承和多态只是面向对象程序设计中最基础的3个概念。在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能。

我们会讨论多重继承、定制类、元类等概念。

## 第38课 使用\_\_slots\_\_

正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。先定义class：

**class Student(object):**

**pass**

然后，尝试给实例绑定一个属性：

>>> s = Student()

>>> s.name = 'Michael' *# 动态给实例绑定一个属性*

>>> print(s.name)

Michael

还可以尝试给实例绑定一个方法：

>>> **def** **set\_age**(self, age): *# 定义一个函数作为实例方法*

... self.age = age

...

>>> **from** types **import** MethodType

>>> s.set\_age = MethodType(set\_age, s) *# 给实例绑定一个方法*

>>> s.set\_age(25) *# 调用实例方法*

>>> s.age *# 测试结果*

25

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

>>> s2 = Student() # 创建新的实例

>>> s2.set\_age(25) # 尝试调用方法

Traceback (most recent call last):

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

AttributeError: 'Student' object has no attribute 'set\_age'

为了给所有实例都绑定方法，可以给class绑定方法：

>>> **def** **set\_score**(self, score):

... self.score = score

...

>>> Student.set\_score = set\_score

给class绑定方法后，所有实例均可调用：

>>> s.set\_score(100)

>>> s.score

100

>>> s2.set\_score(99)

>>> s2.score

99

通常情况下，上面的set\_score方法可以直接定义在class中，但动态绑定允许我们在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。

### 使用\_\_slots\_\_

但是，如果我们想要限制实例的属性怎么办？比如，只允许对Student实例添加name和age属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性：

**class Student(object):**

\_\_slots\_\_ = ('name', 'age') *# 用tuple定义允许绑定的属性名称*

然后，我们试试：

>>> s = Student() *# 创建新的实例*

>>> s.name = 'Michael' *# 绑定属性'name'*

>>> s.age = 25 *# 绑定属性'age'*

>>> s.score = 99 *# 绑定属性'score'*

Traceback (most recent call last):

File "<stdin>", line 1, **in** <module>

AttributeError: 'Student' object has no attribute 'score'

由于'score'没有被放到\_\_slots\_\_中，所以不能绑定score属性，试图绑定score将得到AttributeError的错误。

使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的：

>>> **class GraduateStudent(Student):**

... **pass**

...

>>> g = GraduateStudent()

>>> g.score = 9999

除非在子类中也定义\_\_slots\_\_，这样，子类实例允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。

## 第39课 使用@property属性\*

在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数，导致可以把成绩随便改：

s = Student()

s.score = 9999

这显然不合逻辑。为了限制score的范围，可以通过一个set\_score()方法来设置成绩，再通过一个get\_score()来获取成绩，这样，在set\_score()方法里，就可以检查参数：

**class Student(object):**

**def** **get\_score**(self):

**return** self.\_score

**def** **set\_score**(self, value):

**if** **not** isinstance(value, int):

**raise** ValueError('score must be an integer!')

**if** value < 0 **or** value > 100:

**raise** ValueError('score must between 0 ~ 100!')

self.\_score = value

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

>>> s = Student()

>>> s.set\_score(60) *# ok!*

>>> s.get\_score()

60

>>> s.set\_score(9999)

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的Python程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的@property装饰器就是负责把一个方法变成属性调用的：

**class Student(object):**

@property

**def** **score**(self):

**return** self.\_score

@score.setter

**def** **score**(self, value):

**if** **not** isinstance(value, int):

**raise** ValueError('score must be an integer!')

**if** value < 0 **or** value > 100:

**raise** ValueError('score must between 0 ~ 100!')

self.\_score = value

@property的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上@property就可以了，此时，@property本身又创建了另一个装饰器@score.setter，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

>>> s = Student()

>>> s.score = 60 *# OK，实际转化为s.set\_score(60)*

>>> s.score *# OK，实际转化为s.get\_score()*

60

>>> s.score = 9999

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

注意到这个神奇的@property，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

**class Student(object):**

@property

**def** **birth**(self):

**return** self.\_birth

@birth.setter

**def** **birth**(self, value):

self.\_birth = value

@property

**def** **age**(self):

**return** 2015 - self.\_birth

上面的birth是可读写属性，而age就是一个**只读**属性，因为age可以根据birth和当前时间计算出来。

小结

@property广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

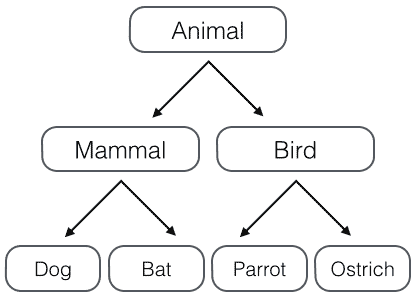
## 第40课 多重继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。

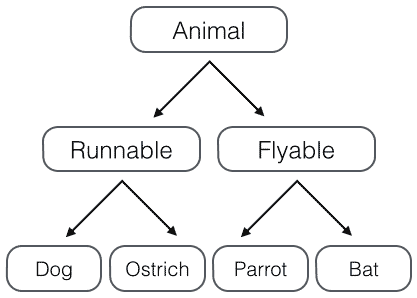
回忆一下Animal类层次的设计，假设我们要实现以下4种动物：

* Dog - 狗狗；
* Bat - 蝙蝠；
* Parrot - 鹦鹉；
* Ostrich - 鸵鸟。

如果按照哺乳动物和鸟类归类，我们可以设计出这样的类的层次：



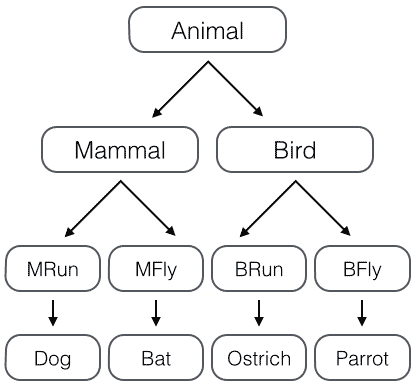
但是如果按照“能跑”和“能飞”来归类，我们就应该设计出这样的类的层次：



如果要把上面的两种分类都包含进来，我们就得设计更多的层次：

* 哺乳类：能跑的哺乳类，能飞的哺乳类；
* 鸟类：能跑的鸟类，能飞的鸟类。

这么一来，类的层次就复杂了：



如果要再增加“宠物类”和“非宠物类”，这么搞下去，类的数量会呈指数增长，很明显这样设计是不行的。

正确的做法是采用多重继承。首先，主要的类层次仍按照哺乳类和鸟类设计：

**class Animal(object):**

**pass**

*# 大类:*

**class Mammal(Animal):**

**pass**

**class Bird(Animal):**

**pass**

*# 各种动物:*

**class Dog(Mammal):**

**pass**

**class Bat(Mammal):**

**pass**

**class Parrot(Bird):**

**pass**

**class Ostrich(Bird):**

**pass**

现在，我们要给动物再加上Runnable和Flyable的功能，只需要先定义好Runnable和Flyable的类：

**class Runnable(object):**

**def** **run**(self):

print('Running...')

**class Flyable(object):**

**def** **fly**(self):

print('Flying...')

对于需要Runnable功能的动物，就多继承一个Runnable，例如Dog：

**class Dog(Mammal, Runnable):**

**pass**

对于需要Flyable功能的动物，就多继承一个Flyable，例如Bat：

**class Bat(Mammal, Flyable):**

**pass**

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

### MixIn

在设计类的继承关系时，通常，主线都是单一继承下来的，例如，Ostrich继承自Bird。但是，如果需要“混入”额外的功能，通过多重继承就可以实现，比如，让Ostrich除了继承自Bird外，再同时继承Runnable。这种设计通常称之为MixIn。

为了更好地看出继承关系，我们把Runnable和Flyable改为RunnableMixIn和FlyableMixIn。类似的，你还可以定义出肉食动物CarnivorousMixIn和植食动物HerbivoresMixIn，让某个动物同时拥有好几个MixIn：

**class Dog(Mammal, RunnableMixIn, CarnivorousMixIn):**

**pass**

MixIn的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个MixIn的功能，而不是设计多层次的复杂的继承关系。

Python自带的很多库也使用了MixIn。举个例子，Python自带了TCPServer和UDPServer这两类网络服务，而要同时服务多个用户就必须使用多进程或多线程模型，这两种模型由ForkingMixIn和ThreadingMixIn提供。通过组合，我们就可以创造出合适的服务来。

比如，编写一个多进程模式的TCP服务，定义如下：

**class MyTCPServer(TCPServer, ForkingMixIn):**

**pass**

编写一个多线程模式的UDP服务，定义如下：

**class MyUDPServer(UDPServer, ThreadingMixIn):**

**pass**

如果你打算搞一个更先进的协程模型，可以编写一个CoroutineMixIn：

**class MyTCPServer(TCPServer, CoroutineMixIn):**

**pass**

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

### 小结

由于Python允许使用多重继承，因此，MixIn就是一种常见的设计。

只允许单一继承的语言（如Java）不能使用MixIn的设计。

## 第41课 定制类

看到类似\_\_slots\_\_这种形如\_\_xxx\_\_的变量或者函数名就要注意，这些在Python中是有特殊用途的。

\_\_slots\_\_我们已经知道怎么用了，\_\_len\_\_()方法我们也知道是为了能让class作用于len()函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

### \_\_str\_\_

我们先定义一个Student类，打印一个实例：

>>> **class Student(object):**

... **def** **\_\_init\_\_**(self, name):

... self.name = name

...

>>> print(Student('Michael'))

<\_\_main\_\_.Student object at 0x109afb190>

打印出一堆<\_\_main\_\_.Student object at 0x109afb190>，不好看。

怎么才能打印得好看呢？只需要定义好\_\_str\_\_()方法，返回一个好看的字符串就可以了：

>>> **class Student(object):**

... **def** **\_\_init\_\_**(self, name):

... self.name = name

... **def** **\_\_str\_\_**(self):

... **return** 'Student object (name: %s)' % self.name

...

>>> print(Student('Michael'))

Student object (name: Michael)

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用print，打印出来的实例还是不好看：

>>> s = Student('Michael')

>>> s

<\_\_main\_\_.Student object at 0x109afb310>

这是因为直接显示变量调用的不是\_\_str\_\_()，而是\_\_repr\_\_()，两者的区别是\_\_str\_\_()返回用户看到的字符串，而\_\_repr\_\_()返回程序开发者看到的字符串，也就是说，\_\_repr\_\_()是为调试服务的。

解决办法是再定义一个\_\_repr\_\_()。但是通常\_\_str\_\_()和\_\_repr\_\_()代码都是一样的，所以，有个偷懒的写法：

**class Student(object):**

**def** **\_\_init\_\_**(self, name):

self.name = name

**def** **\_\_str\_\_**(self):

**return** 'Student object (name=%s)' % self.name

\_\_repr\_\_ = \_\_str\_\_

### \_\_iter\_\_

如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的\_\_next\_\_()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

**class Fib(object):**

**def** **\_\_init\_\_**(self):

self.a, self.b = 0, 1 *# 初始化两个计数器a，b*

**def** **\_\_iter\_\_**(self):

**return** self *# 实例本身就是迭代对象，故返回自己*

**def** **\_\_next\_\_**(self):

self.a, self.b = self.b, self.a + self.b *# 计算下一个值*

**if** self.a > 100000: *# 退出循环的条件*

**raise** StopIteration();

**return** self.a *# 返回下一个值*

现在，试试把Fib实例作用于for循环：

>>> **for** n **in** Fib():

... print(n)

...

1

1

2

3

5

...

46368

75025

### \_\_getitem\_\_

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

>>> Fib()[5]

Traceback (most recent call last):

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

TypeError: 'Fib' object does not support indexing

要表现得像list那样按照下标取出元素，需要实现\_\_getitem\_\_()方法：

**class Fib(object):**

**def** **\_\_getitem\_\_**(self, n):

a, b = 1, 1

**for** x **in** range(n):

a, b = b, a + b

**return** a

现在，就可以按下标访问数列的任意一项了：

>>> f = Fib()

>>> f[0]

1

>>> f[1]

1

>>> f[2]

2

>>> f[3]

3

>>> f[10]

89

>>> f[100]

573147844013817084101

但是list有个神奇的切片方法：

>>> list(range(100))[5:10]

[5, 6, 7, 8, 9]

对于Fib却报错。原因是\_\_getitem\_\_()传入的参数可能是一个int，也可能是一个切片对象slice，所以要做判断：

**class Fib(object):**

**def** **\_\_getitem\_\_**(self, n):

**if** isinstance(n, int): *# n是索引*

a, b = 1, 1

**for** x **in** range(n):

a, b = b, a + b

**return** a

**if** isinstance(n, slice): *# n是切片*

start = n.start

stop = n.stop

**if** start **is** None:

start = 0

a, b = 1, 1

L = []

**for** x **in** range(stop):

**if** x >= start:

L.append(a)

a, b = b, a + b

**return** L

现在试试Fib的切片：

>>> f = Fib()

>>> f[0:5]

[1, 1, 2, 3, 5]

>>> f[:10]

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

但是没有对step参数作处理：

>>> f[:10:2]

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

也没有对负数作处理，所以，要正确实现一个\_\_getitem\_\_()还是有很多工作要做的。

此外，如果把对象看成dict，\_\_getitem\_\_()的参数也可能是一个可以作key的object，例如str。

与之对应的是\_\_setitem\_\_()方法，把对象视作list或dict来对集合赋值。最后，还有一个\_\_delitem\_\_()方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

### \_\_getattr\_\_

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。比如定义Student类：

**class Student(object):**

**def** **\_\_init\_\_**(self):

self.name = 'Michael'

调用name属性，没问题，但是，调用不存在的score属性，就有问题了：

>>> s = Student()

>>> print(s.name)

Michael

>>> print(s.score)

Traceback (most recent call last):

...

AttributeError: 'Student' object has no attribute 'score'

错误信息很清楚地告诉我们，没有找到score这个attribute。

要避免这个错误，除了可以加上一个score属性外，Python还有另一个机制，那就是写一个\_\_getattr\_\_()方法，动态返回一个属性。修改如下：

**class Student(object):**

**def** **\_\_init\_\_**(self):

self.name = 'Michael'

**def** **\_\_getattr\_\_**(self, attr):

**if** attr=='score':

**return** 99

当调用不存在的属性时，比如score，Python解释器会试图调用\_\_getattr\_\_(self, 'score')来尝试获得属性，这样，我们就有机会返回score的值：

>>> s = Student()

>>> s.name

'Michael'

>>> s.score

99

返回函数也是完全可以的：

**class Student(object):**

**def** **\_\_getattr\_\_**(self, attr):

**if** attr=='age':

**return** **lambda**: 25

只是调用方式要变为：

>>> s.age()

25

注意，只有在没有找到属性的情况下，才调用\_\_getattr\_\_，已有的属性，比如name，不会在\_\_getattr\_\_中查找。

此外，注意到任意调用如s.abc都会返回None，这是因为我们定义的\_\_getattr\_\_默认返回就是None。要让class只响应特定的几个属性，我们就要按照约定，抛出AttributeError的错误：

**class Student(object):**

**def** **\_\_getattr\_\_**(self, attr):

**if** attr=='age':

**return** **lambda**: 25

**raise** AttributeError('\'Student\' object has no attribute \'%s\'' % attr)

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

举个例子：

现在很多网站都搞REST API，比如新浪微博、豆瓣啥的，调用API的URL类似：

* <http://api.server/user/friends>
* <http://api.server/user/timeline/list>

如果要写SDK，给每个URL对应的API都写一个方法，那得累死，而且，API一旦改动，SDK也要改。

利用完全动态的\_\_getattr\_\_，我们可以写出一个链式调用：

**class Chain(object):**

**def** **\_\_init\_\_**(self, path=''):

self.\_path = path

**def** **\_\_getattr\_\_**(self, path):

**return** Chain('%s/%s' % (self.\_path, path))

**def** **\_\_str\_\_**(self):

**return** self.\_path

\_\_repr\_\_ = \_\_str\_\_

试试：

>>> Chain().status.user.timeline.**list**

'/status/user/timeline/list'

这样，无论API怎么变，SDK都可以根据URL实现完全动态的调用，而且，不随API的增加而改变！

还有些REST API会把参数放到URL中，比如GitHub的API：

GET /users/:user/repos

调用时，需要把:user替换为实际用户名。如果我们能写出这样的链式调用：

Chain().users('michael').repos

就可以非常方便地调用API了。有兴趣的童鞋可以试试写出来。

### \_\_call\_\_

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用instance.method()来调用。能不能直接在实例本身上调用呢？在Python中，答案是肯定的。

任何类，只需要定义一个\_\_call\_\_()方法，就可以直接对实例进行调用。请看示例：

**class Student(object):**

**def** **\_\_init\_\_**(self, name):

self.name = name

**def** **\_\_call\_\_**(self):

print('My name is %s.' % self.name)

调用方式如下：

>>> s = Student('Michael')

>>> s() *# self参数不要传入*

My name **is** Michael.

