从 C#到 Python

——写给 C#程序员的简明 Python 指南

闫小勇

Email：kaiseryxy@163.com

整理自作者的博客园 Blog (http://www.cnblogs.com/yanxy/)，2010 年 4 月 5 日

目　录

0 前言：进入 Python 的世界

0.0 写在前面

本文整理自我在博客园（http://www.cnblogs.com/yanxy/）上发表的《从 C#到 Python》系列连载文章。原文从 2010 年 2 月 25 日开始写作，到 4 月 5 日全部完工。现将这篇连载整理成 pdf 文档，方便感兴趣的朋友下载阅读或打印。与原版本相比，这个整理版在语句上略有一些修改，删掉了一些废话，个别章节进行了少许调整。

写这个文章的目的，一方面是整理下我自己学习 Python 的心得体会，留着以后忘记时备查；另一方面就是希望能对有一定 C#编程基础、同时对 Python 感兴趣的朋友有所帮助。

需要说明的是，本文并不是一个完备的 Python 语言教程，而是希望能为 C#的使用者提供一个简短的 Python 语言入门，节省大家的学习时间。作为一个 C#程序员，你掌握的编程知识已经足够多了。要知道，Python 和 C#这两门语言是非常相像的，它们之间的关系类似方言与普通话的关系，而不是英语和汉语的关系。你只是需要花点时间了解 Python 与 C#之间的一些微小差别，然后动手编几个小程序熟悉它。你会发现，Python 比你想象的要更简单。

此外，写这个连载的目的并不是比较两门语言的高下，更不是鼓动任何人完全从 C#转向 Python。我的想法很简单：把学习 Python 当作一个兴趣，并能在适当的时候使用它。

考虑到目前两门语言各自版本的普及情况，本文将主要结合 C#的 2.0 版本和 Python 的 2.6 版本来进行介绍。对于一些在 Python 2.6 中已存在的但是在 C# 3.0 及 4.0 里才新增的特性，也将会适当提及。

0.1 安装和配置 Python 开发环境

在 Python 的官方网站可以下载到 Windows 下的安装包（目前是 2.6.4 版本），按照提示一路 Next 下去就可以了。记得安装完成后将 Python 所在的目录（如 C:\Python26）加入系统 PATH 变量。

Python 的安装包自带了一个简单的集成开发环境——IDLE，你也可以选一个自己喜欢的 IDE。我个人推荐 PythonWin，它的语法提示功能很不错，适合初学者使用。

0.2 第一个程序：Hello, world!

现在你可以打开 IDLE 或 PythonWin，新建一个 py 为扩展名的 Python 脚本文件，输入以下内容：

代码 1：第一个 Python 程序

```
1 print "Hello, world!"
```

保存并运行它，如果输出 >>> Hello, world! ，说明你已经成功编写了第一个 Python 程序，congratulations!

上面的“Hello World”几乎是学习任何一门新语言的必经之路，正如提出这个程序的 Simon Cozens 所说：“它是编程之神的传统咒语，可以帮助你更好的学习语言”。

为了比较 Python 和 C#在编码风格等方面的差异，下边给出一个稍复杂些的“Hello, world”程序以及它的 C#对照版本。

代码 2：唐僧版的 Hello, world 程序

```
1 # -*- coding: utf-8 -*-
2 """
3 我的第 2 个 Python 程序
4 仅为和 C#作对比，谢绝效仿：）
5 """
6 import sys
7 def Main():
8     sys.stdout.write("Hello, world!\n")
9 #下面的语句看起来很怪，一会我们再解释它是干什么滴
10 if __name__ =="__main__":
11     Main()
```

注意，代码 2 第 1 行# -*- coding: utf-8 -*-，是为了让 Python 支持中文，这个是必须的。如果你嫌麻烦，可以在 IDE 里作一个只包含这一行代码的模版文件，以后每次新建 Python 脚本的时候自动打开这个模版，这样会比较省事。

代码 3：C#的对照版本

```
1 /*
2 我的第 1001 个 C#程序
3 可能还不到，没写过这么多
4 */
5 using System;
6 class Program{
7     static void Main(){
8     if (1==1) //干什么？要搞 SQL 注入啊？
9             Console.WriteLine("Hello, world!");
10    }
11 }
```

0.3 认识二者最基本的差异

0.3.1 差异之一：编码风格

比较代码 2、3，可以很容易看出两种语言在编码风格上的差异，下面分别来说。

(1)代码块与缩进

C#使用 C/C++风格的编码形式，除了要求用{}组织代码块外，语句间的缩进可以是任意的。

Python 强制所有程序都有相同的编码风格，它通过缩进来组织代码块。缩进相同的语句被认为是处于同一个代码块中，在 if/else 等语句及函数定义式末尾会有一个冒号，指示代码块的开始。Python 这种强制缩进的做法可以省去{}或者 begin/end 等，使程序的结构更为清晰（有的人认为恰好相反），同时也减少了无效的代码行数。

此外需要注意，尽量使用 4 个空格作为 Python 代码的一个缩进单位，最好不使用 Tab，更不要混用 Tab 和空格，这也算是 Python 的一个非强制性约定吧。

(2)语句结尾

C#语句用分号结尾，Python 不用任何符号（类似 BASIC）。

实际上 Python 也可以使用分号结尾，像这样 a = 1 ; b =2 ; c = 3 ; print a,b,c ，不过 Python 中这种风格多用于调试，应为你可以很容易注释掉这一行就删除了所有调试代码。

另外，当一行很长时，Python 可以用 \ 符号折行显示代码。

(3)注释方法

C#用//进行单行注释，用/**/进行多行注释；而 Python 用#符号进行单行注释，用三引号（可单可双）进行多行注释。

(4)条件表达式

C#的条件表达式必须要加括号，见代码 3 第 8 行；而 Python 的条件表达式加不加括号均可（Python 程序员一般不加，C/C++/C#程序员一般会加：）

0.3.2 差异之二：入口方法

C#语言必须要有入口方法 Main()，这是程序开始执行的地方。

Python 语言中没有入口方法（函数），作为解释型的语言，Python 代码会自动从头执行（所以在代码 2 中除了第 8 行，其它行均属于废话）。

如过你对这点不习惯，可以使用 Python 代码的内置属性__name__。__name__属性会根据 Python 代码的运行条件变化：当 Python 代码以单个文件运行时，__name__便等于"__main__"，当你以模块形式导入使用 Python 代码时，__name__属性便是这个模块的名字。

当然，Python 中的__name__属性并不是为了照顾 C/C++/C#程序员的编程习惯而准备的，它主要目的是用于模块测试。想象一下在 C# 中编写一个组件或类代码时，一般还得同时编写一个调用程序来测试它。而 Python 中可以把二者合二为一，这就是__name__属性的真正作用。

0.3.3 差异之三：import 和 using

在 Python 写的代码 2 中，我们首先 import sys，这是导入了 Python 的 sys 模块，然后在代码里我们可以引用 sys 模块中的对象 stdout 及它的 write 方法。在 Python 中这是必须的，否则你无法调用 sys 模块中的任何东西。

在 C#写的代码 3 中，我们首先 using System，这是引入 System 命名空间，using 表明该程序正在使用给定命名空间下的名称（如 Console）。对 C#这是可选的，如果不事先 using System，你可以在代码中使用全限定名，如 System.Console.WriteLine。

简单的说，Python 中的 import 相当于 C#中的程序集引用。一个程序集可能包括一个或多个命名空间，C#中的 using 是用来引入命名空间 的。Python 如果想实现和 using 类似的功能，就需要用到 from import 语句。例如对于代码 2，可以将 import sys 改写为 from sys import *，这样在程序中就可以直接使用 stdout.write 了。

最后，import 可以出现在代码的任何位置，只要在引用它之前出现就可以。不过为了提高程序可读性，建议还是在所有代码开头书写 import。

0.4 小结

作为 Python 语言的入门，本章没有涉及到过多的编程细节，主要比较了 C#与 Python 两种语言在编码风格等方面最基本的差别，要点如下：

(1)Python 使用强制缩进的编码风格，并以此组织代码块；

(2)Python 语句结尾不用分号；

(3)Python 标明注释用#（单行）或三引号（多行）；

(4)Python 语言没有入口方法（Main），代码会从头到尾顺序执行；

(5)Python 语言用 import 引入所需要的模块(关于模块和包的具体内容将在第 5 章介绍)

以上这几点是 C#程序员初学 Python 时最易犯错误的地方，希望引起大家注意。

1 变量和数据类型

“一切数据是对象，一切命名是引用”。

如果你能理解这句话，说明对 Python 的变量与数据类型已经有了不错的认识，那么我建议你直接跳到 1.4 节的总结部分，看看 C#与 Python 在变量与数据类型方面的差异就可以。如果你还有疑惑，那么就请完整的读一下这一章。

1.1 变量声明和定义

1.1.1 变量声明和定义

与 C#不同，Python 在使用变量之前无须定义它的类型，试着运行下面的例子：

```
1 i = 1
2  print i
```

从上边我们可以看到，变量 i 在使用前并不需要定义，但是必须声明以及初始化该变量。试着运行下面的例子：

```
1 i = 1
2  print i + j
```

