附录A 我的名字叫Python

亲爱的朋友，能够认识你们这样能干的C++程序员，我是如此的愉快和荣幸。我是Python，一门对您们来说大概会有些陌生的动态解释语言。但请放心，我生来就是一门友好的语言。在设计之初，我定位于和C语言可以友好的结合，那么，我同时就会友好的支持所有可兼容C的语言。这当然包括了C++，您知道，它是C的近亲。

在语法方面，我更是很自豪的以简单易学著称。学会Python只需要很少的时间，不会给各位C++程序员带来更多的负担。

以下将会介绍我的方方面面，这并不是一个完备的Python语言教程，但是，这足以使您学会够用的知识，帮助您完成大部分的混合开发任务。这份文档专门为C++程序员准备，所以会有针对性的将Python的部分内容与C++的对应技术进行比对，这并不是为了分个高下——厚此薄彼不是我的性格，取长补短才是大智慧——而是为了帮助读者更好的了解我。

第1节 预备知识

1.1、 依赖于缩进的语法

文章内容并不依赖复杂的环境，使用Python内置的IDLE甚至Shell就可以。需要注意的是，我的语法依赖于缩进。每一级缩进表示一个新的代码块，从属于这个语句块的上一级缩进。因此，您选择的编辑器应该可以处理tab和空格的转换。通常我们推荐每级4个空格。以下这个游戏技能的定义代码表现了多级Python代码之间的缩进关系：

```
class AntiAttack(object):
    def __init__(self, sender, target):
        self.name = u'反击'
        self.world = sender.world
        self.sender = sender
        self.target = target

    def __call__(self):
        target = self.target
        value = self.sender.str * 5

        hit = Dice(self.sender.dex).Roll()
        volt = Dice(target.dex).Roll()
        hits = hit - volt #hit = volt

        if hits > 0 :
            harm = target.BeHurt(value * hits/hit+1)
            self.world.PostMessage(AttackMsg(self, self.sender,
```

```
target, harm))       #版面有限导致这里折行，实际上与上面是同一行
        else:
            self.world.PostMessage(TextMsg(u'%s 闪过了%s 的反击!
'%(target.name, self.sender.name))) #版面有限导致这里折行

    def checked(self):
        return True
```

1.2、 注释

我没有多行注释，#注释符相当于 C++中的//注释。您的编辑器如果支持块注释最好，否则可能需要手工操作多行注释，或者利用一些非正规的技巧。

1.3、 多语言脚本

如果您的脚本中包括英文之外的字符，没关系，我可以认出它们。不过需要您在程序开头用注释写一行编码说明，就像下面这样：

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

我会根据这行注释指定的编码读取您编写的脚本。实际上，如果您觉得麻烦，完全可以去掉 coding:code 格式前后的-*-修饰，或者替换成您习惯的修饰符。但是，有些编辑器可能只接受默认的形式。

下面，我们从对象模型开始。

第2节 我的对象家族

这一章将会介绍我家族中的一些重要的成员。也许这会使您觉得乏味。没关系，您可以直接从第 2 节开始，在任何感觉有必要的时候回头来参阅这部分内容。

2.1、 我的名字叫 object

我是 object，不瞒您说，我对自己的身份非常自豪。我现在已经是 Python 所有内置数据类型的基类，作为推荐的类型构造方式，所有您自定义的类型，也都应该从我继承。在即将到来的 Python3.0，所有的对象将尽出于我。因为我于 Python 中无处不在，下面，我们一起来看一些代码：

```
>>> object
<type 'object'>
>>> type(object)
<type 'type'>
```

```
>>> issubclass(type(object), object)
True
>>>
```

这里我向您介绍几个 Python 族类的工具，type 函数获取对象的类型，而这个类型也是一个对象——于是，如您所见，它也是我的子类型，这一点可以通过内置函数 issubclass 证实。由此我们得知 type 类型是 object 的子类型。下面我们再看几种 Python 内置类型：

```
>>> type(1)
<type 'int'>
>>> issubclass(int, object)
True
>>> type('')
<type 'str'>
>>> issubclass(str, object)
True
>>> type([])
<type 'list'>
>>> issubclass(list, object)
True
>>> type({})
<type 'dict'>
>>> issubclass(dict, object)
True
```

关于我的故事有很多，如果要全部介绍一遍，会是一本很厚的书。因此，我在这里只介绍一个与 C++对象有很大不同的地方——动态对象模型：

```
>>> x = 1
>>> x
1
>>> x = 'b'
>>> x
'b'
>>> class c(object):pass

>>> x = c()
>>> dir(x)
['__class__', '__delattr__', '__dict__', '__doc__',
'__getattribute__', '__hash__', '__init__', '__module__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__', '__weakref__']
>>> x.pix_x = 0
>>> x.pix_y = 0
>>> dir(x)
```

```
['__class__', '__delattr__', '__dict__', '__doc__',
'__getattribute__', '__hash__', '__init__', '__module__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__', '__weakref__', 'pix_x', 'pix_y']
>>> x.pix_x
0
>>> x.pix_y
0
>>> x.pix_z

Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    x.pix_z
AttributeError: 'c' object has no attribute 'pix_z'
>>>
```

这里您想必已经看到了我的一些与 C++对象的不同之处，这里变量 x 并没有类型定义，而是直接通过赋值创建。C++变量在定义时，就决定了一个变量的类型，这使得编译器可以确保它的类型安全。而 Python 的对象采用了完全不同的命名—对象绑定机制。每一个像我这样的对象，都通过与我绑定的命名访问。相对来说，你所见到的变量 x，其实是与对象绑定的命名，所以，变量不需要预定义，绑定后即可访问。在 x=1 之后，进行 x = 'b'操作，只是将命名 x 重新绑定于字符串对象'b'。当然，动态类型的代价就是，您不能在 Python 中针对命名进行类型约束。

更进一步，c 被定义为一个空类型，除了继承自我的成员，c 没有定义任何内容。可以通过占位语句 pass 实现这个功能。通过 dir 函数，我们可以观察对象的内部成员。这里您可以看到，我们在程序运行时可以随时为它添加成员，演示中的代码为 c 类型的对象 x（嗯，善变的 x 又绑定到了一个新对象上）添加了两个新成员。事实上，您几乎可以在运行期任意改变对象和类型的行为，在后面的章节中，我和我的同族们还会给您介绍其它的对象行为。

2.2、 我的名字叫 str

2.2.1、 字符串定义

我是 str，Python 的字符串。在几乎每一种编程语言中，字符串都是很重要的东西，Python 也不例外。Python 拥有强大的文本处理功能，这当然与我密不可分。我是一个比 C 的 char*，甚至 C++的 std::string 都要复杂的多的数据结构，我既是基础的文本值类型，也是包含字符内容的序列容器。要定义一个字符串值很容易，对于 Python，用单引号或双引号包围的内容就是一个字符串：

```
>>> s0 = "Python"
>>> s1 = 'C++'
>>> s0
```

```
'Python'
>>> s1
'C++'
>>> s0 + "&" + s1
'Python&C++'
```