\_\_call\_\_()还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个Callable对象，比如函数和我们上面定义的带有\_\_call\_\_()的类实例：

>>> callable(Student())

True

>>> callable(max)

True

>>> callable([1, 2, 3])

False

>>> callable(None)

False

>>> callable('str')

False

通过callable()函数，我们就可以判断一个对象是否是“可调用”对象。

### 小结

Python的class允许定义许多定制方法，可以让我们非常方便地生成特定的类。

本节介绍的是最常用的几个定制方法，还有很多可定制的方法，请参考[Python的官方文档](http://docs.python.org/3/reference/datamodel.html#special-method-names)。

## 第42课 使用枚举类

当我们需要定义常量时，一个办法是用大写变量通过整数来定义，例如月份：

JAN = 1

FEB = 2

MAR = 3

...

NOV = 11

DEC = 12

好处是简单，缺点是类型是int，并且仍然是变量。

更好的方法是为这样的枚举类型定义一个class类型，然后，每个常量都是class的一个唯一实例。Python提供了Enum类来实现这个功能：

from **enum** **import** Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))

这样我们就获得了Month类型的枚举类，可以直接使用Month.Jan来引用一个常量，或者枚举它的所有成员：

**for** name, member **in** Month.\_\_members\_\_.items():

print(name, '=>', member, ',', member.value)

value属性则是自动赋给成员的int常量，默认从1开始计数。

如果需要更精确地控制枚举类型，可以从Enum派生出自定义类：

**from** enum **import** Enum, unique

@unique

**class Weekday(Enum):**

Sun = 0 *# Sun的value被设定为0*

Mon = 1

Tue = 2

Wed = 3

Thu = 4

Fri = 5

Sat = 6

@unique装饰器可以帮助我们检查保证没有重复值。

访问这些枚举类型可以有若干种方法：

>>> day1 = Weekday.Mon

>>> print(day1)

Weekday.Mon

>>> print(Weekday.Tue)

Weekday.Tue

>>> print(Weekday['Tue'])

Weekday.Tue

>>> print(Weekday.Tue.value)

2

>>> print(day1 == Weekday.Mon)

True

>>> print(day1 == Weekday.Tue)

False

>>> print(Weekday(1))

Weekday.Mon

>>> print(day1 == Weekday(1))

True

>>> Weekday(7)

Traceback (most recent call last):

...

ValueError: 7 **is** **not** a valid Weekday

>>> **for** name, member **in** Weekday.\_\_members\_\_.items():

... print(name, '=>', member)

...

Sun => Weekday.Sun

Mon => Weekday.Mon

Tue => Weekday.Tue

Wed => Weekday.Wed

Thu => Weekday.Thu

Fri => Weekday.Fri

Sat => Weekday.Sat

可见，既可以用成员名称引用枚举常量，又可以直接根据value的值获得枚举常量。

### 小结

Enum可以把一组相关常量定义在一个class中，且class不可变，而且成员可以直接比较。

## 第43课 使用元类

### type()

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说我们要定义一个Hello的class，就写一个hello.py模块：

**class Hello(object):**

**def** **hello**(self, name='world'):

print('Hello, %s.' % name)

当Python解释器载入hello模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个Hello的class对象，测试如下：

>>> from hello **import** Hello

>>> h = Hello()

>>> h.hello()

Hello, world.

>>> print(type(Hello))

<**class 'type'>**

**>>> print(type(h))**

**<class 'hello.Hello'>**

type()函数可以查看一个类型或变量的类型，Hello是一个class，它的类型就是type，而h是一个实例，它的类型就是class Hello。

我们说class的定义是运行时动态创建的，而创建class的方法就是使用type()函数。

type()函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过type()函数创建出Hello类，而无需通过class Hello(object)...的定义：

>>> def fn(self, name='world'): # 先定义函数

... print('Hello, %s.' % name)

...

>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello **class**

**>>> h = Hello()**

**>>> h.hello()**

**Hello, world.**

**>>> print(type(Hello))**

**<class 'type'>**

**>>> print(type(h))**

**<class '\_\_main\_\_.Hello'>**

要创建一个class对象，type()函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
3. class的方法名称与函数绑定，这里我们把函数fn绑定到方法名hello上。

通过type()函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用type()函数创建出class。

正常情况下，我们都用class Xxx...来定义类，但是，type()函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

### metaclass\*

除了使用type()动态创建类以外，要控制类的创建行为，还可以使用metaclass。

metaclass，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据metaclass创建出类，所以：先定义metaclass，然后创建类。

连接起来就是：先定义metaclass，就可以创建类，最后创建实例。

所以，metaclass允许你创建类或者修改类。换句话说，你可以把类看成是metaclass创建出来的“实例”。

metaclass是Python面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会碰到需要使用metaclass的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。

我们先看一个简单的例子，这个metaclass可以给我们自定义的MyList增加一个add方法：

定义ListMetaclass，按照默认习惯，metaclass的类名总是以Metaclass结尾，以便清楚地表示这是一个metaclass：

*# metaclass是类的模板，所以必须从`type`类型派生：*

**class ListMetaclass(type):**

**def** **\_\_new\_\_**(cls, name, bases, attrs):

attrs['add'] = **lambda** self, value: self.append(value)

**return** type.\_\_new\_\_(cls, name, bases, attrs)

有了ListMetaclass，我们在定义类的时候还要指示使用ListMetaclass来定制类，传入关键字参数metaclass：

**class MyList(list, metaclass=ListMetaclass):**

**pass**

当我们传入关键字参数metaclass时，魔术就生效了，它指示Python解释器在创建MyList时，要通过ListMetaclass.\_\_new\_\_()来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

\_\_new\_\_()方法接收到的参数依次是：

1. 当前准备创建的类的对象；
2. 类的名字；
3. 类继承的父类集合；
4. 类的方法集合。

测试一下MyList是否可以调用add()方法：

>>> L = MyList()

>>> L.add(1)

>> L

[1]

而普通的list没有add()方法：

>>> L2 = list()

>>> L2.add(1)

Traceback (most recent call last):

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

AttributeError: 'list' object has no attribute 'add'

动态修改有什么意义？直接在MyList定义中写上add()方法不是更简单吗？正常情况下，确实应该直接写，通过metaclass修改纯属变态。

但是，总会遇到需要通过metaclass修改类定义的。ORM就是一个典型的例子。

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

让我们来尝试编写一个ORM框架。

编写底层模块的第一步，就是先把调用接口写出来。比如，使用者如果使用这个ORM框架，想定义一个User类来操作对应的数据库表User，我们期待他写出这样的代码：

**class User(Model):**

*# 定义类的属性到列的映射：*

id = IntegerField('id')

name = StringField('username')

email = StringField('email')

password = StringField('password')

*# 创建一个实例：*

u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')

*# 保存到数据库：*

u.save()

其中，父类Model和属性类型StringField、IntegerField是由ORM框架提供的，剩下的魔术方法比如save()全部由metaclass自动完成。虽然metaclass的编写会比较复杂，但ORM的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该ORM。

首先来定义Field类，它负责保存数据库表的字段名和字段类型：

**class Field(object):**

**def** **\_\_init\_\_**(self, name, column\_type):

self.name = name

self.column\_type = column\_type

**def** **\_\_str\_\_**(self):

**return** '<%s:%s>' % (self.\_\_class\_\_.\_\_name\_\_, self.name)

在Field的基础上，进一步定义各种类型的Field，比如StringField，IntegerField等等：

**class StringField(Field):**

**def** **\_\_init\_\_**(self, name):

super(StringField, self).\_\_init\_\_(name, 'varchar(100)')

**class IntegerField(Field):**

**def** **\_\_init\_\_**(self, name):

super(IntegerField, self).\_\_init\_\_(name, 'bigint')

下一步，就是编写最复杂的ModelMetaclass了：

**class ModelMetaclass(type):**

**def** **\_\_new\_\_**(cls, name, bases, attrs):

**if** name=='Model':

**return** type.\_\_new\_\_(cls, name, bases, attrs)

print('Found model: %s' % name)

mappings = dict()

**for** k, v **in** attrs.items():

**if** isinstance(v, Field):

print('Found mapping: %s ==> %s' % (k, v))

mappings[k] = v

**for** k **in** mappings.keys():

attrs.pop(k)

attrs['\_\_mappings\_\_'] = mappings *# 保存属性和列的映射关系*

attrs['\_\_table\_\_'] = name *# 假设表名和类名一致*

**return** type.\_\_new\_\_(cls, name, bases, attrs)

以及基类Model：

**class Model(dict, metaclass=ModelMetaclass):**

**def** **\_\_init\_\_**(self, \*\*kw):

super(Model, self).\_\_init\_\_(\*\*kw)

**def** **\_\_getattr\_\_**(self, key):

**try**:

**return** self[key]

**except** KeyError:

**raise** AttributeError(r"'Model' object has no attribute '%s'" % key)

**def** **\_\_setattr\_\_**(self, key, value):

self[key] = value

**def** **save**(self):

fields = []

params = []

args = []

**for** k, v **in** self.\_\_mappings\_\_.items():

fields.append(v.name)

params.append('?')

args.append(getattr(self, k, None))

sql = 'insert into %s (%s) values (%s)' % (self.\_\_table\_\_, ','.join(fields), ','.join(params))

print('SQL: %s' % sql)

print('ARGS: %s' % str(args))

当用户定义一个class User(Model)时，Python解释器首先在当前类User的定义中查找metaclass，如果没有找到，就继续在父类Model中查找metaclass，找到了，就使用Model中定义的metaclass的ModelMetaclass来创建User类，也就是说，metaclass可以隐式地继承到子类，但子类自己却感觉不到。

在ModelMetaclass中，一共做了几件事情：

1. 排除掉对Model类的修改；
2. 在当前类（比如User）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个\_\_mappings\_\_的dict中，同时从类属性中删除该Field属性，否则，容易造成运行时错误（实例的属性会遮盖类的同名属性）；
3. 把表名保存到\_\_table\_\_中，这里简化为表名默认为类名。

在Model类中，就可以定义各种操作数据库的方法，比如save()，delete()，find()，update等等。

我们实现了save()方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出INSERT语句。

编写代码试试：

u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')

u.save()

输出如下：

Found model: User

Found mapping: email ==> <StringField:email>

Found mapping: password ==> <StringField:password>

Found mapping: id ==> <IntegerField:uid>

Found mapping: name ==> <StringField:username>

SQL: insert into User (password,email,username,id) values (?,?,?,?)

ARGS: ['my-pwd', 'test@orm.org', 'Michael', 12345]

可以看到，save()方法已经打印出了可执行的SQL语句，以及参数列表，只需要真正连接到数据库，执行该SQL语句，就可以完成真正的功能。

不到100行代码，我们就通过metaclass实现了一个精简的ORM框架。

### 小结

metaclass是Python中非常具有魔术性的对象，它可以改变类创建时的行为。这种强大的功能使用起来务必小心。

## 第44课 错误、调试和测试

在程序运行过程中，总会遇到各种各样的错误。

有的错误是程序编写有问题造成的，比如本来应该输出整数结果输出了字符串，这种错误我们通常称之为bug，bug是必须修复的。

有的错误是用户输入造成的，比如让用户输入email地址，结果得到一个空字符串，这种错误可以通过检查用户输入来做相应的处理。

还有一类错误是完全无法在程序运行过程中预测的，比如写入文件的时候，磁盘满了，写不进去了，或者从网络抓取数据，网络突然断掉了。这类错误也称为异常，在程序中通常是必须处理的，否则，程序会因为各种问题终止并退出。

Python内置了一套异常处理机制，来帮助我们进行错误处理。

此外，我们也需要跟踪程序的执行，查看变量的值是否正确，这个过程称为调试。Python的pdb可以让我们以单步方式执行代码。

最后，编写测试也很重要。有了良好的测试，就可以在程序修改后反复运行，确保程序输出符合我们编写的测试。

## 第45课 错误处理

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数open()，成功时返回文件描述符（就是一个整数），出错时返回-1。

用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

**def** **foo**():

r = some\_function()

**if** r==(-1):

**return** (-1)

*# do something*

**return** r

**def** **bar**():

r = foo()

**if** r==(-1):

print('Error')

**else**:

**pass**

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套try...except...finally...的错误处理机制，Python也不例外。

### try

让我们用一个例子来看看try的机制：

**try**:

print('try...')

r = 10 / 0

print('result:', r)

**except** ZeroDivisionError **as** e:

print('except:', e)

**finally**:

print('finally...')

print('END')

当我们认为某些代码可能会出错时，就可以用try来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即except语句块，执行完except后，如果有finally语句块，则执行finally语句块，至此，执行完毕。

上面的代码在计算10 / 0时会产生一个除法运算错误：

**try**...

**except**: division by zero

**finally**...

END

从输出可以看到，当错误发生时，后续语句print('result:', r)不会被执行，except由于捕获到ZeroDivisionError，因此被执行。最后，finally语句被执行。然后，程序继续按照流程往下走。

如果把除数0改成2，则执行结果如下：

**try**...

result: 5

**finally**...

END

由于没有错误发生，所以except语句块不会被执行，但是finally如果有，则一定会被执行（可以没有finally语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的except语句块处理。没错，可以有多个except来捕获不同类型的错误：

**try**:

print('try...')

r = 10 / int('a')

print('result:', r)

**except** ValueError **as** e:

print('ValueError:', e)

**except** ZeroDivisionError **as** e:

print('ZeroDivisionError:', e)

**finally**:

print('finally...')

print('END')

int()函数可能会抛出ValueError，所以我们用一个except捕获ValueError，用另一个except捕获ZeroDivisionError。

此外，如果没有错误发生，可以在except语句块后面加一个else，当没有错误发生时，会自动执行else语句：

**try**:

print('try...')

r = 10 / int('2')

print('result:', r)

**except** ValueError **as** e:

print('ValueError:', e)

**except** ZeroDivisionError **as** e:

print('ZeroDivisionError:', e)

**else**:

print('no error!')

**finally**:

print('finally...')

print('END')

Python的错误其实也是class，所有的错误类型都继承自BaseException，所以在使用except时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

**try**:

foo()

**except** ValueError **as** e:

print('ValueError')

**except** UnicodeError **as** e:

print('UnicodeError')

第二个except永远也捕获不到UnicodeError，因为UnicodeError是ValueError的子类，如果有，也被第一个except给捕获了。

Python所有的错误都是从BaseException类派生的，常见的错误类型和继承关系看这里：

<https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

使用try...except捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数main()调用foo()，foo()调用bar()，结果bar()出错了，这时，只要main()捕获到了，就可以处理：

**def** **foo**(s):

**return** 10 / int(s)

**def** **bar**(s):

**return** foo(s) \* 2

**def** **main**():

**try**:

bar('0')

**except** Exception **as** e:

print('Error:', e)

**finally**:

print('finally...')

