# Python 学习笔记

王纯业
LeafWang@163bj.com or AnnCharles@tom.com

2003年08月26日

# 目 录

**第一章 Python介绍** **1**

§1.1 开始 . . . 2

§1.1.1 如何运行Python程序 . . . 2

§1.2 一切皆为对象 . . . 3

§1.3 基本类型 . . . 3

§1.3.1 整型(integer) . . . 3

§1.3.2 浮点型(float) . . . 3

§1.3.3 None类型 . . . 3

§1.3.4 逻辑表达式 . . . 3

§1.4 列表(list) . . . 6

§1.4.1 创建list . . . 6

§1.4.2 list的下标和子list . . . 6

§1.4.3 子list的提取 . . . 7

§1.4.4 处理list的方法 . . . 8

§1.4.5 用list 模拟其他常用数据结构 . . . 10

§1.4.6 list comprehension . . . 12

§1.5 字符串(string) . . . 13

§1.5.1 字符串的表示 . . . 13

§1.5.2 转换成为其他类型 . . . 14

§1.5.3 字符串的操作 . . . 14

§1.6 元组(tuple) . . . 22

§1.7 序列(sequence) . . . 23

§1.8 字典(dictionary) . . . 23

§1.8.1 简单例子 . . . 24

§1.8.2 dictionary的操作 . . . 24

§1.9 程序流程 . . . 27

§1.9.1 分支结构if . . . 27

§1.9.2 循环结构for，while，break，continue，range() . . . 29

§1.10 函数 . . . . . 32
§1.10.1 基本函数的用法 . . . . . 32
§1.10.2 参数个数可选，参数有默认值 . . . . . 33
§1.10.3 改变函数参数赋值顺序 . . . . . 34
§1.10.4 个数可变参数 . . . . . 34
§1.10.5 Doc String 函数描述 . . . . . 36
§1.10.6 lambda函数 . . . . . 36
§1.10.7 函数的作用域(scope) . . . . . 37
§1.10.8 嵌套函数(nested) . . . . . 38
§1.10.9 function的参数传递 . . . . . 38
§1.11 模块(module)和包(package) . . . . . 39
§1.11.1 创建一个module . . . . . 39
§1.11.2 怎么查找module . . . . . 41
§1.11.3 package(包) . . . . . 42
§1.12 name space(命名空间) . . . . . 42
§1.13 类 . . . . . 46
§1.13.1 初始化函数 . . . . . 46
§1.13.2 方法(method) . . . . . 47
§1.13.3 属性(property) . . . . . 48
§1.13.4 继承(inherit) . . . . . 49
§1.13.5 重载(overload) . . . . . 49
§1.13.6 class 的attribute . . . . . 49
§1.13.7 Abstrace Class . . . . . 50
§1.13.8 Python的面向对象编程 . . . . . 50
§1.13.9 Python 中class 特殊的methord . . . . . 50
§1.14 异常处理(exception) . . . . . 57
§1.14.1 什么是异常处理，为什么要有异常处理 . . . . . 57
§1.14.2 捕获exception(异常) . . . . . 59
§1.14.3 抛出exception . . . . . 63

**第二章 开发Python 使用的工具** **66**

§2.1 使用Emacs 编辑Python 程序 . . . . . 66

§2.1.1 安装python mode . . . . . 66
§2.1.2 python mode 的基本特性 . . . . . 67
§2.1.3 常用功能举例 . . . . . 67
§2.2 其他编辑器 . . . . . 69
§2.3 调试程序 . . . . . 70
§2.3.1 使用DDD 和pydb 调试python 程序 . . . . . 70

**第三章 Python 的常用模块 71**
§3.1 内置模块 . . . . . 71
§3.1.1 常用函数 . . . . . 71
§3.1.2 类型转换函数 . . . . . 73
§3.1.3 用于执行程序的内置函数 . . . . . 75
§3.2 和操作系统相关的调用 . . . . . 78
§3.2.1 打开文件 . . . . . 81
§3.2.2 读写文件 . . . . . 81
§3.2.3 关闭文件 . . . . . 82
§3.3 regular expression . . . . . 82
§3.3.1 简单的regexp . . . . . 83
§3.3.2 字符集合 . . . . . 83
§3.3.3 重复 . . . . . 85
§3.3.4 使用原始字符串 . . . . . 85
§3.3.5 使用re 模块 . . . . . 85
§3.3.6 高级regexp . . . . . 87
§3.3.7 分组(Group) . . . . . 88
§3.3.8 Compile Flag . . . . . 90
§3.4 用struct模块处理二进制数据 . . . . . 91
§3.5 用Cmd 模块编写简单的命令行接口 . . . . . 93
§3.5.1 简单的例子 . . . . . 94
§3.5.2 定义默认命令 . . . . . 95
§3.5.3 处理EOF . . . . . 95
§3.5.4 处理空命令 . . . . . 96
§3.5.5 命令行自动补齐 . . . . . 96

§3.5.6 改变标准输入输出 . . . . . 96
§3.5.7 改变提示符 . . . . . 97
§3.5.8 提供在线帮助功能 . . . . . 97
§3.5.9 运行Shell 的功能 . . . . . 98
§3.5.10 getattr 功能的使用 . . . . . 98
§3.6 处理命令行选项 . . . . . 99
§3.6.1 一个简单的例子 . . . . . 99
§3.6.2 带有参数的命令行选项 . . . . . 100
§3.6.3 optparser 模块 . . . . . 101
§3.7 关于时间的模块 . . . . . 109

**第四章 Tkinter 编程** **111**
§4.1 Tkinter介绍 . . . . . 111
§4.1.1 什么是Tkinter . . . . . 111
§4.2 Hello Tkinter 程序 . . . . . 112
§4.2.1 简单的例子 . . . . . 112
§4.2.2 另一个简单程序 . . . . . 114
§4.3 Widget 的配置 . . . . . 115
§4.4 Geometry Manager(几何管理器) . . . . . 116
§4.4.1 Pack 管理器 . . . . . 117
§4.4.2 Grid 管理器 . . . . . 120
§4.5 Widget 的样式 . . . . . 122
§4.5.1 颜色 . . . . . 122
§4.5.2 字体 . . . . . 123
§4.5.3 文字格式化 . . . . . 126
§4.5.4 边框 . . . . . 126
§4.5.5 鼠标 . . . . . 128
§4.6 事件和事件的帮定 . . . . . 130
§4.6.1 一个简单的例子 . . . . . 130
§4.6.2 事件处理函数 . . . . . 132
§4.6.3 事件的层次 . . . . . 132
§4.6.4 同一个事件的多个处理函数 . . . . . 135

§4.6.5 和WM 相关的事件绑定 . . . . . 136
§4.7 常用的应用程序使用的Widget . . . . . 137
§4.7.1 基本窗口 . . . . . 137
§4.7.2 菜单 . . . . . 137
§4.7.3 工具栏 . . . . . 141
§4.7.4 状态栏 . . . . . 141
§4.7.5 标准对话框 . . . . . 143

**第五章 Python的扩展** **149**
§5.1 用C编写扩展模块 . . . . . 149

# 表 格

1.5-1 字符串中的转意字符 . . . . . 14
1.5-2 字符串转换为其他类型 . . . . . 14
1.5-3 判断字符串是否属于某种类别的函数 . . . . . 16

3.1-1 Python 的重用内置函数 . . . . . 72
3.1-2 类型转换函数 . . . . . 74
3.2-3 系统调用举例 . . . . . 78
3.4-4 struct 模块中的格式定义 . . . . . 92
3.4-5 字节顺序定义 . . . . . 93
3.5-6 命令行编辑的快捷键 . . . . . 94
3.7-7 表示时间的 . . . . . 110

4.5-1 tkFont 支持的option . . . . . 125
4.6-2 事件描述举例 . . . . . 133
4.6-3 描述事件的参数 . . . . . 133

# 第一章

# Python介绍

python是一种解释性的面向对象的语言。python有以下特点。

**解释型** 速度偏慢，开发周期短，调试容易，自我扩展性。由于有eval的功能，编写的程序可以在运行时继续开发。这是很多解释型语言的共性。如VB 有Script Engineer，Lisp有EVAL，导致Emacs的强大扩展性。

**面向对象** 在Python中，类(class)，函数(function)，模块(module) 等等都是对象。夸张的说，他比Java，C++ 更加OO( Object Oriented)。别让我卷入“什么叫做更OO? Java更OO?”的讨论中，用过就知道了。每个语言都有各自的优缺点，而Python 吸取了更多语言的优点。

**可扩展** Python 使用C语言写的，很自然的用C或者C++语言扩展Python的特性。如编写新的模块，增加新的数据类型等等。

**可嵌入** Python的解释器引擎可以很容易的嵌入到用户自己的C或者C++语言程序项目中，使你的程序可以支持脚本编程，扩展功能。现在就有Python的模块可以扩展VIM和Emacs的功能。

**小内核** Python的核心解释器是很小的。

**动态类型** 不需要事先声明(declare)变量的类型，直接使用变量就像很多其他解释型语言VB，VB Script，JScript，PERL一样。

**强类型** 尽管变量不用声明(declare)类型，但是一旦变量有了值，那么这个值是有一个类型的。不能将string类型的变量直接赋值给一个integer类型的变量，而是需要类型转换。这和VB，VBScript，JScript，PERL不一样，和Java 像。

## §1.1 开始

### §1.1.1 如何运行Python程序

Python 是一种解释型的编程语言，和其他解释型编程语言，如Perl，VB，Bash，Awk，Sed 都有类似的，运行办法如下。

- 第一种方法是交互式。假设你是在Unix 环境下，$ 是命令行提示符号。

```
$python
Python 2.2 (#1, Apr 12 2002, 15:29:57)
[GCC 2.96 20000731 (Red Hat Linux 7.2 2.96-109)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print "Hello World"
Hello World
>>>                                  #按 CTRL+D 退出
```

在Windows 中也有类似的环境。本文中大多数例子都是在交互式运行的。>>>和...表示Python 的提示符号。Bash 下的很多命令行编辑命令都是可以使用的，如C-a ，C-e ，C-w 等等，因为他们都是使用readline 库。

- 还可以把程序写在一个文件中，用python 解释器调用。

```
$cat test.py
print "Hello World"
$python test.py
Hello World
$cat test.py | python #管道操作
Hello World
$python < test.py      #输入重定向
Hello World
```

- 把程序作为一个可以运行的Unix 脚本文件。

```
$cat test.py
#!/usr/bin/python
print "Hello World"
$ls -l test.py
-rw-r--r--    1 chunywan mnw            40 Sep 12 13:34 test.py
$chmod +x test.py
$ls -l test.py
```

```
-rwxr-xr-x    1 chunywan mnw            40 Sep 12 13:34 test.py
$./test.py
Hello World
```

注意到第一行的特殊注释语句，这是Unix 下通用的方法。Perl，Tl/Tcl，Sed，awk，Bash等等都是这么用的。

值得注意的是注释语句，#后面表示注释。和bash一样。注释语句在调试程序的时候，是很有用的，还可以提高程序的可读性。

## §1.2 一切皆为对象

类(class)，函数(function)，模块(module) 等等都是对象。“一切都是对象，对象都有名字”，这是我的一个总结。

## §1.3 基本类型

### §1.3.1 整型(integer)

看起来像整数的表达式就是整型。有以下运算

| 操作符 | 意义 | 例子 | 结果 |
|---|---|---|---|
| + | 加法 | 1+1 | 2 |
| − | 减法 | 1-1 | 0 |
| * | 乘法 | 2*10 | 20 |
| / | 除法 | 10/3 | 3 |
| % | 模 | 10/3 | 1 |
| ** | 幂 | 2**4 | 16 |
| divmod | 除法 | divmod(10，3) | (3，1) |

### §1.3.2 浮点型(float)

看起来像浮点数的表达式就是浮点数。和整型类似。注意，1是整型，1.0是浮点型。逻辑表达式1 == 1.0 是真值。注意，浮点数的运算是有误差的。用于科学计算的时候，需要其他的模块支持。这是因为浮点运算器的只能表示有限位数。

### §1.3.3 None类型

None是一个特殊的常量，表示出错。在以后可以看到他的用处。

### §1.3.4 逻辑表达式

Python中没有boolean类型。就像C语言中，除了0以外，其他都是真。但是Python中的“假”有很多，包括

None,0,0.0,"" (空字符串),[](空list),()(空tuple),{}(空dictionary)

Python中有以下逻辑运算符。

| | |
|---|---|
| and | 逻辑与 |
| or | 逻辑或 |
| not | 逻辑反 |

```
>>> def getTrue():
...     print "debug: true"
...     return 1
...
>>> def getFalse():
...     print "debug: false"
...     return 0
...
>>> if getTrue() and getFalse():
...     print "ok"
... else:
...     print "bad"
...
debug: true          # 先运行 getTrue
debug: false         # 如果 getTrue 返回真,那么运行getFalse
bad                  # 整个结果是 getFalse 的返回结果.
>>> if getFalse() and getTrue():
...     print "ok"
... else:
...     print "bad"
...
debug: false      #先运行 getFalse
bad               #如果 getFalse 返回假,那么就不运行 getTrue
>>>
```