这种单双引号的混用带来很多便利，当我的内容中包含双引号时，您可以用单引号定义我，反之亦然：

```
>>> s0 = '"C++" 是一门强大的计算机语言'
>>> print s0
"C++" 是一门强大的计算机语言
```

print 是一个 python shell 指令，它会打印 python 对象的显示信息，对于我和其它的大部分内置类型，这就是我们的字面值，对于您自定义类型，您也可能通过重载__repr__方法来定义自己的 print 实现。

Python 没有字符类型，再短的文本也是字符串。但是，字符串有普通字符串和 unicode 字符串的区别，后面将会请我的 unicode 兄弟出来自我介绍一下。

2.2.2、 *字符串转义*

除了利用单双引号的嵌套，我还可以用类似 C 的转义字符串来表达内容中的引号：

```
>>> s = 'I\'m Python.'
>>> s
"I'm Python."
```

我支持的转义功能当然不止这么简单，基本上 c 语言字符串中的转义定义在我这里都可以使用：

```
>>> print 'Python\t2.5'
Python      2.5
>>> print 'Python\n2.5'
Python
2.5
>>> print 'Python\\string'
Python\string
```

如果不想让内容中的\发生转义，可以在字符串定义前加一个字符 r，这表示“raw string”：

```
>>> print 'Python\n2.5'
Python
2.5
>>> print r'Python\n2.5'
Python\n2.5
```

2.2.3、 字符串池

告诉您一个秘密，我基于池机制。在整个 Python 进程运行期间生成的所有字符串值，都会缓冲于虚拟机的字符串池中。也就是说，两个值相同的 Python 字符串，其实是指向同一个值，下面我们利用 Python 的内置函数 id()来证实这一点：

```
>>> s0 = "Python"
>>> s = 'Python'
>>> id(s)
11416064
>>> id(s0)
11416064
```

id 函数以一个 Python 对象为参数，返回它的内部标识符。通过这个唯一标识我们可以看出，两个值相同的字符串变量实际上总是引用同一个对象。这带来了很多好处。但是也有几点需要注意的：

1、我是 COW（copy on write）的，如果修改了字符串变量的值，不管是通过什么方式，都会使变量指向一个不同的字符串对象。也因此，我的所有相关的改写方法，都不是 in-place 方式的，也就是说，它们不会修改原有的字符串，而是生成一个新的。

2、Python 通常不会回收字符串池的空间，所有运行时生成的字符串对象都会保存到进程结束。所以，尽管如前所见，我支持连加号，但大量的连加操作会生成很多中间字符串，这些对象可能永远都不会被调用。因此，如果你需要连接若干字符串，我更推荐您使用我的成员方法 join：

```
>>> s = 'Python'
>>> s1 = 'C++'
>>> '&'.join([s1, s])
'C++&Python'
```

join 方法接受一个字符串序列对象（关于序列，我们在后面会进一步讨论），以字符串自身做为间隔联接序列中的每一个字符串元素。Join 函数会一次性生成结果字符串，不会生成多余的中间变量。

3、甚至遍历我，也会生成很多新的临时对象，当然，这通常不是太大的问题，全部英文字母也不过 26 个，即使汉字，通常不会超过两万个，这种程度的字符串池对现代的计算机都不是什么大问题。下面的代码，会帮助您理解为什么出现这个现象：

```
>>> id('abc')
13860064
>>> for c in 'abc':
    print c, id(c)
a 11487040
b 11487104
c 11489248
>>>
```

Python 的 for 循环也许会让您想起 C++的 STL 迭代器，没错，它们颇有共通之处。后面我们有专门的章节来讨论 Python 优雅的 for 语法。

2.2.4、 文档字符串

使用三个连续的引号标记的称为文档字符串，这种方式使您可以轻松定义复杂的字符串。

```
>>> #以下下划线部分是 doc 变量的内容
>>> doc = """
>>> x = int(raw_input("Please enter an integer: "))
>>> if x < 0:
...       x = 0
...       print 'Negative changed to zero'
... elif x == 0:
...       print 'Zero'
... elif x == 1:
...       print 'Single'
... else:
...       print 'More'
...
"""
>>> print doc

>>> x = int(raw_input("Please enter an integer: "))
>>> if x < 0:
...       x = 0
...       print 'Negative changed to zero'
... elif x == 0:
...       print 'Zero'
... elif x == 1:
...       print 'Single'
... else:
...       print 'More'
...
```

这里我把 Python Tutorial 的一段示例代码完整的放在了变量 doc 中。文档字符串可以方便的保留文本中的换行信息，用来在代码中书写大段的说明很方便，所以它经常用于块注释

（其实，被定义为文档字符串以后，还是会被解释器读取并放进虚拟机，但是，它不会给您带来更多的麻烦了）、定义说明等，在后面的章节您会见到如何在 Python 定义中加入说明信息。

2.2.5、 字符串切割

切割（split）是指通过在[]中指定起止点和步长，从现有的线性序列中生成新的序列。这是一种强大的能力，它使得线性容器可以使用简洁的语法快速映射生成新的序列。我当然也支持这个能力，下面是一些字符串切割的示例：

```
>>> #以下这行具有浓郁函数化编程风味的代码生成了 a 到 z 的英文字母串
>>> s = ''.join(chr(o) for o in range(ord('a'), ord('z')+1))
>>> s
'abcdefghijklmnopqrstuvwxyz'
>>> s[10]   #这个当然不算是切割，只是字符串索引，等同于 c 里数组的[]用法
'k'
>>> s[:10]  #从字符串中切割出前 10 个字母
'abcdefghij'
>>> s[10:]  #切割出 10 以后的字母，
'klmnopqrstuvwxyz'
>>> s[:10]+s[10:]   #s[:x]+s[x:]=s，这称为序列切割的不变性
'abcdefghijklmnopqrstuvwxyz'
>>> s[0:10]         #这个操作实际上等同于 s[:10]
'abcdefghij'
>>> s[1:10]         #当然可以从指定位置开始
'bcdefghij'
>>> s[1:10:2]       #也可以指定步长
'bdfhj'
>>> s[:]            #这个操作相当于复制了 s，这是常用的序列浅复制方式
'abcdefghijklmnopqrstuvwxyz'
>>> s[::-1]         #因为有:区分位置，我们可以任意省略不需要的参数
'zyxwvutsrqponmlkjihgfedcba'
>>> s[-5:-1]        #负数索引相当于从末尾反向切割
'vwxy'
>>> s[-1:-5:-1]     #步长也可以是负数，这表示反方向
'zyxw'
```

2.3、 我的名字叫 unicode

2.3.1、 unicode 字符串及定义

我是 unicode 字符串，str 的兄弟。我们两个有几乎一样的使用方法。定义一个 unicode

字符串，只需要在定义字符串时，在前面加上一个字符 u：

```
>>> u'中文'
u'\u4e2d\u6587'
>>> print u'中文'
中文
>>> us = u'中文'
>>> print us
中文
```