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写try...except...finally的麻烦。

### 调用堆栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看err.py：

*# err.py:*

**def** **foo**(s):

**return** 10 / int(s)

**def** **bar**(s):

**return** foo(s) \* 2

**def** **main**():

bar('0')

main()

执行，结果如下：

$ python3 err.py

Traceback (most recent **call** **last**):

File "err.py", line 11, **in** <**module**>

main()

File "err.py", line 9, **in** main

bar('0')

File "err.py", line 6, **in** bar

return foo(s) \* 2

File "err.py", line 3, **in** foo

return 10 / **int**(s)

ZeroDivisionError: division **by** zero

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：

错误信息第1行：

Traceback (most recent **call** **last**):

告诉我们这是错误的跟踪信息。

第2~3行：

File "err.py", line 11, in <module>

main()

调用main()出错了，在代码文件err.py的第11行代码，但原因是第9行：

File "err.py", line 9, **in** main

bar('0')

调用bar('0')出错了，在代码文件err.py的第9行代码，但原因是第6行：

File "err.py", line 6, **in** bar

**return** foo(s) \* 2

原因是return foo(s) \* 2这个语句出错了，但这还不是最终原因，继续往下看：

File "err.py", line 3, **in** foo

**return** 10 / int(s)

原因是return 10 / int(s)这个语句出错了，这是错误产生的源头，因为下面打印了：

ZeroDivisionError: integer division **or** modulo by zero

根据错误类型ZeroDivisionError，我们判断，int(s)本身并没有出错，但是int(s)返回0，在计算10 / 0时出错，至此，找到错误源头。

### 记录错误

如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的logging模块可以非常容易地记录错误信息：

*# err\_logging.py*

**import** logging

**def** **foo**(s):

**return** 10 / int(s)

**def** **bar**(s):

**return** foo(s) \* 2

**def** **main**():

**try**:

bar('0')

**except** Exception **as** e:

logging.exception(e)

main()

print('END')

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

$ python3 err\_logging.py

ERROR:root:division by zero

Traceback (most recent **call** **last**):

File "err\_logging.py", line 13, **in** main

bar('0')

File "err\_logging.py", line 9, **in** bar

return foo(s) \* 2

File "err\_logging.py", line 6, **in** foo

return 10 / **int**(s)

ZeroDivisionError: division **by** zero

**END**

通过配置，logging还可以把错误记录到日志文件里，方便事后排查。

### 抛出错误

因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用raise语句抛出一个错误的实例：

*# err\_raise.py*

**class FooError(ValueError):**

**pass**

**def** **foo**(s):

n = int(s)

**if** n==0:

**raise** FooError('invalid value: %s' % s)

**return** 10 / n

foo('0')

执行，可以最后跟踪到我们自己定义的错误：

$ python3 err\_raise.py

Traceback (most recent **call** **last**):

File "err\_throw.py", line 11, **in** <**module**>

foo('0')

File "err\_throw.py", line 8, **in** foo

raise FooError('invalid value: %s' % s)

\_\_main\_\_.FooError: invalid **value**: 0

只有在必要的时候才定义我们自己的错误类型。如果可以选择Python已有的内置的错误类型（比如ValueError，TypeError），尽量使用Python内置的错误类型。

最后，我们来看另一种错误处理的方式：

*# err\_reraise.py*

**def** **foo**(s):

n = int(s)

**if** n==0:

**raise** ValueError('invalid value: %s' % s)

**return** 10 / n

**def** **bar**():

**try**:

foo('0')

**except** ValueError **as** e:

print('ValueError!')

**raise**

bar()

在bar()函数中，我们明明已经捕获了错误，但是，打印一个ValueError!后，又把错误通过raise语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。好比一个员工处理不了一个问题时，就把问题抛给他的老板，如果他的老板也处理不了，就一直往上抛，最终会抛给CEO去处理。

raise语句如果不带参数，就会把当前错误原样抛出。此外，在except中raise一个Error，还可以把一种类型的错误转化成另一种类型：

**try**:

10 / 0

**except** ZeroDivisionError:

**raise** ValueError('input error!')

只要是合理的转换逻辑就可以，但是，决不应该把一个IOError转换成毫不相干的ValueError。

### 小结

Python内置的try...except...finally用来处理错误十分方便。出错时，会分析错误信息并定位错误发生的代码位置才是最关键的。

程序也可以主动抛出错误，让调用者来处理相应的错误。但是，应该在文档中写清楚可能会抛出哪些错误，以及错误产生的原因。

## 第46课 调试

程序能一次写完并正常运行的概率很小，基本不超过1%。总会有各种各样的bug需要修正。有的bug很简单，看看错误信息就知道，有的bug很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复bug。

第一种方法简单直接粗暴有效，就是用print()把可能有问题的变量打印出来看看：

**def** **foo**(s):

n = int(s)

print('>>> n = %d' % n)

**return** 10 / n

**def** **main**():

foo('0')

main()

执行后在输出中查找打印的变量值：

$ python3 err.py

>>> n = 0

Traceback (most recent **call** **last**):

...

ZeroDivisionError: **integer** division **or** modulo **by** zero

用print()最大的坏处是将来还得删掉它，想想程序里到处都是print()，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

### 断言

凡是用print()来辅助查看的地方，都可以用断言（assert）来替代：

（assert语句用来声明某个条件是真的，当assert语句失败的时候，会引发一AssertionError。assert还有一个异常参数，就是在断言表达式后添加字符串信息，用来解释断言并更好的知道是哪里出了问题。格式如下：

assert expression [, arguments]

）

**def** **foo**(s):

n = int(s)

**assert** n != 0, 'n is zero!'

**return** 10 / n

**def** **main**():

foo('0')

assert的意思是，表达式n != 0应该是True，否则，根据程序运行的逻辑，后面的代码肯定会出错。

如果断言失败，assert语句本身就会抛出AssertionError：

$ python3 err.py

Traceback (most recent **call** **last**):

...

AssertionError: n **is** zero!

程序中如果到处充斥着assert，和print()相比也好不到哪去。不过，启动Python解释器时可以用-O参数来关闭assert：

$ python3 -O err.py

Traceback (most recent **call** **last**):

...

ZeroDivisionError: division **by** zero

关闭后，你可以把所有的assert语句当成pass来看。

### logging

把print()替换为logging是第3种方式，和assert比，logging不会抛出错误，而且可以输出到文件：

**import** logging

s = '0'

n = **int**(s)

logging.info('n = %d' % n)

print(10 / n)

logging.info()就可以输出一段文本。运行，发现除了ZeroDivisionError，没有任何信息。怎么回事？

别急，在import logging之后添加一行配置再试试：

**import** logging

logging.basicConfig(level=logging.INFO)

看到输出了：

$ python3 err.py

INFO:root:n = 0

Traceback (most recent **call** **last**):

File "err.py", line 8, **in** <**module**>

print(10 / n)

ZeroDivisionError: division **by** zero

这就是logging的好处，它允许你指定记录信息的级别，有debug，info，warning，error等几个级别，当我们指定level=INFO时，logging.debug就不起作用了。同理，指定level=WARNING后，debug和info就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

logging的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

### pdb

第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

*# err.py*

s = '0'

n = int(s)

**print**(10 / n)

然后启动：

$ python3 -m pdb err.py

> /Users/michael/Github/learn-python3/samples/debug/err.py(2)<**module**>()

-> s = '0'

以参数-m pdb启动后，pdb定位到下一步要执行的代码-> s = '0'。输入命令l来查看代码：

(Pdb) l

1 *# err.py*

2 -> s = '0'

3 n = int(s)

4 **print**(10 / n)

输入命令n可以单步执行代码：

(Pdb) n

> /Users/michael/Github/learn-python3/samples/debug/err.py(3)<**module**>()

-> n = int(s)

(Pdb) n

> /Users/michael/Github/learn-python3/samples/debug/err.py(4)<**module**>()

-> print(10 / n)

任何时候都可以输入命令p 变量名来查看变量：

(Pdb) p s

'0'

(Pdb) p n

0

输入命令q结束调试，退出程序：

(Pdb) q

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

### pdb.set\_trace()

这个方法也是用pdb，但是不需要单步执行，我们只需要import pdb，然后，在可能出错的地方放一个pdb.set\_trace()，就可以设置一个断点：

*# err.py*

**import** pdb

s = '0'

n = int(s)

pdb.set\_trace() *# 运行到这里会自动暂停*

print(10 / n)

运行代码，程序会自动在pdb.set\_trace()暂停并进入pdb调试环境，可以用命令p查看变量，或者用命令c继续运行：

$ python3 err.py

> /Users/michael/Github/learn-python3/samples/debug/err.py(7)<**module**>()

-> print(10 / n)

(Pdb) p n

0

(Pdb) c

Traceback (most recent call last):

File "err.py", line 7, **in** <**module**>

print(10 / n)

ZeroDivisionError: division by zero

这个方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

### IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有PyCharm：

<http://www.jetbrains.com/pycharm/>

另外，[Eclipse](http://eclipse.org/)加上[pydev](http://pydev.org/)插件也可以调试Python程序。

### 小结

写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。

虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。

## 第47课 单元测试

如果你听说过“测试驱动开发”（TDD：Test-Driven Development），单元测试就不陌生。（TDD要求在编写某个功能的代码之前先编写测试代码，然后只编写使测试通过的功能代码，通过测试来推动整个开发的进行。这有助于编写简洁可用和高质量的代码，并加速开发过程。盖房子的时候，工人师傅砌墙，会先用桩子拉上线，以使砖能够垒的笔直，因为垒砖的时候都是以这根线为基准的。TDD就像这样，先写测试代码，就像工人师傅先用桩子拉上线，然后编码的时候以此为基准，只编写符合这个测试的功能代码。）

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数abs()，我们可以编写出以下几个测试用例：

1. 输入正数，比如1、1.2、0.99，期待返回值与输入相同；
2. 输入负数，比如-1、-1.2、-0.99，期待返回值与输入相反；
3. 输入0，期待返回0；
4. 输入非数值类型，比如None、[]、{}，期待抛出TypeError。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对abs()函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对abs()函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

我们来编写一个Dict类，这个类的行为和dict一致，但是可以通过属性来访问，用起来就像下面这样：

>>> d = Dict(a=1, b=2)

>>> d['a']

1

>>> d.a

1

mydict.py代码如下：

**class Dict(dict):**

**def** **\_\_init\_\_**(self, \*\*kw):

super().\_\_init\_\_(\*\*kw)

**def** **\_\_getattr\_\_**(self, key):

**try**:

**return** self[key]

**except** KeyError:

**raise** AttributeError(r"'Dict' object has no attribute '%s'" % key)

**def** **\_\_setattr\_\_**(self, key, value):

self[key] = value

为了编写单元测试，我们需要引入Python自带的unittest模块，编写mydict\_test.py如下：

**import** unittest

**from** mydict **import** Dict

**class TestDict(unittest.TestCase):**

**def** **test\_init**(self):

d = Dict(a=1, b='test')

self.assertEqual(d.a, 1)

self.assertEqual(d.b, 'test')

self.assertTrue(isinstance(d, dict))

**def** **test\_key**(self):

d = Dict()

d['key'] = 'value'

self.assertEqual(d.key, 'value')

**def** **test\_attr**(self):

d = Dict()

d.key = 'value'

self.assertTrue('key' **in** d)

self.assertEqual(d['key'], 'value')

**def** **test\_keyerror**(self):

d = Dict()

**with** self.assertRaises(KeyError):

value = d['empty']

**def** **test\_attrerror**(self):

d = Dict()

**with** self.assertRaises(AttributeError):

value = d.empty

编写单元测试时，我们需要编写一个测试类，从unittest.TestCase继承。

以test开头的方法就是测试方法，不以test开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个test\_xxx()方法。由于unittest.TestCase提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是assertEqual()：

**self**.assertEqual(abs(-1), 1) *# 断言函数返回的结果与1相等*

另一种重要的断言就是期待抛出指定类型的Error，比如通过d['empty']访问不存在的key时，断言会抛出KeyError：

with **self**.assertRaises(KeyError):

value = d['empty']

而通过d.empty访问不存在的key时，我们期待抛出AttributeError：

with **self**.assertRaises(AttributeError):

value = d.**empty**

### 运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在mydict\_test.py的最后加上两行代码：

**if** \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

这样就可以把mydict\_test.py当做正常的python脚本运行：

$ python3 mydict\_test.py

另一种方法是在命令行通过参数-m unittest直接运行单元测试：

$ python3 -m unittest mydict\_test

.....

----------------------------------------------------------------------

Ran 5 tests **in** 0.000s

OK

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

### setUp与tearDown

可以在单元测试中编写两个特殊的setUp()和tearDown()方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

setUp()和tearDown()方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在setUp()方法中连接数据库，在tearDown()方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

**class TestDict(unittest.TestCase):**

**def** **setUp**(self):

print('setUp...')

**def** **tearDown**(self):

print('tearDown...')

可以再次运行测试看看每个测试方法调用前后是否会打印出setUp...和tearDown...。

### 小结

单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。

单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。

单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能有bug。

单元测试通过了并不意味着程序就没有bug了，但是不通过程序肯定有bug。

### 附super()

子类里访问父类的同名属性，而又不想直接引用父类的名字，因为说不定什么时候会去修改它，所以数据还是只保留一份的好。其实呢，还有更好的理由不去直接引用父类的名字，这就是super()。

<https://docs.python.org/3.1/library/functions.html#super>

class A(object):#不继承object会报错

def m(self):

print('A')

class B(A):

def m(self):

print('B')

super().m()

B().m()

当然 Python 2 里super() 是一定要参数的，所以得这么写：

class B(A):

def m(self):

print('B')

super(B, self).m()

## 第48课 文档测试\*

如果你经常阅读Python的官方文档，可以看到很多文档都有示例代码。比如[re模块](https://docs.python.org/2/library/re.html)就带了很多示例代码：

>>> **import** re

>>> m = re.search('(?<=abc)def', 'abcdef')

>>> m.group(0)

'def'

可以把这些示例代码在Python的交互式环境下输入并执行，结果与文档中的示例代码显示的一致。

这些代码与其他说明可以写在注释中，然后，由一些工具来自动生成文档。既然这些代码本身就可以粘贴出来直接运行，那么，可不可以自动执行写在注释中的这些代码呢？

答案是肯定的。

当我们编写注释时，如果写上这样的注释：

**def** **abs**(n):

'''

Function to get absolute value of number.

Example:

>>> abs(1)

1

>>> abs(-1)

1

>>> abs(0)

0

'''

**return** n **if** n >= 0 **else** (-n)

无疑更明确地告诉函数的调用者该函数的期望输入和输出。

并且，Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。

doctest严格按照Python交互式命令行的输入和输出来判断测试结果是否正确。只有测试异常的时候，可以用...表示中间一大段烦人的输出。

让我们用doctest来测试上次编写的Dict类：

*# mydict2.py*

**class Dict(dict):**

'''

Simple dict but also support access as x.y style.

>>> d1 = Dict()

>>> d1['x'] = 100

>>> d1.x

100

>>> d1.y = 200

>>> d1['y']

200

>>> d2 = Dict(a=1, b=2, c='3')

>>> d2.c

'3'

>>> d2['empty']

Traceback (most recent call last):

...

KeyError: 'empty'

>>> d2.empty

Traceback (most recent call last):

...

AttributeError: 'Dict' object has no attribute 'empty'

'''

**def** **\_\_init\_\_**(self, \*\*kw):

super(Dict, self).\_\_init\_\_(\*\*kw)

**def** **\_\_getattr\_\_**(self, key):

**try**:

**return** self[key]

**except** KeyError:

**raise** AttributeError(r"'Dict' object has no attribute '%s'" % key)

**def** **\_\_setattr\_\_**(self, key, value):

self[key] = value

**if** \_\_name\_\_=='\_\_main\_\_':

**import** doctest

doctest.testmod()

运行python3 mydict2.py：

$ python3 mydict2.py

什么输出也没有。这说明我们编写的doctest运行都是正确的。如果程序有问题，比如把\_\_getattr\_\_()方法注释掉，再运行就会报错：

$ python3 mydict2.py

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py", line 10, in \_\_main\_\_.Dict

Failed example:

d1.x

Exception raised:

Traceback (most recent **call** **last**):

...

AttributeError: 'Dict' object has **no** attribute 'x'

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py", line 16, **in** \_\_main\_\_.Dict

Failed example:

d2.c

**Exception** raised:

Traceback (most recent **call** **last**):

...