值得注意的是and 和or ，对于and 来说，例如expr1 and expr2，如果expr1 是逻辑假，那么就总的表达式直接返回expr1 ，也许是空字符串，或者空list，而不再计算expr2，也就是说expr2 中如果有函数调用的话，也不会执行。如果expr1 是逻辑真，那么才会计算expr2 ，整个逻辑表达式就返回expr2。对于or 来说，例如expr1 or expr2，如果expr1 是逻辑真，那么就总的表达式直接返回expr1 ，而不再计算expr2，也就是说expr2 中如果有函数调用的话，也不会执行。如果expr1 是逻辑假，那么才会计算expr2 ，整个逻辑表达式就返回expr2。这样就有一种常用的特殊表达方式:

```
<cond> and <expr_true> or <expr_false>
```

他有一个很有用的语义是: 如果cond 成立，那么返回expr_true，否则返回expr_false。他没有违反什么规则，只是从左到右依次。让我们来检查一下为什么:

| cond | expr_true | expr_false | 总的结果 |
|---|---|---|---|
| 真 | 真 | 任何(不计算) | expr_true |
| 假 | 任何(不计算) | 真 | expr_false |
| 假 | 任何(不计算) | 假 | expr_false |
| 真 | 假 | 假 | expr_false |
| 真 | 假 | 真 | expr_false |

从这个表上可以看到:

- 头一组说明了，如果cond 是真，那么整个表达式就是expr_true
- 第二组说明了，如果cond 是假，那么整个表达式就是expr_false
- 最后一组说明了，我说谎了。记住如果expr_true 有可能是假的时候，不要这么做，因为有可能打破原来的语义，cond 是真的时候，整个表达式却是expr_false。解决这个问题的办法就是:

  ```
  (<cond> and (<expr_true>,) or (<expr_false>,))[0]
  ```

  最外面的括号把整个表达式括起来，改变计算优先级，否则[0] 的优先级最高。中间的括号和逗号，是一个tuple 的表达式。这样(<expr_true>,)永远是真，因为他是含有一个元素的tuple，最后的[0] 表示提取出这个元素来。幸运的是，通常这个表达式是用在比较简单的情况下，简单的赋值语句，expr_true和expr_false 一般都是常量，而且是真。

  ```
  >>> x= cond and "OK" or "BAD"
  >>> x
  'OK'
  >>> cond = 0
  >>> x= cond and "OK" or "BAD"
  >>> x
  'BAD'
  >>>
  ```

  同样道理:

```
<cond> or <expr_false> and <expr_true>
```

也有类似的语义，但是不要去使用这种表达方式，不是因为这个表达式不合法，而是因为他不符合习惯。编程语言和自然语言有类似的地方，就象我们平时会说，“走，吃饭去”，

但是我们不会说，“走，进餐去”，或者“走，进食了”，不是因为那么说就是错的，是因为不合习惯。那就会问，为什么不合习惯，习惯从哪里来？原因是从C 语言中来。C 语言中有

```
cond ? true_expr: false_expr
```

这是一个三元操作符，众多的C 语言的使用者用了几十年了，和第一种语义是类似的。总之，忘了后面那种方式，只知道第一种就好了。

## §1.4 列表(list)

list 类似于C++中的vector，C 语言中的数组，用于存储顺序结构。如，

一年中每个月的天数，可以表示为

```
[ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]
```

一年中每个月的名称，可以表示为

```
[ 'January', 'February', 'March', 'April', 'May',
                'June', 'July', 'August', 'September', 'October',
                'November', 'December' ]
```

每个星期的名字，可以表示为

```
[ 'Monday','Tuesday', 'Wednesday', 'Thursday' ,
      'Friday', 'Saturday', 'Sunday']
```

### §1.4.1 创建list

```
a=['1','2']
```

用[]括起来，表示一个list，中间的各个元素可以是任何类型，用逗号分隔。

### §1.4.2 list的下标和子list

list的下标从0开始，和C语言类似，但是增加了负下标的使用。

| | |
|---|---|
| -len | 第一个元素 |
| ... | ... |
| -2 | 倒数第二个元素 |
| -1 | 最后一个元素 |
| 0 | 第一个元素 |
| 1 | 第二个元素 |
| ... | ... |
| len-1 | 最后一个元素 |

```
>>> a=[0,1,2,3,4,5,6]
```

```
>>> i=-len(a)
>>> while i < len(a):
...     print "a[",i,"]=",a[i]
...     i=i+1
...
a[ -7 ]= 0
a[ -6 ]= 1
a[ -5 ]= 2
a[ -4 ]= 3
a[ -3 ]= 4
a[ -2 ]= 5
a[ -1 ]= 6
a[ 0 ]= 0
a[ 1 ]= 1
a[ 2 ]= 2
a[ 3 ]= 3
a[ 4 ]= 4
a[ 5 ]= 5
a[ 6 ]= 6
```

注意while语句的缩进。会在后面介绍。a[-8] 和a[9] 都会引起越界。

```
>>> a[-8]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> a[7]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

### §1.4.3 子list的提取

可通过下标指定范围，用于提取出一个list的一部分。下标表明位置，一个起始位置，一个结束位置，中间用冒号分隔。如果不指定起始位置，起始位置是0 ，如果不指定结束位置，结束位置是−1。子list 表示包括起始位置处的元素，一致到结束位置，但是不包括结束位置的元素。

负下标也可以参与定义起始位置，或者结束位置。

```
>>> a[1:4]
```

```
[1, 2, 3]
>>> a[:]
[0, 1, 2, 3, 4, 5, 6]
>>> a[1:]
[1, 2, 3, 4, 5, 6]
>>> a[:3]
[0, 1, 2]
>>> a[:]
[0, 1, 2, 3, 4, 5, 6]
>>> a[1:-1]
[1, 2, 3, 4, 5]
```

得到的子list 是原来list 的一个拷贝，一个副本，改变子list 不会改变原来的list。

### §1.4.4 处理list的方法

- `L.append(var)` ，追加元素。

```
>>> a=[0,1,2,3,4,5]
>>> a.append(6)
>>> a
[0, 1, 2, 3, 4, 5, 6]
>>>
```

- `L.count(var)` 计算var 在L中出现的次数。

```
>>> a=[0,2,2,2,3,3]
>>> a.count(2)
3
>>> a.count(3)
2
```

- `len(L)` 返回L 的长度。

```
>>> a=[1,2,3]
>>> len(a)
3
```

- `L.extend(list)` ，将list追加在L后面。

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

```
>>> a
[1, 2, 3, 4, 5, 6]
```

- `L.index(var)` 返回var在L中的位置，若无，则抛出异常。

```
>>> a=[1,2,3,4,4,5]
>>> a.index(4)
3
>>> a.index(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
```

- `L.insert(index,var)` 在index处，插入var，其余元素向后推。如果index大于list的长度，就在最后添加。如果index小于0，就在最开始处添加。

```
>>> a=[1,2]
>>> a.insert(0,1)
>>> a
[1, 1, 2]
>>> a.insert(100,100)
>>> a
[1, 1, 2, 100]
>>> a.insert(2,3)
>>> a
[1, 1, 3, 2, 100]
```

- `L.pop()` 返回最后一个元素，并且删除最后一个元素。`L.pop(index)` 返回index处的元素，并且删除该元素。

```
>>> a=[0,1,2,3,]
>>> a.pop()
3
>>> a.pop(0)
0
>>> a
[1, 2]
>>>
```

- `L.remove(var)` 找到var，并且删除之，若无，抛出异常。

```
>>> a=["a","b","a","c","a","d"]
>>> a.remove("a")
>>> a
['b', 'a', 'c', 'a', 'd']
>>> a.remove("a")
>>> a
['b', 'c', 'a', 'd']
```

- `L.reverse()` 将L倒序。

```
>>> a=[1,2,3,4,5]
>>> a.reverse()
>>> a
[5, 4, 3, 2, 1]
```

- `L.sort()`，将a排序，a中元素若类型不同，结果自己看一看吧。一般不会这么做。既然你都不确定list中的元素的类型是否一样，你还要排序，那么你期待是什么结果?

```
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
>>> a=[1,"abc",2,"xxx"]
>>> a.sort()
>>> a
[1, 2, 'abc', 'xxx']
```

`L.sort(func)`，将用func作为比较函数，将排序。`func(x,y)`，返回−1，0，1分别表示小于，等于，大于。

### §1.4.5 用list 模拟其他常用数据结构

**用list模拟堆栈stack**

```
>>> a=[]
>>> a.append(0) #压栈
>>> a.append(1)
>>> a.append(2)
>>> a
[0, 1, 2]
>>> a.pop()      #出栈
2
```

```
>>> a.pop()
1
>>> a.pop()
0
>>>
```

用list模拟队列queue

```
>>> a.insert(-1,0) #入队
>>> a.insert(-1,1)
>>> a.insert(-1,2)
>>> a
[2, 1, 0]
>>> a.pop()        #出队
0
>>> a.pop()
1
>>> a.pop()
2
>>> a
[]
```

或者

```
>>> a.append(0)
>>> a.append(1)
>>> a.append(2)
>>> a.append(3)
>>> a
[0, 1, 2, 3]
>>> a.pop(0)
0
>>> a.pop(0)
1
>>> a.pop(0)
2
>>> a.pop(0)
3
```

用list模拟树tree

```
>>> leaf1=[0,1]
>>> leaf2=[2,3]
>>> leaf3=[4,5]
>>> leaf4=[6,7]
>>> branch1=[leaf1,leaf2]
>>> branch2=[leaf3,leaf4]
>>> root=[branch1,branch2]
>>> root
[[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
```

### §1.4.6 list comprehension

语法:

```
[ <expr1> for k in L if <expr2> ]
```

语义:

```
returnList=[]
for k in L:
   if <expr2>: returnList.append(<expr1>)
return returnList;
```

返回一个list，list的元素由每一个expr1 组成，if 语句用于过滤，可有可无。

```
>>> a=["123","456","abc","Abc","AAA"]
>>> [ k.center(9) for k in a ]
['   123   ', '   456   ', '   abc   ', '   Abc   ', '   AAA   ']
#得到a中仅由字母组成的字符串，并把他变成大写的。
>>> [ k.upper() for k in a if k.isalpha() ]
['ABC', 'ABC', 'AAA']
#得到a中仅由大写字母组成的字符串，并把他变成小写。
>>> [ k.lower() for k in a if k.isupper() ]
['aaa']
#得到a中仅有数字构成的字符串，并把他变成整数类型。
>>> [ int(k) for k in a if k.isdigit() ]
[123, 456]
```

过滤(filter)，和映射(mapping)和二为一。

尽管这个用法很简单，但是却十分有用，程序变得简洁，可读性强。有很多时候，我们使用循环语句，仅仅做很简单的事情，但是循环语句本身是很不好读的，因为循环语句不符

合自然语言中的习惯。例如：上面的例子中，有一个表示“把list 中的每个字符串中的字符变成大写”。但是如果用循环语句表示就是: “创建一个空list，对于list 中的每一个元素，把这个字符串变成大写的，添加到新建的list中。”

还有很多例子，让我们很自然的应用这种用法。

把list中的所有元素加 1

```
[ k+1 for k in list]
```

提取出 list 中所有整数变量

```
[ k for k in list if type(k)==types.IntType]
```

把 list 中的所有整数元素增加一

```
[ k + 1 for k in list if type(k)==types.IntType ]
```

我非常喜欢这种表达方式!

## §1.5 字符串(string)

### §1.5.1 字符串的表示

用单引号或双引号构成字符串

```
'hello'
"world"
```

有一些字符是无法用键盘输入的，所以用字符组合表示特殊字符。如表1.5-1。

这是和C 语言十分类似的，还有echo 命令，Java 语言都是这么表示一个特殊字符的。

如果要输入一个十分长的字符串，一行不方便，就用两行。两个字符串中间由空白字符连接，表示一个字符串。如

```
"abc" \
'def'
```

表示“abcdef”，注意和"abc"+"def"是有区别的，带+的表示两个字符串连接到一起，用空白字符连接的，表示一个字符串。

如果要输入一个十分十分长的字符串，就用三个连续的双引号串。如

```
"""
this is a very very long string.
And it contains a lot line.
it is suitable to write document of function.
"""
```

| 代码 | 表示的字符 | 意义 | 备注 |
|---|---|---|---|
| *newline* | 无 | 忽略 | 用于连接多个短行，构成一个长行 |
| \\ | \ | 反斜杠 | |
| \’ | ’ | 单引号 | |
| \" | " | 双引号 | |
| \a | 0x07 | 响铃 | ASCII 控制字符 |
| \b | 0x08 | 退格 | ASCII 控制字符 |
| \f | 0x0c | Formfeed (FF) | ASCII 控制字符 |
| \n | 0x0a | 换行 | ASCII 控制字符 |
| \N{name} | | Unicode 字符 | 只针对于Unicode |
| \r | 0x0d | 回车 | ASCII 控制字符 |
| \t | 0x09 | 水平制表符 | |
| \v | 0x0b | 垂直制表符 | |
| \u{xxxx} | | Unicode编码 | 只针对于Unicode |
| \ooo | 0x*ooo* | ooo 表示的八进制字符 | |
| \xhh | 0x*hh* | hh 表示的十六进制字符 | |

表 1.5-1: 字符串中的转意字符

| 函数名称 | 功能 | 例子 | 结果 |
|---|---|---|---|
| float(str) | 变成浮点数 | `float("1e-1")` | 0.1 |
| int(str) | 变成整型 | `int("12")` | 12 |
| int(str,base) | 变成base进制整数 | `int("11",2)` | 2 |
| long(str) | 变成长整型 | `int("12")` | 12 |
| long(str,base) | 变成base进制长整数 | `int("11",2)` | 2 |

表 1.5-2: 字符串转换为其他类型

### §1.5.2 转换成为其他类型

### §1.5.3 字符串的操作

**联接两个字符串**