我会将每一个字都做为一个单位（一个 unicode 字符）存储，无论它是英文字母还是汉字、日文这样的表意文字。所以在长度和位置的计算上，我和我的 str 兄弟有所不同：

```
>>> us = u'中文'
>>> len(us)
2
>>> for c in us:
...     print c
...
中
文
>>> len(us)
2
```

对比我的操作，我们看看 str 兄弟会如何处理这个字符串：

```
>>> s = '中文'
>>> len(s)
4
>>> for c in s:
    print c

Ö
Ð
Î
Ä
```

str 会将双字节字符视为两个单字节字符，而我将其做为一个完整的字符。所以相对来说，str 比我可以更准确的表达两进制层面的内容，例如，可以用来存储二进制流或 C 结构。而我更适合面向内容的表达，例如东方语言的文本处理。

2.3.2、 字符串编解码——str 与 unicode 的互相转换

对于 Python 解释器，str 是平台相关的，它使用的内码依赖于操作系统环境，而我，unicode 是平台无关的，是 Python 内部的字符串存储方式。Unicode 可以通过编码（encode）成为特定编码的 str，而 str 也可以通过解码（decode）成为 unicode：

```
>>> us = u'中文'
>>> s = us.encode('mbcs')
>>> s
'\xd6\xd0\xce\xc4'
>>> print s
中文
>>> us = s.decode('mbcs')
>>> us
u'\u4e2d\u6587'
>>> print us
中文
```

注意 encode 和 decode 的参数，一定要与字符串使用的实际编码相符，否则会发生异常或者产生乱码。这里我使用了 MS Windows XP SP2 操作系统，发布于这个平台的 Python2.5 shell，使用 MBCS 编码。

Encode 和 decode 在需要显式编码操作时很有用，例如 Windows Xp 平台上使用 IDLE 时，因为 IDLE 使用 ANSI 内码，直接定义 unicode 字符串会出错，这个时候使用 str 的 decode 方法才会得到正确结果：

```
>>> s = '中文'
>>> print s
中文
>>> us = u'中文'
>>> us
u'\xd6\xd0\xce\xc4'
>>> print us
ÖÐÎÄ
>>> print us.encode('mbcs')
?D??
>>> us = s.decode('mbcs')
>>> print us
中文
>>> us
u'\u4e2d\u6587'
```

注意两次直接打印 us，观察到的编码不同，在多语言环境中，如果遇到编解码错误，可以通过这种方式解决。这在电子邮件处理或 web 服务器等应用领域很常见。

2.4、 我是 bool

我是 bool，我觉得其实自己没什么可说的，True or False，这就是我的世界，很简单。不过，袭承 C 的风格，逻辑值可以与其它类型进行混合运算。在这种情况下，0、空字符串、空序列都会被当做 False，通过下面的 and/or 计算演示，可以比较清楚的看出来：

```
>>> True and 0
0
>>> True or 0
True
>>> True and ''
''
>>> True or ''
True
>>> True and []
[]
>>> True or []
True
```

应用这种方法，可以构造出一些简短的代码，但是要当心不要为了追求简短影响可读性。

2.5、 我们是 python 数值对象

2.5.1、 基本数值类型

Python 家族拥有完整的数据类型支持，我想这对您没有什么奇怪的。Python 的数值定义、进制转换和隐式转换规则类似 C，支持四则运算和幂运算。也有浮点数和整数之分：

```
>>> 1+2532
2533
>>> 25/6    #这里的除数和被除数都是整数，返回商也是整数
4
>>> 25/6.0  #6.0 表示浮点数 6，返回值为浮点数
4.166666666666667
>>> 2*4
8
>>> 4-25
-21
>>> 2**3    #Python 的幂运算用**运算符表示
8
```

Python 中的整型对应 C 中的 long，浮点型对应 C 中的 double。除了这些基本的数值类型，Python 还提供了一些特殊用途的类型。

2.5.2、 我的名字叫 long

Python 中，我，长整型 long 代表一种“无限长整数”：

```
>>> x = 2**64
>>> type(x)
<type 'long'>
>>> x
18446744073709551616L
>>>
```

我在理论上讲是没有最大值的，适用于大数计算，实际上唯一限制 long 的因素是计算机内存。只要内存够，想定义多大都可以。

也许您已经发现了，可以在数值最后加一个大写的”L”来表示我：

```
>>> type(1)
<type 'int'>
>>> type(1L)
<type 'long'>
```

2.5.3、 我的名字叫 complex

我是 complex，在通用编程语言中见到内置的复数类型也许让您有些意外。在 Python 中定义和使用复数非常容易：

```
>>> 1+1j + 3          #我与实数的混合计算
(4+1j)
>>> (4j)*5
20j
>>> (6+4j)*(-6+2j)
(-44-12j)
>>> (2.5-4.3j)*(3+6j)        #我的虚部和实部也可以使用不同类型的实数
(33.299999999999997+2.1000000000000014j)
>>> (2.5-4.3j)/(3+6j)
(-0.40666666666666662-0.62j)
>>> (2.5-4.3j)**(3+6j)       #我也支持幂运算
(63292.59329535532+13461.996107772202j)
```

从某种意义上讲，Python 的数值计算体系基于复数体系，而不是通常的实数，因为我可以参与几乎所有的运算。这是一项很了不起的能力。在应用方面，可以很容易实现二维矢量运算之类的计算。

2.5.4、 我的名字是 Decimal

由于二进制和十进制互换的差异问题，Python 像 C/C++一样，存在浮点数误差问题：

```
>>> 50.10
50.100000000000001
```

而我，Decimal，就是为了满足精确计算的需要而出现的：

```
>>> from decimal import Decimal
>>> x = Decimal(1)
>>> x
Decimal("1")
>>> y = Decimal(3)
>>> x/y
Decimal("0.33333333333333333333333333333")
>>> y**6
Decimal("729")
>>> print y
3
```

尽管我很有用，但是目前我还不能与 complex 兄弟兼容：

```
>>> Decimal(5)*(1+1j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'Decimal' and
'complex'
```

您可以在 Python 的帮助文档中查阅到有关于我的详细说明。

2.6、 我的名字叫 list

在前面的内容里，您已经见过了一些用'[]'标示的数据类型。也知道了它的名字叫 list，是的，list 就是我，Python 中最常见的线性容器。

我很像是您在数据结构课程中学习过的链表。C++程序员们总要用到一些线性容器，从最基本的 C 数组，到 std::vector、std::list 这些强大的通用容器类型。在学习如何使用我的过程中，您可以参阅已有的知识。

在 list 对象中，您可以混合放入不同的元素，如果这让您难以接受，请把它看做是 std::list<PyObject*>：

```
>>> l = ['a', 2, [6, 7, 8]]
>>> l
['a', 2, [6, 7, 8]]
```