AttributeError: 'Dict' object has **no** attribute 'c'

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1 items had failures:

2 **of** 9 **in** \_\_main\_\_.Dict

\*\*\*Test Failed\*\*\* 2 failures.

注意到最后3行代码。当模块正常导入时，doctest不会被执行。只有在命令行直接运行时，才执行doctest。所以，不必担心doctest会在非测试环境下执行。

## 第49课 IO编程

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。（inout和output一般相对内存而言）

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络IO获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

很明显，使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步IO的复杂度远远高于同步IO。

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。我们后面会详细讨论Python的IO编程接口。

注意，本章的IO编程都是同步模式，异步IO由于复杂度太高，后续涉及到服务器端程序开发时我们再讨论。

## 第50课 文件读写

读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

### 读文件

要以读文件的模式打开一个文件对象，使用Python内置的open()函数，传入文件名和标示符：

>>> f = open('/Users/michael/test.txt', 'r')

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在，open()函数就会抛出一个IOError的错误，并且给出错误码和详细的信息告诉你文件不存在：

>>> f=open('/Users/michael/notfound.txt', 'r')

Traceback (most recent **call** **last**):

File "<stdin>", line 1, **in** <**module**>

FileNotFoundError: [Errno 2] **No** such file **or** directory: '/Users/michael/notfound.txt'

如果文件打开成功，接下来，调用read()方法可以一次读取文件的全部内容，Python把内容读到内存，用一个str对象表示：

>>> f.read()

'Hello, world!'

最后一步是调用close()方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

>>> f.close()

由于文件读写时都有可能产生IOError，一旦出错，后面的f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally来实现：

**try**:

f = open('/path/to/file', 'r')

print(f.read())

**finally**:

**if** f:

f.close()

但是每次都这么写实在太繁琐，所以，Python引入了with语句来自动帮我们调用close()方法：

**with** open('/path/to/file', 'r') **as** f:

print(f.read())

这和前面的try ... finally是一样的，但是代码更佳简洁，并且不必调用f.close()方法。

with 语句适用于对资源进行访问的场合，确保不管使用过程中是否发生异常都会执行必要的“清理”操作，释放资源，比如文件使用后自动关闭、线程中锁的自动获取和释放等。

语法：

with context\_expression [as target(s)]:

with-body

这里 context\_expression 要返回一个上下文管理器对象，该对象并不赋值给 as 子句中的 target(s) ，如果指定了 as 子句的话，会将上下文管理器的 \_\_enter\_\_() 方法的返回值赋值给 target(s)。target(s) 可以是单个变量，或者由“()”括起来的元组（不能是仅仅由“,”分隔的变量列表，必须加“()”）。

调用read()会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用read(size)方法，每次最多读取size个字节的内容。另外，调用readline()可以每次读取一行内容，调用readlines()一次读取所有内容并按行返回list。因此，要根据需要决定怎么调用。

如果文件很小，read()一次性读取最方便；如果不能确定文件大小，反复调用read(size)比较保险；如果是配置文件，调用readlines()最方便：

**for** line **in** f.readlines():

print(line.strip()) *# 把末尾的'\n'删掉*

### file-like Object

像open()函数返回的这种有个read()方法的对象，在Python中统称为file-like Object。除了file外，还可以是内存的字节流，网络流，自定义流等等。file-like Object不要求从特定类继承，只要写个read()方法就行。

StringIO就是在内存中创建的file-like Object，常用作临时缓冲。

### 二进制文件

前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。要读取二进制文件，比如图片、视频等等，用'rb'模式打开文件即可：

>>> f = open('/Users/michael/test.jpg', 'rb')

>>> f.read()

b'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' *# 十六进制表示的字节*

### 字符编码

要读取非UTF-8编码的文本文件，需要给open()函数传入encoding参数，例如，读取GBK编码的文件：

>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk')

>>> f.read()

'测试'

遇到有些编码不规范的文件，你可能会遇到UnicodeDecodeError，因为在文本文件中可能夹杂了一些非法编码的字符。遇到这种情况，open()函数还接收一个errors参数，表示如果遇到编码错误后如何处理。最简单的方式是直接忽略：

>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk', errors='ignore')

### 写文件

写文件和读文件是一样的，唯一区别是调用open()函数时，传入标识符'w'或者'wb'表示写文本文件或写二进制文件：

>>> f = open('/Users/michael/test.txt', 'w')

>>> f.write('Hello, world!')

>>> f.close()

你可以反复调用write()来写入文件，但是务必要调用f.close()来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用close()方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用close()的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用with语句来得保险：

**with** open('/Users/michael/test.txt', 'w') **as** f:

f.write('Hello, world!')

要写入特定编码的文本文件，请给open()函数传入encoding参数，将字符串自动转换成指定编码。

### 小结

在Python中，文件读写是通过open()函数打开的文件对象完成的。使用with语句操作文件IO是个好习惯。

### 附二进制文件与文本文件

【具体到物理保存时都是二进制的，关键是保存前对数据的编码有区别。】  
文本文件与二进制文件在计算机文件系统中的物理存储都是二进制的，也就是在物理存储方面没有区别都是01码，这个没有异议，他们的区别主要在逻辑存储上，也就是编码上。  
文本文件格式存储时是将值作为字符然后存入其字符编码的二进制，文本文件用‘字符’作为单位来表示和存储数据，比如对于1这个值，文本文件会将其看做字符‘1’然后保存其ASCII编码值（这里假定是ASCII编码），这样在物理上就是0x31这个二进制值，而若是二进制保存1，则直接保存其二进制值，比如如果程序中是处理1为整数则保存的二进制值就是 0x00000001 (4字节）。  
当然如果程序本来就是按字符保存的 也就是 char ch ='1' ; 则二进制保存后值就是其ASCII码，因为该变量的二进制本来就是其ASCII码。可以总结出二进制文件就是值本身的编码，那么就是不定长的编码了，因为值本身就是不等字节的，如整数4个字节那么保存在二进制文件就是这四个字节的原生二进制值。  
  
综上，可以知道文本文件与二进制文件就是编码方式不一样而已，而这个是用户行为，把一个数据以什么样的编码（字符还是值本身）存入文件是由用户主动选择的，也就是写入的接口选择，如果以二进制接口方式写入文件那么就是一个二进制文件，如果以字符方式写入文件就是一个文本文件了。既然有写入时候的编码也就会有读出的编码，只有两个编码对应才能读出正确的结果，如用记事本打开一个二进制文件会呈现乱码的，这里稍微提一下后缀名，后缀名并不能确定其是否就是文本文件，二进制文件也可以是txt后缀名，后缀名只是用来关联打开程序，给用户做备注用的，与文件的具体编码没有关系。  
  
可以使用字符接口读写二进制文件，只需要做些处理即可，所以所谓的二进制文件，文本文件主要体现在读写方式这里。  
此外windows有一个明显的区别是对待文本文件读写的时候，会将换行 \n自动替换成 \r\n。  
  
最后文本文件和二进制文件主要是windows下的概念，UNIX/Linux并没有区分这两种文件，他们对所有文件一视同仁，将所有文件都看成二进制文件。

## 第51课 StringIO和BytesIO

### StringIO

很多时候，数据读写不一定是文件，也可以在内存中读写。

StringIO顾名思义就是在内存中读写str。

要把str写入StringIO，我们需要先创建一个StringIO，然后，像文件一样写入即可：

>>> **from** io **import** StringIO

>>> f = StringIO()

>>> f.write('hello')

5

>>> f.write(' ')

1

>>> f.write('world!')

6

>>> print(f.getvalue())

hello world!

getvalue()方法用于获得写入后的str。

要读取StringIO，可以用一个str初始化StringIO，然后，像读文件一样读取：

>>> **from** io **import** StringIO

>>> f = StringIO('Hello!\nHi!\nGoodbye!')

>>> **while** True:

... s = f.readline()

... **if** s == '':

... **break**

... print(s.strip())

...

Hello!

Hi!

Goodbye!

### BytesIO

StringIO操作的只能是str，如果要操作二进制数据，就需要使用BytesIO。

BytesIO实现了在内存中读写bytes，我们创建一个BytesIO，然后写入一些bytes：

>>> **from** io **import** BytesIO

>>> f = BytesIO()

>>> f.write('中文'.encode('utf-8'))

6

>>> print(f.getvalue())

b'\xe4\xb8\xad\xe6\x96\x87'

请注意，写入的不是str，而是经过UTF-8编码的bytes。

和StringIO类似，可以用一个bytes初始化BytesIO，然后，像读文件一样读取：

>>> **from** io **import** StringIO

>>> f = BytesIO(b'\xe4\xb8\xad\xe6\x96\x87')

>>> f.read()

b'\xe4\xb8\xad\xe6\x96\x87'

### 小结

StringIO和BytesIO是在内存中操作str和bytes的方法，使得和读写文件具有一致的接口。

## 第52课 操作文件和目录

如果我们要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如dir、cp等命令。

如果要在Python程序中执行这些目录和文件的操作怎么办？其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python内置的os模块也可以直接调用操作系统提供的接口函数。

打开Python交互式命令行，我们来看看如何使用os模块的基本功能：

>>> **import** os

>>> os.name *# 操作系统类型*

'posix'

如果是posix，说明系统是Linux、Unix或Mac OS X，如果是nt，就是Windows系统。

要获取详细的系统信息，可以调用uname()函数：

>>> os.uname()

posix.uname\_result(sysname='Darwin', nodename='MichaelMacPro.local', release='14.3.0', version='Darwin Kernel Version 14.3.0: Mon Mar 23 11:59:05 PDT 2015; root:xnu-2782.20.48~5/RELEASE\_X86\_64', machine='x86\_64')

注意uname()函数在Windows上不提供，也就是说，os模块的某些函数是跟操作系统相关的。

环境变量

在操作系统中定义的环境变量，全部保存在os.environ这个变量中，可以直接查看：

>>> os.environ

environ({'VERSIONER\_PYTHON\_PREFER\_32\_BIT': 'no', 'TERM\_PROGRAM\_VERSION': '326', 'LOGNAME': 'michael', 'USER': 'michael', 'PATH': '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/mysql/bin', ...})

要获取某个环境变量的值，可以调用os.environ.get('key')：

>>> os.environ.get('PATH')

'/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/mysql/bin'

>>> os.environ.get('x', 'default')

'default'

操作文件和目录

操作文件和目录的函数一部分放在os模块中，一部分放在os.path模块中，这一点要注意一下。查看、创建和删除目录可以这么调用：

*# 查看当前目录的绝对路径:*

>>> os.path.abspath('.')

'/Users/michael'

*# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来:*

>>> os.path.join('/Users/michael', 'testdir')

'/Users/michael/testdir'

*# 然后创建一个目录:*

>>> os.mkdir('/Users/michael/testdir')

*# 删掉一个目录:*

>>> os.rmdir('/Users/michael/testdir')

把两个路径合成一个时，不要直接拼字符串，而要通过os.path.join()函数，这样可以正确处理不同操作系统的路径分隔符。在Linux/Unix/Mac下，os.path.join()返回这样的字符串：

part-1/part-2

而Windows下会返回这样的字符串：

part-1\part-2

同样的道理，要拆分路径时，也不要直接去拆字符串，而要通过os.path.split()函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

>>> os.path.split('/Users/michael/testdir/file.txt')

('/Users/michael/testdir', 'file.txt')

os.path.splitext()可以直接让你得到文件扩展名，很多时候非常方便：

>>> os.path.splitext('/path/to/file.txt')

('/path/to/file', '.txt')

这些合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

文件操作使用下面的函数。假定当前目录下有一个test.txt文件：

*# 对文件重命名:*

>>> os.rename('test.txt', 'test.py')

*# 删掉文件:*

>>> os.remove('test.py')

但是复制文件的函数居然在os模块中不存在！原因是复制文件并非由操作系统提供的系统调用。理论上讲，我们通过上一节的读写文件可以完成文件复制，只不过要多写很多代码。

幸运的是shutil模块提供了copyfile()的函数，你还可以在shutil模块中找到很多实用函数，它们可以看做是os模块的补充。

最后看看如何利用Python的特性来过滤文件。比如我们要列出当前目录下的所有目录，只需要一行代码：

>>> [x **for** x **in** os.listdir('.') **if** os.path.isdir(x)]

['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash', '.vim', 'Applications', 'Desktop', ...]

要列出所有的.py文件，也只需一行代码：

>>> [x **for** x **in** os.listdir('.') **if** os.path.isfile(x) **and** os.path.splitext(x)[1]=='.py']

['apis.py', 'config.py', 'models.py', 'pymonitor.py', 'test\_db.py', 'urls.py', 'wsgiapp.py']

是不是非常简洁？

小结

Python的os模块封装了操作系统的目录和文件操作，要注意这些函数有的在os模块中，有的在os.path模块中。

## 第53课 序列化

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个dict：

d = dict(name='Bob', age=20, score=88)

可以随时修改变量，比如把name改成'Bill'，但是一旦程序结束，变量所占用的内存就被操作系统全部回收。如果没有把修改后的'Bill'存储到磁盘上，下次重新运行程序，变量又被初始化为'Bob'。

我们把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称之为serialization，marshalling，flattening等等，都是一个意思。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python提供了pickle模块来实现序列化。

首先，我们尝试把一个对象序列化并写入文件：

>>> **import** pickle

>>> d = dict(name='Bob', age=20, score=88)

>>> pickle.dumps(d)

b'\x80\x03}q\x00(X\x03\x00\x00\x00ageq\x01K\x14X\x05\x00\x00\x00scoreq\x02KXX\x04\x00\x00\x00nameq\x03X\x03\x00\x00\x00Bobq\x04u.'

pickle.dumps()方法把任意对象序列化成一个bytes，然后，就可以把这个bytes写入文件。或者用另一个方法pickle.dump()直接把对象序列化后写入一个file-like Object：

>>> f = open('dump.txt', 'wb')

>>> pickle.dump(d, f)

>>> f.close()

看看写入的dump.txt文件，一堆乱七八糟的内容，这些都是Python保存的对象内部信息。

当我们要把对象从磁盘读到内存时，可以先把内容读到一个bytes，然后用pickle.loads()方法反序列化出对象，也可以直接用pickle.load()方法从一个file-like Object中直接反序列化出对象。我们打开另一个Python命令行来反序列化刚才保存的对象：

>>> f = open('dump.txt', 'rb')

>>> d = pickle.load(f)

>>> f.close()

>>> d

{'age': 20, 'score': 88, 'name': 'Bob'}

变量的内容又回来了！

当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。

Pickle的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于Python，并且可能不同版本的Python彼此都不兼容，因此，只能用Pickle保存那些不重要的数据，不能成功地反序列化也没关系。

### JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型对应如下：

|  |  |
| --- | --- |
| JSON类型 | Python类型 |
| {} | dict |
| [] | list |
| "string" | str |
| 1234.56 | int或float |
| true/false | True/False |
| null | None |

Python内置的json模块提供了非常完善的Python对象到JSON格式的转换。我们先看看如何把Python对象变成一个JSON：

>>> **import** json

>>> d = dict(name='Bob', age=20, score=88)

>>> json.dumps(d)

'{"age": 20, "score": 88, "name": "Bob"}'

dumps()方法返回一个str，内容就是标准的JSON。类似的，dump()方法可以直接把JSON写入一个file-like Object。

要把JSON反序列化为Python对象，用loads()或者对应的load()方法，前者把JSON的字符串反序列化，后者从file-like Object中读取字符串并反序列化：

>>> json\_str = '{"age": 20, "score": 88, "name": "Bob"}'

>>> json.loads(json\_str)

{'age': 20, 'score': 88, 'name': 'Bob'}

由于JSON标准规定JSON编码是UTF-8，所以我们总是能正确地在Python的str与JSON的字符串之间转换。

### JSON进阶

Python的dict对象可以直接序列化为JSON的{}，不过，很多时候，我们更喜欢用class表示对象，比如定义Student类，然后序列化：

**import** json

**class Student(object):**

**def** **\_\_init\_\_**(self, name, age, score):

self.name = name

self.age = age

self.score = score

s = Student('Bob', 20, 88)

print(json.dumps(s))

运行代码，毫不留情地得到一个TypeError：

Traceback (most recent **call** **last**):

...