```
>>> "Hello" + " " + "World"
’Hello World’
```

**大小写转换**

```
>>> "this is test".capitalize() #首字母大写
```

```
'This is test'
>>> "THIS IS TEST".capitalize()
'This is test'
>>> "THIS IS TEST".lower()      #全部变成小写
'this is test'
>>> "this is test".upper()      #全部变成大写
'THIS IS TEST'
>>> "This Is Test".swapcase()   #大小写互换
'tHIS iS tEST'
```

### 得到字符串长度

```
>>> len("Hello World")
11
```

### 得到字符串的子串

可以把一个字符串看成一个含有很多字符的list，这样可以用和list相同的方式得到子串。下标从0开始。负数代表反向。

```
>>> str="Hello World"
>>> str="Hello"
>>> i=-len(str)
>>> while i < len(str):
...     print "str[",i,"]=",str[i]
...     i=i+1
...
str[ -5 ]= H
str[ -4 ]= e
str[ -3 ]= l
str[ -2 ]= l
str[ -1 ]= o
str[ 0 ]= H
str[ 1 ]= e
str[ 2 ]= l
str[ 3 ]= l
str[ 4 ]= o
>>>
```

冒号可以表示范围，得到子字符串。冒号两边的数表示下界和上界，下界和上界都是可选，缺省上届为-1，下界为0。

```
>>> str="Hello"
>>> str[:]
'Hello'
>>> str[1:2]
'e'
>>> str[1:-1]
'ell'
>>> str[-1:2]
''
```

负下界只是代表位置，不代表大小。所以从这个意义上讲，−1 要比2 大。str[-1:2]是空字符串。应该注意求子字符串是真正的原来的字符串的拷贝，改变得到的字符串是不会改变原来的字符串的。不可以用子字符串的方式改变原来的字符串。如:

```
>>> a="abcd"
>>> b=a[1:2]
>>> b="x"       #改变 b
>>> a
'abcd'          #不会影响 a
>>> a[1]='abc'  #
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>>
```

### 判断字符串是否属于某种类别

表1.5-3中的方法可以判断字符串中是否符合某些条件。字符串满足条件，就返回1，否则，返回0。

| 名称 | 条件 |
|---|---|
| s.isalnum() | 所有字符都是数字0123456789 或者字母A-Z，a-z |
| s.isalpha() | 所有字符都是字母a-z 或者A-Z |
| s.isdigit() | 所有字符都是数字0123456789 |
| s.islower() | 所有字符都是小写字母a-z |
| s.isupper() | 所有字符都是大写字母A-Z |
| s.istitle() | 所有单词都是首字母大写，像个title |
| s.isspace() | 所有字符都是空白字符\n,\t,\r,' ' |

表 1.5-3: 判断字符串是否属于某种类别的函数

### 在字符串中查找子串

- s.find(substring,[start [,end]]) 如果找到，返回索引值，表明是在哪最先找的。如果找不到，返回−1

```
#ok后面的abc，a 的开头位置为 3，
>>> "ok abc abc abc".find("abc")
3
#从位置4以后开始查找。相当于 s[4:].find("abc")
>>> "ok abc abc abc".find("abc",4)
7
#相当于 s[4:9].find("abc")
>>> "ok abc abc abc".find("abc",4,9)
-1
#相当于 s[4:10].find("abc")
>>> "ok abc abc abc".find("abc",4,10)
7
```

- s.rfind(substring,[start [,end]]) 和find()类似，但是是反向查找。

```
>>> "ok abc abc abc".rfind("abc")
11
```

- s.index(substring,[start [,end]] 和find()类似，但是如果找不的substring，就产生一个ValueError的异常。两种方式的不同在于错误处理的不同。

```
>>> "ok abc abc abc".index("abc")
3
>>> "ok abc abc abc".index("abcd")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: substring not found in string.index
```

- s.rindex(substring,[start [,end]]
  和s.index()类似。不同之处在于是反向查找。
- s.count(substring,[start [,end]]) 返回找到substring的次数。

```
>>> "ok abc abc abc".count("abc")
3
```

### 格式化字符串

基本用法是 `s % <tuple>` ，tuple 表示一个参数列表，类似于C语言中printf，把tuple 中的每一个值用字符串表示，表示的格式由s 来确定。

```
>>> print "%s's height is %dcm"%("Charles",180)
Charles's height is 180cm
```

注意到%s,%d和C语言中的printf是类似的，术语叫做conversion specifier，(转换定义)

现在着重讨论conversion specifier。conversion specifier 有以下几个部分构成，

```
%[<mapping key>][<conversion flag>][<Minimum field width>]|
[<precision>][<lenght modifier>]<conversion type>|
```

看着很复杂，其实只要一个%接一个conversion type就可以了。其他的都是可选的。conversion type如下:

- d 表示有符号十进制整数。

  ```
  >>> "%d and %d" % (-1,2)
  -1 and 2
  ```

- i 同d

- o 表示无符号八进制数。

  ```
  >>> "%o and %o" % (16,8)
  20 and 10
  ```

- u 表示无符号整数。

  ```
  >>> "%u and %u"  %(-10,10)
  '4294967286 and 10'
  ```

  注意−10 被解释成为无符号的数时候，表示一个很大的正数。这是因为在计算机中用补码表示负数，负数的最高位是1，在被解释成为无符号数的时候，就会很大。

- x 表示无符号十六进制(小写)

  ```
  >>> "%x and %x" % (100,200)
  '64 and c8'
  ```

- X 表示无符号十六进制(大写)

  ```
  >>> "%X and %X" % (100,200)
  '64 and C8'
  ```

- e 浮点数，科学表达式法(小写)

```
>>> "%e and %e"%(-100,200)
2.000000e+02
```

- E 浮点数，科学表达式法(大写)

```
>>> "%E and %E"%(-100,200)
2.000000E+02
```

- f 表示浮点数

```
>>> "%f and %f"%(-0.00001,200.0)
'-0.000010 and 200.000000'
```

- g 表示浮点数，如果是小于0.0001或者不够精度，就用科学表示法。

```
>>> "%g and %g"%(-0.00001,2000000.0)
'-1e-05 and 2e+06'
```

- c 把ASCII 整数表示单个字符

```
>>> "%c and %c"%(67,68)
'C and D'
```

可用于变整数为ASCII字符

- r 表示一个字符串，这个字符串使用repr()的返回值

```
>>> "%r"%({"one":1,"two":2})
"{'two': 2, 'one': 1}"
```

和print expr(obj) 打印出来的效果一样。

- s 表示一个字符串，这个字符串使用str()函数的返回值

```
>>> "%s"%({"one":1,"two":2})
"{'two': 2, 'one': 1}"
```

和print str(obj) 打印出来的效果一样。
注意str 和expr 函数的不同。str 是得到一个对象的非正式的描述。expr 是的到一个字符串，是一个合法的Python 表达式语句，可以的到对象本身。

mapping key的作用是用dictionary 代替tuple ，用mapping key作为key表示dictionary中变量。这样带来的好处是，提高了程序的可读性，%d和%(height)比较，后者的可读性要好。而且，同样的一个变量出现两次以上，在tuple中就要写两次，而且要注意tuple中元素的顺序和个数不要错，用mapping key的方式就好多了。

```
>>> print "%s's height is %d cm, %s's weight is %d kg" % \
... ("Charles",170,"Charles",70)
Charles's height is 170 cm, Charles's weight is 70 kg
>>> print "%(name)s's height is %(height)d cm" \
...       ",%(name)s's weight is %(weight)d kg" % \
...     {"name":"Charles","height":170,"weight":70}
Charles's height is 170 cm,Charles's weight is 70 kg
```

name出现两次，在mapping key中，就不用写两次，而且参数的顺序也可以随便写。和locals()函数相配合，会有更好的效果，locals()返回local name space (局部命名空间)，是一个dictionary，其中key是变量的名称，value是变量的值。globals()返回global name space (全局命名空间) 这里只举个例子。在后面的name space(命名空间)有详细介绍。

```
>>> def fun(a,b):
...     print "a is %(a)d,b is %(b)d" %locals()
...
>>> fun(1,2)
a is 1,b is 2
```

minimum field width 表示最小长度。

```
>>> print "%10d,%10d"%(1,2)
         1,         2
```

precision用来表示精度，只对浮点数有作用，带有四舍五入的功能。

```
>>> print "%10.2f,%10.2f"%(10.123456,20.126456)
     10.12,     20.13
```

conversion flag 是对某些类型起作用。#表示用alternate form，如果是8进制，那么就在前面加0，如果是十六进制就在前面加0x 或者0X 对其他类型不起作用。

```
>>> print "%#x,%#X,%#o" %(16,16,16)
0x10,0X10,020
```

0 表示用0补全，使之满足minimum field width。

```
>>> print "%010d,%010d"%(1,2)
0000000001,0000000002
```

-表示左对齐，如果和0同时使用那么0就不起作用。

```
>>> print "%-010d,%-010d"%(1,2)
1         ,2
```

␣，一个空格，表示0和正数前面加个空格，负数前面加-，

```
>>> print "% 10d,% 10d,% 10d"%(-1,2,-0)
        -1,         2,         0
```

+ 表示显示符号，正数显示+，负数显示-

```
>>> print "%+10d,%+10d,%+10d"%(-1,2,0)
        -1,        +2,        +0
```

lenght modifier 被忽略。

**其他简单的格式化功能**

- `s.ljust(width)` 左对齐如果s长度没有width那么长，就在后面补空格，得到一个长度为width的字符串。否则得到字符串s。

```
>>> "Hello World".ljust(1)
'Hello World'
>>> "Hello World".ljust(12)
'Hello World '
>>> "Hello World".ljust(16)
'Hello World     '
```

- `s.rjust(width)` 右对齐和左对齐类似。

```
>>> "Hello World".rjust(16)
'     Hello World'
>>> "Hello World".rjust(1)
'Hello World'
```

- `s.center(width)` 居中类似，在两头补空格，使s的长度为width，如果s的长度本身比width大，那么什么也不做。

```
>>> "Hello World".center(1)
'Hello World'
>>> "Hello World".center(16)
'  Hello World   '
```

- 去掉空白字符`s.lstrip()` 去掉左边的空白字符`s.rstrip()` 去掉右边的空白字符`s.lstrip()` 去掉两边的空白字符

```
>>> "\t a  \t".lstrip()
'a  '
>>> "\t a  \t".rstrip()
```

```
' a'
>>> "\t a  \t".strip()
'a'
```

新版的python 2.2.3中增加了一个可选参数*chars*，表示删除出现在*chars*中的字符。旧版的`s.strip` 相当于`s.strip("\t \n\r")`。

**字符串的合并**(join)**和分解**(split)

`s.join(words)`，`words`是一个只含有字符串的tuple 或者list 。`join`用`s`作为分隔符将`words`中的字符串连接起来，合并为一个字符串。

```
>>> "\n".join(["Hello","World","Python","Said"])
'Hello\nWorld\nPython\nSaid'
>>> print "\n".join(["Hello","World","Python","Said"])
Hello
World
Python
Said
```

`s.split(words)`，`words`是一个字符串，表示分隔符。`split`的操作和`join`相反。将用`s`分解成一个list。

```
>>> "This is a book".split(" ")
['This', 'is', 'a', 'book']
```

## §1.6 元组(tuple)

记住以下规则，基本上就可以掌握tuple。

- tuple是常量list。tuple不能pop，remove，insert等方法。
- tuple 用() 表示，如a=(0，1，2，3，4)，括号可以省略。
- tuple 可以用下标返回元素或者子tuple。
- tuple 可以用于多个变量的赋值。

```
>>> a,b=(1,2)
>>> print a,b
1 2
>>> t=(1,2)
>>> a,b=t
>>> print a,b
1 2
```

```
>>> a,b=b,a+1
>>> print a,b
>>> 2,2
```

- 表示只含有一个元素的tuple的方法是: (1，) 后面有个逗号，用来和单独的变量相区分。

tuple 比list 的性能好，也就是说，不用提供动态内存管理的功能。

## §1.7 序列(sequence)

sequence 包括string，list，tuple。他们都有以下一些通用的操作。

- in 判断某个object 是不是在一个sequence 中。

```
>>> x=12
>>> l=[12,13]
>>> if x in l : print "x is in l"
...
x is in l
>>> str="abcd"
>>> if "a" in str: print " ok "
...
 ok
>>> if 2 in T : print "OK"
...
OK
```

- 得到sequence的长度`len(seq)`
- 通过下标取其中一个元素。 `seq[i]`
- 通过带冒号的下标取子sequence，`seq[start:end]`
- 用+ 表示连接两个sequence
- 用* 表示重复一个sequence，`"a"*3` 表示aaa，`(1,2)*3`表示`(1,2,1,2,1,2)`
- 可以使用list comprehension。

## §1.8 字典(dictionary)

dictionary 是一个无序存储结构。每一元素是一个pair，包括key和value两个部分。key的类型是integer或者string[1]。value的类型随便什么都可以。dictionary中没有重复的key。dictionary中的每一个元素是一个tuple，tuple中有两个元素，这个tuple叫pair，pair中前面的是key，后面的value。可以用`D[key]` 的方式得到value。

[1]或者任何同时含有__hash__和__cmp__ 方法的对象

### §1.8.1 简单例子