也可以像 C 数组一样，按位置索引：

```
>>> l[0]
'a'
>>> l[1]
2
>>> l[2]
[6, 7, 8]
```

2.6.1、 我与等差数列

内置函数 range 是我的好伙伴，在熟悉我的过程中，您可以使用它快速生成等差数列做为学习用的数据。在实用中，这也是非常重要的一个工具。调用 range(x)。就可以生成一个 0 到 x 的前闭后开的自然数列 list：

```
>>> range(10)        #生成的 list 最后一个元素是 9 而不是 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1)
[0]
>>> range(0)         #这一行和下一行代码演示了 range 对边界条件的处理
[]
>>> range(-1)
[]
```

range 可以指定起点、终点和步长：

```
>>> range(2, 12)             #带上下界的调用方式
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> range(2, 12, 2)          #带上下界和步长的调用方式
[2, 4, 6, 8, 10]
>>> range(10, 5)             #这样看起来没什么效果
[]
>>> range(10, 5, -1)         #但是指定步长为负数就可以了
[10, 9, 8, 7, 6]
>>> range(-10, 10, 2)        #也可以从负数开始
[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8]
```

2.6.2、 我的对象定义

定义一个 list 很简单，除了用 range，当然您也可以直接定义，以下几种方式都是合法的：

```
>>> l = []
>>> l
[]
```

```
>>> l = [1]
>>> l
[1]
>>> l = [1, 2, 3]
>>> l
[1, 2, 3]
>>> l = list()      #这会不会让您想起 C++语法？
>>> l
[]
```

2.6.3、 对象引用

从数据结构的角度讲，我和 str 是近亲，我们都是线性容器。不同的是，我是可改变的，对比一下字符串和我。

以下示例展示了 str 的只读特性，注意 a 修改前后，a 和 s 的值及 id：

```
>>> s = 'abcdefghijklmnopqrstuvwxyz'
>>> a = s
>>> id(s)
12502984
>>> id(a)
12502984
>>> a += 'abc'
>>> a
'abcdefghijklmnopqrstuvwxyzabc'
>>> s
'abcdefghijklmnopqrstuvwxyz'
>>> id(s)
12502984
>>> id(a)
11783488
```

以下示例展示了我的引用特性：

```
>>> l = ['a', 'b', 'c']
>>> x = l
>>> x += ['a', 'b', 'c']
>>> x
['a', 'b', 'c', 'a', 'b', 'c']
>>> l
['a', 'b', 'c', 'a', 'b', 'c']
```

如果您需要隔离两个变量所发生的修改，就需要复制一个新的 list，浅复制可以使用 str 向您介绍的切割方法（这通用于所有的 Python 线性容器），深复制则需要内置的 copy 模块。

下面演示了如何利用切割方法进行复制：

```
>>> l = ['a', 'b', 'c']
>>> x = l
>>> x = l[:]
>>> l
['a', 'b', 'c']
>>> x
['a', 'b', 'c']
>>> x += ['a', 'b', 'c']
>>> l
['a', 'b', 'c']
>>> x
['a', 'b', 'c', 'a', 'b', 'c']
```

2.6.4、 我是链表

除了可以用[]按位置索引我的元素，还可以把我当做链表进行操作，以下是一些来自 Python 官方文档 Python Tutorial 的示例：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Python 2.5 Tutorial 5.1

2.6.5、 我是堆栈

通常来说，堆栈操作是链表操作的一部分。我也不例外：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Python 2.5 Tutorial 5.1.1

2.6.6、 我是队列

同样，我也支持队列操作：

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

Python Tutorial 5.1.2

2.6.7、 列表推导式

列表推导式是一种通过函数化的编程方式简洁的构造 list 的方法。这是一种强有力的表达方式。例如，我们回顾一下前面的例子，给定 s = 'abcdefghijklmnopqrstuvwxyz'，可以这样生成小写英文字母表的 list：

```
>>> s
'abcdefghijklmnopqrstuvwxyz'
>>> l = [c for c in s]
>>> l
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
```

```
'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
'y', 'z']
```

在推导式中，还可以通过 if 生成进一步的过滤条件：

```
>>> #在四种类型中查找带有 has_key 方法的
>>> types = [list, set, tuple, dict]
>>> haskey = [t for t in types if hasattr(t, 'has_key')]
>>> haskey
[<type 'dict'>]
```

也可以使用一些复杂的生成逻辑：

```
>>> #平方数列
>>> l = [x**2 for x in range(10)]
>>> l
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> #九九乘法表
>>> l = [(x, y, x*y) for x in range(1, 10) for y in range(1,
10) if x>=y]
>>> l
[(1, 1, 1), (2, 1, 2), (2, 2, 4), (3, 1, 3), (3, 2, 6), (3, 3,
9), (4, 1, 4), (4, 2, 8), (4, 3, 12), (4, 4, 16), (5, 1, 5),
(5, 2, 10), (5, 3, 15), (5, 4, 20), (5, 5, 25), (6, 1, 6), (6,
2, 12), (6, 3, 18), (6, 4, 24), (6, 5, 30), (6, 6, 36), (7, 1,
7), (7, 2, 14), (7, 3, 21), (7, 4, 28), (7, 5, 35), (7, 6,
42), (7, 7, 49), (8, 1, 8), (8, 2, 16), (8, 3, 24), (8, 4,
32), (8, 5, 40), (8, 6, 48), (8, 7, 56), (8, 8, 64), (9, 1,
9), (9, 2, 18), (9, 3, 27), (9, 4, 36), (9, 5, 45), (9, 6,
54), (9, 7, 63), (9, 8, 72), (9, 9, 81)]
```

如果注意到[]实际上相当于我的构造函数，那么链表推导式的参数表达式完全可以从[]中抽像出来。事实上也确是如此，Python 中已经可以使用生成器推导式做为参数传递给函数，而对于接收它的函数，这是一个抽象的迭代器。这种技术大大增强了 Python 对程序行为的抽象能力。

2.7、 我的名字叫 tuple

我叫 tuple，你可以称我为元组。如果要在 C++中找一个对应的东西，著名的 C++ 代码库 Boost 中，有一个 tuple 类型。

实际上可以把我看做是一个只读的 list。构造 tuple 实例很简单，可以用()标示的序列就是一个 tuple：

```
>>> tu = (1, 2, 3, 4, 5)
>>> tu
(1, 2, 3, 4, 5)
>>> type(tu)
<type 'tuple'>
```

也可以直接使用 tuple 构造函数，通常使用这种方法自 list 中构造我的对象，当然，也可以直接使用生成器推导式：

```
>>> x = tuple([235, 224, 120])
>>> x
(235, 224, 120)
>>> x = tuple(x**2 for x in range(1, 4))
>>> x
(1, 4, 9)
```

因为我也是 COW 的，就像 str 那样，所以修改一个 tuple 变量不会影响其它引用：

```
>>> x = tuple(x**2 for x in range(1, 4))
>>> x
(1, 4, 9)
>>> y = x
>>> id(x)
12665560
>>> id(y)
12665560
>>> x += (6,)        #注意这个单元素 tuple 的定义
>>> x
(1, 4, 9, 6)
>>> id(x)
11371888
>>> y
(1, 4, 9)
>>> id(y)
12665560
```