TypeError: <\_\_main\_\_.Student object **at** 0x10603cc50> **is** **not** JSON serializable

错误的原因是Student对象不是一个可序列化为JSON的对象。

如果连class的实例对象都无法序列化为JSON，这肯定不合理！

别急，我们仔细看看dumps()方法的参数列表，可以发现，除了第一个必须的obj参数外，dumps()方法还提供了一大堆的可选参数：

<https://docs.python.org/3/library/json.html#json.dumps>

这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把Student类实例序列化为JSON，是因为默认情况下，dumps()方法不知道如何将Student实例变为一个JSON的{}对象。

可选参数default就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为Student专门写一个转换函数，再把函数传进去即可：

**def** **student2dict**(std):

**return** {

'name': std.name,

'age': std.age,

'score': std.score

}

这样，Student实例首先被student2dict()函数转换成dict，然后再被顺利序列化为JSON：

>>> **print**(json.dumps(s, **default**=student2dict))

{"age": 20, "name": "Bob", "score": 88}

不过，下次如果遇到一个Teacher类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意class的实例变为dict：

**print**(json.dumps(s, **default**=lambda obj: obj.\_\_dict\_\_))

因为通常class的实例都有一个\_\_dict\_\_属性，它就是一个dict，用来存储实例变量。也有少数例外，比如定义了\_\_slots\_\_的class。

同样的道理，如果我们要把JSON反序列化为一个Student对象实例，loads()方法首先转换出一个dict对象，然后，我们传入的object\_hook函数负责把dict转换为Student实例：

**def** **dict2student**(d):

**return** Student(d['name'], d['age'], d['score'])

运行结果如下：

>>> json\_str = '{"age": 20, "score": 88, "name": "Bob"}'

>>> print(json.loads(json\_str, object\_hook=dict2student))

<\_\_main\_\_.Student object at 0x10cd3c190>

打印出的是反序列化的Student实例对象。

### 小结

Python语言特定的序列化模块是pickle，但如果要把序列化搞得更通用、更符合Web标准，就可以使用json模块。

json模块的dumps()和loads()函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性。

## 第54课 进程和线程

很多同学都听说过，现代操作系统比如Mac OS X，UNIX，Linux，Windows等，都是支持“多任务”的操作系统。

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听MP3，一边在用Word赶作业，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个Word就启动了一个Word进程。

有些进程还不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像Word这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多核CPU才可能实现。

我们前面编写的所有的Python程序，都是执行单任务的进程，也就是只有一个线程。如果我们要同时执行多个任务怎么办？

有两种解决方案：

一种是启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。

还有一种方法是启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。

当然还有第三种方法，就是启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有3种方式：

* 多进程模式；
* 多线程模式；
* 多进程+多线程模式。

同时执行多个任务通常各个任务之间并不是没有关联的，而是需要相互通信和协调，有时，任务1必须暂停等待任务2完成后才能继续执行，有时，任务3和任务4又不能同时执行，所以，多进程和多线程的程序的复杂度要远远高于我们前面写的单进程单线程的程序。

因为复杂度高，调试困难，所以，不是迫不得已，我们也不想编写多任务。但是，有很多时候，没有多任务还真不行。想想在电脑上看电影，就必须由一个线程播放视频，另一个线程播放音频，否则，单线程实现的话就只能先把视频播放完再播放音频，或者先把音频播放完再播放视频，这显然是不行的。

Python既支持多进程，又支持多线程，我们会讨论如何编写这两种多任务程序。

### 小结

线程是最小的执行单元，而进程由至少一个线程组成。如何调度进程和线程，完全由操作系统决定，程序自己不能决定什么时候执行，执行多长时间。

多进程和多线程的程序涉及到同步、数据共享的问题，编写起来更复杂。

## 第55课 多进程

要让Python程序实现多进程（multiprocessing），我们先了解操作系统的相关知识。

Unix/Linux操作系统提供了一个fork()系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是fork()调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回0，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用getppid()就可以拿到父进程的ID。

Python的os模块封装了常见的系统调用，其中就包括fork，可以在Python程序中轻松创建子进程：

**import** os

print('Process (%s) start...' % os.getpid())

*# Only works on Unix/Linux/Mac:*

pid = os.fork() #返回两个进程

**if** 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))

运行结果如下：

Process (876) **start**...

I (876) just created a child process (877).

I am child process (877) **and** my parent **is** 876.

由于Windows没有fork调用，上面的代码在Windows上无法运行。由于Mac系统是基于BSD（Unix的一种）内核，所以，在Mac下运行是没有问题的，推荐大家用Mac学Python！

有了fork调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的Apache服务器就是由父进程监听端口，每当有新的http请求时，就fork出子进程来处理新的http请求。

### multiprocessing

如果你打算编写多进程的服务程序，Unix/Linux无疑是正确的选择。由于Windows没有fork调用，难道在Windows上无法用Python编写多进程的程序？

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。multiprocessing模块就是跨平台版本的多进程模块。

multiprocessing模块提供了一个Process类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结束：

**from** multiprocessing **import** Process

**import** os

*# 子进程要执行的代码*

**def** **run\_proc**(name):

print('Run child process %s (%s)...' % (name, os.getpid()))

**if** \_\_name\_\_=='\_\_main\_\_':

print('Parent process %s.' % os.getpid())

p = Process(target=run\_proc, args=('test',))

print('Child process will start.')

p.start()

p.join()

print('Child process end.')

执行结果如下：

Parent process 928.

Process will **start**.

Run child process test (929)...

Process **end**.

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动，这样创建进程比fork()还要简单。

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

### Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

**from** multiprocessing **import** Pool

**import** os, time, random

**def** **long\_time\_task**(name):

print('Run task %s (%s)...' % (name, os.getpid()))

start = time.time()

time.sleep(random.random() \* 3)

end = time.time()

print('Task %s runs %0.2f seconds.' % (name, (end - start)))

**if** \_\_name\_\_=='\_\_main\_\_':

print('Parent process %s.' % os.getpid())

p = Pool(4)

**for** i **in** range(5):

p.apply\_async(long\_time\_task, args=(i,))

print('Waiting for all subprocesses done...')

p.close()

p.join()

print('All subprocesses done.')

执行结果如下：

**Parent** process 669.

Waiting **for** all subprocesses done...

Run task 0 (671)...

Run task 1 (672)...

Run task 2 (673)...

Run task 3 (674)...

Task 2 runs 0.14 seconds.

Run task 4 (673)...

Task 1 runs 0.27 seconds.

Task 3 runs 0.86 seconds.

Task 0 runs 1.41 seconds.

Task 4 runs 1.91 seconds.

All subprocesses done.

代码解读：

对Pool对象调用join()方法会等待所有子进程执行完毕，调用join()之前必须先调用close()，调用close()之后就不能继续添加新的Process了。

请注意输出的结果，task 0，1，2，3是立刻执行的，而task 4要等待前面某个task完成后才执行，这是因为Pool的默认大小在我的电脑上是4，因此，最多同时执行4个进程。这是Pool有意设计的限制，并不是操作系统的限制。如果改成：

p = Pool(5)

就可以同时跑5个进程。

由于Pool的默认大小是CPU的核数，如果你不幸拥有8核CPU，你要提交至少9个子进程才能看到上面的等待效果。

### 子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。

subprocess模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。

下面的例子演示了如何在Python代码中运行命令nslookup www.python.org，这和命令行直接运行的效果是一样的：

**import** subprocess

print('$ nslookup www.python.org')

r = subprocess.call(['nslookup', 'www.python.org'])

print('Exit code:', r)

运行结果：

$ nslookup www.python.org

Server: 192.168.19.4

Address: 192.168.19.4*#53*

Non-authoritative answer:

www.python.org canonical name = python.map.fastly.net.

Name: python.map.fastly.net

Address: 199.27.79.223

Exit code: 0

如果子进程还需要输入，则可以通过communicate()方法输入：

**import** subprocess

print('$ nslookup')

p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

output, err = p.communicate(b'set q=mx\npython.org\nexit\n')

print(output.decode('utf-8'))

print('Exit code:', p.returncode)

上面的代码相当于在命令行执行命令nslookup，然后手动输入：

**set** q=mx

python.org

exit

运行结果如下：

$ nslookup

Server: 192.168.19.4

Address: 192.168.19.4*#53*

Non-authoritative answer:

python.org mail exchanger = 50 mail.python.org.

Authoritative answers can be found from:

mail.python.org internet address = 82.94.164.166

mail.python.org has AAAA address 2001:888:2000:d::a6

Exit code: 0

### 进程间通信

Process之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python的multiprocessing模块包装了底层的机制，提供了Queue、Pipes等多种方式来交换数据。

我们以Queue为例，在父进程中创建两个子进程，一个往Queue里写数据，一个从Queue里读数据：

**from** multiprocessing **import** Process, Queue

**import** os, time, random

*# 写数据进程执行的代码:*

**def** **write**(q):

print('Process to write: %s' % os.getpid())

**for** value **in** ['A', 'B', 'C']:

print('Put %s to queue...' % value)

q.put(value)

time.sleep(random.random())

*# 读数据进程执行的代码:*

**def** **read**(q):

print('Process to read: %s' % os.getpid())

**while** True:

value = q.get(True)

print('Get %s from queue.' % value)

**if** \_\_name\_\_=='\_\_main\_\_':

*# 父进程创建Queue，并传给各个子进程：*

q = Queue()

pw = Process(target=write, args=(q,))

pr = Process(target=read, args=(q,))

*# 启动子进程pw，写入:*

pw.start()

*# 启动子进程pr，读取:*

pr.start()

*# 等待pw结束:*

pw.join()

*# pr进程里是死循环，无法等待其结束，只能强行终止:*

pr.terminate()

运行结果如下：

Process to write: 50563

Put A to queue...

Process to read: 50564

Get A **from** queue.

Put B to queue...

Get B **from** queue.

Put C to queue...

Get C **from** queue.

在Unix/Linux下，multiprocessing模块封装了fork()调用，使我们不需要关注fork()的细节。由于Windows没有fork调用，因此， multiprocessing需要“模拟”出fork的效果，父进程所有Python对象都必须通过pickle序列化再传到子进程去，所有，如果multiprocessing在Windows下调用失败了，要先考虑是不是pickle失败了。

### 小结

在Unix/Linux下，可以使用fork()调用实现多进程。

要实现跨平台的多进程，可以使用multiprocessing模块。

进程间通信是通过Queue、Pipes等实现的。

## 第56课 多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。

我们前面提到了进程是由若干线程组成的，一个进程至少有一个线程。

由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python也不例外，并且，Python的线程是真正的Posix Thread，而不是模拟出来的线程。

Python的标准库提供了两个模块：\_thread和threading，\_thread是低级模块，threading是高级模块，对\_thread进行了封装。绝大多数情况下，我们只需要使用threading这个高级模块。

启动一个线程就是把一个函数传入并创建Thread实例，然后调用start()开始执行：

**import** time, threading

*# 新线程执行的代码:*

**def** **loop**():

print('thread %s is running...' % threading.current\_thread().name)

n = 0

**while** n < 5:

n = n + 1

print('thread %s >>> %s' % (threading.current\_thread().name, n))

time.sleep(1)

print('thread %s ended.' % threading.current\_thread().name)

print('thread %s is running...' % threading.current\_thread().name)

t = threading.Thread(target=loop, name='LoopThread')

t.start()

t.join()

print('thread %s ended.' % threading.current\_thread().name)

执行结果如下：

thread MainThread **is** running...

thread LoopThread **is** running...

thread LoopThread >>> 1

thread LoopThread >>> 2

thread LoopThread >>> 3

thread LoopThread >>> 4

thread LoopThread >>> 5

thread LoopThread ended.

thread MainThread ended.

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python的threading模块有个current\_thread()函数，它永远返回当前线程的实例。主线程实例的名字叫MainThread，子线程的名字在创建时指定，我们用LoopThread命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，如果不起名字Python就自动给线程命名为Thread-1，Thread-2……

### Lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

来看看多个线程同时操作一个变量怎么把内容给改乱了：

**import** time, threading

*# 假定这是你的银行存款:*

balance = 0

**def** **change\_it**(n):

*# 先存后取，结果应该为0:*

**global** balance

balance = balance + n

balance = balance - n

**def** **run\_thread**(n):

**for** i **in** range(100000):

change\_it(n)

t1 = threading.Thread(target=run\_thread, args=(5,))

t2 = threading.Thread(target=run\_thread, args=(8,))

t1.start()

t2.start()

t1.join()

t2.join()

print(balance)

我们定义了一个共享变量balance，初始值为0，并且启动两个线程，先存后取，理论上结果应该为0，但是，由于线程的调度是由操作系统决定的，当t1、t2交替执行时，只要循环次数足够多，balance的结果就不一定是0了。

原因是因为高级语言的一条语句在CPU执行时是若干条语句，即使一个简单的计算：

balance = balance + n

也分两步：

1. 计算balance + n，存入临时变量中；
2. 将临时变量的值赋给balance。

也就是可以看成：

x = balance + n

balance = x

由于x是局部变量，两个线程各自都有自己的x，当代码正常执行时：

初始值 balance = 0

t1: x1 = balance + 5 *# x1 = 0 + 5 = 5*

t1: balance = x1 *# balance = 5*

t1: x1 = balance - 5 *# x1 = 5 - 5 = 0*

t1: balance = x1 *# balance = 0*

t2: x2 = balance + 8 *# x2 = 0 + 8 = 8*

t2: balance = x2 *# balance = 8*

t2: x2 = balance - 8 *# x2 = 8 - 8 = 0*

t2: balance = x2 *# balance = 0*

结果 balance = 0

但是t1和t2是交替运行的，如果操作系统以下面的顺序执行t1、t2：

初始值 balance = 0

t1: x1 = balance + 5 *# x1 = 0 + 5 = 5*

t2: x2 = balance + 8 *# x2 = 0 + 8 = 8*

t2: balance = x2 *# balance = 8*

t1: balance = x1 *# balance = 5*

t1: x1 = balance - 5 *# x1 = 5 - 5 = 0*

t1: balance = x1 *# balance = 0*

t2: x2 = balance - 8 *# x2 = 0 - 8 = -8*

t2: balance = x2 *# balance = -8*

结果 balance = -8

究其原因，是因为修改balance需要多条语句，而执行这几条语句时，线程可能中断，从而导致多个线程把同一个对象的内容改乱了。

两个线程同时一存一取，就可能导致余额不对，你肯定不希望你的银行存款莫名其妙地变成了负数，所以，我们必须确保一个线程在修改balance的时候，别的线程一定不能改。

如果我们要确保balance计算正确，就要给change\_it()上一把锁，当某个线程开始执行change\_it()时，我们说，该线程因为获得了锁，因此其他线程不能同时执行change\_it()，只能等待，直到锁被释放后，获得该锁以后才能改。由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过threading.Lock()来实现：

balance = 0

lock = threading.Lock()

**def** **run\_thread**(n):

**for** i **in** range(100000):

*# 先要获取锁:*

lock.acquire()

**try**:

*# 放心地改吧:*

change\_it(n)

**finally**:

*# 改完了一定要释放锁:*

lock.release()

当多个线程同时执行lock.acquire()时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用try...finally来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了。其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部挂起，既不能执行，也无法结束，只能靠操作系统强制终止。

### 多核CPU

如果你不幸拥有一个多核CPU，你肯定在想，多核应该可以同时执行多个线程。

如果写一个死循环的话，会出现什么情况呢？

打开Mac OS X的Activity Monitor，或者Windows的Task Manager，都可以监控某个进程的CPU使用率。

我们可以监控到一个死循环线程会100%占用一个CPU。

如果有两个死循环线程，在多核CPU中，可以监控到会占用200%的CPU，也就是占用两个CPU核心。

要想把N核CPU的核心全部跑满，就必须启动N个死循环线程。

试试用Python写个死循环：

**import** threading, multiprocessing

**def** **loop**():

x = 0

**while** True:

x = x ^ 1

**for** i **in** range(multiprocessing.cpu\_count()):

t = threading.Thread(target=loop)

t.start()

启动与CPU核心数量相同的N个线程，在4核CPU上可以监控到CPU占用率仅有102%，也就是仅使用了一核。

但是用C、C++或Java来改写相同的死循环，直接可以把全部核心跑满，4核就跑到400%，8核就跑到800%，为什么Python不行呢？

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非重写一个不带GIL的解释器。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。

不过，也不用过于担心，Python虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个Python进程有各自独立的GIL锁，互不影响。

### 小结

多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。

Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。多线程的并发在Python中就是一个美丽的梦。

注：python GIL <http://www.oschina.net/translate/pythons-hardest-problem>

## 第57课 ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而全局变量的修改必须加锁。

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

**def** **process\_student**(name):

std = Student(name)

*# std是局部变量，但是每个函数都要用它，因此必须传进去：*

do\_task\_1(std)

do\_task\_2(std)

**def** **do\_task\_1**(std):

do\_subtask\_1(std)

do\_subtask\_2(std)

**def** **do\_task\_2**(std):

do\_subtask\_2(std)

do\_subtask\_2(std)

每个函数一层一层调用都这么传参数那还得了？用全局变量？也不行，因为每个线程处理不同的Student对象，不能共享。

如果用一个全局dict存放所有的Student对象，然后以thread自身作为key获得线程对应的Student对象如何？

global\_dict = {}

**def** **std\_thread**(name):

std = Student(name)

*# 把std放到全局变量global\_dict中：*

global\_dict[threading.current\_thread()] = std

do\_task\_1()

do\_task\_2()

**def** **do\_task\_1**():

*# 不传入std，而是根据当前线程查找：*

std = global\_dict[threading.current\_thread()]

...