```
# 创建一个简单的 dictionary 使用 {} 花括号括起来，
# 每个pair用逗号分隔，pair中的key在前，value在后，冒号分隔。
>>> pricelist={"clock":12,"table":100,"xiao":100 }
# 引用 dictionary 中的一个元素的 value时，用 key 做下标 。
>>> pricelist["clock"]
12
>>> del pricelist["clock"]
>>> pricelist
{'table': 100, 'xiao': 100}
# dict(L) 可以构造一个 dictionary
# 其中 L 是一个 list， L中的每一个元素是一个tuple，
# tuple中有两个元素，前面的是key，后面的value。
>>> pricelist=dict([('clock',12),("table",100),("xiao",100)])
>>> pricelist
{'table': 100, 'xiao': 100, 'clock': 12}
>>> pricelist=dict([(x,10*x) for x in [1,2,3]])
>>> pricelist
{1: 10, 2: 20, 3: 30}
#若对一个不存在的key做赋值操作，则增加一个pair。
>>> pricelist["apple"]=12   #增加一个 元素
>>> pricelist
{'table': 100, 'apple': 12, 'xiao': 100, 'clock': 12}
#如果读不存在的key，会抛出异常。
>>> pricelist["appled"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: appled
```

### §1.8.2 dictionary的操作

假设 `a={"one":1,"two":2,"three":3,"four":4}`

**清空**dictionary

函数名称`D.clear()`

```
>>> a.clear()
>>> a
{}
```

清空一个dictionary

D.copy()

得到一个dictionary 的一个拷贝

```
>>> a
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
>>> b=a.copy()
>>> b
{'four': 4, 'one': 1, 'three': 3, 'two': 2}
>>> b["four"]=5
>>> b
{'four': 5, 'one': 1, 'three': 3, 'two': 2}
>>> a
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
>>> c=a
>>> c["four"]=5
>>> a
{'four': 5, 'three': 3, 'two': 2, 'one': 1}
>>> c
{'four': 5, 'three': 3, 'two': 2, 'one': 1}
```

得到拷贝。从上面的例子可以看到，如果直接赋值`c=a`，那么c和a一样，修改c的内容同样会影响a。但是如果`b=a.copy()`，那么b是独立的，改b不会影响a。

D.get(key)

若D中有key，那么返回D[key]，否则返回None

```
>>> a
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
>>> print a.get("abc")
None
>>> print a.get("four")
4
>>> print a.get("abc",100)
100
```

`D.get(key,default)` 和`D.get(key)`的不同在于，若无key，则返回default，而不是None 。`D.get(key)`相当于`D.get(key,None)`。

D.haskey(key)

用于判断D 中是否含有key，相当于key in D.keys()。

```
>>> a
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
>>> print a.has_key("one")
1
>>> print a.has_key("abc")
0
```

D.items()

得到一个list，其中的每个元素是一个含有两个元素的tuple(pair)，tuple中前面是key，后面是value

```
>>> a
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
>>> a.items()
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

D.copy()相当于dict(D.items())，但是若欲得到副本，用copy()更快。

D.keys()

得到list，含有D中的所有key。

```
>>> a
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
>>> a.keys()
['four', 'three', 'two', 'one']
```

a.copy()还可以写成dict([(k,a[k]) for k in a.keys() ])

```
>>> dict([ (k,a[k]) for k in a.keys()])
{'four': 4, 'one': 1, 'three': 3, 'two': 2}
```

当然，如果仅仅是为了copy，不要这么干，用a.copy()，因为更快。

D.values()

得到list，含有D中的所有value。

```
>>> a
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
>>> a.values()
[4, 3, 2, 1]
```

D.update(E)

从E中得到新的数据。意思是

```
for k in E.keys():
    D[k]=E[k]
```

例如：

```
>>> a
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
>>> b={"five":5,"three":30}
>>> b
{'five': 5, 'three': 30}
>>> b.update(a)
>>> b
{'four': 4, 'five': 5, 'one': 1, 'three': 3, 'two': 2}
```

这种方法可以用来从一个dictionary 向另一个dictionary 赋值。

D.popitem()

得到一个pair，并且删除他。若D是空的，抛出异常。

```
>>> print a.popitem()
('four', 4)
>>> print a.popitem()
('three', 3)
>>> print a.popitem()
('two', 2)
>>> print a.popitem()
('one', 1)
>>> print a.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: popitem(): dictionary is empty
```

## §1.9 程序流程

编程语言的程序流程有三种，顺序，分支，循环。

### §1.9.1 分支结构if

语法

```
if <expr1>:  <one_line_statement>
```
或者
```
if <expr1>:
    <statement-block>
```
或者
```
if <expr1:
    <statement-block>
else:
    <statement-block>
```
或者
```
if <expr1>:
    <statement-block>
elif <expr2>:
    <statement-block>
elif <expr3>:
    <statement-block>
...
else:
    <statement-block>
```

这里应该注意以下几点

- if 后面的表达式可以是任何表达式，除了None，0，""(空字符串)，[] (空list)，{}( 空dictionary ) ()(空tuple )以外，其他都是真。
- 表达式以冒号分隔。
- 若是一行的简单表达式，可以直接跟在后面。
- 若是多行的表达式，就要用缩进的方式，表示一组语句。在Python中没有Pascal 中的`begin ... end`，或者C 语言中的`{ }` 来表示一组语句的开始和结束。Python认为同样缩进长度的语句是一组。这是一个仁者见仁的事情。好处就是，提高了可读性，任何人编写的程序都是类似的，不像C中有各种各样的风格，有的把“{” 紧接在`if`后面，有的则另起一行。不好的地方就是，这是个新点子，很多人都不习惯，人们一向认为编程语言中空白字符不应该是语法的一部分。而且有的编辑器不能分别tab和4（或8）个空格的区别，就会有一些莫名其妙的语法错误。
- Python中好像没有`switch`语句，`elif`用来完成这个功能[2]。
- `else`跟表示条件不满足时应该执行的语句，别忘了后面的冒号。

```
>>> if x>0:
```

[2]个人认为如果有switch 语句会更好

```
...     print "x is positive"
... elif x==0:
...     print "x is zero"
... elif x < 0:
...     print "x is negative"
... else:
...     print "x is not a number"
...
```

### §1.9.2 循环结构for，while，break，continue，range()

for循环语法

```
for x in <sequence>:
    <statement-block>
else:
    <else-block>
```

- sequence表示任何string，tuple，dictionary

```
>>> for x in "abc" : print x
...
a
b
c
>>> for x in [1,2,3]: print x
...
1
2
3
>>> D={"one":1,"two":2,"three":3}
>>> for x in D: print "D[",x,"]=",D[x]
...
D[ three ]= 3
D[ two ]= 2
D[ one ]= 1
```

- break语句可以强行退出循环，和C一样。

- else是可有可无的。若有，则表示如果每个a中的元素都循环到了(没有break)，那么就执行else-block。这一点很好，因为我在C语言中，经常要这么作。

```
for(i=0;i<len;i++){
  if(a[i]==100) break;
}
if(i==len){/*100 not found*/
  /*do something*/
}
```

而在Python中

```
>>> a
[1, 2, 3, 4]
>>> for x in a:
...     if x==100: break
... else:
...     print "100 not found"
...
100 not found
```

else 语句可以理解为：如果循环没有被break 语句强行中断，或者说循环正常结束后，就会执行else中的语句。这还是生硬的语法描述，语义描述会更加清楚，例如上面的例子，意思是说如果在a中找不到100，那么就会执行else中的语句。语义在各个上下文的情况下会有所不同。一般情况下，break都会在一个if 语句后面[3]，else就是表示，如果那个if 语句在循环中从来都没有成立过，就会执行else 中的语句。上面的例子中，if 的条件是x==100，那么如果a中任意元素x，都有not x==100，那么else中的语句就会执行。

- 修改x并不能够修改x在a中对应的内容。

```
>>> a=[1,2,3,4]
>>> for x in a:
...     x=x+1
...
>>> x
5
>>> a
[1, 2, 3, 4]
```

[3]否则循环就失去意义了

注意，是不能修改x在a中对应的内容，如果是Dictionary，那么没有修改x，而是直接修改Dictionary 中的内容。看过name space 的概念觉得这很自然。

```
>>> D={"one":1,"two":2,"three":3}
>>> for x in D:
...     D[x] += 1;
...
>>> D
{'three': 4, 'two': 3, 'one': 2}
```

若要修改，就用range()函数。range([start,]stop,[,step])，表示从start开始，到stop结束，每步增加step，start缺省为0，step缺省为1，返回一个list。

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,10,2)
[1, 3, 5, 7, 9]
>>> range(1,10,-1)
[]
```

若可能产生无限个数，那么是空list，不会有死循环。
利用range()函数修改list

```
>>> a
[1, 2, 3, 4]
>>> for x in range(len(a)):
...     a[x]=a[x]+2
...
>>> a
[3, 4, 5, 6]
>>>
```

while循环语法。

```
while <expr1>:
    <block>
else:
    <else-block>
```

这个循环的意思是，只要expr1是真，那么就一直执行block中的语句。如果没break跳出循环，也就是说，在循环结束的时候expr1是假，那么执行else-block中的语句。

```
>>> i=0
>>> while i < 3:
...     print i ; i=i+1
...
0
1
2
>>> while i < 3:
...     print i; i = i +1
... else:
...     print i , "is 3"
...
0
1
2
i is 3
```

千万别忘了i=i+1，否则就会有死循环。

## §1.10 函数

函数是一小段可以重复使用的代码。

### §1.10.1 基本函数的用法

语法:

```
def <function_name> ( <parameters_list> ):
        <code block>
```

说明:

- function_name是函数名称，冒号用来分隔函数的内容。
- parameters_list是参数列表。参数也没有类型，可以传递任何类型的值给函数，由函数的内容定义函数的接口，如果传递的参数的类型不是函数想要的，那么函数可以抛出异常。
- code block是和if，while一样的代码。注意应该有相同的缩进。

- 函数没有返回值类型，return可以返回任何类型。

例子:

```
>>> def add(a,b):
...     return a+b;
...
>>> print add(1,2)
3
>>> print add("abc","def")
abcdef
>>> print add("abc",12)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in add
TypeError: cannot concatenate 'str' and 'int' objects
>>> myadd=add
>>> myadd(1,2)
3
```

函数名称只是一个变量，一个对象。记得“一切都是对象”。所以可以像普通对象一样，把一个函数赋值给另一个变量。myadd=add，这有点像函数指针。术语叫做function alias，函数别名。

### §1.10.2 参数个数可选，参数有默认值

记得C++，VB中的可变参数，Python一样有类似的机制。例子:

```
>>> def myjoin(str,sep=","):
...     return sep.join(str)
...
>>> myjoin(["a","b","c"])
'a,b,c'
>>> myjoin(["a","b","c"],";")
'a;b;c'
```

sep的缺省值是“，”，用逗号分割，如果没有提供sep的参数，那么，就用“，”。但是如果一个参数是可选的，那么他后面的所有参数都应该是可选参数。

```
>>> def myrange(start=0,stop,step=1):
...   print stop,start,step
...
```

```
SyntaxError: non-default argument follows default argument
```

### §1.10.3 改变函数参数赋值顺序

如果定义了一个以上的可选参数，本来是不能改变参数的传递顺序的，

```
>>> def printValues(name,height=170,weight=70):
...     print("%s's height is %d cm,"
...         " his weigth is %d kg\n"%(name,height,weight))
...
>>> printValues("Charles")
Charles's height is 170 cm, his weigth is 70 kg
>>> printValues("Charles",175)
Charles's height is 175 cm, his weigth is 70 kg
>>> printValues("Charles",175,85)
Charles's height is 175 cm, his weigth is 85 kg
>>>
```

但是可以通过下面的方式改变参数的顺序。

```
>>> printValues("Charles",weight=85)
Charles's height is 170 cm, his weigth is 85 kg
```

### §1.10.4 个数可变参数

如果用过C语言中的printf()，那么就会对参数个数可变的意义比较了解。尽管可选参数的机制令函数的参数个数是可变的，但是还是有限制，参数个数有最大限制，而且还要指明哪些是可选参数。而下面这个机制，可以接受任意多个参数。

```
>>> def printf(format,*arg):
...     print format%arg
...
>>> printf ("%d is greater than %d",1,2)
1 is greater than 2
```

其中*arg必须是最后一个参数，*表示接受任意多个参数，除了前面的参数后，多余的参数都作为一个tuple传递个函数printf，可以通过arg来访问。

```
>> def printf(format,*arg):
...     print type(arg)
...     print format%arg
>>> printf("a",1)
<type 'tuple'>
```

在函数中，arg 是一个tuple，可以通过访问tuple的方法，来访问arg。

还有一种方式来实现任意个数参数，就是参数按照dictionary的方式传递个函数，函数同样可以接受任意多个参数。

```
>>> def printf(format,**keyword):
...     for k in keyword.keys():
...             print "keyword[%s] is %s"%(k,keyword[k])
...
>>> printf("ok",One=1,Two=2,Three=3)
keyword[Three] is 3
keyword[Two] is 2
keyword[One] is 1
```

同上一种机制类似，只不过**keyword是用**表示接受任意个数的有名字的参数传递，但是调用函数时，要指明参数的名字，One=1,Two=2,Three=3，在函数中，可以用dictionary的方式来操作keyword，其中keys是["One","Two","Three"]，values是[1,2,3]。还可将两种机制和在一起。这时，*arg，要放在**keyword前面。