注意上例中，为 x 添加一个单元素 tuple 时，在元素后面加了一个逗号，这样解释器就可以正确区分出这是一个 tuple。

像 x, y = a, b 这样的形式，解释器会将其转为 tuple 进行计算，然后再分解为单独的元素，这称为 tuple 封装和解封。由于这种语句广泛应用于交换变量、函数返回值和传值等应用，Python 的解释器不断对其进行优化，现在的封装和拆封操作已经效率很高了。例如，在 Python2.5 中使用 a, b = b, a 交换变量值，比 tmp = a; a = b; b = tmp 更快。

2.8、 我的名字叫 set

我是 Python 中一个年轻的成员，set。我是一个不重复的元素集合。可以通过调用 set(...)构造函数来生成 set 对象。传入的重复元素会自动合并。我还支持交、并、差、补等操作，以下是来自于 Python Tutorial 的一些示例：

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> fruit = set(basket)               # create a set without
duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership
testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two
words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              # letters in a but not
in b
set(['r', 'd', 'b'])
>>> a | b                              # letters in either a
or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                              # letters in both a and
b
set(['a', 'c'])
>>> a ^ b                              # letters in a or b but
not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

2.9、 我的名字叫 dict

我是 python 中的键-值映射对象 dict，相当于 C++中的 std::map。我要求 key 传入的必须是不可变的类型。比如整数或是 str，由于 tuple 是 COW 类型，也可以做为键，相应的，list 不可以做为 dict 的键。

2.9.1、 字典定义

定义一个字典很容易，用{}标示的就是字典：

```
>>> d = {}
>>> d = {'a':'a', 'b':'x', 4:'y', 'abc':'e'}
>>> d
{'a': 'a', 'b': 'x', 4: 'y', 'abc': 'e'}
>>> d['a'] = 'a'              #字典中没有的键会自动添加进去
>>> d[(1, 2, 3)] = 4          #tuple 也可以作为键
>>> d[[1, 2, 3]] = 6          #list 不行
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

2.9.2、 我是无序的

类似 std::map，我存储键值对时，会根据优化性能的原则进行排列，所以外部访问时应该视我为一个无序的键值对容器。如果需要顺序查找数据，可以调用我的成员变量 keys()，得到所有键的列表后，再对其进行排序，然后根据这个列表访问字典对象。

```
>>> d = dict((chr(x), x) for x in range(ord('a'), ord('z')+1))
>>> d
{'a': 97, 'c': 99, 'b': 98, 'e': 101, 'd': 100, 'g': 103, 'f':
102, 'i': 105, 'h': 104, 'k': 107, 'j': 106, 'm': 109, 'l':
108, 'o': 111, 'n': 110, 'q': 113, 'p': 112, 's': 115, 'r':
114, 'u': 117, 't': 116, 'w': 119, 'v': 118, 'y': 121, 'x':
120, 'z': 122}
>>> for k, v in d.items(): #
...         print k, v
...
a 97
c 99
b 98
e 101
d 100
g 103
f 102
i 105
h 104
k 107
j 106
m 109
l 108
o 111
n 110
q 113
p 112
```

```
s 115
r 114
u 117
t 116
w 119
v 118
y 121
x 120
z 122
```

这里我向您演示了一个字典遍历的示例。首先我定义了一个英文字母和 ascii 码对应的表。您结合前面的列表推导式，相信您能够理解这个字典的构造代码。如您所见，尽管我按顺序插入了 26 个字母，但是输出时，却不能得到原有的顺序。注意我的 items 方法，这个方法返回一个 list，存储的是键-值对元组。这里使用了隐式的元组自动拆封。

下面我们看一下如何按顺序遍历我的对象：

```
>>> for k in sorted(d.keys()):
...         print k, d[k]
...
a 97
b 98
c 99
d 100
e 101
f 102
g 103
h 104
i 105
j 106
k 107
l 108
m 109
n 110
o 111
p 112
q 113
r 114
s 115
t 116
u 117
v 118
w 119
x 120
y 121
z 122
```

函数 sorted 接收一个可迭代对象（例如，线性容器）。返回一个有序的 list。利用 keys 方法和 list 函数，您可以按顺序检索了。

2.9.3、 默认值

回顾前一个例子的字典对象：

```
>>> d = dict((chr(x), x) for x in range(ord('a'), ord('z')+1))
>>> d
{'a': 97, 'c': 99, 'b': 98, 'e': 101, 'd': 100, 'g': 103, 'f':
102, 'i': 105, 'h': 104, 'k': 107, 'j': 106, 'm': 109, 'l':
108, 'o': 111, 'n': 110, 'q': 113, 'p': 112, 's': 115, 'r':
114, 'u': 117, 't': 116, 'w': 119, 'v': 118, 'y': 121, 'x':
120, 'z': 122}
```

如果我们读取一个不存在于 d 中的键：

```
>>> d['A']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'A'
```

这里解释器抛出了一个 KeyError 异常。有的时候可能您不希望这样的结果，而是想要一个默认值，那么，您可以用 get 方法：

```
>>> d.get('A', -1)
-1
>>> d.get('A', ord('A'))
65
>>> d.get('A')
>>>
```

2.9.4、 关于我的事

我对于 Python，有着非常重要的意义，这里只是简单介绍了一些特别重要和常见的内容。关于我的更多的故事，有些会在后面的章节中提到。有些您可以查阅 Python 的帮助文档，特别是 Python Tutorial 和 Python Reference Manual。

第3节 我的流程控制

3.1、 选择分支

您一定在 C++中使用过 if...else...语法。我也有这样的逻辑选择分支功能。从外观和使用细节上，我的 if 语法与 C++都有些不同，但是它们都完成同样的工作：根据逻辑条件执行对

应的选择分支。

3.1.1、 二重选择

基本的逻辑选择分支功能很直观：

```
>>> x = int(raw_input('请输入一个整数:'))
请输入一个整数:10
>>> if x % 3 == 0:
...         print 'x 能被 3 整除'
... else:
...         print 'x 不能被 3 整除'
...
x 不能被 3 整除
```

3.1.2、 多路分支

我没有 switch/case 这样的语句，多路选取择分支通过 if/elif/else 结构实现：

```
>>> x = int(raw_input('请输入一个整数:'))
请输入一个整数:10
>>> if x % 3 == 0:
...         print 'x 能被 3 整除'
... elif x % 4 == 0:
...         print 'x 能被 4 整除'
... elif x % 5 == 0:
...         print 'x 能被 5 整除'
... else:
...         print 'x 不能被 3 或 4 或 5 整除'
...
x 能被 5 整除
```

3.1.3、 三元表达式

我一直没有加入在 c 中很重要的三元运算符?:。以前的版本中，我通过一个逻辑运算组合：...and...or...实现这个功能。自从 2.5，我加入了一个专门的逻辑分支表达式：