**def** **do\_task\_2**():

*# 任何函数都可以查找出当前线程的std变量：*

std = global\_dict[threading.current\_thread()]

...

这种方式理论上是可行的，它最大的优点是消除了std对象在每层函数中的传递问题，但是，每个函数获取std的代码有点丑。

有没有更简单的方式？

ThreadLocal应运而生，不用查找dict，ThreadLocal帮你自动做这件事：

**import** threading

*# 创建全局ThreadLocal对象:*

local\_school = threading.local()

**def** **process\_student**():

*# 获取当前线程关联的student:*

std = local\_school.student

print('Hello, %s (in %s)' % (std, threading.current\_thread().name))

**def** **process\_thread**(name):

*# 绑定ThreadLocal的student:*

local\_school.student = name

process\_student()

t1 = threading.Thread(target= process\_thread, args=('Alice',), name='Thread-A')

t2 = threading.Thread(target= process\_thread, args=('Bob',), name='Thread-B')

t1.start()

t2.start()

t1.join()

t2.join()

执行结果：

Hello, Alice (**in** Thread-A)

Hello, Bob (**in** Thread-B)

全局变量local\_school就是一个ThreadLocal对象，每个Thread对它都可以读写student属性，但互不影响。你可以把local\_school看成全局变量，但每个属性如local\_school.student都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题，ThreadLocal内部会处理。

可以理解为全局变量local\_school是一个dict，不但可以用local\_school.student，还可以绑定其他变量，如local\_school.teacher等等。

ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

### 小结

一个ThreadLocal变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。ThreadLocal解决了参数在一个线程中各个函数之间互相传递的问题。

## 第58课 进程 vs 线程

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常我们会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责分配任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用fork调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作，即将关闭”，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS的稳定性就不如Apache。为了缓解这个问题，IIS和Apache现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

### 线程切换

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？

我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核CPU执行多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备新环境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新任务的执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，操作系统可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于假死状态。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

### 计算密集型 vs. IO密集型

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

### 异步IO

考虑到CPU和IO之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待IO操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们才需要多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为**事件驱动模型（**鼠标的一个点击，移动，键盘的按键按下等等操作，都是对应操作系统的一个事件，然后应用程序接受你的操作进行处理**）**，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是一个主要的趋势。

对应到Python语言，单进程的异步编程模型称为**协程**，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

## 第59课 分布式进程\*

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的multiprocessing模块不但支持多进程，其中managers子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于managers模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

举个例子：如果我们已经有一个通过Queue通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务的进程分布到两台机器上。怎么用分布式进程实现？

原有的Queue可以继续使用，但是，通过managers模块把Queue通过网络暴露出去，就可以让其他机器的进程访问Queue了。

我们先看服务进程，服务进程负责启动Queue，把Queue注册到网络上，然后往Queue里面写入任务：

*# task\_master.py*

**import** random, time, queue

**from** multiprocessing.managers **import** BaseManager

*# 发送任务的队列:*

task\_queue = queue.Queue()

*# 接收结果的队列:*

result\_queue = queue.Queue()

*# 从BaseManager继承的QueueManager:*

**class QueueManager(BaseManager):**

**pass**

*# 把两个Queue都注册到网络上, callable参数关联了Queue对象:*

QueueManager.register('get\_task\_queue', callable=**lambda**: task\_queue)

QueueManager.register('get\_result\_queue', callable=**lambda**: result\_queue)

*# 绑定端口5000, 设置验证码'abc':*

manager = QueueManager(address=('', 5000), authkey=b'abc')

*# 启动Queue:*

manager.start()

*# 获得通过网络访问的Queue对象:*

task = manager.get\_task\_queue()

result = manager.get\_result\_queue()

*# 放几个任务进去:*

**for** i **in** range(10):

n = random.randint(0, 10000)

print('Put task %d...' % n)

task.put(n)

*# 从result队列读取结果:*

print('Try get results...')

**for** i **in** range(10):

r = result.get(timeout=10)

print('Result: %s' % r)

*# 关闭:*

manager.shutdown()

print('master exit.')

请注意，当我们在一台机器上写多进程程序时，创建的Queue可以直接拿来用，但是，在分布式多进程环境下，添加任务到Queue不可以直接对原始的task\_queue进行操作，那样就绕过了QueueManager的封装，必须通过manager.get\_task\_queue()获得的Queue接口添加。

然后，在另一台机器上启动任务进程（本机上启动也可以）：

*# task\_worker.py*

**import** time, sys, queue

**from** multiprocessing.managers **import** BaseManager

*# 创建类似的QueueManager:*

**class QueueManager(BaseManager):**

**pass**

*# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:*

QueueManager.register('get\_task\_queue')

QueueManager.register('get\_result\_queue')

*# 连接到服务器，也就是运行task\_master.py的机器:*

server\_addr = '127.0.0.1'

print('Connect to server %s...' % server\_addr)

*# 端口和验证码注意保持与task\_master.py设置的完全一致:*

m = QueueManager(address=(server\_addr, 5000), authkey=b'abc')

*# 从网络连接:*

m.connect()

*# 获取Queue的对象:*

task = m.get\_task\_queue()

result = m.get\_result\_queue()

*# 从task队列取任务,并把结果写入result队列:*

**for** i **in** range(10):

**try**:

n = task.get(timeout=1)

print('run task %d \* %d...' % (n, n))

r = '%d \* %d = %d' % (n, n, n\*n)

time.sleep(1)

result.put(r)

**except** Queue.Empty:

print('task queue is empty.')

*# 处理结束:*

print('worker exit.')

任务进程要通过网络连接到服务进程，所以要指定服务进程的IP。

现在，可以试试分布式进程的工作效果了。先启动task\_master.py服务进程：

$ python3 task\_master.py

Put task 3411...

Put task 1605...

Put task 1398...

Put task 4729...

Put task 5300...

Put task 7471...

Put task 68...

Put task 4219...

Put task 339...

Put task 7866...

Try get results...

task\_master.py进程发送完任务后，开始等待result队列的结果。现在启动task\_worker.py进程：

$ python3 task\_worker.py

Connect to server 127.0.0.1...

run task 3411 \* 3411...

run task 1605 \* 1605...

run task 1398 \* 1398...

run task 4729 \* 4729...

run task 5300 \* 5300...

run task 7471 \* 7471...

run task 68 \* 68...

run task 4219 \* 4219...

run task 339 \* 339...

run task 7866 \* 7866...

worker exit.

task\_worker.py进程结束，在task\_master.py进程中会继续打印出结果：

Result: 3411 \* 3411 = 11634921

Result: 1605 \* 1605 = 2576025

Result: 1398 \* 1398 = 1954404

Result: 4729 \* 4729 = 22363441

Result: 5300 \* 5300 = 28090000

Result: 7471 \* 7471 = 55815841

Result: 68 \* 68 = 4624

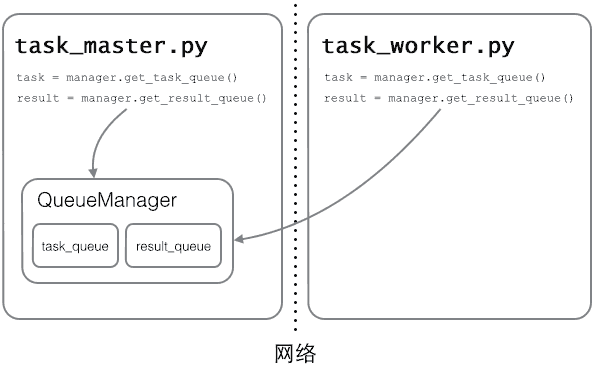
Result: 4219 \* 4219 = 17799961

Result: 339 \* 339 = 114921

Result: 7866 \* 7866 = 61873956

这个简单的Master/Worker模型有什么用？其实这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算n\*n的代码换成发送邮件，就实现了邮件队列的异步发送。

Queue对象存储在哪？注意到task\_worker.py中根本没有创建Queue的代码，所以，Queue对象存储在task\_master.py进程中：



而Queue之所以能通过网络访问，就是通过QueueManager实现的。由于QueueManager管理的不止一个Queue，所以，要给每个Queue的网络调用接口起个名字，比如get\_task\_queue。

authkey有什么用？这是为了保证两台机器正常通信，不被其他机器恶意干扰。如果task\_worker.py的authkey和task\_master.py的authkey不一致，肯定连接不上。

### 小结

Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。

注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。

## 第60课 正则表达式

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，虽然可以编程提取@前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的Email的方法是：

1. 创建一个匹配Email的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用\d可以匹配一个数字，\w可以匹配一个字母或数字，所以：

* '00\d'可以匹配'007'，但无法匹配'00A'；
* '\d\d\d'可以匹配'010'；
* '\w\w\d'可以匹配'py3'；

.可以匹配任意字符，所以：

* 'py.'可以匹配'pyc'、'pyo'、'py!'等等。

要匹配变长的字符，在正则表达式中，用\*表示任意个字符（包括0个），用+表示至少一个字符，用?表示0个或1个字符，用{n}表示n个字符，用{n,m}表示n-m个字符：

来看一个复杂的例子：\d{3}\s+\d{3,8}。

我们来从左到右解读一下：

1. \d{3}表示匹配3个数字，例如'010'；
2. \s可以匹配一个空格（也包括Tab等空白符），所以\s+表示至少有一个空格，例如匹配' '，' '等；
3. \d{3,8}表示3-8个数字，例如'1234567'。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配'010-12345'这样的号码呢？由于'-'是特殊字符，在正则表达式中，要用'\'转义，所以，上面的正则是\d{3}\-\d{3,8}。

但是，仍然无法匹配'010 - 12345'，因为带有空格。所以我们需要更复杂的匹配方式。

### 进阶

要做更精确地匹配，可以用[]表示范围，比如：

* [0-9a-zA-Z\\_]可以匹配一个数字、字母或者下划线；
* [0-9a-zA-Z\\_]+可以匹配至少由一个数字、字母或者下划线组成的字符串，比如'a100'，'0\_Z'，'Py3000'等等；
* [a-zA-Z\\_][0-9a-zA-Z\\_]\*可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；
* [a-zA-Z\\_][0-9a-zA-Z\\_]{0, 19}更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

A|B可以匹配A或B，所以[P|p]ython可以匹配'Python'或者'python'。

^表示行的开头，^\d表示必须以数字开头。

$表示行的结束，\d$表示必须以数字结束。

你可能注意到了，py也可以匹配'python'，但是加上^py$就变成了整行匹配，就只能匹配'py'了。

### re模块

有了准备知识，我们就可以在Python中使用正则表达式了。Python提供re模块，包含所有正则表达式的功能。由于Python的字符串本身也用\转义，所以要特别注意：

s = 'ABC\\-001' *# Python的字符串*

*# 对应的正则表达式字符串变成：*

*# 'ABC\-001'*

因此我们强烈建议使用Python的r前缀，就不用考虑转义的问题了：

s = r'ABC\-001' *# Python的字符串*

*# 对应的正则表达式字符串不变：*

*# 'ABC\-001'*

先看看如何判断正则表达式是否匹配：

>>> **import** re

>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')

<\_sre.SRE\_Match object; span=(0, 9), match='010-12345'>

>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')

>>>

match()方法判断是否匹配，如果匹配成功，返回一个Match对象，否则返回None。常见的判断方法就是：

test = '用户输入的字符串'

**if** re.match(r'正则表达式', test):

print('ok')

**else**:

print('failed')

### 切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

>>> 'a b c'.split(' ')

['a', 'b', '', '', 'c']

嗯，无法识别连续的空格，用正则表达式试试：

>>> re.split(r'\s+', 'a b c')

['a', 'b', 'c']

无论多少个空格都可以正常分割。加入,试试：

>>> re.split(r'[\s\,]+', 'a,b, c d')

['a', 'b', 'c', 'd']

再加入;试试：

>>> re.split(r'[\s\,\;]+', 'a,b;; c d')

['a', 'b', 'c', 'd']

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

### 分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用()表示的就是要提取的分组（Group）。比如：

^(\d{3})-(\d{3,8})$分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')

>>> m

<\_sre.SRE\_Match object; span=(0, 9), match='010-12345'>

>>> m.group(0)

'010-12345'

>>> m.group(1)

'010'

>>> m.group(2)

'12345'

如果正则表达式中定义了组，就可以在Match对象上用group()方法提取出子串来。

注意到group(0)永远是原始字符串，group(1)、group(2)……表示第1、2、……个子串。

提取子串非常有用。来看一个更凶残的例子：

>>> t = '19:05:30'

>>> m = re.match(r'^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])$', t)

>>> m.groups()

('19', '05', '30')

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

'^(0[1-9]|1[0-2]|[0-9])-(0[1-9]|1[0-9]|2[0-9]|3[0-1]|[0-9])$'

对于'2-30'，'4-31'这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

### 贪婪匹配

最后需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的0：

>>> re.match(r'^(\d+)(0\*)$', '102300').groups()

('102300', '')

由于\d+采用贪婪匹配，直接把后面的0全部匹配了，结果0\*只能匹配空字符串了。

必须让\d+采用非贪婪匹配（也就是尽可能少匹配），才能把后面的0匹配出来，加个?就可以让\d+采用非贪婪匹配：

>>> re.match(r'^(\d+?)(0\*)$', '102300').groups()

('1023', '00')

### 编译

当我们在Python中使用正则表达式时，re模块内部会干两件事情：

1. 编译正则表达式，如果正则表达式的字符串本身不合法，会报错；
2. 用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

>>> **import** re

*# 编译:*

>>> re\_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')

*# 使用：*

>>> re\_telephone.match('010-12345').groups()

('010', '12345')

>>> re\_telephone.match('010-8086').groups()

('010', '8086')

编译后生成Regular Expression对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

### 小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要讲清楚正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

## 第61课 访问数据库

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

|  |  |
| --- | --- |
| **名字** | **成绩** |
| Michael | 99 |
| Bob | 85 |
| Bart | 59 |
| Lisa | 87 |

你可以用一个文本文件保存，一行保存一个学生，用,隔开：

Michael,99

Bob,85

Bart,59

Lisa,87

你还可以用JSON格式保存，也是文本文件：

[

{"name":"Michael","score":99},

{"name":"Bob","score":85},

{"name":"Bart","score":59},

{"name":"Lisa","score":87}

]