```
>>> printf("%d is greater than %d",2,1,Apple="red",One="1")
2 is greater than 1
keyword[Apple]=red
keyword[One]=1
```

还可以把这两种机制和可选参数机制合在一起使用。

```
>>> def testfun(fixed,optional=1,*arg,**keywords):
...      print "fixed parameters is ",fixed
...      print "optional parameter is ",optional
...      print "Arbitrary parameter is ", arg
...      print "keywords parameter is ",keywords
...
>>> testfun(1,2,"a","b","c",one=1,two=2,three=3)
fixed parameters is  1
optional parameter is  2
Arbitrary parameter is  ('a', 'b', 'c')
keywords parameter is  {'three': 3, 'two': 2, 'one': 1}
```

函数接受参数的顺序，先接受固定参数，然后是可选参数，然后接受任意参数，最后是带名字的任意参数。

### §1.10.5 Doc String 函数描述

将文档写在程序里，是LISP中的一个特色，Python也借鉴过来。每个函数都是一个对象，每个函数对象都有一个__doc__的属性，函数语句中，如果第一个表达式是一个string，这个函数的__doc__就是这个string，否则__doc__是None。

```
>>> def testfun():
...     """
... this function do nothing, just demostrate the use of the
... doc string.
...     """
...     pass
...
>>> print testfun.__doc__
this function do nothing, just demostrate the use of the
doc string.
>>>
```

pass语句是空语句，什么也不干，就像C语言中的{}。通过显示__doc__，们可以察看一些内部函数的帮助信息。

```
>>> print " ".join.__doc__
S.join(sequence) -> string
Return a string which is the concatenation of the strings in the
sequence.  The separator between elements is S.
>>> print range.__doc__
range([start,] stop[, step]) -> list of integers
Return a list containing an arithmetic progression of
integers. range(i, j) returns [i, i+1, i+2, ..., j-1];
start (!) defaults to 0.When step is given, it specifies
 the increment (or decrement). For example, range(4) returns
[0, 1, 2, 3].  The end point is omitted!
These are exactly the valid indices for a list of
4 elements.
```

用这种办法，察看帮助，是十分有效的。文档和程序在一起，有助于保持文档和程序的一致性。

### §1.10.6 lambda函数

熟悉functional programming的人，不会对lambda函数感到陌生。LISP是一

种functional programming language。不熟悉functional programming 的人，可以认为lambda函数就是一种简单的匿名函数。

```
>>> f = lambda a,b: a+b
>>> f(1,2)
3
>>> f("abc","def")
'abcdef'
```

f 等价于def f(a,b): return a+b。

一个函数可以返回一个lambda函数。如

```
>>> def incfun(a):
...     return lambda x: x+a
...
>>> incfun(2)(12)
14
>>> incfun(2)(13)
15
>>> incfun(4)(13)
17
>>>
```

### §1.10.7 函数的作用域(scope)

这里记住一个LGB的规则，python得到一个变量的名字先找local name space (局部命名空间)，在找global name space(全局命名空间)，然后是buildin name space (内在命名空间) 用global语句可以改变某些变量的命名空间。

```
>>> a=1
>>> def testfun():
...     a=2
...     print a
...
>>> testfun()
2
>>> a
1
```

在函数testfun()中a是在local name space 中，修改a不会改变global name space 中的a但是，用global语句可以修改一个变量的所在的name space。

```
>>> a=1
>>> def testfun():
...     global a
...     a=2
...     print a
...
>>> testfun
<function testfun at 0x8176264>
>>> testfun()
2
>>> a
2
>>>
```

### §1.10.8 嵌套函数(nested)

Python中函数是可以嵌套的。

```
>>> def outfun(a,b):
...     def innerfun(x,y):
...             return x+y
...     return innerfun(a,b)
...
>>> outfun(1,2)
3
```

这在C语言中是不允许的，因为C中的static 函数可以代替这种功能。这种机制提供了一个函数范围的概念，某些函数只在某些函数的内部，才看得到，如果他要公开的话，可以通过返回值，返回内部函数。注意，和Pascal 不一样，嵌套函数不能访问外层函数的变量。我认为嵌套函数的用处不大。

### §1.10.9 function的参数传递

python是按值传递的参数的，但是还是有可能改变参数所指的值。

```
>>> def change(x,y):
...     x=2
...     y[0]="hello"
...
>>> x=1
>>> y=[1,2]
```

```
>>> change(x,y)
>>> x
1
>>> y
['hello', 2]
```

可以看到，x的值没有改变，但是y的值却改变了。在介绍name space 的概念后会觉得这是在自然不过的事情了。参数x，y 不过是在function 的local name space 中创建的一个新的name ，和function 的调用者传进来的参数指向同一个对象。在function 内部x=2，只是改变了name x的bind 关系，让这个name 指向一个新的对象。所以不会影响调用者传进来的参数所指的对象。而y[0]="hello" 并没有改变y 的bind 关系，而是直接修改y所指向的对象，那么当然就会改变调用者传进来的参数了。看了后面name space 的概念会对这个有更深的理解。

## §1.11 模块(module)和包(package)

### §1.11.1 创建一个module

一个module 不过是一些函数，class放在一个文件中。例如，在当前目录(current directory)创建一个文件叫testmodule.py，内容是

```
"""
this only is a very simple test module
"""
age=0  # a sample attribute

def sayHello(): # a sample function in a module
print "Hello"

if __name__ == "__main__"
sayHello()
```

一个module 也有类似function 的doc string。除了注释的第一条语句如果是一个字符串的话，那么这个字符串就是doc string，可以查看一个doc string来得到帮助。import 一个module 有两种方法。

```
>>> import testmodule
>>> print testmodule.__doc__
        this only is a very simple test module
>>>
```

可以访问testmodule中的attribute

```
>>> testmodule.age
0
>>> testmodule.age=1
>>> testmodule.age
1
>>> testmodule.sayHello
<function sayHello at 0x8174a34>
>>> testmodule.sayHello()
Hello
>>> sayHello=testmodule.sayHello
>>> sayHello()
Hello
>>> sayHello
<function sayHello at 0x8174a34>
>>>
>>> othermodule=testmodule
>>> othermodule.age=100
>>> testmodule.age
100
```

可以看到，通过modname.attribute的方式可以访问，module 中的attribute ，在module 中定义的变量，函数，class 都是module 的attribute 。用name space 的概念来解释就方便多了，module 中的name space 中的一切都是module的一个attribute，反过来说module 的attribute 是module 的name space 中的一个name。

sayHello=testmodule.sayHello()，可以在global name space 中建立一个name，map(映射到) testmodule 的name space 中的一个name 所指的object。可以看到两个sayHello 指向了同一个function object。尽管他们的都是sayHello，他们也指向同一个对象，但是他们在两个不同的name space 中。module 本身也是一个object，testmodule就是他的name，可以在global name space 中建立另一个name和testmodule指向同一个object，一个module object。othermodule = testmodule。如果一个module中有很多有用的functions，但是通过modulename.funcname的方式，就显得很麻烦。可以用第二种import module 的办法。重新进入python，然后。

```
>>> from testmodule import age, sayHello
>>> age
0
>>> sayHello
```

```
<function sayHello at 0x81631a4>
>>> sayHello()
Hello
>>> testmodule
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'testmodule' is not defined
>>>
```

可以看到，这种方法相当于

```
import testmodule
age=testmodule.age
sayHello=testmodule.sayHello
```

但是有些不同，用`from`的方法，`testmodule`这个名字，并没有被import进来，也就是说，当前name space中没有`testmodule`这样一个name，而`import testmodule`就可以在当前name space中创建一个`testmodule`这样一个的name。可以用`from testmodule import *`将testmodule中name space中所有name引进到当前的name space中，通过这些name，访问name所指的对象。

而且`import testmodule`后，如果`testmodule.py`修改过了，可以用`reload(testmodule)`重新加载`testmodule`。

### §1.11.2 怎么查找module

python用以下步骤查找一个module，以`import testmodule`为例子。

- 在当前目录中查找`testmodule.py`。
- 若没找到，在环境变量(environment variables) `PYTHONPATH`中查找，就象在`PATH`中查找可执行文件一样。
- 若无`PYTHONPATH`变量，那么在安装目录中查找，在unix中是`.:/usr/local/lib/python`

其实，python是在`sys.path`中的所有目录中查找module的。可以查看`sys.path`。

```
>>> import sys
>>> print sys.path
['', '/usr/lib/python2.2', '/usr/lib/python2.2/plat-linux2',
 '/usr/lib/python2.2/lib-dynload',
 '/usr/lib/python2.2/site-packages']
```

从查找顺序上看，我们可以知道如果你在当前目录建立了一个和标准库中带有的标准module有相同的名字那么会用你的module 代替系统module，会产生莫名其妙的问题，所以我们要注意，自己的module 名字不要和系统的module名字相同。

### §1.11.3 package(包)

package 是一组module 的集合。用以下方法创建一个package。

- 现在当前目录创建一个目录testpackage
- 在testpackage 创建一个空文件`__init__.py`
- 在testpackage 中创建一个testmodule。py，里面含有一些代码。这里为了简单，就和`testmodule.py`一样。

起动python

```
>>> import testpackage.testmodule
>>> testpackage.testmodule.sayHello()
Hello
```

package 是一种组织module 的方法，提供了一个name space，防止发生名字冲突。package 中还可以有package，所以这种方式可以很好的组织一个树状结构，用来管理modules。

## §1.12 name space(命名空间)

在前面的介绍中，不止一次的提到name space，现介绍name space 的概念，这是一个高级概念，但是如果理解他，很多东西都很容易理解。同时也会感觉到python 是一种形式简单，概念清晰，概念统一的语言。

什么是name space? name space 是从名称(name)到对象(object)上的映射(map)。当一个name 映射到一个object 上时，我们说这个name 和这个object有帮定(bind)关系，或者说这个name 指向这个object。

重复的说，python 中一切都是object ，包括function ，module ，class ，package 本身。这些objects都有在内存中真真正正的存在，就像生活在世上的每一个人。但是我们怎么找到这些objects呢? 用name! 给每个object 起个名字。每个name 只对应一个object 而一个object可以有多个名字。但是name 不是object 本身，就象一个人名字叫Charles，但是Charles 不是这个人，是这个人的名字，这个人是有血有肉会呼吸的，Charles 不会，Charles是可以被写在纸上的。一个人也可以有多个名字。

Python中有很多的name space，常用的有: build-in name space (内在的命名空间) global name space (全局命名空间)，local name space(局部命名空间)。在不同name space 中的name 是一点关系也没有的。甚至字面上相同name，只要他们在不同的name space

中，那么他们就毫无关联。改变local name space 中的name的bind 关系，不会影响到global name space 中相同的name。

每个object 都有自己的name space。可以通过`object.name` 的方式访问object 的name space 中的name。每个object 的name space 是独立的。“独立的”的意思是说，一个object 的name space 中的一个name，和其它name space 中的name 是一点关系也没有的，甚至两个字面上看起来一样的name，也是毫不相干的。就像“车子” 这个object，在他的name space 中有一个“颜色”，这样一个name ，表示一个车子的颜色，而“房子”这个object ，在他的name space 中也有一个“颜色”，这个name 表示房子的颜色，他和车子一点关系也没有，尽管他们都叫“颜色”。

我认为尽管name space 是一个高级概念，但是他是很清晰的概念，一个容易理解的概念。简单的概念后面隐藏着大故事。name space 是这个笔记中说得最多，最罗嗦的一个概念。我认为，他是python 的核心概念，就像file 这个概念unix 中一样。

每一个object ，如module，class，function，一个instance ，都有自己的name space。name space 是动态创建的。每一个name space 的生存时间也不一样。

一个module 的name space 是这个module 被import 的时候创建的。

有一个特殊的module ，一进入python的解释器，就建立了一个的module，这个module 的name space 就是global name space，一个全局唯一的name space。这个module 的一个内部的attribute，`__name__` 用来标识module，这个module的`__name__` 是`__main__`。

```
>>> print __name__
__main__
```

每一个module 都有一个`__name__`的attribute，用来表示当前module 的名字。所以每一个module 都有这样的代码。

```
if __name__ == "__main__"
    sayHello() # 测试代码
```

用于modoule 测试。因为在一个module 在被`import`的时候，他的name space 中创建了一个内置的name，`__name__`，用来表示module 的文件名称。所以在被`import` 的时候，测试代码就象不存在一样，因为`__name__`，永远不会是“`__main__`”。但是如果用`python testmodule.py` 的方式直接运行一个module 的时候，而不是`import`，那么module 的name space 就是global name space ，这时python 自动把global name space 中的`__name__` 设为“`__main__`”，那么就可以运行测试代码了。

当一个调用一个function的时候，function的local name space 被创建了，当function返回的时候，或者function 抛出exception的时候，local name space 被删除了。