```
>>>x = int(raw_input('请输入一个整数:'))
请输入一个整数:10
>>> print '偶数' if x % 2 == 0 else '奇数'
偶数
```

注意逻辑条件在 if...else 之间。

3.2、 我是 for

我是 Python 的 for 关键字。我对自己的重要地位非常自信。是的，我是 Python 中最具个性的语法之一。理解我，对学习 Python 很重要。

3.2.1、 集合迭代

我与 Pascal 那种 for seed step 形式的语法有本质的不同。我其实不会通过索引下标去访问一个集合，而是直接通过迭代器——是的，就是你在 STL 中经常见到的 iterator——来迭代遍历集合。这个过程通过 Python 的一系列定义约定来抽象。Python 内置的集合对象都可以通过直接或间接的方式支持我。以下是一个典型的 list 迭代：

```
>>> l = [1, 2, 4, 8, 16]
>>> for i in l:
...         print i
...
1
2
4
8
16
```

常用的通过 range 遍历容器的方式，其实也是迭代。函数 range 会生成一个等差数列 list。通过在这个等差数列上迭代，得到用来访问容器索引：

```
>>> l = [1, 2, 4, 8, 16]
>>> for i in range(len(l)):
...         print l[i]
...
1
2
4
8
16
```

3.2.2、 抽象迭代能力

实际上，我的迭代推导能力不限于容器，就像 C++里可以定义不依赖于容器的迭代器，我也可以迭代一个与容器无关联的独立迭代器对象。您会在后面的章节读到具体的迭代器定义语法。而这里，我们先观察一个现成的独立迭代器对象 xrange。

函数 xrange 在使用上与 range 基本无异，唯一的区别是它不返回 list，而是返回一个迭代器对象：

```
>>> print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print xrange(10)
xrange(10)
>>> for i in range(10):
...         print i
...
0
1
2
3
4
5
6
7
8
9
>>> for i in xrange(10):
...         print i
...
0
1
2
3
4
5
6
7
8
9
```

函数 xrange 不返回完整的 list，而是逐步迭代。所以在生成数列的时候会比 range 节省内存。通常大数列枚举使用 xrange 更有效率。

3.2.3、 生成器推导式

生成器推导式是我的一种衍生形式。也是 Python 最强有力的语法之一。我们回顾前面的几个例子，首先是著名的列表推导：

```
>>> #九九乘法表
>>> l = [(x, y, x*y) for x in range(1, 10) for y in range(1,
10) if x>=y]
>>> l
[(1, 1, 1), (2, 1, 2), (2, 2, 4), (3, 1, 3), (3, 2, 6), (3, 3,
9), (4, 1, 4), (4, 2, 8), (4, 3, 12), (4, 4, 16), (5, 1, 5),
(5, 2, 10), (5, 3, 15), (5, 4, 20), (5, 5, 25), (6, 1, 6), (6,
```

```
2, 12), (6, 3, 18), (6, 4, 24), (6, 5, 30), (6, 6, 36), (7, 1,
7), (7, 2, 14), (7, 3, 21), (7, 4, 28), (7, 5, 35), (7, 6,
42), (7, 7, 49), (8, 1, 8), (8, 2, 16), (8, 3, 24), (8, 4,
32), (8, 5, 40), (8, 6, 48), (8, 7, 56), (8, 8, 64), (9, 1,
9), (9, 2, 18), (9, 3, 27), (9, 4, 36), (9, 5, 45), (9, 6,
54), (9, 7, 63), (9, 8, 72), (9, 9, 81)]
```

然后是通过生成器推导式生成容器：

```
>>> d = dict((chr(x), x) for x in range(ord('a'), ord('z')+1))
>>> d
{'a': 97, 'c': 99, 'b': 98, 'e': 101, 'd': 100, 'g': 103, 'f':
102, 'i': 105, 'h': 104, 'k': 107, 'j': 106, 'm': 109, 'l':
108, 'o': 111, 'n': 110, 'q': 113, 'p': 112, 's': 115, 'r':
114, 'u': 117, 't': 116, 'w': 119, 'v': 118, 'y': 121, 'x':
120, 'z': 122}
>>> t = tuple(i**2 for i in range(5))
>>> t
(0, 1, 4, 9, 16)
```

我们也可以自定义一个函数，接受生成器推导式做为参数：

```
>>> def prtfoo(its):
...         for it in its:
...                 print it
...
>>> prtfoo(i for i in xrange(5, 20, 3))
5
8
11
14
17
```

Python 原本通过一组内置的高阶函数 map、fliter、reduce 来实现函数化的集合容器操作，现在由于迭代技术的发展，这一组函数可能会在不远的将来退出 Python 世界。

3.2.4、 我的 else

这一点或许会出乎您的意料，我，for，也有 else 语法。当 for 迭代到达结尾后，会执行 else 再退出。如果自 break 跳出，则不会执行 else，这在执行搜索之类的遍历操作时很有用：

```
>>> s = 'abserwroh'
>>> for c in s:
...         if c == 'z':
```

```
...                 print 'find!'
...                 break
... else:
...         print 'no'
...
no
>>> for c in s:
...         if c == 'w':
...                 print 'find!'
...                 break
... else:
...         print 'no'
...
find!
```

当然，这只是个示例，实际上如果要判断一个字符串 c 是否是字符串 s 的一部分，可以直接用 c in s 运算。

3.3、 我是 while

我是一个比较容易理解的关键字。循环，我感兴趣的就是循环。只要 while 条件满足，就会一直执行循环体：

```
>>> i = 0
>>> while i < 20:
...         i += 5
...         i -= 3
...         print i
...
2
4
6
8
10
12
14
16
18
20
```

其实，我也有 else 分支，类似 for，如果因为判断表达式为 False 退出循环，就执行 else，否则不执行：

```
>>> while i < 10:
...         if i in pr:
```

```
...                 print i
...                 break
... else:
...         print i
...         print 'else'
...
2
>>> while i < 10:
...         i += 1
... else:
...         print i
...         print 'else'
...
10
else
```

3.4、 我是 pass

如果你需要一个不执行任何代码的空语句，类似 C++中的;，那么，找我就对了。比如，有时候程序员会需要一个没有任何定义的类，他们可以这样写：

```
>>> class c(object):pass
```

3.5、 异常处理

和很多现代语言一样，我也有异常处理。利用 try/except/finally 结构，可以很方便的捕获异常。而需要手动抛出异常的话，也只需要一个 raise 语句。

3.5.1、 语法错误与异常

实际上，我会将程序中的错误分为两类：语法错误与异常。一个程序，如果存在语法错误，导致无法被解释器理解，就会发生错误异常，例如下面这个类定义，少写了冒号：

```
>>> class c(object) pass
  File "<stdin>", line 1
    class c(object) pass
                    ^
SyntaxError: invalid syntax
```

当然，我们知道，即使程序通过了语法检查，一样有可能出错，这个时候我的虚拟机会抛出异常，就像下面这样：

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

其实手动抛出异常也很简单，在程序中加入 raise 就可以：