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

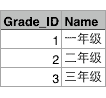
不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

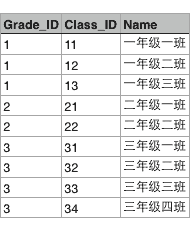
数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

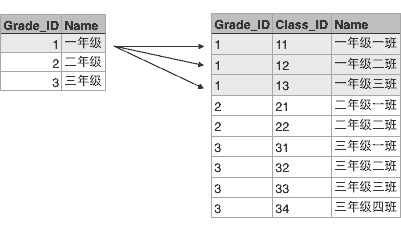
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：



每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：



这两个表格有个映射关系，就是根据Grade\_ID可以在班级表中查找到对应的所有班级：



也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

**SELECT** \* **FROM** classes **WHERE** grade\_id = '1';

结果也是一个表：

*---------+----------+----------*

grade\_id | class\_id | name

*---------+----------+----------*

1 | 11 | 一年级一班

*---------+----------+----------*

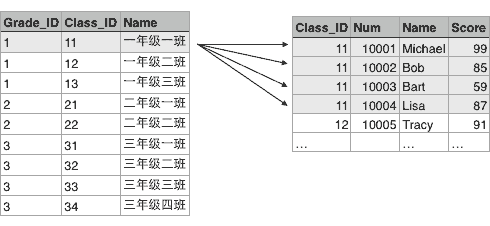
1 | 12 | 一年级二班

*---------+----------+----------*

1 | 13 | 一年级三班

*---------+----------+----------*

类似的，Class表的一行记录又可以关联到Student表的多行记录：



由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，推荐Coursera课程：

英文：<https://www.coursera.org/course/db>

中文：<http://c.open.163.com/coursera/courseIntro.htm?cid=12>

### NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

### 数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

* Oracle，典型的高富帅；
* SQL Server，微软自家产品，Windows定制专款；
* DB2，IBM的产品，听起来挺高端；
* Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

* MySQL，大家都在用，一般错不了；
* PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
* sqlite，嵌入式数据库，适合桌面和移动应用。

作为Python开发工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

为了能继续后面的学习，你需要从MySQL官方网站下载并安装[MySQL Community Server 5.6](http://dev.mysql.com/downloads/mysql/)，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。

## 第62课 使用SQLite

SQLite是一种嵌入式数据库，它的数据库就是一个文件。由于SQLite本身是C写的，而且体积很小，所以，经常被集成到各种应用程序中，甚至在iOS和Android的App中都可以集成。

Python就内置了SQLite3，所以，在Python中使用SQLite，不需要安装任何东西，直接使用。

在使用SQLite前，我们先要搞清楚几个概念：

表是数据库中存放关系数据的集合，一个数据库里面通常都包含多个表，比如学生的表，班级的表，学校的表，等等。表和表之间通过外键关联。

要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection；

连接到数据库后，需要打开游标，称之为Cursor，通过Cursor执行SQL语句，然后，获得执行结果。

Python定义了一套操作数据库的API接口，任何数据库要连接到Python，只需要提供符合Python标准的数据库驱动即可。

由于SQLite的驱动内置在Python标准库中，所以我们可以直接来操作SQLite数据库。

我们在Python交互式命令行实践一下：

*# 导入SQLite驱动:*

>>> **import** sqlite3

*# 连接到SQLite数据库*

*# 数据库文件是test.db*

*# 如果文件不存在，会自动在当前目录创建:*

>>> conn = sqlite3.connect('test.db')

*# 创建一个Cursor:*

>>> cursor = conn.cursor()

*# 执行一条SQL语句，创建user表:*

>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')

<sqlite3.Cursor object at 0x10f8aa260>

*# 继续执行一条SQL语句，插入一条记录:*

>>> cursor.execute('insert into user (id, name) values (\'1\', \'Michael\')')

<sqlite3.Cursor object at 0x10f8aa260>

*# 通过rowcount获得插入的行数:*

>>> cursor.rowcount

1

*# 关闭Cursor:*

>>> cursor.close()

*# 提交事务:*

>>> conn.commit()

*# 关闭Connection:*

>>> conn.close()

我们再试试查询记录：

>>> conn = sqlite3.connect('test.db')

>>> cursor = conn.cursor()

*# 执行查询语句:*

>>> cursor.execute('select \* from user where id=?', ('1',))

<sqlite3.Cursor object at 0x10f8aa340>

*# 获得查询结果集:*

>>> values = cursor.fetchall()

>>> values

[('1', 'Michael')]

>>> cursor.close()

>>> conn.close()

使用Python的DB-API时，只要搞清楚Connection和Cursor对象，打开后一定记得关闭，就可以放心地使用。

使用Cursor对象执行insert，update，delete语句时，执行结果由rowcount返回影响的行数，就可以拿到执行结果。

使用Cursor对象执行select语句时，通过featchall()可以拿到结果集。结果集是一个list，每个元素都是一个tuple，对应一行记录。

如果SQL语句带有参数，那么需要把参数按照位置传递给execute()方法，有几个?占位符就必须对应几个参数，例如：

cursor.execute('**select** \* **from** **user** **where** name=? **and** pwd=?', ('abc', 'password'))

SQLite支持常见的标准SQL语句以及几种常见的数据类型。具体文档请参阅SQLite官方网站。

小结

在Python中操作数据库时，要先导入数据库对应的驱动，然后，通过Connection对象和Cursor对象操作数据。

要确保打开的Connection对象和Cursor对象都正确地被关闭，否则，资源就会泄露。

如何才能确保出错的情况下也关闭掉Connection对象和Cursor对象呢？请回忆try:...except:...finally:...的用法。

## 第63课 使用mysql

MySQL是Web世界中使用最广泛的数据库服务器。SQLite的特点是轻量级、可嵌入，但不能承受高并发访问，适合桌面和移动应用。而MySQL是为服务器端设计的数据库，能承受高并发访问，同时占用的内存也远远大于SQLite。

此外，MySQL内部有多种数据库引擎，最常用的引擎是支持数据库事务的InnoDB。

（ACID，指数据库事务正确执行的四个基本要素的缩写。包含：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。一个支持事务（Transaction）的数据库，必需要具有这四种特性，否则在事务过程（Transaction processing）当中无法保证数据的正确性，交易过程极可能达不到交易方的要求。）

### 安装MySQL

可以直接从MySQL官方网站下载最新的[Community Server 5.6.x](http://dev.mysql.com/downloads/mysql/5.6.html)版本。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入root用户的口令，请务必记清楚。如果怕记不住，就把口令设置为password。

在Windows上，安装时请选择UTF-8编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在/etc/my.cnf或者/etc/mysql/my.cnf：

**[client]**

default-character-set = utf8

**[mysqld]**

default-storage-engine = INNODB

character-set-server = utf8

collation-server = utf8\_general\_ci

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

$ mysql -u root -p

Enter password:

Welcome to the MySQL monitor...

...

mysql> **show** variables **like** '%char%';

+*--------------------------+--------------------------------------------------------+*

| Variable\_name | Value |

+*--------------------------+--------------------------------------------------------+*

| character\_set\_client | utf8 |

| character\_set\_connection | utf8 |

| character\_set\_database | utf8 |

| character\_set\_filesystem | binary |

| character\_set\_results | utf8 |

| character\_set\_server | utf8 |

| character\_set\_system | utf8 |

| character\_sets\_dir | /usr/local/mysql-5.1.65-osx10.6-x86\_64/share/charsets/ |

+*--------------------------+--------------------------------------------------------+*

8 rows in **set** (0.00 sec)

看到utf8字样就表示编码设置正确。

注：如果MySQL的版本≥5.5.3，可以把编码设置为utf8mb4，utf8mb4和utf8完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

### 安装MySQL驱动

由于MySQL服务器以独立的进程运行，并通过网络对外服务，所以，需要支持Python的MySQL驱动来连接到MySQL服务器。MySQL官方提供了mysql-connector-python驱动，但是安装的时候需要给pip命令加上参数--allow-external：

$ pip install mysql-connector-python --allow-external mysql-connector-python

我们演示如何连接到MySQL服务器的test数据库：

*# 导入MySQL驱动:*

>>> **import** mysql.connector

*# 注意把password设为你的root口令:*

>>> conn = mysql.connector.connect(user='root', password='password', database='test')

>>> cursor = conn.cursor()

*# 创建user表:*

>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')

*# 插入一行记录，注意MySQL的占位符是%s:*

>>> cursor.execute('insert into user (id, name) values (%s, %s)', ['1', 'Michael'])

>>> cursor.rowcount

1

*# 提交事务:*

>>> conn.commit()

>>> cursor.close()

*# 运行查询:*

>>> cursor = conn.cursor()

>>> cursor.execute('select \* from user where id = %s', ('1',))

>>> values = cursor.fetchall()

>>> values

[('1', 'Michael')]

*# 关闭Cursor和Connection:*

>>> cursor.close()

True

>>> conn.close()

由于Python的DB-API定义都是通用的，所以，操作MySQL的数据库代码和SQLite类似。

### 小结

* 执行INSERT等操作后要调用commit()提交事务；
* MySQL的SQL占位符是%s。

## 第64课 使用SQLAlchemy\*

数据库表是一个二维表，包含多行多列。把一个表的内容用Python的数据结构表示出来的话，可以用一个list表示多行，list的每一个元素是tuple，表示一行记录，比如，包含id和name的user表：

[

('1', 'Michael'),

('2', 'Bob'),

('3', 'Adam')

]

Python的DB-API返回的数据结构就是像上面这样表示的。

但是用tuple表示一行很难看出表的结构。如果把一个tuple用class实例来表示，就可以更容易地看出表的结构来：

**class User(object):**

**def** **\_\_init\_\_**(self, id, name):

self.id = id

self.name = name

[

User('1', 'Michael'),

User('2', 'Bob'),

User('3', 'Adam')

]

这就是传说中的ORM技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

在Python中，最有名的ORM框架是SQLAlchemy。我们来看看SQLAlchemy的用法。

首先通过pip安装SQLAlchemy：

$ pip install sqlalchemy

然后，利用上次我们在MySQL的test数据库中创建的user表，用SQLAlchemy来试试：

第一步，导入SQLAlchemy，并初始化DBSession：

*# 导入:*

**from** sqlalchemy **import** Column, String, create\_engine

**from** sqlalchemy.orm **import** sessionmaker

**from** sqlalchemy.ext.declarative **import** declarative\_base

*# 创建对象的基类:*

Base = declarative\_base()

*# 定义User对象:*

**class User(Base):**

*# 表的名字:*

\_\_tablename\_\_ = 'user'

*# 表的结构:*

id = Column(String(20), primary\_key=True)

name = Column(String(20))

*# 初始化数据库连接:*

engine = create\_engine('mysql+mysqlconnector://root:password@localhost:3306/test')

*# 创建DBSession类型:*

DBSession = sessionmaker(bind=engine)

以上代码完成SQLAlchemy的初始化和具体每个表的class定义。如果有多个表，就继续定义其他class，例如School：

**class School(Base):**

\_\_tablename\_\_ = 'school'

id = ...

name = ...

create\_engine()用来初始化数据库连接。SQLAlchemy用一个字符串表示连接信息：

'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'

你只需要根据需要替换掉用户名、口令等信息即可。

下面，我们看看如何向数据库表中添加一行记录。

由于有了ORM，我们向数据库表中添加一行记录，可以视为添加一个User对象：

*# 创建session对象:*

session = DBSession()

*# 创建新User对象:*

new\_user = User(id='5', name='Bob')

*# 添加到session:*

session.add(new\_user)

*# 提交即保存到数据库:*

session.commit()

*# 关闭session:*

session.close()

可见，关键是获取session，然后把对象添加到session，最后提交并关闭。DBSession对象可视为当前数据库连接。

如何从数据库表中查询数据呢？有了ORM，查询出来的可以不再是tuple，而是User对象。SQLAlchemy提供的查询接口如下：

*# 创建Session:*

session = DBSession()

*# 创建Query查询，filter是where条件，最后调用one()返回唯一行，如果调用all()则返回所有行:*

user = session.query(User).filter(User.id=='5').one()

*# 打印类型和对象的name属性:*

**print**('type:', type(user))

**print**('name:', user.name)

*# 关闭Session:*

session.close()

运行结果如下：

type: <class '\_\_main\_\_.User'>

name: Bob

可见，ORM就是把数据库表的行与相应的对象建立关联，互相转换。

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联，相应地，ORM框架也可以提供两个对象之间的一对多、多对多等功能。

例如，如果一个User拥有多个Book，就可以定义一对多关系如下：

**class User(Base):**

\_\_tablename\_\_ = 'user'

id = Column(String(20), primary\_key=True)

name = Column(String(20))

*# 一对多:*

books = relationship('Book')

**class Book(Base):**

\_\_tablename\_\_ = 'book'

id = Column(String(20), primary\_key=True)

name = Column(String(20))

*# “多”的一方的book表是通过外键关联到user表的:*

user\_id = Column(String(20), ForeignKey('user.id'))

当我们查询一个User对象时，该对象的books属性将返回一个包含若干个Book对象的list。

### 小结

ORM框架的作用就是把数据库表的一行记录与一个对象互相做自动转换。

正确使用ORM的前提是了解关系数据库的原理。

## 第65课 动态类型（内存角度）

动态类型(dynamic typing)是Python另一个重要的核心概念。我们之前说过，Python的变量(variable)不需要声明，而在赋值时，变量可以重新赋值为任意值。这些都与动态类型的概念相关。

### 动态类型

在我们接触的对象中，有一类特殊的对象，是用于存储数据的。常见的该类对象包括各种数字，字符串，表，词典。在C语言中，我们称这样一些数据结构为变量。而在Python中，这些是对象。

对象是储存在内存中的实体。但我们并不能直接接触到该对象。我们在程序中写的对象名，只是指向这一对象的引用(reference)。

引用和对象分离，是动态类型的核心。引用可以随时指向一个新的对象：

a = 3

a = 'at'

第一个语句中，3是储存在内存中的一个整数对象。通过赋值，引用a指向对象3。

第二个语句中，内存中建立对象‘at’，是一个字符串(string)。引用a指向了'at'。此时，对象3不再有引用指向它。Python会自动将没有引用指向的对象销毁(destruct)，释放相应内存。

(对于小的整数和短字符串，Python会缓存这些对象，而不是频繁的建立和销毁。)

a = 5

b = a

a = a + 2

再看这个例子。通过前两个句子，我们让a,b指向同一个整数对象5(b = a的含义是让引用b指向引用a所指的那一个对象)。但第三个句子实际上对引用a重新赋值，让a指向一个新的对象7。此时a,b分别指向不同的对象。我们看到，即使是多个引用指向同一个对象，如果一个引用值发生变化，那么实际上是让这个引用指向一个新的引用，并不影响其他的引用的指向。从效果上看，就是各个引用各自独立，互不影响。

其它数据对象也是如此:

L1 = [1,2,3]

L2 = L1

L1 = 1

但注意以下情况

L1 = [1,2,3]

L2 = L1

L1[0] = 10

print L2

在该情况下，我们不再对L1这一引用赋值，而是对L1所指向的表的元素赋值。结果是，L2也同时发生变化。

原因何在呢？因为L1，L2的指向没有发生变化，依然指向那个表。表实际上是包含了多个引用的对象（每个引用是一个元素，比如L1[0]，L1[1]..., 每个引用指向一个对象，比如1,2,3), 。而L1[0] = 10这一赋值操作，并不是改变L1的指向，而是对L1[0], 也就是表对象的一部份(一个元素)，进行操作，所以所有指向该对象的引用都受到影响。

（与之形成对比的是，我们之前的赋值操作都没有对对象自身发生作用，只是改变引用指向。）

列表可以通过引用其元素，改变对象自身(in-place change)。这种对象类型，称为可变数据对象(mutable object)，词典也是这样的数据类型。

而像之前的数字和字符串，不能改变对象本身，只能改变引用的指向，称为不可变数据对象(immutable object)。

我们之前学的元组(tuple)，尽管可以调用引用元素，但不可以赋值，因此不能改变对象自身，所以也算是immutable object.