```
>>> def testfun():
...     print locals()
```

```
...     print globals()
...
>>>
>>> testfun()
{}
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__',
 'testfun': <function testfun at 0x8174dd4>, '__doc__': None}
>>>
```

locals()和globals()分别返回global name space 和local name space。name space 通常用dictionary 来表示，dictionary 是一个(key，name)这样的pair 的集合，key 是name，value是name 所指的object。

什么是scope(作用域)? scope 就是python 中的代码从字面意义(textually)上可以“直接访问”的name 的集合。这里解释两个概念:

- “直接访问”的意思是用unqualified reference name 就可以直接找到name 所指的对象。
- unqualified reference 就是不含有.的name。a 就是一个unqualified name，a.b 就不是。

LGB的规则用scope的概念来解释就是: 在任何代码执行的时候，都至少有三个scope，一个是local name space 组成的scope。一个是当前module 的global name space 组成的scope，还有buildin name space 组成的scope。从内到外依次查找一个unqualified name。L 指local，G 指gloabl，B 指buildin。

global语句可以改变把一个unqualified name 的scope，直接到global name space 中查找。

function 的alias 的意思是指alias=module_name.funcname 的方式创建的一个function 的别名。

```
>>> def sayHello():
...     print "Hello"
...
>>> saidHello=sayHello
>>> saidHello
<function sayHello at 0x8174bb4>
>>> sayHello
<function sayHello at 0x8174bb4>
>>> sayHello()
Hello
```

```
>>> saidHello()
Hello
```

可以看到sayHello，saidHello他们不过是两个名字，他们指的是同样的object。就象一个人有两个名字一样。a=[1,2]这样语句，会在最内部的name space 上创建一个name和[1,2]这个object 帮定(bind)。这种赋值语句不会拷贝数据，只是在namespace中创建了一个新的name，指向一个object。这很重要。就像java 中所有的变量都是引用(reference)一样，还和C 中pointer(指针)有些类似。在理解一些程序的时候，这是很有帮助的。

用del语句可以从name space删除一个name。

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

删除一个变量的意思是从最内部的name space 清除一个name 和object 的帮定(bind) 关系，如果没有其他name和这个对象bind，那么才真正删除object。这和java 中的垃圾回收有些类似。和C和C++中智能指针的技术很像。

实际python 中就是使用了智能指针的技术，每个name 都和指向object的指针相关联，每一个object都有一个referece counter 记住有多少个name 和这个object bind。每次bind，reference count 都加1，每次删除bind 关系，都减少1，只有reference counter 变成0的时候才真正删除对象。

python 是一种强类型，动态类型的语言。

强类型是指，每一个对象都有一个类型如tuple ，dictionary ，list ，string ，int ，long ，float ，module ，function ，class ，methord 和任何用户定义或者module 定义的class。Basic 就不是一种强类型，所以Basic容易引起歧义，例如

```
print "1" + "2"
12
print 1 + "2"
3
```

也许"1"+"2"，你期望是"3" 或者"12"，但是搞不懂你到底要什么。

每个对象都有自己独立的name space。每种类型有自己独立的操作方法。对象的name space 中name 会和某个操作方法相对应。变量仅仅是对象的名字。

动态类型是指每个变量都不用声明，可以指向任何类型的对象。或者叫变量(variable)为名字(name)更为合适。变量只是一个对象的名字，每个变量在某一时刻指向一个对象，但是一个对象可以同时有多个变量指向他，就像函数，module ，class 都可以有alias (别名)。变量的生存期要和对象的生存期是不一样的。当没有任何变量指向一个对象的时候，这个对象才被真正删除。而一个变量也可以随时改变他的帮定关系，转指向其他对象。

这种表达方式和我们的现实世界是一致。

## §1.13 类

要接近Python的核心了。面向对象(OO)是一种编程思想而不是一种语言。Python是用C语言是实现的面向对象的语言。GTK也是用C接口的面向对象的组件库。Unix操作系统中，“一切都是文件”，某种意义上讲，是一种成功的面向对象，文件就是定义了对应open，write，read，close，ioctrl 等一组内核函数(或者叫方法)的对象。

面向对象的目的是代码重用，减少重复性的开发。面向对象是代码重用机制包括封装(encapsulation)，继承(Inheritance)，多态(Polymorphism)。

面向对象的核心是抽象(abstraction)，分离接口(interface)和实现(implemetation)。

一个最简单的类

```
>>> class SimplestClass:
...     pass
...
```

他什么也没干，但是我们可以自豪的说，我有了我的第一个class。

### §1.13.1 初始化函数

```
>>> class SampleClass:
...     def __init__(self):
...             print "OK, I was born."
...
>>> a=SampleClass()
OK, I was born.
>>> AClass = SampleClass
>>> b=AClass()
OK, I was born.
>>>
```

在Python没有C++中的构造函数，和析构函数。但是有初始化函数，当一个object创建的以后，会自动调用`__init__`。

class 和函数(function)一样，也是一个对象，可以赋值给普通变量，那么这个变量就变成了这个类。这种机制类似于C++ 中的template机制，也就是说，可以把一个class作为参数传递个一个函数。但是和template 还不完全一致。

例子中，`AClass = SampleClass` 就像function alias 一样，建立了一个class alias。

可以说class 关键字创立了另一个独立的name space ，这样理解，后面的东西都是自然而然的事情了。后面介绍的不过是对name space 的概念的进一步解释。会明白为什么所有的成员变量和成员函数都是虚函数(virtual function)了。

每个instance(实例) 都是用函数调用的方式返回的`obj=ClassName()`

每个obj有自己独立的name space。

### §1.13.2 方法(method)

在class 内部定义的函数都是methord。包括`__init__`。每个methord至少有一个参数，`self`，类似于C++，Java中的`this`。

这里有个规则，如果从类的内部调用methord，那么一定要包括`self`作为第一个参数，如果是调用对象的methord，那么直接调用，就好像少了一个self参数一样。

可以理解为: 每一个class 的methord 都至少有一个参数。若obj 是class A 的一个instance，`sayHello(self)` 是class A 的一个methord 的话，那么`obj.sayHello()`和`A.sayHello(obj)`是一致的。

用name space 的概念来理解就更清楚了，class 提供了一个建立一个独立name space 的机制，每一个instance 都有自己独立的name space。

在class 的methord 中，`self` 在methord 的local name space 中，指向class 的instance。

当 `obj=A_Class()` 的时候，创建了一个instance ，这个instance 有自己独立的name space ，这个name space 是`A_Class`的name space 的一个拷贝。

注意，我说的是“拷贝”，class 的instance 的name space 和class 的name space 是完全独立的。

```
>>> class A:
...     def sayHello(self):
...             print "hello"
...
>>> obj=A()
>>> obj.sayHello()
hello
>>> A.sayHello(obj)
hello
```

`self` 代表一个instance 本身。

### §1.13.3 属性(property)

重复的说，class 就是建立了一个独立的name space 。任何内部的变量都是class 的属性。属性是可以随时建立，随时删除的。如果obj 没有a 的属性，那么任何地方都可以用obj.a =1 创建一个新的属性。无论在class 的methord 中用self.a=1的方式还是在外面用obj.a=1的方式，都可以创建一个新的属性。因为每个instance 有自己独立的name space，可以自然的理解到，即使同一个class 的两个不同的instance 也可以有不同的属性。可以用del 删除一个instance 的属性。

```
>>> class A:
...     name="abc"
...
>>> obj1=A()
>>> obj2=A()
>>> obj1.name="xyz" #改变 obj1 都 name
>>> obj2.name       #不会影响obj2
'abc'
>>> obj1.name       #因为他们的 name space
'xyz'               #是独立的.
>>> class B:
...     def __init__(self):
...             self.a=1
...
>>> obj1=B()
>>> obj2=B()
>>> obj1.a       #obj1 和 obj2 都有
1               #属性 a
>>> obj2.a
1
>>> del obj2.a   #删除 obj2 的属性 a
>>> obj2.a       #没有了,不能在用 obj2.a了
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: B instance has no attribute 'a'
>>> obj2.b=1    #创建一个 obj2.b
>>> obj2.b      #obj2 有 b, 没有 a
1               #obj1 有 a, 没有 b 尽管他们同是
                #class B的 instance
```

### §1.13.4 继承(inherit)

继承的语法格式是

```
class <name>(superclass1,superclass2,...):
```

可以有多个父类。用name space 的概念理解就是把所有superclass 中的name space 合并，并且增加自身定义的新的name 到name space 中，从而生成一个新的name space。重复的说，class 几乎就是一个创建独立name space 的机制。当然还可以用`superclass.methord(self,...)` 的方法来访问父类的方法。更加感到name space 简单的强大。

### §1.13.5 重载(overload)

几乎不用像C++ 中那样特别的解释什么是重载。因为class 可以建立一个独立的name space ，每一个instance 都有自己独立name space。class 中建立的新的name，如果和任何父类name space 中的的相同，都会重新覆盖原来的bind 关系。当然还可以用`superclass.methord(childInstance)`的方式访问父类的方法。再次看到name space 简单的强大。

### §1.13.6 class 的attribute

也许你用过C++或者Java中静态成员变量，就是同一个class 的不同instance 共享同一个变量。python 也有这种机制，但是它不是引进的新的术语，还是用name space 的机制，实现了我们知道每一个instance 有独立的name space，而class 本身也是一个object ，也有一个独立name space 。每个instance 的name space 中都有一个`__class__`的name ，用来指向创建该instance 的class 对象。`__class__`由python 自动创建，`obj.__class__` 指向obj 的class。这样就不难理解下面的程序了。

```
>>> class A:
...     count = 0
...     def __init__(self):
...             self.__class__.count= self.__class__.count + 1
...             #注意不是self.count
>>> A.count #还没有A的 instance
0
>>> a=A() # 创建一个A的 instance
>>> a.count # a.count 和 A.count 是在两个不同的 name space 中
1
>>> a.count = 2 # 改变 a.count
>>> a.count
2
```

```
>>> A.count      # 不会影响 A.count
1               # 现在A 只有一个instance
>>> a.__class__  # a 的name space 有一个 __class__ 的 name
<class __main__.A at 0x81566d4>
>>> a.__class__ is A # 这个 name 和 A 都和同一个 class 对象有bind 关系.
1                    # 也就是说,他们是一回事,指的是同一个东西
>>> b=A()           #在创建一个 instance
>>> A.count         # __init__ 中 self.__class__ 增加 1
2                   # self.__class__ 和 A 指的是统一的东西.
                    # 现在 A有两个 instance了.
```

### §1.13.7 Abstrace Class

Java 有interface ， C++ 中abstract class ， 那么python 呢?

```
class AbstractClass:
        def AbstrctMethord(self):
                raise NotImplementedError
```

除了异常exception之外，没有介绍新的概念。同时也满足了abstract class 的要求。初学面向对象编程，觉得这是无聊的代码，什么也没有干，学过之后，发现这是面向对象的核心思想之一，分离接口和实现。

### §1.13.8 Python的面向对象编程

OO (面向对象) 是一种编程思想，不是编程语言，甚至用汇编语言一样可以面向对象，当然没有人会那么干有些语言为OO 提供了很多便利，很容易实现封装，继承，重载，多态。Python，java，C++，SmallTalk 等等。Python 的实现机制是十分简单的，name space。这种方式是简单而强大的，但是也是容易写出很烂的代码，其它语言也一样，谁也阻止不了你写很烂的代码。本来想举很多例子，但是我觉得这个笔记是用来描述Pyton语言本身的，这个笔记不能使你成为编程高手，只能熟悉一门语言，就像学会了说话，但是不一定都会演讲，而且谁也阻止不了你胡说八道，语言越强大，越容易不知所云，相反，如果很好的掌握一种编程思想(不是编程语言)，用什么语言都可以讲的很清楚，很明白，很漂亮。

### §1.13.9 Python 中class 特殊的methord

在Python 中有很多class 的特殊methord，像__init__就是其中一个。

- `__init__` 已经介绍过了，就是在`obj=A_Class(arg,...)`时候自动调用，它的参数除了`self` 还有`A_Class` 中传递的参数。
- `__del__(self)` 在对象被删除的时候，自动调用。就是说没有name 和这个object 有bind 关系的时候，会调用他。

- `__repr__(self)` 使用repr() 函数调用的时候，返回和eval() 兼容的对象字符串表达式，用于重建对象。
- `__str__(self)` 随便返回一个字符串描述对象本身。下面的例子说明了`__repr__`和`__str__` 的不同。

```
>>> class A:
...     def __repr__(self):
...             return "A()"
...     def __str__(self):
...             return "ok, I am here"
...
>>> x=A()
>>> print x
ok, I am here
>>> print repr(x)
A()
>>> b = eval(repr(x))
>>> b
A()
>>>
```

eval(str) 是用来计算一个python 表达式，并且返回这个表达式。这是解释语言通用的特性，可以在程序中写程序，但是把这种代码翻译成C++ 是比较困难的，因为需要有解释执行表达式的模块。这可不是一个容易的工作。

- `__cmp__(self,other)` 一个比较操作符。0 表示相等，1表示大于，−1表示小于。

```
>>> class Dog:
...     def __cmp__(self,other):
...             if other > 0 : return 1;
...             elif other < 0 : return -1;
...             else: return 0
...
>>> dog = Dog()
>>> if dog > -10 : print "ok"
...
>>> if dog < -10 : print "ok"
...
ok
>>> if dog == 0 : print "ok"
```

```
...
ok
>>>
```

这段代码只是说明__cmp__ 的用法，其实这是很混乱的一段代码，狗和数字有什么好比较的？他只是说明了python的语言特点，就算学会了一门语言，也一样会语无伦次。

- __nozero__(self)，以前我们注意到None，"" ，() 等等都表示逻辑假，其他都是真，其实不完全对，这里可以定义一个对象是否是逻辑假。如果返回0 表示逻辑假，返回1 表示逻辑真。