```
>>> raise 'error'
__main__:1: DeprecationWarning: raising a string exception is
deprecated
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
error
```

语句 raise 可以支持任何 Python 对象。当然，在比较正规的场合，还是鼓励使用统一的异常类型定义。

3.5.2、 异常捕获与处理

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：

```
import sys

try:
    f = open('thefile.txt')
    s = f.readline()
    ...
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
finally:
    f.close()
```

3.5.3、 异常与 else

甚至，我也有 else。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）。

因此，一个完整的异常管理结构可能会像下面这样：

```
try:
   block-1 ...
except Exception1:
   handler-1 ...
except Exception2:
   handler-2 ...
else:
   else-block
finally:
   final-block
```

Python 2.5 What's New 6 PEP 341

第4节 我的定义语法

4.1、 我的名字叫函数

函数定义是最基本的行为抽象代码。而我，就是 Python 中的函数。

4.1.1、 基本定义方式

以下代码定义了一个求最大公约数的算法：

```
def HCF(x, y):
    "Highest common factor of x and y."
    a = x
    b = y
    r = a%b

    while r != 0:
            a = b
            b = r
            r = a%b
    return b
```

没有返回值的函数等同于返回 None。

细心的您可能已经注意到了，Python 函数定义的时候，形参没有类型约束。这根源于 Python 的动态类型思想。因此，您可以方便的抽象出代码行为；也因此，您不能通过不同参数类型定义函数重载，包括类成员函数的定义也是如此。不过您可以通过定义不同的参数表来区分不同的函数定义。

函数可以有默认参数，这样就可以更方便的调用函数：

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return True
        if ok in ('n', 'no', 'nop', 'nope'): return False
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint
```

Python Tutorial 4.7.1

4.1.2、 函数默认值构造

有一个很重要的问题是，我的定义相当于一次类型构造，默认值参数只在此时解析一次。而函数调用时不会重新执行默认参数的构造。所以。如果函数使用了字典、列表这样的可变数据类型，而又要在函数体内修改它，可能会出现您意想不到的效果：

```
>>> def grow(l=[]):
...         l.append(len(l))
...         return l
...
>>> grow([2, 2])
[2, 2, 2]
>>> grow([3])
[3, 1]
>>> grow()
[0]
>>> grow()
[0, 1]
>>> grow()
[0, 1, 2]
>>> grow()
[0, 1, 2, 3]
>>>
```

如果不希望出现这样的副作用，可以这样定义函数：

```
>>> def grow(l=None):
...         if l == None:
...                 return []
...         l.append(len(l))
...         return l
```

```
...
>>> grow()
[]
>>> grow()
[]
```

4.1.3、 关键字参数

函数可以通过形如 key=value 的形式定义可选的参数，在函数调用时，可以按顺序传入参数值，也可以通过参数名指定传入所需的参数。

```
>>> def foo(a, b, x = 1, y = 1):
...         return a*x + b*y
...
>>> foo(6, 7)
13
>>> foo(6, 7, 4, 2)
38
>>> foo(6, 7, 4)
31
>>> foo(2, 5, y = 9)
47
>>> foo(5, 5, x=12, y=7)
95
>>> foo(a=12, b=7, x=4, y=3)
69
>>> foo(x=7, y=12, a=9, b=3)
99
```

4.1.4、 参数封装与拆封

如果需要将一个序列中的各元素做为参数来调用函数，可以使用*运算符，这称为参数拆封：

```
>>> args = [5, 25, 5]
>>> range(*args)
[5, 10, 15, 20]
```

相应的，在定义函数的时候，可以通过*运算符指定一个变长的参数列。函数体执行的时候会得到这个 tuple 类型的参数列：

```
>>> def foo(*args):
...         return args
```

```
...
>>> foo(*[3, 3, 2])
(3, 3, 2)
>>>
```

也可以通过**操作符定义关键字参数的拆封和解封：

```
>>> def foo(**args):
...         return args
...
>>> foo(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
>>> foo(**{'a':1, 'b':2, 'c':3})
{'a': 1, 'c': 3, 'b': 2}
```

**操作的对象是一个字典。对于函数 foo，以上两种调用方式是等价的（笔者非常喜欢用这种方式实现简单的 ORM 功能）。

4.1.5、 我的名字叫 lambda

在 python 家族中，我是函数的一种简写形式，通常用于在运行期定义临时的匿名函数：

```
>>> f = lambda x, y:x**2+y**2
>>> f(20, 5)
425
```

4.1.6、 文档字符串

在 python 定义后面加入一行字符串的话，Python 解释器会视其为该定义的说明文档，可以通过 help 得到它：

```
>>> def foo(**args):
...         'Sample for Unpacking Argument Dictionary'
...         return args
...
>>> help(foo)
Help on function foo in module __main__:

foo(**args)
    Sample for Unpacking Argument Dictionary
```

4.2、 我的名字叫类

我是类，大家轮流介绍自己的时候，我觉得我已经等太久了。其实，你在前面一定已经

听说过很多与我有关的事情了。定义一个类型其实很简单：

```
>>> class c(object):pass
...
>>> c
<class '__main__.c'>
```

想必您在前面已经见过这样的例子了。定义一个最简单的类型是很容易的，下面我向您介绍如何在定义中加入更多的内容：

4.2.1、 成员函数

为一个类型加入成员方法并不复杂：

```
>>> class c(object):
...         def foo(self, a, b):
...                 return a%b
...
>>> o = c()
>>> o.foo(5, 2)
1
```

这是普通的调用方法，与C++的对象成员调用并无二致。注意 foo 的定义，我们引入了三个形式参数：self，a，b，但是对象调用的时候只使用了两个。实际上 Python 类型的成员方法总是会有至少一个参数，这个参数在对象调用方法的时候是不可见的。它就是 C++中的 this 指针。在 Python 中，这个参数必须显式定义。通常这个参数命名为 self，这只是个约定，您当然可以用其它的名字，但有些 Python 的开发工具可能会依赖这个变量名。下面这种非常规的调用方法可能会对您理解 self 参数有所帮助：

```
>>> c.foo(o, 5, 2)
1
```

上例中的 c 和 o 来自前一个例子。我们可以看到，对象引用在实质上相当于将对象传入 self 形参。

在 python 中，所有的成员方法都是虚的，这充分满足了动态类型的需要。

4.2.2、 构造函数与成员变量定义

如前所见，在 Python 中，对象的自引用变量需要在运行时显式引入。所以，您不能像定义 C++类型那样，在定义类型结构时直接定义成员变量。所有需要在对象建立时就存在的成员变量，可以在构造函数中生成。Python 类型的定义中，名为__init__的函数即为构造函数。因为构造函数是一个成员方法，您可以通过引用 self 为对象添加成员变量：

```
>>> class vector3(object):
...         def __init__(self, x, y, z):
...                 self.x=x
...                 self.y=y
...                 self.z=z
...         def __abs__(self):
...                 return
(self.x**2+self.y**2+self.z**2)**(.5)
...
>>> v = vector3(1, 1, 1)
>>> abs(v)
1.7320508075688772
```