上面的代码会产生一个异常：“NameError: name ’j’ is not defined”，Python 提示变量 j 没有定义。这点和 BASIC 等弱类型的语言不一样。在 BASIC 中，执行上述代码的时候不会产生异常，你可以在 EXCEL 的 VBA 开发环境里试一下，把 print 改为 MsgBox 就可以，结果会输出 1 。这说明 Python 并不是一种类似 BASIC 的弱类型语言。

另一方面，Python 与 C#有一个很大的差异就是在程序运行过程中，同一变量名可以（在不同阶段）代表不同类型的数据，看看下边的例子：

```
1 i = 1
2  print i,type(i),id(i)
3 i = 10000000000
4  print i,type(i),id(i)
5 i = 1.1
6  print i,type(i),id(i)
```

变量 i 的类型在程序执行过程中分别经历了 int、long 和 float 的变化，这和静态类型语言（如 C 等）有很大不同。静态语言只要一个变量获得了一个数据类 型，它就会一直是这个类型，变量名代表的是用来存放数据的内存位置。而 Python 中使用的变量名只是各种数据及对象的引用，用 id()获取的才是存放数 据的内存位置，我们输入的 1、10000000000 和 1.1

三个数据均会保存在 id()所指示的这些内存位置中，直到垃圾回收车把它拉走（在系统确定你 不再使用它的时候）。这是动态语言的典型特征，它确定一个变量的类型是在给它赋值的时候。

另一方面，Python 又是强类型的，试着运行下边的例子：

```
1 # -*- coding: utf-8 -*-
2  i = 10; j = 'ss'
3  print i+j
4  #正确的写法是 print str(i)+j，输出 10ss
```

会产生一个异常：“TypeError: unsupported operand type(s) for +: 'int' and 'str'”。在 BASIC 等弱类型的语言中，上边的例子会正常运行并返回（虽然有时候是不可预期的）结果。

所以，我们说 Python 既是一种动态类型语言，同时也是一种强类型的语言， 这点是和 C#不同的地方。对于 Python 的这种变量的声明、定义和使用方式，C#程序员可能要花一段时间去适应，不过相信你会很快就喜欢上它，因为它让 事情变得更加简单（而且不会不安全）。而且，C# 4.0 已经开始用类似的方式定义和使用变量（通过在变量名前加关键字 dynamic），如果你先学了 Python，将能够更快的适应 C# 4.0 的动态编程特征。

1.1.2 变量的命名规则

Python 与 C#的变量（以及函数、类等其它标识符）的命名规则基本一样，同样对大小写敏感。不一样的地方是，Python 中以下划线开始或者结 束的标识符通常有特殊的意义。例如以一个下划线开始的标识符(如 _foo)不能用 from module import *语句导入。前后均有两个下划线的标识符，如__init__，被特殊方法保留。前边有两个下划线的标识符，如__bar，被用来实现类私有属性，这个将在“类和面向对象编程”中再说。

最后，Python 的关键字不能作为标识符（这个大家都知道），不过 Python 的关键字比 C#要少得多，可以 google 一下，这里就不列出了。

1.1.3 常量

Python 没有常量，如果你非要定义常量，可以引入 const 模块（我没用过，在 C#中我也很少用常量）。

1.2 简单数据类型

Python 程序中的一切数据都是对象，包括自定义对象及基本数据类型。这点和 C#一样，它们都是完全面向对象的语言，所以我想 C#程序员会很容易理解 Python 的“一切数据是对象”这个口号。

与 C#不同的是，Python 不区分值类型和引用类型，你可以把所有的类型都理解为 C#的引用类型（当然，它们的实现方式是不一样的，这里只是一个类比）。

Python 内建的数据类型有 20 多种，其中有些不常用到，有些即将被合并。本文将主要介绍空类型、布尔类型、整型、浮点型和字符串、元组、列表、集合、字典等 9 种 Python 内置的数据类型。

在这里，我将前 4 种称为“简单数据类型”，将后 5 种称为“高级数据类型”，实际上 Python 语言本身没有这种叫法，这样分类是我自己设定的，主要是为了和 C#中的相关概念对照方便，希望不要误导大家。

1.2.1 空类型

空类型（None）表示该值是一个空对象，比如没有明确定义返回值的函数就返回 None。空类型没有任何属性，经常被用做函数中可选参数的默认值。None 的布尔值为假。

Python 的 None 和 C#中的可空类型 Nullable<T>类似，比如 C#可以定义 Nullable<double> i = null，与 Python 的空类型类似，但实现原理和用途都不一样。

1.2.2 布尔类型

Python 中用 True 和 False 来定义真假，你可以直接用 a = True 或 a = False 来定义一个布尔型变量。但在 Python 2.6 里，True、False 以及 None 却都不是关键字，在 Python 3.0 里它们已经是关键字了，这个有点乱，我们可以不用管它，直接使用就 OK 了。

注意和 C#不同的是，Python 中 True 和 False 的首字母要大写。

最后一点，在 C#中布尔类型和其他类型之间不存在标准的转换。但在 Python 中，None、任何数值类型中的 0、空字符串’’、空元组()、空列 表[]、空字典{}都被当作 False，其他对象均为 True，这点和 C++差不多，要提起注意。请思考一下，下面的 Python 代码会输出什么？

```
1 if 0:
2     print 'True'
3  else:
4     print 'False'
```

1.2.3 数值类型

Python 拥有四种数值类型：整型，长整型，浮点类型以及复数类型。

整数类型（int）用来表示从-2147483648 到 2147483647 之间的任意整数(在某些电脑系统上这个范围可能会更大，但绝不会比这个更小)；长整数（long）可以表示任意范围的整数。实际上我 们把 Python 的 long 和 int 理解为同一种类型就可以了，因为当一个整数超过

int 的范围后，Python 会自动将其升级为长整型。所以，请忘掉 C#中的 byte、sbyte、short、ushort、int、uint、long 和 ulong 吧，Python 只有一种整数。

Python 中只有 64 位双精度浮点数，与 C#中的 double 类型相同（注意在 Python 中浮点数类型名字是 float 而不是 double），Python 不支持 32 位单精度的浮点数。

除了整数和实数，Python 还提供了 C#中不支持（当然可以通过自定义类来扩展）的一种数据类型：复数（complex）。复数使用一对浮点数表示，复数 z 的实部和虚部分别用 z.real 和 z.imag 访问。

在数值运算中，整数与浮点数运算的结果是浮点数，这就是所谓的“提升规则”，也就是“小”类型会被提升为“大”类型参与计算。这一点与 C#是一样的，提升的顺序依次为：int、long、 float、complex。

作为数值类型的最后一个问题，C#程序员需要注意的是，Python 没有内建 decimal 类型，但可以导入 decimal 模块用来完成与货币处理相关的计算。

1.3 高级数据类型

1.3.1 序列（字符串、列表和元组）

Python 中的序列是由非负整数索引的对象的有序集合（真拗口，其实意思就是下标从 0 开始），它包括字符串、Unicode 字符串、列表、元组、xrange 对象以及缓冲区对象。后两种类型我们先不介绍，后边用到时再说明。

1.3.1.1 字符串类型

Python 拥有两种字符串类型：标准字符串（str）是单字节字符序列，Unicode 字符串（unicode）是双字节字符序列。

在 Python 中定义一个标准字符串（str）可以使用单引号、双引号甚至三引号， 这使得 Python 输入文本比 C#更方便。比如当 str 的内容中包含双引号时，就可以用单引号定义，反之亦然。当字符中有换行符等特殊字符时，可以直接使 用三引号定义。这样就方便了很多，不用去记那些乱七八糟的转义字符。当然 Python 也支持转义字符，且含义和 C#基本一样。不过既然有简单的东西用，谁还去自找麻烦呢？

下边是一个例子，来说明以上几点：

```
1 str1 = 'I am "Python"\n'
2 str2 = "I am 'Python'\r"
3 str3 = """
4 I'm "Python",
5 <a href="http://Csharp.com">you are C#</a>
6  """     #你可以把 html 代码之类的东西直接弄进来而不需要做特殊处理
7  print str1,str2,str3
```

在 Python 中定义一个 Unicode 字符串，需要在引号前面加上一个字符 u，例如

```
1 # -*- coding: utf-8 -*-
2  print u'我是派森'
```

这点没有 C#方便，因为 C#字符串默认就是 Unicode 的，我想 Python 如果要改进，应该把两种字符串合二为一，这样可以为初学者减少很多麻 烦（你看网上有多少帖子是在问 Python 怎么支持中文？根源都在这里）。同时注意，当使用 utf-8 编码时，非 unicode 字符中一个汉字的长度是 3，而使用 gb2312 时是 2，见下边代码：

```
 1 # -*- coding: utf-8 -*-
 2  unicode = u'我'
 3 str = '我'
 4  print len(unicode),len(str)
 5  #输出 1 3
 6
 7  # -*- coding: gb2312 -*-
 8  unicode = u'我'
 9 str = '我'
10  print len(unicode),len(str)
11  #输出 1 2
```