```
>>> class Dog:
...     alive=0
...     def __nonzero__(self):
...             if self.alive==0 : return 0
...             else: return 1
...
>>> dog = Dog()
>>> dog.alive
0
>>> if dog : print "the dog is alive"
... else: print " the dog is dead"
...
 the dog is dead
>>> dog.alive =1
>>> if dog : print "the dog is alive"
... else: print " the dog is dead"
...
the dog is alive
>>>
```

这段代码看起来比上一个例子还是容易懂。至少他没有胡说八道。

- __len__(self)，在用内置函数len调用对象的时候，会被调用，返回object的长度。如果没有定义这个函数，而调用len(obj)会导致异常。

```
>>> class EmptyClass:
...     pass
...
>>> obj=EmptyClass()
```

```
>>> len(obj)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: EmptyClass instance has no attribute '__len__'
>>> class Dog :
...     def __len__(self):
...             return 1000
...
>>> dog = Dog()
>>> len(dog)
1000
```

前面一部分说明了没有定义__len__ 会导致异常，后面的例子说明了定义了__len__ 就可以了，还是为了说明语言本身，写了一段糊涂的代码，狗怎么会有长度呢？len(dog)得到一个狗的长度? 毫无意义。但是在你知道有这么一种机制可以做到长度，也许你在开发一个EmailBody 的class 的时候就可以用len(a_email_body) 返回一个电子邮件的内容的长度。后面还会写一些无意义的代码，只是为了说明语言的特点。

- __getitem__(self,key)，返回self[key] 用来模拟list，dictionary，tuple 等数据结构。

```
>>> class Zoo:
...     def __getitem__(self,key):
...             if key == "dog" : return "dog"
...             elif key == "pig" : return "pig"
...             elif key == "wolf" : return "wolf"
...             else : return "unknown"
...
>>> zoo=Zoo()
>>> zoo["dog"]
'dog'
>>> zoo["pig"]
'pig'
>>> zoo["wolf"]
'wolf'
```

注意，__getitem__ 可以使一个对象像sequence 一样。在python 中的术语是，object 实现了__getitem__ 的protocol (协议)，类似java 中interface。也就是说一些内置的语句，for 等会使用这样的的协议。

```
>>> class A:
...     def __getitem__(self,key):
...         print "debug: key is %s"%key
...         if key >=0 and key <=5: return key * key
...         raise IndexError
...         return None
...
>>> a=A()
>>> for k in a: print k
...
debug: key is 0
0
debug: key is 1
1
debug: key is 2
4
debug: key is 3
9
debug: key is 4
16
debug: key is 5
25
debug: key is 6
>>>
```

raise IndexError 是表示结束。return None 变得没有用了。for 语句从0 开始，用一个不断递增的key 从a 中取得对象。用while描述他就是

```
__tmp__ = 0
while(1):
    try :
        k = a[__tmp__]
        ... # for 内的语句
        __tmp__ = __tmp__ +1
    except IndexError:
        break
```

这是为了说明 for k in a 的语义，当然用for 语句更加简练。你永远不会使用这种while语句代替for 语句。

- __setitem__(self,key,value)，模拟 obj[key]=value。

```
>>> class A:
...     def __setitem__(self,key,value):
...         print "debug: you want to set [%s] to [%s]"%
...                                     (key,value)
...
>>> a=A()
>>> a["key"]="value"
debug: you want to set [key] to [value]
>>>
```

- __delitem__(self,key)，模拟 del obj[key] 时，调用该函数

```
>>> class A:
...     def __delitem__(self,key):
...         print "debug: you want to del [%s]"%(key,)
...
>>> a=A()
>>> del a["key"]
debug: you want to del [key]
```

__delitem__，__setitem__，__getitem__ 只是提供一种扩展类型的语法，具体语义，应该由你的程序决定，可以是好程序，也可以写出让人搞不懂的的程序。
__getslice__(self,i,j)，__setslice__(self,i,j,value)，
__delsclice__(self,i,j) 是类似的。注意如果没有指定上限或者下限，使用一个最大的正整数和0代替。

```
>>> class A:
...     def __getslice__(self,i,j):
...         print "you want to get the slice %d:%d"%(i,j)
...     def __setslice__(self,i,j,value):
...         print "you want to set the slice %d:%d to %s"%
...                 (i,j,value)
...     def __delslice__(self,i,j):
...         print "you want to del the slice %d:%d"%(i,j)
...
>>> a=A()
>>> print a[1:2]
you want to get the slice 1:2
```

```
None
>>> print a[:]
you want to get the slice 0:2147483647
None
>>> a[2:3]= [1,2]
you want to set the slice 2:3 to [1, 2]
>>> del a[:]
you want to del the slice 0:2147483647
>>>
```

- `__contains__(self,other)` 可以使object 像sequence 一样，处理in 语句

```
>>> class A:
...     def __contains__(self,other):
...         if other=="ok" : return 1
...         return 0
...
>>> a=A()
>>> if "ok" in a: print  "ok in a"
... else : print "ok not in a"
...
ok in a
```

- `__call__(self,arg1,arg2,...)` 让object 像函数一样，可以调用。

```
>>> class A:
...     def __call__(self,*arg):#这里使用可变参数机制,打印所有传进来的参
数
...             for k in arg:
...                     print k
...
>>> a=A()
>>> a("hello","World")
hello
World
>>>
```

- 其他算术操作符

| 方法 | 结果 |
|---|---|
| __add__(self,other) | self + other |
| __sub__(self,other) | self - other |
| __mul__(self,other) | self * other |
| __div__(self,other) | self / other |
| __mod__(self,other) | self % other |
| __divmod__(self,other) | divmod(self,other) |
| __pow__(self,other) | self ** other |
| __pow__(self,other,modulo) | pow(self,other,modulo) |
| __lshift__(self,other) | self << other |
| __rshift__(self,other) | self >> other |

还有很多，但是我想笔记还是不要写得像参考手册一样。总之，有很多函数实现了操作符重载的机制，几乎所有的python 操作符都有这种重载机制。你可以写一个自己的class，让他看起来，完全像list ，dictionary 一样，支持所有的list 的所有的操作符和语句。在python 的module lib 中已经有这样的class，可以察看他们的源代码，非常的短，但是很有价值。在lunix 中可以在/usr/lib/python2/ 中找到UserList.py UserDict.py。

## §1.14 异常处理(exception)

### §1.14.1 什么是异常处理，为什么要有异常处理

异常处理是一种出错处理的机制。一定要对出错处理给于高度重视，一个强壮的安全的程序的源代码中，充满了出错处理，如果不要出错处理，那么源代码的规模会小很多。你可以找一个项目的例子，看看里面有多大的比例的源代码是出错处理，还有多大比例是调试代码。你可能会发现真正干活的代码占很小的比例。就算一个正常运行的程序，他的大部分时间运行的代码只是你所写的代码中的极小的一部分。但是不能没有这些代码，他们就象消防员，平时并没有从事生产工作，没有创造价值，但是不能没有他们，任何人都知道消防队员的重要性。C 中错误处理的方法是

```
int fun1()
{
   ...
    if ( open(...) == -1 ){
        return -1;
    }
   ...
}
int fun2()
```

```
{
    ...
    if (fun1()==-1){
        return -1;
    }
    ...
}
int main()
{
     ...
     if(fun2()==-1){
         return -1;
     }
}
```

从这个例子中，我们可以看到C语言的出错处理有个以下几个特点：

- 程序的每一级都要做出错处理，用来把错误通知上一级函数。
  因为底层函数不知道应该怎么处理错误，只有函数的调用者，也就是上一级函数才能知道，所以必须这样一级一级的上报。
  这样带来的问题是，到处都是判断出错，代码的大部分都是出错处理。
- 程序通过返回值来表示错误。
  这样带来的问题，返回值有歧义。返回值不但要返回正常的程序想要的数据，还要返回错误状态，这样，程序“有可能”会把错误状态当成返回的数据来处理。例如:

  ```
  unsigned int r;
  r=read(fd,buffer,len_of_buffer);
  for(i=0;i<r;i++){
     // do something
  }
  ```

  这段程序，是很明显的有问题，如果read 出错，会返回−1 但是程序没有判断，for 会成为死循环。当然，一般不会写出这样烂的代码，对r 赋值也会有编译警告，类型不匹配。但是这个代码说明了你“有可能”写出这样的代码，尤其是程序比较大的时候，你可能有修改了某个函数的返回值的事先约定。那么所有函数的调用者都有可能出现这样的问题。
- 出错信息太少。因为返回值身兼数职，所以不能够携带更多的出错信息。例如系统调用read 返回−1表示出错，还要errno 来表示出错码，还需要其他的函数把出错码变成错误信息的文字描述。

exception的机制是令一种出错处理的方式，他有以下特点:

- 和return 一样，抛出异常同样会使函数返回。
- 和return 也有区别，如果return，函数的调用者会得到返回值，如果发生异常，函数的调用者不会得到返回值。
- 自动支持错误的上报。异常会在堆栈中向上冒泡，直到有人捕获，如果某个函数抛出了异常，而函数的调用者又没有处理异常(捕获异常)，那么就象在函数调用处发生异常一样，函数的调用者也会返回。
- 将出错处理和返回值的功能分开。只要得到返回值，那么他就是程序想要的数据，不会有其他意思。

例如:

```
def fun1()
    ...
    open file
    ...
def fun2()
    ...
    fun1()
    ...
def main()
    try :
       fun2()
    except exp :
       print "error message"
       return.
```

### §1.14.2 捕获exception(异常)

有两种捕获异常的方法。第一种是

```
try :
    ... # statements 1
except ExceptionType :
    ... # statements 2
```

描述如下:

- 首先执行statement 1，如果没有exception 发生，那么就一切正常，就像没有try 语句一样，继续执行后面的语句。

- 如果发生了exception ，那么就把exception 的类型和`except` 语句后面的`ExceptionType` 比较，看exception 的类型是否和`ExceptioinType` 一致，
- 如果一致，就执行`statement 2`。
- 如果不一致就向直接返回，把exception 抛给函数的调用者，然后就这样一级一级的向上抛，直到有try 语句捕获了exception。
- 若没有任何try 语句可以捕获，那么程序就异常退出，在解释器中执行的话，就会像以前的例子一样，打印出来异常发生的详细信息。可以用`1/0` 看一下产生被0除的异常。

可见，exception 是另一种函数return 的方式，C语言或者其他没有exception 机制的语言，函数只有通过返回值来指明函数出错，所以每一级函数都要检查函数的返回值。exception 的机制就是提供除了返回值之外的另一种出错处理的方法。这种方法支持错误的向上冒泡，如果某一级函数不知道该怎么处理错误，那么就不处理，留给更上一级函数处理。尤其是lib 中的module ，他们的出错处理大多数都抛出exception 。一个`try` 可以有多个`except` 语句。

```
try :
    ... # statements 1
except (ExceptionType1,ExceptionType2) :
    ... # statements 2
except (ExceptionType3,ExceptionType4) :
    ... # statements 3
except:
    ... # statements 4
```

最后一个`except`没有指定exception 的类型，表示捕获任何类型的exception。`try` 语句可以有一个`else` 语句，一定放在最后，表示没有发生exception 的时候执行`else` 带的一些语句。例如:

```
try:
    f = open("/home/chunywang/tmp/test.py","r")
except IOError:
    print "can not open file"
else:
    for i in f.readlines(): print i
```

表示如果打开文件成功，也就是说，打开文件的时候没有发生异常就读文件。通过比较以下几个例子，可以知道这样做的好处。

```
try:
    f = open("/home/chunywang/tmp/test.py","r")
```

```
    for i in f.readlines(): print i
except IOError:
    print "can not open file"
```

这个例子把很多语句都放在try 中，这样做的话，会带来问题，如果出现了IOError，也不一定是open 的时候产生的，readlines() 也可以产生这个exception。所以不要把很多语句放在try 中，因为他们中有些也会产生exception，而这些exception 并不是你想要的，在本例子中，except只想要打开文件出错的exception，而不是读文件出错。如果要处理读文件出错。可以

```
try:
    try:
        f = open("/home/chunywang/tmp/test.py","r")
    except IOError:
        print "can not open file"
    else:
        for i in f.readlines(): print i
except IOError:
    print "read faid"
```

甚至就像原来的例子一样，不做任何处理，让上一级函数去处理。或者:

```
    try:
        f = open("/home/chunywang/tmp/test.py","r")
    except IOError:
        print "can not open file"
    else:
            try:
            for i in f.readlines(): print i
        except IOError:
            print "read faid"
```

另一个比较的例子是

```
try:
    f = open("/home/chunywang/tmp/test.py","r")
except IOError:
    print "can not open file"
    return None
for i in f.readlines(): print i
```

这种方法中，如果不加return 语句，那么在文件打开失败的时候，readlines 还会执行，这显然是不行的，读一个没有打开的文件，是很傻的，自己给自己找麻烦。但是return 语句也限制，因为函数直接返回了，也许你并不想这么干。举个例子，假设这段语句是用来从配置文件中读取一些系统参数，如果打不开配置文件，就设置为缺省值。然后再作其他操作。最好用下面的方案。

```
try:
   f = open (" 配置文件")
except IOError:
   设置缺省值
else :
   从文件中读取系统参数
#现在系统参数是有效的了
作其他操作
```

如果不用else 的方法，还有一个方案是