这里演示了一个简单的三维向量类型，系统函数 abs 调用参数对象的__abs__成员并返回其结果。注意在成员方法中引用对象成员变量时，一定要完整给出 self 前缀。

4.2.3、 一些特殊方法

您可以在定义类型时使用一些特殊方法来得到需要的功能，这里我们介绍几种常用的方法：

__str__，这个方法供 str()函数调用，为对象提供字符串转型行为。

__repr__，这个方法为对象提供描述能力，理想状态下，eval(x.__repr__())==x。当然，这个目标并不保证达到。

__del__，您可以将这个方法理解为对象的析构函数。它在对象实际被虚拟机回收时调用。可以用来回收文件句柄或网络端口等资源。因为 Python 虚拟机依据引用计数管理内存，可能说无法准确预期__del__被调用的时机。

类型继承

Python 的继承并不复杂，前面的类型定义中您已经见过了这种定义，class c(object)实际上就是继承了 object 类型：

4.2.4、 关键字 del

关键字 del 用于删除命名或者对象引用。也用于删除序列元素：

```
>>> l = [1, 2, 3, 4]
>>> del l[2]
>>> l
[1, 2, 4]
>>> d = {'x':1, 'y':1, 'z':1}
>>> del d['y']
>>> d
{'x': 1, 'z': 1}
```

```
>>> del d
>>> d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined
```

4.2.5、 *类型信息*

Python 类型包含丰富的运行时信息，例如我们可以查看__dict__来查看对象的成员（系统函数 dir 内部执行这个操作）。您可以在一些专业的 Python 书籍和 Python Help 查看详细的内容。

```
>>> class c(object):
...         def foo(self):
...                 return 'a c object'
...
>>> class d(c):
...         def foo(self):
...                 return 'a d object'
...
>>> x = c()
>>> x.foo()
'a c object'
>>> y = d()
>>> y.foo()
'a d object'
>>> x.f()
'c.f'
>>> y.f()
'c.f'
>>> issubclass(d, c)
True
```

从上例中我们可以看到虚函数继承和重载的效果。

4.2.6、 *迭代器与生成器*

回顾 xrange 的使用，您可能还记得这并不是一个真正的序列，而是一个迭代对象，我们可以通过一种称为生成器（generator）的技术来方便的实现这个功能：

```
>>> def xrange(start, end, step):
...         while end > start:
...                 start += step
...                 yield start
```

```
...
>>> xrange(20, 50, 10)
<generator object at 0x01AF3AA8>
>>> xrange(20, 50, 10)
<generator object at 0x01AF3AA8>
>>> for i in xrange(2, 10, 2):
...         print i
...
4
6
8
10
```

关键字 yield 不同于 return 之处在于，yield 返回值之后并不结束函数，而是重新回到函数内部继续执行。而现代的 Python yield 关键字还允许向函数内部传入参数。这带来了很多有待发掘的设计思想。有关参数化生成器的详细信息，您可以在更为专业的 Python 技术文章中见到。

第5节 我的代码组织

5.1、 我是模块

5.1.1、 模块定义

Python 脚本可以通过本文格式保存，并且做为运行时组件在运行时导入代码中，人们通常称我为模块。模块名与代码文件的主文件名相同。例如，如果您编写了一个名为 Items.py 的程序，可以使用 import Items 来引入。

5.1.2、 模块引用

以下代码从标准模块 sys 中导入了环境路径变量 path：

```
>>> from sys import path
>>> path
['', 'C:\\WINDOWS\\system32\\python25.zip',
'd:\\Python25\\DLLs', 'd:\\Python25\\lib',
'd:\\Python25\\lib\\plat-win', 'd:\\Python25\\lib\\lib-tk',
'd:\\Python25', 'd:\\Python25\\lib\\site-packages',
'd:\\Python25\\lib\\site-packages\\PIL',
'd:\\Python25\\lib\\site-packages\\win32',
'd:\\Python25\\lib\\site-packages\\win32\\lib',
'd:\\Python25\\lib\\site-packages\\Pythonwin',
'd:\\Python25\\lib\\site-packages\\wx-2.8-msw-unicode']
```

您也可以导入模块，并利用限定名引用其中的成员：

```
>>> import sys
>>> sys.path
['', 'C:\\WINDOWS\\system32\\python25.zip',
'd:\\Python25\\DLLs', 'd:\\Python25\\lib',
'd:\\Python25\\lib\\plat-win', 'd:\\Python25\\lib\\lib-tk',
'd:\\Python25', 'd:\\Python25\\lib\\site-packages',
'd:\\Python25\\lib\\site-packages\\PIL',
'd:\\Python25\\lib\\site-packages\\win32',
'd:\\Python25\\lib\\site-packages\\win32\\lib',
'd:\\Python25\\lib\\site-packages\\Pythonwin',
'd:\\Python25\\lib\\site-packages\\wx-2.8-msw-unicode']
```

解释器执行 import 的时候，会先在程序执行目录下查找，如果没有的话，就查找 sys.path 中包含的路径，这个路径相当于系统环境变量 PYTHONPATH。如果在这个路径下也没有找到，会查找 Python 的默认依赖路径，在 UNIX 下，这个路径通常是 .:/usr/local/lib/python。

5.1.3、 Python 模块的编译

实际上，Python 解释器在第一次 import 一个 py 文件的时候，就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。您甚至可以只发布 pyc 文件。

5.1.4、 文件编码

如果代码中包含英语之外的字符，您需要用一种支持多语言的编码形式保存这个文件，并在文件的第一行写入形如下例的编码声明：

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

uft-8 是一种广受欢迎的选择。如果在混合开发时，您使用的 C++框架不能很好的支持 unicode，也可以使用系统相关的编码方式，例如 Windows 上可以使用 mbcs。

5.2、 我是包

Python 的模块可以按目录组织为包（package），当解释器读取到某个目录下存在一个名为__init__.py 的脚本文件时，就视其为一个包。如果您不需要定制包的结构，这个文件可以为空。多级目录还可以递归引用：

```
>>> import os.path
>>> dir(os.path)
['__all__', '__builtins__', '__doc__', '__file__', '__name__',
'_getfullpathname', 'abspath', 'altsep', 'basename',
'commonprefix', 'curdir', 'defpath', 'devnull', 'dirname',
```

```
 'exists', 'expanduser', 'expandvars', 'extsep', 'getatime',
 'getctime', 'getmtime', 'getsize', 'isabs', 'isdir', 'isfile',
 'islink', 'ismount', 'join', 'lexists', 'normcase',
 'normpath', 'os', 'pardir', 'pathsep', 'realpath', 'sep',
 'split', 'splitdrive', 'splitext', 'splitunc', 'stat',
 'supports_unicode_filenames', 'sys', 'walk']
```

以上代码引入了系统包 os 的子模块 path。

关于模块和包的更详细规则，您可以在 Python 的帮助文档，特别是 Python Tutorial 的第 6 章，Modules 中见到。