另外，Python 没有 C#中的字符类型，再短的文本也是字符串，这点稍微注意一下就可以，因为现在使用 C#的也很少用 char 了吧？

最后，关于字符串的操作方法，基本上 C#有的 Python 都有，可以看看 Python 手册之类的资料，我就不多说了。唯一提一点就是在 Python 中提取一个字符串的子串时，记得用“切片”语句（后边讲列表和元组时还会介绍），而不要再去找 SubString 了，见下边的例子：

```
1 # -*- coding: utf-8 -*-
2  str1 =  u'我是派森'
3  print str1[2:4]
4  #输出 '派森'
```


1.3.1.2 列表(list)

Python 中的列表(list)类似于 C#中的可变数组（ArrayList），用于顺序存储结构。

列表用符号[]表示，中间的元素可以是任何类型（包括列表本身，以实现多维数组），元素之间用逗号分隔。取值或赋值的时候可以像 C 数组一样，按位置索引：

```
1 # -*- coding: utf-8 -*-
2  array = [1,2,3]
3  print array[0]
4  #输出 1
5  array[0] = 'a'
```

```
6  print array
7  #输出 ['a', 2, 3]
```

从上边的代码中你可能发现一个有趣的事情：在 Python 的列表中可以混合使用不同类型的数据，像['a', 2, 3]这样，不过我不建议你这样做，我觉得没什么好处（虽然个别场合下可能会比较方便）。

另外还可以看到，列表是可变的序列，也就是说我们可以在“原地”改变列表上某个位置所存储的对象(的值)。

C#中 ArrayList 支持的多数操作（如追加、插入、删除、清空、排序、反转、计数等），Python 中的 list 也都支持，同时 list 也支 持“切片”这种操作。切片指的是抽取序列的一部分，其形式为：list[start:end:step]。其抽取规则是：从 start 开始，每次加上 step，直到 end 为止。默认的 step 为 1；当 start 没有给出时，默认从 list 的第一个元素开始；当 end=-1 时表示 list 的最后一个元素，依此类推。一些简单的例子见下边代码：

```
1 # -*- coding: utf-8 -*-
2  test = ['never', 1, 2, 'yes', 1, 'no', 'maybe']
3  print test[0:3] # 包括 test[0],不包括 test[3]
4  print test[0:6:2] # 包括 test[0],不包括 test[6],而且步长为 2
5  print test[:-1] # 包括开始,不包括最后一个
6  print test[-3:] # 抽取最后 3 个
```

字符串、列表、元组都支持切片操作，这个很方便，应该学会熟练使用它。

最后，list 是 Python 中最基础的数据结构，你可以把它当作链表、堆栈或队列来使用，效率还不错。Python 中没有固定长度数组，如果你确实很在意性能，可以导入 array 模块来创建一个 C 风格的数组，它的效率很高，这里就不详细介绍了。

1.3.1.3 元组(tuple)

元组与列表非常相似，它是用()而不是[]括起来的序列。元组比列表的速度更快，但元组是一个不可变的序列，也就是与 str 一样，无法在原位改变它的值。除此之外，其他属性与列表基本一致。

元组定义的方法与列表类似，不过在定义只包含一个元素的元组时，注意在后边加一个逗号。请体会以下几句语句的差异：

```
1 # -*- coding: utf-8 -*-
2  test = [0]          #列表可以这样定义
3  print type(test)    #输出<type 'list'>
4  test = [0,]         #也可以这样定义
5  print type(test)    #输出<type 'list'>
6  test = (0,)         #元组可以这样定义
7  print type(test)    #输出<type 'tuple'>
8  test = (0)          #但不能这样定义，Python 会认为它是一个括号表达式
9  print type(test)    #输出<type 'int'>
```

```
10  test = 0,           #也可以省略括号，但要注意与 C 的逗号表达式不同
11  print type(test)    #输出<type 'tuple'>
```

利用元组的这个特性，可以简化 Python 变量的初始化过程，如：

```
1 x,y,z=1,2,3
```

还可以很简单地进行数据交换。比如：

```
1 a = 1
2 b = 2
3 a,b = b,a
```

以上这类语句在 Python 中被广泛应用于变量交换、函数传值等应用，因此 Python 的解释器在不断对其进行优化，现在已经具备了相当高的效率。所以以上代码在 Python 2.5 以后的版本中，比 tmp = a; a = b; b = tmp 这种常规语句更快。

1.3.2 集合(set)

Python 中的 set 和 C#中的集合（collection）不是一个概念，这是翻译的问题。Python 中的集合是指无序的、不重复的元素集，类似数学中的集合概念，可对其进行交、并、差、补等逻辑运算。

常见集合的语法为：s = set(['a', 'b', 'c'])。不过 set 在 Python 3.0 中发生了较大的变化，创建一个集合的语法变成了：s = {1,2,3}，用花括弧的方法，与后边要提到的 dict 类似。

如果在 set 中传入重复元素，集合会自动将其合并。这个特性非常有用，比如去除列表里大量的重复元素，用 set 解决效率很高，示例如下：

```
1 # -*- coding: utf-8 -*-
2  a = [11,22,33,44,11,22,11,11,22,22,33,33,33]
3 b = set(a)
4  print b
5  #输出 set([33, 11, 44, 22])
```

另一个例子，找出两个 list 里面相同的元素（集合求交，其它类推），代码如下：

```
1 # -*- coding: utf-8 -*-
2  a = ["11","22","33"]
3 b = ["11","33"]
4 c = set(a)&set(b)
5  print c
6  #输出 set(['11', '33'])
```

想想你如果自己实现这个算法会怎么写？然后可以找两个大一点的列表，比比和 set 实现的效率，你就会有体会了。以后在程序里多用 set 吧。
目前 C#的 Collections 中好像还没有 Set，但是 C++ STL 里是有的，不知道 C#为什么不实现这个有趣的东西。

1.3.3 字典(dict)

用过 C#中 Collections 的人对 Hashtable 应该不会陌生，Python 里的哈希表就是字典（dict）了。与 set 类似，字典是一种无序存储结构，它包括关键字（key）和关键字对应的值（value）。

C#程序员需要了解的就是，在 Python 中 dict 是一种内置的数据类型，定义方式为：dictionary = {key:value}，当有多个键值对时，使用逗号进行分割。

字典里的关键字为不可变类型，如字符串、整数、只包含不可变对象的元组，列表等不能作为关键字。字典中一个键只能与一个值关联，对于同一个键，后添加的值会覆盖之前的值。

学过数据结构的人对字典的散列查找效率应该都有认识，所以我建议在可能的情况下尽量多用字典，其它的就不多写了。关于 Python 中 dict 类型 （以及 list、tuple、set）提供的主要方法，可以参考专门介绍 Python 的各种书籍，大多会提供一个详细的方法列表。

1.4 小结

本章讨论了 Python 中变量和数据类型的使用方法，要点如下：

(1) Python 是一种动态的强类型语言，在使用变量之前无须定义其类型，但是必须声明和初始化；

(2) “一切命名是引用”，Python 中变量名是对象的引用，同一变量名可以在程序运行的不同阶段代表不同类型的数据；

(3) “一切数据是对象”，Python 的所有数据类型都是对象，（相较 C#）具有一致的使用方法；

(4) “把问题想得更简单一点”，Python 的数值类型可以说只有两种：整形和浮点，忘掉 C#里的各种数值类型吧；

(5) 注意区别 str 和 unicode，Python 的字符串类型有时候会让人发晕，请试着习惯它，另外不要忘了“切片”这个好工具。

(6) 多使用 list, tuple, set 和 dict 这几种很 pythonic 的数据类型，它们分别用[]、()、([])和{}定义。

2 运算符、表达式和流程控制

本章介绍 Python 的运算符、表达式、程序流程控制语句以及异常处理语句，在这方面，Python 和 C#是非常类似的，我们仅需要注意它们之间的一些细微差异。另外，在本章我还会简要介绍 Python 语言中的两项有趣功能——列表内涵和动态表达式，虽然它们严格来说属于函数部分的内容，不过我觉得还是放在表达式一章比较合适。

2.1 运算符和表达式

无论使用什么语言，我们编写的大多数代码（逻辑行）都包含表达式。一个表达式可以分解为运算符和操作数，运算符的功能是完成某件事，它们由一些数学 运算符号或者其他特定的关键字表示；运算符需要数据来进行运算，这样的数据被称为操作数。例如，2 + 3 是一个简单的表达式，其中+是运算符，2 和 3 是操作数。

2.1.1 算术运算符与算术表达式

算术运算符是程序设计语言最基本的运算符。Python 提供的算术运算符除了+、-、*、/、%（求余）之外，还提供了两种 C#中未提供的运算符：求幂（**）和取整除（//）。下面我们通过一段代码来理解这两个算术运算符：

```
#-*-coding:utf-8-*-
 x = 3.3
y = 2.2
a = x**y
 print a
 #输出 13.827086118,即 3.3 的 2.2 次幂，在 C#中可用 Pow 方法实现幂运算
 b = x//y
 print b
 #输出 1.0,取整除返回商的整数部分
 c = x/y
 print c
 #输出 1.5,注意体会普通除与取整除的区别
```