### 从动态类型看函数的参数传递

函数的参数传递，本质上传递的是引用。比如说：

[复制代码](javascript:void(0);)

def f(x):

x = 100

print x

a = 1

f(a)

print a

[复制代码](javascript:void(0);)

参数x是一个新的引用，指向a所指的对象。如果参数是不可变(immutable)的对象，a和x引用之间相互独立。对参数x的操作不会影响引用a。这样的传递类似于C语言中的值传递。

如果传递的是可变(mutable)的对象，那么改变函数参数，有可能改变原对象。所有指向原对象的引用都会受影响，编程的时候要对此问题留心。比如说：

[复制代码](javascript:void(0);)

def f(x):

x[0] = 100

print x

a = [1,2,3]

f(a)

print a

[复制代码](javascript:void(0);)

动态类型是Python的核心机制之一。可以在应用中慢慢熟悉。

### 总结

引用和对象的分离，对象是内存中储存数据的实体，引用指向对象。

可变对象，不可变对象

函数值传递

## 第66课 python内存管理机制

*作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！*

语言的内存管理是语言设计的一个重要方面。它是决定语言性能的重要因素。无论是C语言的手工管理，还是Java的垃圾回收，都成为语言最重要的特征。这里以Python语言为例子，说明一门动态类型的、面向对象的语言的内存管理方式。

### 对象的内存使用

赋值语句是语言最常见的功能了。但即使是最简单的赋值语句，也可以很有内涵。Python的赋值语句就很值得研究。

a = 1

整数1为一个对象。而a是一个引用。利用赋值语句，引用a指向对象1。Python是动态类型的语言(参考[动态类型](http://www.cnblogs.com/vamei/archive/2012/07/10/2582795.html))，对象与引用分离。Python像使用“筷子”那样，通过引用来接触和翻动真正的食物——对象。



 引用和对象

为了探索对象在内存的存储，我们可以求助于Python的内置函数id()。它用于返回对象的身份(identity)。其实，这里所谓的身份，就是该对象的内存地址。

a = 1

print(id(a))

print(hex(id(a)))

在我的计算机上，它们返回的是:

11246696  
'0xab9c68'

分别为内存地址的十进制和十六进制表示。

在Python中，整数和短小的字符，Python都会缓存这些对象，以便重复使用。当我们创建多个等于1的引用时，实际上是让所有这些引用指向同一个对象。

a = 1

b = 1

print(id(a))

print(id(b))

上面程序返回

11246696

11246696

可见a和b实际上是指向同一个对象的两个引用（内存地址，变量名？）。

为了检验两个引用指向同一个对象，我们可以用is关键字。is用于判断两个引用所指的对象是否相同。

[复制代码](javascript:void(0);)

# True

a = 1

b = 1

print(a is b)

# True

a = "good"

b = "good"

print(a is b)

# False

a = "very good morning"

b = "very good morning"

print(a is b)

# False

a = []

b = []

print(a is b)

[复制代码](javascript:void(0);)

上面的注释为相应的运行结果。可以看到，由于Python缓存了整数和短字符串，因此每个对象只存有一份。比如，所有整数1的引用都指向同一对象。即使使用赋值语句，也只是创造了新的引用，而不是对象本身。长的字符串和其它对象可以有多个相同的对象，可以使用赋值语句创建出新的对象。

在Python中，每个对象都有存有指向该对象的引用总数，即引用计数(reference count)。

我们可以使用sys包中的getrefcount()，来查看某个对象的引用计数。需要注意的是，当使用某个引用作为参数，传递给getrefcount()时，参数实际上创建了一个临时的引用。因此，getrefcount()所得到的结果，会比期望的多1。

[复制代码](javascript:void(0);)

from sys import getrefcount

a = [1, 2, 3]

print(getrefcount(a))  
  
b = a  
print(getrefcount(b))

[复制代码](javascript:void(0);)

由于上述原因，两个getrefcount将返回2和3，而不是期望的1和2。

### 对象引用对象

Python的一个容器对象(container)，比如表、词典等，可以包含多个对象。实际上，容器对象中包含的并不是元素对象本身，是指向各个元素对象的引用。

我们也可以自定义一个对象，并引用其它对象:

[复制代码](javascript:void(0);)

class from\_obj(object):

def \_\_init\_\_(self, to\_obj):

self.to\_obj = to\_obj

b = [1,2,3]

a = from\_obj(b)

print(id(a.to\_obj))

print(id(b))

[复制代码](javascript:void(0);)

可以看到，a引用了对象b。

对象引用对象，是Python最基本的构成方式。即使是a = 1这一赋值方式，实际上是让词典的一个键值"a"的元素引用整数对象1。该词典对象用于记录所有的全局引用。该词典引用了整数对象1。我们可以通过内置函数globals()来查看该词典。

当一个对象A被另一个对象B引用时，A的引用计数将增加1。

[复制代码](javascript:void(0);)

from sys import getrefcount

a = [1, 2, 3]

print(getrefcount(a))

b = [a, a]

print(getrefcount(a))

[复制代码](javascript:void(0);)

由于对象b引用了两次a，a的引用计数增加了2。

容器对象的引用可能构成很复杂的拓扑结构。我们可以用objgraph包来绘制其引用关系，比如

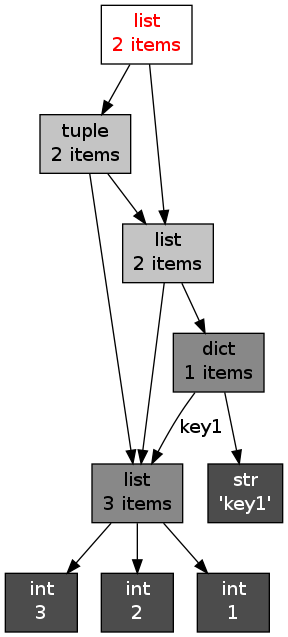
x = [1, 2, 3]

y = [x, dict(key1=x)]

z = [y, (x, y)]

import objgraph

objgraph.show\_refs([z], filename='ref\_topo.png')



objgraph是Python的一个第三方包。安装之前需要安装xdot。

sudo apt-get install xdot

sudo pip install objgraph

[objgraph官网](http://mg.pov.lt/objgraph/)

两个对象可能相互引用，从而构成所谓的引用环(reference cycle)。

a = []

b = [a]

a.append(b)

即使是一个对象，只需要自己引用自己，也能构成引用环。

a = []

a.append(a)

print(getrefcount(a))

引用环会给垃圾回收机制带来很大的麻烦，我将在后面详细叙述这一点。

### 引用减少

某个对象的引用计数可能减少。比如，可以使用del关键字删除某个引用:

[复制代码](javascript:void(0);)

from sys import getrefcount

a = [1, 2, 3]

b = a

print(getrefcount(b))

del a

print(getrefcount(b))

[复制代码](javascript:void(0);)

del也可以用于删除容器元素中的元素，比如:

a = [1,2,3]

del a[0]

print(a)

如果某个引用指向对象A，当这个引用被重新定向到某个其他对象B时，对象A的引用计数减少:

[复制代码](javascript:void(0);)

from sys import getrefcount  
  
a = [1, 2, 3]

b = a

print(getrefcount(b))

a = 1

print(getrefcount(b))

[复制代码](javascript:void(0);)

### 垃圾回收

吃太多，总会变胖，Python也是这样。当Python中的对象越来越多，它们将占据越来越大的内存。不过你不用太担心Python的体形，它会乖巧的在适当的时候“减肥”，启动垃圾回收(garbage collection)，将没用的对象清除。在许多语言中都有垃圾回收机制，比如Java和Ruby。尽管最终目的都是塑造苗条的提醒，但不同语言的减肥方案有很大的差异 (这一点可以对比本文和[Java内存管理与垃圾回收](http://www.cnblogs.com/vamei/archive/2013/04/28/3048353.html)

)。



从基本原理上，当Python的某个对象的引用计数降为0时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾了。比如某个新建对象，它被分配给某个引用，对象的引用计数变为1。如果引用被删除，对象的引用计数为0，那么该对象就可以被垃圾回收。比如下面的表:

a = [1, 2, 3]

del a

del a后，已经没有任何引用指向之前建立的[1, 2, 3]这个表。用户不可能通过任何方式接触或者动用这个对象。这个对象如果继续待在内存里，就成了不健康的脂肪。当垃圾回收启动时，Python扫描到这个引用计数为0的对象，就将它所占据的内存清空。

然而，减肥是个昂贵而费力的事情。垃圾回收时，Python不能进行其它的任务。频繁的垃圾回收将大大降低Python的工作效率。如果内存中的对象不多，就没有必要总启动垃圾回收。所以，Python只会在特定条件下，自动启动垃圾回收。当Python运行时，会记录其中分配对象(object allocation)和取消分配对象(object deallocation)的次数。当两者的差值高于某个阈值时，垃圾回收才会启动。

我们可以通过gc模块的get\_threshold()方法，查看该阈值:

import gc

print(gc.get\_threshold())

返回(700, 10, 10)，后面的两个10是与分代回收相关的阈值，后面可以看到。700即是垃圾回收启动的阈值。可以通过gc中的set\_threshold()方法重新设置。

我们也可以手动启动垃圾回收，即使用gc.collect()。

### 分代回收

Python同时采用了分代(generation)回收的策略。这一策略的基本假设是，存活时间越久的对象，越不可能在后面的程序中变成垃圾。我们的程序往往会产生大量的对象，许多对象很快产生和消失，但也有一些对象长期被使用。出于信任和效率，对于这样一些“长寿”对象，我们相信它们的用处，所以减少在垃圾回收中扫描它们的频率。



小家伙要多检查

Python将所有的对象分为0，1，2三代。所有的新建对象都是0代对象。当某一代对象经历过垃圾回收，依然存活，那么它就被归入下一代对象。垃圾回收启动时，一定会扫描所有的0代对象。如果0代经过一定次数垃圾回收，那么就启动对0代和1代的扫描清理。当1代也经历了一定次数的垃圾回收后，那么会启动对0，1，2，即对所有对象进行扫描。

这两个次数即上面get\_threshold()返回的(700, 10, 10)返回的两个10。也就是说，每10次0代垃圾回收，会配合1次1代的垃圾回收；而每10次1代的垃圾回收，才会有1次的2代垃圾回收。

同样可以用set\_threshold()来调整，比如对2代对象进行更频繁的扫描。

import gc

gc.set\_threshold(700, 10, 5)

### 孤立的引用环

引用环的存在会给上面的垃圾回收机制带来很大的困难。这些引用环可能构成无法使用，但引用计数不为0的一些对象。

a = []

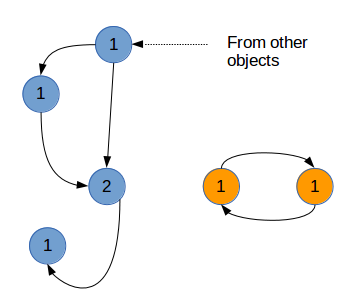
b = [a]

a.append(b)

del a

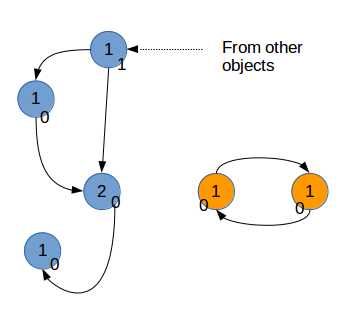
del b

上面我们先创建了两个表对象，并引用对方，构成一个引用环。删除了a，b引用之后，这两个对象不可能再从程序中调用，就没有什么用处了。但是由于引用环的存在，这两个对象的引用计数都没有降到0，不会被垃圾回收。



孤立的引用环

为了回收这样的引用环，Python复制每个对象的引用计数，可以记为gc\_ref。假设，每个对象i，该计数为gc\_ref\_i。Python会遍历所有的对象i。对于每个对象i引用的对象j，将相应的gc\_ref\_j减1。



遍历后的结果

在结束遍历后，gc\_ref不为0的对象，和这些对象引用的对象，以及继续更下游引用的对象，需要被保留。而其它的对象则被垃圾回收。

### 总结

Python作为一种动态类型的语言，其对象和引用分离。这与曾经的面向过程语言有很大的区别。为了有效的释放内存，Python内置了垃圾回收的支持。Python采取了一种相对简单的垃圾回收机制，即引用计数，并因此需要解决孤立引用环的问题。Python与其它语言既有共通性，又有特别的地方。对该内存管理机制的理解，是提高Python性能的重要一步。

### **附：引用计数变化；is与==**

引用计数增加

1.对象被创建：x=4

2.另外的别人被创建：y=x

3.被作为参数传递给函数：foo(x)

4.作为容器对象的一个元素：a=[1,x,'33']

引用计数减少

1.一个本地引用离开了它的作用域。比如上面的foo(x)函数结束时，x指向的对象引用减1。

2.对象的别名被显式的销毁：del x ；或者del y

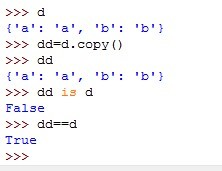
3.对象的一个别名被赋值给其他对象：x=789

4.对象从一个窗口对象中移除：myList.remove(x)

5.窗口对象本身被销毁：del myList，或者窗口对象本身离开了作用域。

python中的对象包含三要素：id、type、value。  
其中id用来唯一标识一个对象，type标识对象的类型，value是对象的值。  
is判断的是a对象是否就是b对象，是通过id来判断的。  
==判断的是a对象的值是否和b对象的值相等，是通过value来判断的。

看下面的例子：



## 第67课 常用内置函数

## [Python的zip函数](http://www.cnblogs.com/frydsh/archive/2012/07/10/2585370.html)

zip函数接受任意多个（包括0个和1个）序列作为参数，返回一个tuple列表。具体意思不好用文字来表述，直接看示例：

1.示例1：

[复制代码](javascript:void(0);)

x = [1, 2, 3]

y = [4, 5, 6]

z = [7, 8, 9]

xyz = zip(x, y, z)

print xyz

[复制代码](javascript:void(0);)

运行的结果是：

[(1, 4, 7), (2, 5, 8), (3, 6, 9)]

从这个结果可以看出zip函数的基本运作方式。

2.示例2：

x = [1, 2, 3]

y = [4, 5, 6, 7]

xy = zip(x, y)

print xy

运行的结果是：

[(1, 4), (2, 5), (3, 6)]

从这个结果可以看出zip函数的长度处理方式。

3.示例3：

x = [1, 2, 3]

x = zip(x)

print x

运行的结果是：

[(1,), (2,), (3,)]

从这个结果可以看出zip函数在只有一个参数时运作的方式。

4.示例4：

x = zip()

print x

运行的结果是：

[]

从这个结果可以看出zip函数在没有参数时运作的方式。

5.示例5：

[复制代码](javascript:void(0);)

x = [1, 2, 3]

y = [4, 5, 6]

z = [7, 8, 9]

xyz = zip(x, y, z)

u = zip(\*xyz)

print u

[复制代码](javascript:void(0);)

运行的结果是：

[(1, 2, 3), (4, 5, 6), (7, 8, 9)]

一般认为这是一个unzip的过程，它的运行机制是这样的：

在运行zip(\*xyz)之前，xyz的值是：[(1, 4, 7), (2, 5, 8), (3, 6, 9)]

那么，zip(\*xyz) 等价于 zip((1, 4, 7), (2, 5, 8), (3, 6, 9))

所以，运行结果是：[(1, 2, 3), (4, 5, 6), (7, 8, 9)]

注：在函数调用中使用\*list/tuple的方式表示将list/tuple分开，作为位置参数传递给对应函数（前提是对应函数支持不定个数的位置参数）

6.示例6：

x = [1, 2, 3]

r = zip(\* [x] \* 3)

print r

运行的结果是：

[(1, 1, 1), (2, 2, 2), (3, 3, 3)]

它的运行机制是这样的：

[x]生成一个列表的列表，它只有一个元素x

[x] \* 3生成一个列表的列表，它有3个元素，[x, x, x]

zip(\* [x] \* 3)的意思就明确了，zip(x, x, x)