2.1.2 赋值运算符与赋值表达式

赋值就是给一个变量赋一个新值，除了简单的 = 赋值之外，Python 和 C#都支持复合赋值，例如 x += 5，等价于 x = x + 5。

Python 不支持 C#中的自增和自减运算符，例如 x++这种语句在 Python 中会被提示语法错误。C#程序员可能用惯了这种表达方式（要不为什么叫 C++++呢），在 Python 中，请老老实实的写 x += 1 就是了。

2.1.3 逻辑运算符与逻辑表达式

Python 的逻辑运算符与 C#有较大区别，Python 用关键字 and、or、not 代替了 C#语言中的逻辑运算符&&、|| 和！，此外 Python 中参与逻辑运算的操作数不限于布尔类型，任何类型的值都可以参与逻辑运算，参见 1.2.2 节（布尔类型）的讨论。

用逻辑运算符将操作数或表达式连接起来就是逻辑表达式。与 C#一样，Python 中的逻辑表达式是“短路”执行的，也就是说只有需要时才会进行逻辑 表达式右边值的计算，例如表达式 a and b 只有当 a 为 True 时才计算 b。思考一下，if (0 and 10/0): 这条语句会引发除数为零的异常吗？

此外还要注意：在 Python 中，and 和 or 所执行的逻辑运算并不返回布尔值，而是返回它们实际进行比较的值之一。下边是一个例子：

```
1 print 'a' and 'b'
2  #输出 b
3  print '' and 'b'
4  #输出空串
```

2.1.4 关系运算符与关系表达式

关系运算实际上是逻辑运算的一种，关系表达式的返回值总是布尔值。Python 中的比较操作符与 C#是完全一样的，包括==、!=、>、<、>=和<=共 6 种。

除了基本的变量比较外，Python 的关系运算符还包括身份运算符 is。在 Python 中，is 用来检验两个对象在内存中是否指向同一个对象（还记 得“一切数据皆对象吗，一切命名皆引用”吗？）。注意 Python 中 is 的含义和 C#有所不同，在 C#中，is 操作符被用于动态地检查运行时对象类型是否和给定的类型兼容。例如，运算 e is T ，其中 e 是一个对象，T 是一个类型，返回值是一个布尔值，它表示 e 是否能转换于 T 类型。

2.1.5 三元运算符

三元运算符是 C/C++/C#一系语言所特有的一类运算符，例如，对表达式 b? x: y，先计算条件 b，然后进行判断，如果 b 的值为 true，则计算并返回 x 的值，否则计算并返回 y 的值。

在 Python 中，提供了专门的逻辑分支表达式来模拟 C 系中的三元运算，我们也可以在一行语句中完成三元运算，例如

```
print '偶数' if x % 2 == 0 else '奇数'
```

2.1.6 运算符的优先级

你期待我列一个很长的表嘛？自己去查好了，总之一句话：搞不清楚用括号！

2.2 流程控制语句

2.2.1 条件语句

Python 用 if，elif，else 三个关键字进行条件判断，与 C#唯一的区别就是用 elif 取代了 else if，少打两个字，其它都一样。此外别忘了在 if 等语句后加 ： 哦！

如果一个流程控制分支下不做任何事情，记得写一句 pass 语句，不然 Python 会报错。例如：

```
1 if 0:
2     pass  #神经啊！这种例子用来说明什么？
```

在 Python 中没有 switch 语句，你可以使用 if..elif..else 语句来完成同样的工作。如果你觉得繁琐，可以试试 dict 实现方式，下边是个例子，分别对比了两种实现方式。

```
1 # 类 C#伪码，根据输入的不同参数选择程序的不同行为
2  switch(x):
3 case "1":
4     print 'one'; break;
5 case "2":
6     print 'two'; break;
7 default:
8     print 'nothing!'
9
10  # 使用 if 替代
11  if x =='1':
12     print 'one'
13  elif x=='2':
14     print 'two'
15  else:
16     print 'nothing!'
17
18  # 使用 dict
19  numtrans = {
20     1: 'one',
21     2: 'two',
22     ...
23 }
24 try:
```

```
25      print numtrans[x]
26  except KeyError:
27      print 'nothing!'
28
29  # 也可以在分支中使用方法（函数）
30  def print_one():
31      print 'one'
32  def print_two():
33      print 'two'
34  numtrans = {
35      1:print_one,
36      2:print_two,
37  }
38
39  try:
40      numtrans[x]() #注意名字+括号就可以执行方法了，这个实际上很牛 X 的。
41  except KeyError:
42      print 'nothing!'
```

从上面的语句可以看到，dict 用一种更优雅的方式模拟了 switch 选择，集合 lambda 函数，还可以进一步实现更加复杂的逻辑分支语句。关于 lambda 函数的使用，我们到下一章再学习。

2.2.2 循环

Python 支持两种循环语句——while 循环和 for 循环，不支持 C#中的 do-while 循环。Python 的 while 循环和 C#基本一致，此处我们着重比较两种语言中 for 循环的区别。

说的简单一点，Python 中的 for 语句相当于 C#中的 foreach 语句，它常用于从集合对象（list、str、tuple 等）中遍历数据。例如：

```
1 for i in [1,2,3,4,5]:
2    print i
```

这与 C#中的 foreach 语法基本是一样的，下边是 C#中的对应代码：

```
1 IEnumerable<int> numbers = Enumerable.Range(0, 5);
2 foreach( int i in numbers)
3     Console.WriteLine(i);
```

如何实现类似 C#中 for(int i = 0; i < 10; i++)这种 for 循环呢？答案是使用 range 或 xrange 对象，见下边的代码：

```
1 # range(10)也可以用 xrange(10)代替
2 for i in range(10):
3     print i
```

```
#等价于以下 C#语句
#for(int i = 0; i<10;i++)
#    Console.WriteLine(i);
```

内建函数 range([i,]j[,stride])建立一个整数列表，列表内容为 k(i <= k < j)。第一个参数 i 和第三个参数 stride 是可选的，默认值分别为 0 和 1。内建函数 xrange([i,]j[,stride])与 range 有相似之处，但 xrange 返回的是一个不可改变的 XRangeType 对象。这是一个迭代器，也就是只有用到那个数时才临时通过计算提供值。当 j 值很大时，xrange 能更有效地利用内存。

Python 中的 while 和 for 循环中支持 break 和 continue 语句。break 语句用于立刻中止循环，continue 语句用于直接 进入下一次循环(忽略当前循环的剩余语句)。break 和 continue 语句在 C#与 Python 中的用法是一致的，只用于语句所在的当前循环。如果需 要退出一个多重循环，应该使用异常，因为 Python 中没有提供 goto 语句。

最后，Python 中的循环还支持 else 语句，它只在循环正常完成后运行（for 和 while 循环），或者在循环条件不成立时立即运行(仅 while 循环),或者迭代序列为空时立即执行(仅 for 循环)。如果循环使用 break 语句退出的话,else 语句将被忽略。下面的代码用于说明 else 在循环中的应用：

```
# while-else
while i < 10:
    i = i + 1
else:
    print 'Done'
# for-else
for a in s:
    if a == 'Foo':
        break
else:
    print 'Not found!'
```


2.2.3 异常

Python 和 C#一样支持异常处理，利用 try/except/finally 结构，可以很方便的捕获异常，同时可以用 raise 语句手动抛出异 常（上述四个异常处理的关键字分别对应 C#中的 try/catch/finally/throw）。通过 except，您可以将 try 标示的语句中出现的 错误和异常捕获,except 可以接受参数作为要捕获的异常,如果想要捕获多个异常,可以使用元组(tuple)作为参数。没有参数的 except 被认为 是捕获所有异常。而 finally 则用来在最后执行一定要运行的代码，例如资源回收。下面是一个简单的例子，来说明 Python 中的异常处理方式：

```
try:
    f = open('thefile.txt')
    s = f.readline()
    ...
except IOError, (errno, strerror):
```

```
 6     print "I/O error(%s): %s" % (errno, strerror)
 7 except ValueError:
 8     print "Could not convert data to an integer."
 9 except:
10     print "Unexpected error:", sys.exc_info()[0]
11     raise
12 finally:
13     f.close()
```

最后说明一点，Python 的 try 也支持 else 语句。如果有一些代码要在 try 没有发生异常的情况下才执行，就可以把它放到 else 中（这一点与 finally 不同，finally 分支无论如何都会被执行）。

关于异常处理我们就简单介绍到这里，若需了解更多关于 Python 异常处理类、内建异常类型、自定义异常等内容，可以参考《Python 精要参考（第二版）》。

2.3 列表内涵

列表内涵（List Comprehensions，也译作“列表推导式”）是 Python 最强有力的语法之一，常用于从集合对象中有选择地获取并计算元素，虽然多数情况下可 以使用 for、if 等语句组合完成同样的任务，但列表内涵书写的代码更简洁（当然有时可能会不易读）。

列表内涵的一般形式如下，我们可以把[]内的列表内涵写为一行，也可以写为多行（一般来说多行更易读）。

[表达式 for item1 in 序列 1 ... for itemN in 序列 N if 条件表达式]

上面的表达式分为三部分，最左边是生成每个元素的表达式，然后是 for 迭代过程，最右边可以设定一个 if 判断作为过滤条件。

列表内涵的一个著名例子是生成九九乘法表：

```
s = [(x, y, x*y) for x in range(1, 10) for y in range(1,10) if x>=y]
```

列表内涵可能放在函数编程一章更合适，因为它可以统一实现 map 和 filter 等高阶函数（下一章介绍）。不过我还是倾向于将它看为一种组合的流程 控制语句，而且我个人感觉它与 C#中的 LINQ 有点神似（当然 LINQ 更强大，可以处理数据库和 XML）。下面是两个例子，一个用 LINQ 实现，一个用 Python 的列表内涵实现。

```
1 //C#中用 LINQ 找出 10 以内的偶数
2 var s = from x in Enumerable.Range(0, 10) where x % 2 == 0 select x;
1 #Python 中用列表内涵模拟以上 LINQ 语句
2 s = [x for x in range(0, 10) if x % 2 == 0]
```

当然上边的例子很简单，实际上我们可以用列表内涵完成更复杂的程序设计任务，而且效率一般会比使用 for、if 等的组合语句高（因为中间省略了一些 列表的生成和赋值过程）。

Python 2.5 之后，列表内涵进行了进一步的扩展，如果一个函数接受一个可迭代对象作为参数，那么可以给它传递一个不带中括号的列表内涵，这样就不需要一次生成整个列表，只要将可迭代对象传递给函数。

当然，列表内涵也不能滥用，可能会严重影响代码可读性。

2.4 动态表达式

在 C#语言中，如果需要在文本框中输入 1+2（或更复杂的数学表达式）后计算它的值，可能会用到表达式解析器等。现在我们有了 Python，要完成这个任务可以说是非常简单：只要用内置的 eval()函数，就可以计算并返回任意有效表达式的值。例如：

```
1 str = '1+2'
2 print eval(str)
```

你还可以试验更复杂的表达式，是不是很 Powerful 的一项功能？

除了 eval 函数之外，Python 还提供了 exec 语句将字符串 str 当成有效 Python 代码来执行，看下面的例子：

```
1 #exec.py
2 exec 'a=100'
3 print a
```

另外还有 execfile 函数，它用来执行一个外部的 py 文件。上一个例子存为 exec.py 后，运行下边的代码就知道是怎么回事了：

```
1 execfile(r'c:\exec.py')
```

最后提醒，默认的 eval(),exec,execfile()所运行的代码都位于当前的名字空间中，eval(), exec,和 execfile()函数也可以接受一个或两个可选字典参数作为代码执行的全局名字空间和局部名字空间，具体可以参考 Python 的手册，我就不啰嗦了。

2.5 小结

本章简要比较了 Python 与 C#在运算符、表达式、程序流程控制语句等方面的异同，要点如下：

(1) Python 的算术运算符除了+、-、*、/、%之外，还有求幂（**）和取整除（//）；

(2) Python 支持复合赋值（x+=1），但不支持 C#中的自增和自减运算符（如 x++）；

(3) Python 用 and、or、not 代替了 C#语言中的逻辑运算符&&、||和!，同时 and、or 所执行的逻辑运算不返回布尔值，而是返回实际比较值之一；

(4) Python 的选择语句包括 if、elif 和 else，记得后边加冒号；Python 中没有 switch，但可以用 if-elif-else 或 dict 完成相同的任务模拟；

(5) Python 的循环语句有 while、for，它们都支持 else 语句；注意 Python 的 for 相当于 C#中的 foreach 语句，同时可以用 for + range 对象模拟 C#中的常规 for 循环；

(6) 两种语言的异常处理语句基本一致，不同的是 Python 的异常处理也支持 else。

3 函数及函数编程

在 C#中没有独立的函数存在，只有类的（动态或静态）方法这一概念，它指的是类中用于执行计算或其它行为的成员。在 Python 中，你可以使用类似 C#的方 式定义类的动态或静态成员方法，因为它与 C#一样支持完全的面向对象编程。你也可以用过程式编程的方式来编写 Python 程序，这时 Python 中的函数 与类可以没有任何关系，类似 C 语言定义和使用函数的方式。此外，Python 还支持函数式编程，虽然它对函数式编程的支持不如 LISP 等语言那样完备，但 适当使用还是可以提高我们工作的效率。

本章主要介绍在过程编程模式下 Python 中函数的定义和使用方法，关于在面向对象编程中如何使用函数，我们将在下一章再讨论。此外，我还会简要介绍 Python 中的函数编程功能。

3.1 函数的定义

函数定义是最基本的行为抽象代码，也是软件复用最初级的方式。Python 中函数的定义语句由 def 关键字、函数名、括号、参数（可选）及冒号:组成。下面是几个简单的函数定义语句：

```
# -*- coding: utf-8 -*-
#定义没有参数、也没有返回值的函数
def F1():
    print 'hello kitty!'
#定义有参数和一个返回值的函数
def F2(x,y):
    a = x + y
    return a
#定义有多个返回值的函数，用逗号分割不同的返回值，返回结果是一个元组
def F3(x,y):
    a = x/y
    b = x%y
    return a,b
```


可能你已经注意到了，Python 定义函数的时候并没有约束参数的类型，它以最简单的形式支持了泛型编程。你可以输入任意类型的数据作为参数，只要这些类型支持函数内部的操作（当然必要时需要在函数内部做一些类型判断、异常处理之类的工作）。

3.2 函数的参数

3.2.1 C#与 Python 在函数参数定义方面的差别

4.0 之前的 C#中，方法的参数有四种类型：

(1) 值参数不含任何修饰符；
(2) 引用型参数以 ref 修饰符声明（Python 中没有对应的定义方式）；
(3) 输出参数以 out 修饰符声明（Python 中不需要，因为函数可以有多个返回值）；
(4) 数组型参数以 params 修饰符声明。

Python 中函数参数的形式也有四种类型：

(1) f(arg1,arg2,...) 这是最常用的函数定义方式；
(2) f(arg1=value1,arg2=value2,...,argN=valueN) 这种方式为参数提供了默认值，同时在调用函数时参数顺序可以变化，也称为关键字参数；
(3) f(*arg) arg 代表了一个 tuple，类似 C#中的 params 修饰符作用，可以接受多个参数；
(4) f(**arg) 传入的参数在函数内部是保存在名称为 arg 的 dict 中,调用的时候需要使用如 f(a1=v1,a2=v2)的形式。

可以看出，Python 函数参数定义与 C#相比，最大的两个区别是支持关键字参数和字典参数。不过 C# 4.0 之后已经支持命名参数（即关键字参数）和可选参数（即参数默认值）了。

3.2.2 关键字参数

关键字参数可以使我们调用含有很多参数、同时一些参数具有默认值的 Python 函数变得更简单，也是 Python 实现函数重载的一种重要手段。下面的例子说明了如何定义和调用含关键字参数的函数：


```
1 def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
2     print "-- This parrot wouldn't", action,
3     print "if you put", voltage, "Volts through it."
4     print "-- Lovely plumage, the", type
5     print "-- It's", state, "!"
```

可以用如下几种方式调用：

```
1 parrot(1000)                                  # 缺省值
2  parrot(action = 'VOOOOOM', voltage = 1000000)        # 关键字，缺省值，次序可变
3  parrot('a thousand', state = 'pushing up the daisies')#  位置参数，缺省值，关键字
4  parrot('a million', 'bereft of life', 'jump')        # 位置参数，缺省值
```

但以下几种调用方式是错误的：

```
1 parrot()                          # 非缺省的参数没有提供
2  parrot(voltage=5.0, 'dead')  # 关键字参数后面又出现了非关键字参数
```

```
3  parrot(110, voltage=220)     # 参数值重复提供
4  parrot(actor='John Cleese')  # 未知关键字
```

3.2.3 字典参数

如果形参表中有一个形为**name 的形参，在调用时这个形参可以接收一个字典，字典中包含所有不与任何形参匹配的关键字参数。例如下面的函数：

```
1 def cheeseshop(**keywords):
2     for kw in keywords.keys(): print kw, ':', keywords[kw]
```

就可以象下面这样调用：

```
1 cheeseshop(client='John Cleese',
2             shopkeeper='Michael Palin',
3             sketch='Cheese Shop Sketch')
```

结果显示：

```
    client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```


3.2.4 函数参数调用的顺序

调用 Python 函数时，参数书写的顺序分别为：非关键字参数，关键字参数，元组参数，字典参数。请记住以下几点规则：

* 通过位置分配非关键字参数
* 通过匹配变量名分配关键字参数
* 其他额外的非关键字参数分配到*name 元组中
* 其他额外的关键字参数分配到**name 的字典中
* 用默认值分配给在调用时未得到分配的参数

一般说来，实参表中非关键字参数在前，关键字参数在后，关键字名字必须是形参名字。形参有没有缺省值都可以用关键字参数的形式调用。每一形参至多只能对应一个实参，因此，已经由位置参数传入值的形参就不能在同一调用中再作为关键字参数。

总之，由于 Python 的函数参数定义和调用方式太灵活了，所以一开始容易把人搞晕。不过可以慢慢来，你会越来越发现 Python 的简便所在。

3.3 函数文档

在 C#，可以使用文档 XML 标记对函数进行注释，这样在 VS 等 IDE 中，输入函数名后就会提示函数的功能、参数及返回值等的说明，方便函数的调用 者。在 Python 中，也有类似的功能，即文档字符串（Docstrings）。但 Python 的文档字符串不像 C#中的文档 XML 标记那么丰富，基本等同于 C#中的<summary>标记，下面让我们一起来通过一个例子了解一下：

```
1     def printMax(x, y):
2         '''Prints the maximum of two numbers.
3
4         The two values must be integers.'''
5         x = int(x) # convert to integers, if possible
6          y = int(y)
7
8         if x > y:
9             print x, 'is maximum'
10        else:
11            print y, 'is maximum'
12
13    printMax(3, 5)
14    print printMax.__doc__
```

输出

```
$ python func_doc.py
5 is maximum
Prints the maximum of two numbers. The two values must be integers.
```


在上述函数中，第一个逻辑行的字符串是这个函数的文档字符串。文档字符串的惯例是一个多行字符串，它的首行以大写字母开始，句号结尾。第二行是空行，从第三行开始是详细的描述，描述对象的调用方法、参数说明、返回值等具体信息。

可以使用__doc__（注意双下划线）调用 printMax 函数的文档字符串属性（属于函数的名称，请记住 Python 把每一样东西都作为对象，包括这个函数）。它等同于用 Python 的内建函数 help()读取函数的说明。很多 Python 的 IDE 也依赖于函数的文档字符串进行代码的智能提示， 因此我们在编写函数时应养成编写文档字符串的习惯。

3.4 函数编程

Python 中加入了一些在函数编程语言和 Lisp 中常见的功能，如匿名函数、高阶函数及列表内涵等，关于最后一个问题在第 2 章已经介绍过了，本章只介绍匿名函数和高阶函数。

3.4.1 匿名函数

lambda 函数是匿名函数，用来定义没有名字的函数对象。在 Python 中，lambda 只能包含表达式：lambda arg1, arg2 ... : expression。lambda 关键字后就是逗号分隔的形参列表，冒号后面是一个表达式，表达式求值的结果为 lambda 的返回值。

虽然 lambda 的滥用会严重影响代码可读性，不过在适当的时候使用一下 lambda 来减少键盘的敲击还是有其实际意义的，比如做排序的时候，使用 data.sort(key=lambda o:o.year)显然比

```
1 def get_year(o):
2     return o.year
3 func=get_year(o)
4 data.sort(key=func)
```

要方便许多。

在 C#中也有匿名函数功能。在 C# 1.0 中，通过使用在代码中其他位置定义的方法显式初始化委托来创建委托的实例。C# 2.0 引入了匿名方法的概念，作为一种编写可在委托调用中执行的未命名内联语句块的方式。C# 3.0 引入了 Lambda 表达式，这种表达式与匿名方法的概念类似，但更具表现力并且更简练。这两个功能统称为“匿名函数”。通常，针对 .NET 3.5 及更高版本的应用程序应使用 Lambda 表达式。

Lambda 表达式可以包含表达式和语句，并且可用于创建委托或表达式目录树类型。所有 Lambda 表达式都使用 Lambda 运算符 =>， 运算符的左边是输入参数（如果有），右边包含表达式或语句块。可以将此表达式分配给委托类型，如下所示：

```
1 delegate int del(int i);
2 del myDelegate = x => x * x;
3  int j = myDelegate(5); //j = 25
```


3.4.2 高阶函数

高阶函数（High Order）是函数式编程的基础，常用的高阶函数有：

(1) 映射，也就是将算法施于容器中的每个元素，将返回值合并为一个新的容器。

(2) 过滤，将算法施于容器中的每个元素，将返回值为真的元素合并为一个新的容器。

(3) 合并，将算法（可能携带一个初值）依次施于容器中的每个元素，将返回值作为下一步计算的参数之一，与下一个元素再计算，直至最终获得一个总的结果。

Python 通过 map、filter、reduce 三个内置函数来实现上述三类高阶函数功能。本文不对这三个函数的用法进行详细介绍，仅给出它们使用的示例代码，请细细体会：

```
1
2 #coding:utf-8
3
4 def map_func(lis):
5     return lis + 1
6
7 def filter_func(li):
```

```
 8     if li % 2 == 0:
 9         return True
10     else:
11         return False
12
13 def reduce_func(li, lis):
14     return li + lis
15
16 li = [1,2,3,4,5]
17
18 map_l = map(map_func, li)              #将 li 中所有的数都+1
19  filter_l = filter(filter_func, li)    #得到 li 中能被 2 整除的
20  reduce_l = reduce(reduce_func, li)    #1+2+3+4+5
21
22  print map_l
23  print filter_l
24  print reduce_l
```

运行结果如下：

```
C:\>python test1.py
[2, 3, 4, 5, 6]
[2, 4]
15
```

在 C#中，可以用委托实现函数编程的大部分功能，因为"委托允许将方法作为参数进行传递"。

3.5 小结

本章讨论了 Python 中函数的定义和使用方法，要点如下：

(1) Python 用 def 关键字、函数名、括号、参数（可选）及冒号定义函数，函数可以有 0、1 或多个返回值；

(2) Python 定义函数不需要（也不能）指定参数类型，它天生就是泛型的；

(3) Python 支持（单独或同时使用）普通参数、关键字参数、元组参数和字典参数四种类型的参数，请学会灵活使用它们；

(4) 文档字符串（Docstrings）用于对 Python 的函数提供注释，供自省函数及 IDE 调用；

(5) lambda 关键字用来定义匿名函数，它仅支持表达式，某些情况下可以简化代码编写量，但不要滥用；

(6) Python 通过 map、filter、reduce 三个内置函数实现函数编程中映射、过滤及合并三类高阶函数功能，但注意 map 和 filter 可以用列表内涵替代。

4 类及面向对象

如果你熟悉 C#，那么对类(Class)和面向对象(Object Oriented)应该不会陌生。Python 与 C#一样，能够很好地支持面向对象的编程模式。本章对 Python 中面向对象编程的基本知识进行介绍，并将其与 C#中的对应部分进行比较。

4.1 类的定义与实例化

4.1.1 类的定义

与 C#一样，Python 使用 class 关键字定义一个类。一个最简单的类定义语句如下：

```
1 class A:
2     pass
```

它等价于 C#中的 class A{}。当然，以上语句没有任何实际意义，它只是告诉我们什么是定义一个类所必需的，即：class 关键字，类名和冒号，pass 关键字只用来占位，相当于 C#中花括号的作用。

4.1.2 类的实例化

类是定义对象格式的模板，而对象则是类的实例，通过类创建对象的过程称为类的实例化。在 C#中，需要使用 new 关键字实例化一个类，例如

```
A a = new A();
```

在上条语句中，C#完成了两件事情：首先声明一个类型为 A 的变量 a，然后用 new 运算符创建一个类型为 A 的对象，并将该对象的引用赋值给变量 a。而在 Python 中没有 new 关键字，同时它是一种动态语言，不需要事先指定变量的类型，只需要：

```
a = A()
```

即可创建一个类型为 A 的对象，看起来好像是将类当作一个函数调用，返回值是新创建的对象。

4.2 类的成员变量

4.2.1 为类添加数据

通常我们利用类来定义各种新的数据类型，其中既包含数据内容，又包含对数据内容的操作。前面我们定义的类 A 暂时什么也不能做，因为它既没有数据，也没有操作。本节我们先讨论第一个问题，即如何为类添加数据。

在 C#中，我们需要显示地对类的数据进行定义，例如：

```
1 class A {
2    public int x;
3 }
```

以上代码为类 A 定义了一个名为 x 的成员变量，用来存放整数型数据。对类实例化后，可以对此数据进行访问，例如：

```
1 A a = new A();
2 a.x = 2;
3 Console.Write(a1.x);
```

而 Python 类的数据添加方法与 C#有一些不同，因为 Python 是一种动态语言，变量在使用之前不需要定义，所以你可以不在类定义中添加成员变量，而是在运行时动态地添加它们，例如：

```
1 class A:pass
2 a = A()
3 a.x =1
4  print a.x
```


当然上述方法不是一种好方法，也很少有人真正这么用（虽然语法上没有任何错误），类的成员变量最好是在类的初始化函数里面声明并初始化。

4.2.2 初始化函数

Python 的类提供了类似 C#构造函数的东西：__init__（注意前后是两个下划线），类在实例化时会首先调用这个函数。我们可以通过重写 __init__函数，完成变量的初始化等工作。与 C#不同的地方是，Python 不支持无参数的初始化函数，你至少需要为初始化函数指定一个参数，即对 象实例本身（self）。下面是一段简单的示例代码，在该代码中，我们重写了函数__init__，定义并初始化了一个类的成员变量 x：

```
1 class A:
2     def __init__(self):
3         self.x = 1
4 a = A()
5  print a.x
```

实际上 Python 允许在任何成员函数（不仅限于__init__）中声明类的成员变量，不过最好养成在初始化函数中声明成员变量的习惯。

另外，我们也可以定义带多个参数的初始化函数，如下：

```
1 class A:
2     def __init__(self,x):
3         self.x = x
4 a = A(1)
5  print a.x
```

当然，Python 的类也有类似 C#析构函数的东西：__del__函数，但不推荐使用它，Python 的垃圾回收机制会帮你做好一切事情。

4.2.3 静态成员（类变量）

你可能已经注意到，在前面的代码中，成员变量 x 之前也有一个 self，这说明该成员属于于类的实例（而不是类本身），这类似 C#中的非静态成员。那么如何在 Python 中定义与类的静态成员呢？这个很简单，只需要在类定义中直接初始化一个变量即可，例如：

```
1 class A:
2     y = 2
3  print A.y
4  #输出 2
```

与 C#不同的是，Python 中对象可以直接访问类的静态成员，例如：

```
1 class A:
2     y = 2
3 a = A()
4  print a.y
5  #输出 2
```

但对象不能为类的静态成员赋值，例如：

```
1 class A:
2     y = 2
3 a = A()
4 a.y = 3
5  print a.y
6  print A.y
7  #输出 3  2
```

事实上赋值语句 a.y = 3 相当于为实例 a 添加了一个非静态成员 y，而类 A 的静态成员 A.y 没有发生改变（即使同名也不会冲突，因为它们不在同一个命名空间内）。可以直接用 A.y = 3 为类变量赋值，这点与 C#是一致的。

4.2.4 私有成员

在编写 C#程序时，我们可以对类的成员使用不同的访问修饰符定义它们的访问级别。例如对于公有成员可以使用修饰符 public，而对于私有成员（仅限于类中的成员可以访问）则使用修饰符 private，此外 C#还提供了保护成员（protected 修饰）和内部成员（internal 修饰）等。

但在 Python 中所有的成员都是公有成员，你可以不受限制地访问它们（变量和方法均是如此）。Python 建议在类成员前面加_（一个下划线）的 为私有成员，但这仅仅是个建议，没有语法上的强制限制，也就是说你还是可以调用_开头的变量和方法。不过，Python 里以两个下划线开头的类成员就不能 直接访问了，例如：

```
1 class A:
2     def __init__(self):
3         self.__x = 1
4 a = A()
5  print a.__x
6  #输出 AttributeError: A instance has no attribute '__x'
```

但实际上我们还是可以不受限制地访问这个“私有”成员，只需要在变量名（或方法名）前加下划线和类名称，如：

```
1 print a._A__x
2  #输出 1
```

我认为 Python 应该彻底封掉这个口子，支持真正的私有成员，这样才符合面向对象编程中“封装”的基本思想。

4.3 类的方法

4.3.1 为类添加方法

方法是对类数据内容的操作，在 Python 中，定义类的方法与定义一个普通的函数在语法上基本相同（见上一章）。C#程序员需要注意的是，在类中定义的常规方法的第一个参数总是该类的实例，即 self。同时注意在方法中引用类的另一个方法必须使用类名加方法名的形式，下面是一个定义类方法的简单例子：

```
1 class A:
2    def prt(self):
3       print "My Name is A"
4    def reprt(self):
5        prt(self) #错误，NameError: global name 'prt' is not defined
6         A.prt(self) #正确
7  a = A()
```

```
8 a.prt()
9 a.reprt()
```

不能定义一个不操作实例的方法（这点在开始时经常会被忘记，请注意）：

```
1 class A:
2    def prt():
3       print "My Name is A"
4
5 a = A()
6 a.prt()
7  #TypeError: prt() takes no arguments (1 given)
```

4.3.2 静态方法

Python 与 C#一样支持静态方法。在 C#中需要使用关键字 static 声明一个静态方法，而在 Python 中是通过静态方法修饰符@staticmethod 来实现的，下面是示例代码：

```
1 class A:
2    @staticmethod
3    def prt():
4       print "My Name is A"
5
6 A.prt()
```


如你所见，静态方法可以直接被类调用，它没有常规方法那样的特殊行为（默认的第一个参数是 self 等），你完全可以将静态方法当成一个用属性引用方式调用 的普通函数。任何时候定义静态方法都不是必须的，静态方法能实现的功能都可以通过定义一个普通函数来实现。一般认为，当有一堆函数仅仅为某一特定类编写 时，将这些函数包装成静态这种方式可以提供使用上的一致性。

4.3.3 方法重载

在 C#中，同一个类中的两个或多个方法可以共享相同的名称，只要它们的参数（数量或类型）不同即可，这种过程被称为方法重载（Method overloading），方法重载是 C#实现多态特性的方式之一。

很不幸，Python 不支持同名方式的方法重载，虽然你可以在类中定义多个同名的方法，但前边定义的方法会被后边定义的所覆盖（Python 使用名字来绑定一个对象）。但 Python 作为面向对象语言，自然不会丢掉方法重载这个面向对象的重要特性。

如前所述，C#重载方法的主要方式是定义不同类型或数量的参数。由于 Python 本身是动态语言，方法的参数是没有类型的，当调用传值的时候才确定参数的类型，故对参数类型不同的方法无需考虑重载。对参数数量不同的方法，则（大多数情况下）可以采用参数默认值来实现，具体内容就不再重复介绍了，可以参考上一章《函数及函数编程》。

4.3.4 运算符重载

运算符重载是实现多态的另外一种重要手段。在 C#中，我们通过使用关键字 operator 定义一个运算符方法，并定义与所在类相关的运算符行为。在 Python 中，运算符重载的方式更为简单——每一个类都默认内置了所有可能的运算符方法，只要重写这个方法，就可以实现针对该运算符的重载。例如以下是重载加法操作：

```
1 class A:
2     def __init__(self,sum = 0):
3         self.sum = sum
4     def __add__(self,x):
5         return A(self.sum + x.sum)
6     def __str__(self):
7         return str(self.sum)
8 a = A(1)
9 b = A(2)
10  print a + b
11  #输出 3
```

Python 具体的内建运算符方法列表，请参看 Python 手册及有关教程，本文不再罗列。

4.4 类的继承

面向对象的三个基本特征是封装、继承和多态，前面的内容已经涉及了封装和多态，本节介绍 Python 中类的继承。

继承是创建新类的机制之一，它通过对一个已有类进行修改和扩充来生成新类。这个原始的类被称为基类或超类，新生成的类称为该类的派生类或子类。当通 过继承创建一个类时，它会自动'继承'在基类中定义的属性。一个子类也可以重新定义父类中已有的属性或定义新的属性，这也是实现多态的一种重要方式。

4.4.1 单继承

Python 用类名后加括号的方式实现继承，下面是一个简单的示例：

```
1 class A:
2     x = 1
3  class B(A):
4     x = 2
5  print B.x,B.y
```

与 C#一样，Python 中如果要引用子类的某个属性，会首先在子类中寻找，没有就去到父类中寻找它的定义，再没有的话，就一直向上找下去，知道找到为止（最不利的情况是找到 object，如果还没有就只能报错了）。

方法的寻找方式与属性相同。子类的方法可以重定义父类的方法，也可以在子类中直接调用父类中的方法，方式如下：

```
BaseClass.method(self, arguments)
```

4.4.2 多继承

Python 支持多继承（C#需要用接口实现），实现方式很简单，只需要在类名后的括号中，写入用逗号分隔的多个父类名即可。不过当父类中有重名的属性时，需要了解 Python 的搜索顺序，具体可以参考《Python 精要参考》，本文不再详细介绍。

4.4.3 接口与抽象类

Python 没有 C#中接口的概念，因为可以实现多重继承，使用接口的意义不大，而且 Python 的标准类基本包含 C 中接口的大部分功能（不过据说新版本的 Python 已经考虑要加入接口的功能了）。而 C#中抽象类在 Python 中则可以通过一些变通的方法实现。常用的方法是用 NotImplementedError 异常模拟抽象类，示例代码如下：

```
# -*- coding: utf-8 -*-
def abstract():
    raise NotImplementedError("abstract")

class Car:
    def __init__(self):
        if self.__class__ is Car:
            abstract()
        print "Car"

class BMW(Car):
    def __init__(self):
        Car.__init__(self)
        print "bmw"

bmw = BMW() #子类可以实例化
Car()       #抽象类实例化会报错：NotImplementedError: abstract
```


4.5 获取对象的信息

在 Python 这类动态编程语言中，我们经常需要检查一个类、对象或模块，以确定它是什么、它知道什么或它能做什么。在 Python 中，我们把这类功能叫做自省（Introspection），它非常类似于 C#中的反射（Reflection）。

4.5.1 对象的特殊属性

除了用户自定义的类属性外，Python 中的所有对象都拥有一些内置的特殊属性，它们可以帮助我们回答如下问题：

* 对象的名称是什么（__name__）？
* 这是哪种类型的对象（__class__）？
* 对象知道些什么（__doc__）？
* 对象能做些什么（__dict__）？
* 对象（的类）继承自谁（__bases__）？
* 它在哪一个模块中（__module__）？

限于篇幅，本文不一一解释这些特殊属性的含义，你可以试着找一个对象（Python 中一切皆对象，所以这些属性适用于所有类型的对象，包括字符串、数值、列表、元组、字典、函数、自定义类、类实例和类方法等），打印它的这些属性，不难明白它们是做什么用的。

4.5.2 自省函数

Python 提供了很多有用的自省函数，帮助我们找到对象的有用信息，常用的自省函数包括：

* id() 返回对象唯一的标识符
* repr() 返回对象的标准字符串表达式
* type() 返回对象的类型
* dir() 返回对象的属性名称列表
* vars() 返回一个字典，它包含了对象存储于其__dict__中的属性(键)及值
* hasattr() 判断一个对象是否有一个特定的属性
* getattr() 取得对象的属性
* setattr() 赋值给对象的属性
* delattr() 从一个对象中删除属性
* callable() 测试对象的可调用性
* issubclass() 判断一个类是另一个类的子类或子孙类
* isinstance() 判断一个对象是否是另一个给定类的实例
* super() 返回相应的父类

同样，我建议你在 Shell 中逐项试试这些函数的功能，以更好地体会自省的威力。

4.6 本章小结

本章介绍了 Python 中面向对象编程的基本知识，并与 C#做了简单对比，要点如下：

(1) Python 用 class 关键字、类名、括号加父类（可选）及冒号定义类，实例化类时不需要 new 关键字；

(2) Python 可以对象中动态添加成员变量，但建议在__init__函数中添加成员变量并初始化；

(3) Python 的类中可以通过名称前加双下划线定义私有变量（及私有方法）；

(4) Python 中定义类的方法与定义普通函数在语法上基本相同，区别是非静态方法的第一个参数总是类的实例，即 self；

(5) 通过修饰符@staticmethod 可以定义一个类的静态方法；

(6) Python 对类方法重载和运算符重载的实现方式与 C#有一定区别，不过我认为 Python 的方式更简单一些；

(7) Python 支持类的单继承和多继承，但不支持接口，也不（直接）支持抽象类；

(8) 通过自省，可以在运行时获得对象的有用信息。

5 模块和包

本章是《从 C#到 Python》系列连载的最后一章，内容较简单，主要介绍 Python 中模块与包的使用方法。

5.1 模块

Python 的脚本都是用扩展名为 py 的文本文件保存的，一个脚本可以单独运行，也可以导入另一个脚本中运行。当脚本被导入运行时，我们将其称为模块（module）。模块是 Python 组织代码的基本方式。

模块名与脚本的文件名相同，例如我们编写了一个名为 Items.py 的脚本，则可在另外一个脚本中用 import Items 语句来导入它。在导入时，Python 解释器会先在脚本当前目录下查找，如果没有则在 sys.path 包含的路径中查找。

在导入模块时，Python 会做以下三件事：

(1) 为模块文件中定义的对象创建一个名字空间，通过这个名字空间可以访问到模块中定义的函数及变量；
(2) 在新创建的名字空间里执行模块文件；
(3) 创建一个名为模块文件的对象，该对象引用模块的名字空间，这样就可以通过这个对象访问模块中的函数及变量，如：

```
1 import sys
2  print sys.path
```


如果要同时导入多个模块，可以用逗号分隔，如： import sys, os ；

可以使用 as 关键字来改变模块的引用对象名，如：import os as system ；

也可以用 from 语句将模块中的对象直接导入到当前的名字空间（不创建模块名字空间的引用对象），如：from socket import gethostname ；

from 语句支持逗号分割的对象，也可以使用星号(*)代表模块中除下划线开头的所有对象，如：from socket import *，这是个懒省事的方法，我是经常用，不过用之前最好先搞清楚会不会覆盖当前名字空间中的函数名等，总之不是个好习惯:)

除了 Python 脚本（不仅限于 py，还包括 pyc 和 pyo），import 语句还可以导入 C 或 C++扩展（已编译为共享库或 DLL 文件）、包（包含多个模块，一会介绍)和内建模块（使用 C 编写并已链接到 Python 解释器内）。不过除了包，另外两个我也没用过:(

Python 解释器在第一次 import 一个 py 文件的时候，会尝试将其编译为字节码文件，这个文件的扩展名通常为.pyc，它是已经完成语法检查并转译为虚拟机指令的代码。后边的导入操作会直接读取.pyc 文件而不是.py 文件，一般而言速度会更快。

5.2 包

Python 的模块可以按目录组织为包（package）。一般来说，我们将多个关系密切的模块组织成一个包，以便于维护和使用，同时可有效避免名 字空间冲突。创建一个包的步骤是：建立一个名字为包名字的文件夹，并在该文件夹下创建一个__init__.py 文件，你可以根据需要在该文件夹下存放脚 本文件、已编译扩展及子包。

一个典型的包可能有以下结构:

```
package1/
    __init__.py
    subPack1/
        __init__.py
        module_11.py
        module_12.py
        module_13.py
    subPack2/
        __init__.py
        module_21.py
        module_22.py
    ……
```


只要目录下存在__init__.py，就表明此目录应被作为一个 package 处理。在最简单的例子中，__init__.py 是一个空文件，不过一般我们都要在__init__.py 中做一些包的初始化动作，或是设定一些变量。

最常用的变量是__all__。当使用包的人在用 from pack import * 语句导入的时候，系统会查找目录 pack 下的__init__.py 文件中的__all__这个变量。__all__是一个 list，包含了所有应该被导 入的模块名称，例如：__all__ = ["m1", "m2", "m3"] 表示当 from pack import * 时会 import 这三个 module。

如果没有定义__all__，from pack import * 不会保证所有的子模块被导入。所以要么通过__init.py__，要么显式地 import 以保证子模块被导入，如：import pack.m1, pack.m2, pack.m3。

5.3 本章小结

本章介绍了 Python 中模块与包的使用方法，要点如下：

(1) 模块是一个可以导入的 Python 脚本文件；

(2) 包是一堆按目录组织的模块和子包，目录下的__init__.py 文件存放了包的信息；

(3) 可以用 import, import as , form import 等语句导入模块和包。

总之，模块与包是在物理上组织 Python 代码复用的一种有效方式，它有点类似于 C#中的程序集（Assembly）。初学 Python 不一定要会自己建立模块和包，但一定要学会使用各种包，这正是 Python 的强大之处：Python 带着各种各样的 battery。当你想完成一项功能时，最好先去 搜索是不是有相关的包能够复用（多数情况下是这样的，因为在各种领域都有太多的人在为 Python 作着贡献）。而且，绝大多数 Python 包都是开源的，研读优秀的代码也是提高编程能力的一种有效途径。

参考文献（推荐读物）

本文在写作的过程中参考了大量网络上的 Python 教程和文档，限于篇幅，不再一一列出（我博客的原文里，引用或推荐部分都给出了链接），在此向作者和译者致谢！

下面的这些资料是我认为不错的一些读物，在此推荐给各位：

[1]**《Python 精要参考（第二版）》**，David M Beazley 著，Feather、Weizhong 译。

网址：http://wiki.woodpecker.org.cn/moin/WeiZhong

写作本文参考最多的一本教程，也是我认为最简明扼要、详略得当的一本教程。

[2]**《可爱的 Python》**，作者：哲思社区；电子工业出版社，2009 年出版。

豆瓣链接：http://book.douban.com/subject/3884108/

一本好书，语言简洁幽默，实例生动，适合快速了解和入门 Python。

[3]**《深入 Python》**，Mark Pilgrim 著，CpyUG 译。

网址：http://www.woodpecker.org.cn/diveintopython/toc/index.html

Python 的经典教程，对很多主题有很深入的讨论，适合提高阶段阅读。

第 3 版地址：http://www.woodpecker.org.cn/diveintopython3/

如果你已经用 Python 3 了，那么就应该看这个了，不过我还没开始用呢。

[4]**《我的名字叫 Python》**，某本书的附录，实在查不到作者是谁（抱歉）。

网址：http://www.cublog.cn/u/29309/showart_226898.html

C++程序员的 Python 使用手册，如果你是从 C/C++转向 Python，我向你推荐这本书。

[5]**《用 Python 做科学计算》**，HYRY Studio 著。

网址：http://pyscin.appspot.com/html/index.html

前面的读物多是介绍 Python 语法的，而这本优秀的著作是专门面向学习 Python 的科研工作者的，里面介绍了很多科学计算包的用法，个人强烈推荐！