```
isOpen=false
try:
   f = open (" 配置文件")
   isOpen=true
except IOError:
   isOpen=false
if isOpen :
   从文件中读取系统参数
else:
   设置缺省值
#现在系统参数是有效的了
作其他操作
```

这种方法增加了一个新的变量，和else 语句比较，就没有那么清晰了。第二种异常处理的方法是

```
try:
   ... # statements 1
finally:
   ... # statements 2
```

这种方式不能和第一种方式混用，也就是说，不能带有except 和else ，最好把它看成完全另一种语句。无论是否发生exception ，还是return 语句返回，finally子句statements 2 都会执行，

```
>>> def fun():
...     try:
...         print "ok"
...         return 1
...     finally:
...         print "clean up"
...     return 1
...
>>> fun()
ok
clean up
1
>>>
```

这种方法适合于释放资源。如果一个函数有多个地方return 那么就要在每一个return之前编写几乎同样的释放资源的代码，如关闭文件，关闭网络连接等等。finally 的方法提供了一个在函数返回之前，执行一些代码，来关闭资源，函数返回可能是因为return 语句，或者发生了没有被捕获exception

### §1.14.3 抛出exception

raise 语句用于抛出exception，在抛出exception 的时候，可以是任何object，用于详细描述错误类型。

```
>>> from  UserList import *
>>> try:
...     raise UserList([1,2])
... except UserList,arg:
...     for k in arg: print k
...
1
2
```

在except 中只要exceptionType和object 的类型一致，就可以捕获这个exception 。类型一致的意思是指object 是给类型的一个instance。如果一个object 所属的class 继承了其他的class，那么这个object 可以是多个类型的instance，这些类型包括object 所属的class 的父类，父类的父类，父类的父类的父类...

except 不但可以指定exception 的类型，还可以接收抛出来的object。

上一个例子中arg 就是raise 语句抛出来的UserList([1,2]) 这样，就可以有丰富的信息可以用来描述出错的原因。一般来说，都是抛出Exception 和它的子类，从手册中可以

查到很多Exception 的类型，如: KeyError，IndexError，IOError 等等等等。但是python语言没有限制，你可以抛出任何类型的对象。用户可以自己创建一个`Exception` 的子类，来表示用户子定义的exception。`raise` 语句还可以是`raise classname,objectname`的形式，其中`objectname` 可以和`classname` 没有一点关系，当然这样做很不明智，一般`classname` 和`objectname`还是有关系的，用来表明你的程序中的逻辑关系。

`classname` 对应`except` 语句中的exception 的类型，`objectname` 对应`except` 语句中的`arg`

```
>>> try:
...     raise KeyError, "hello"
... except KeyError, arg:
...     print arg
...
hello
```

可见，抛出和捕获exception 提供了一种方便的出错处理机制。函数不用在通过特殊的返回值来通知上一级函数（函数的调用者)发生了错误，而是通过raise exception 的方式。

返回值就是函数正常返回的时候应该返回的东西。

这种方法可以加快开发速度，在开发的初期，可以不用处理任何错误，以加快程序原型的开发速度。

正常的程序中，会有很多代码是出错处理，一级一级的处理。如果去掉这些处理，会使程序看起来很清楚，明了。当一个包含很少出错处理的程序原型呈现个用户的时候，尽管他不是很强壮，但是可以有一个真正可以使用的程序作为参考，用来收集用户的需求，挖掘用户的需求，这样可以减少项目的风险。

相反，如果一个程序开发了很久，包含很多的出错处理，也很强壮，但是呈现个客户的时候，却发现，原来那不是客户想要的，那是很沮丧的，尴尬，浪费人力物力的事情。

至于出错处理，可以在以后增加，因为函数的返回值和出错处理没有关系，程序的主干是不需要改变的，程序的处理流程不需要变化。也就是说，大楼的主体结构是不用变的，要做只是添砖加瓦。我想画画也是有同样的道理，都是先画个轮廓，然后再一步一步细化。

如果没有exception 机制的话(例如C语言)，你就需要trace (搜索) 到底是哪个函数返回了错误，而且是错误的源头，然后在所有调用这个函数的地方都加判断，上抛错误值，如果有些函数有办法通过返回值来上报错误，例如，一个函数的返回值是一个字符(byte，最多0 – 255)，表示某256 种正常状态，现在要再增加若干状态表示出错，那么要么修改函数的返回值的类型，要么增加新的参数，用于返回错误。总之是需要修改接口，众所周知，修改接口是程序开发的噩梦。

出现这种问题的原因就是“返回值不仅仅是为返回值”。

用exception 的机制，就可以在需要出错处理的地方，增加try 语句。就象是消防队员，

那里有火就奔向哪里，而不是在任何一个可能着火的地方都放消防队员，那样会很挤的，你的系统中人几乎都是消防队员，而真正干活的人却很少。或者说干活的人同时也要接受消防训练，要身兼数职，当办个消防员，但是我们知道“社会分工”是更先进的生产方式。

顺便提一下，Exception 机制需要动态内存管理机制，或者说自动对象回收机制。向C++ 中也有Exception 机制，但我觉得不好用，如果你抛出一个对象，那么谁来回收这个对象哪？为了实现这个方式，会把本来很复杂的C++ ，搞得更加复杂了。Python，Java 就好多了。不用担心抛出异常会产生内存泄漏。

学习Python 的过程，和学习C++ 和Java 的过程，让我感到，学习一门语言是次要的，学习一种编程思想才是主要的。有些语言的某些特性后面都带有一个编程思想在其中。语言是思维的载体。

Python 吸收了很多先进的编程思想，如浆糊语言( glue language )，像Perl，VB Script ，Java Script ，Bash。如原型开发( prototype development )。如面向对象(Object Oriented)，像C++，Java。如强类型(strong type) ，动态类型(dynamic type) ，像Scheme，LISP。如正则表达式(regular expression)，像Perl，Sed，Awk。如Web 服务器编程，像Perl。还有Doc String，lambda函数，像LISP。还有函数的可变参数，任意个数参数，命名参数，像C++，Java ，VB。

他还有一个自己的特性，他把缩进式作为语法的一部分，可见其是如此的强大调可读性。

# 第二章

# 开发Python 使用的工具

## §2.1 使用Emacs 编辑Python 程序

本章假定用户使用过Emacs ，Emacs 是一个著名的程序开发环境，不仅仅是一个编辑器。

python mode 是Emacs 中的一个编辑python 源程序的主模式。

### §2.1.1 安装python mode

XEmacs 21.4.4 的用户省事了，他自带有Python mode 。

如果你的系统中没有python mode 的话，按照下面的安装步骤:

1. 下载python-mode.el 文件，在Python 2.3 的安装包中，`Misc` 目录中含有这个文件。也可以到下面的网址下载

   `http://www.python.org/emacs/python-mode/python-mode.el`

2. 拷贝python-mode.el 到一个目录中，例如:

   ```
   cp python-mode.el ~/emacs/
   ```

3. 在`~/.emacs` 中增加下面的内容。

   ```
   ;;                    Python
   ;;
   ;; 让 emacs 可以找到 python-mode.el
   (add-to-list 'load-path "~/emacs")
   ;; 打开 *.py 的文件的时候，自动加载 python mode
   ```

```
(setq auto-mode-alist
(cons '("\\.py$" . python-mode) auto-mode-alist))
(setq interpreter-mode-alist
(cons '("python" . python-mode)
          interpreter-mode-alist))
(autoload 'python-mode "python-mode" "Python editing mode." t)
;; 语法高亮显示
(setq font-lock-maximum-decoration t)
```

4. 检查安装。在Emacs 中运行，

   `M-x locate-library RET python-mode RET`

   应该可以找到`python-mode.el`，如果找不到，那么应该检查`load-path` 中是不是指定了正确的路径名称。

### §2.1.2 python mode 的基本特性

安装好了Emacs 的python mode 之后，编辑python 源代码就有一些不用学就会的很好的特性。例如：

- 自动处理缩进。
  我们知道，python 利用缩进做为语法的一部分，尤其是TAB 和空格混合的时候，总是有一些莫名其妙的语法错误。使用python mode 基本可以避免这种问题了。因为他十分智能的自己判断缩进，如遇到冒号，而且，用backspace 删除四个缩进的空格时，只用按一下就够了。而且在某一行上按TAB 键，就会自动处理缩进，而且从来不用TAB 字符来表示缩进，都是用空格表示的。带来的好处就是，无论用什么工具查看，缩进的距离都是一样的。
- 自动语法加亮。
  很多流行的好的编辑器的特点。关键字等不同语法结构的元素，用不同的颜色表示，一目了然。
- 当使用backspace 退到上一级的语句块的时候，会自动提示当前缩进的地方是哪一块语句的结束。
- 菜单IM-Python 上含有程序的结构，包括类，成员函数，变量等等。
- 菜单Python 上有很多的常用操作。

### §2.1.3 常用功能举例

1. `C-c ! 或者 M-x py-shell`

   可以启动python 解释器。

2. C-c C-c 或者 M-x py-execute-buffer

   可以运行正在编辑的python 程序。如果启动了python 解释器，那么就会在Python 的解释器中执行，就像一个一个命令输入其中一样。

3. C-c |  或者 M-x py-execute-region

   可以运行region 中的程序。region 是Emacs 中的一个最普通的概念，和选择区域类似。C-@ 或者C-<SPC> 可以定义一个region 的开头，region 的结尾就是当前光标的位置。这个命令可以把region 中的部分发送给解释器执行。如果启动了python 解释器，那么就会在Python 的解释器中执行，就像一个一个命令输入其中一样。

4. C-c RET  或者 M-x py-execute-import-or-reload

   这个命令在开发模块的过程中特别有用，可以把正在编辑的文件作为一个module ，导入到Python 解释器中。开发模块的时候，经常要把这个模块import 或者reload 。有了这个功能，那么开发流程就是，修改模块，C-c RET ，运行测试代码，继续修改模块。这样大大提高了开发速度。这个命令和C-c C-c 相比，有两个优点。

   (a) 所有的函数和变量的定义都是在模块内部，而不是解释器中的全局变量。

   (b) Python 的调试器可以得到对应文件的行数，易于错误定位。

5. C-c C-s  或者 M-x py-execute-string

   会在minibuffer 请求用户输入一段python 代码，然后运行他。这在调试一个小函数的时候，很有帮助。你可以输入函数的名称，运行这个函数。或者当你对某个模块函数的用法不太肯定，或者对某种语法不太肯定的时候，也可很方便的运行简单的一句话代码。或者查看帮助也是很有用的，可以运行help(somefunc) 可以查看帮助，或者运行type(somevar) 查看类型，或者dir(obj) 查看对象的所有属性。

6. C-c C-#  或者 M-x comment-region

   可以把region 部分注释掉。

   M-x uncomment-region

   可以把region 部分被注释掉的部分，去掉注释。这对调试程序很有帮助。

7. C-c C-k 或者 M-x py-mark-block

   可以把光标后面的一组代码块，标记为一个region ，光标会移动到代码块的开头。可以用C-x C-x 来检查region 的开头和结尾。

8. C-M-a 或者 M-x py-beginning-of-def-or-class

`C-M-e` 或者 `M-x py-end-of-def-or-class`

分别表示移动光标到一个def 或class 块的开头或者结尾。可以使用参数。例如: `C-u 1 C-M-a` 表示移动光标到上两级的def 或class 块的开头这个命令几乎在Emacs 所有的编程模式中起作用。

9. `C-M-h` 或者 `M-x py-mark-def-or-class`

   表示把一个def 或者class 的语句块，标记为一个region 。可以用`C-x C-x` 来检查region 的开头和结尾。同样可以使用参数。这个命令几乎在Emacs 所有的编程模式中起作用。

10. `C-c >` 或者 `C-c C-r` 或者 `M-x py-shift-region-right`

    `C-c <` 或者 `C-c C-l` 或者 `M-x py-shift-region-left`

    分别表示把region 向左移动，和向右移动，用来改变代码块的缩进。

11. `C-M-j` 或者 `M-x indent-new-comment-line`

    在编写多行的注释的时候十分有用，表示开始一个新行，自动加上注释符号和正确的缩进。

12. `C-x $` 或者 `M-x set-selective-display`

    表示只显示一个模块中的所有类和函数的名称，而隐藏类和函数的主体，这样很用的看到一个模块中代码的轮廓。这个命令是可以接受参数的，例如: `C-u C-x $` 表示隐藏，`C-x $` 表示显示类和函数的主体。

    本来这个功能不是这个意思，原来的的意思，只显示缩进值小于指定参数的行。例如`C-u 4 C-x $` 如果一行的开头含有的空格数小于4 ，那么就显示他，否则就不显示，`C-x $` 表示显示所有的行。恰好，python 使用缩进来表示语法结构，所以用各个功能可以查看源代码的结构。

    非常的好用! 尤其是你的程序很大的时候，游走于其间，胜似闲庭信步。

## §2.2 其他编辑器

可以在下面的网址，看到许多的编辑器，如果Emacs 不适合你，就选择一个适合你的编辑器。

`http://www.python.org/cgi-bin/moinmoin/PythonEditors`

但是，我强烈推荐使用Emacs 。效果真的不错。

## §2.3 调试程序

调试程序当首推pdb ，他是python 子带的调试模块。类似GDB 的接口，文字界面，不是很好用。但是他是一个底层构架，有其它界面更好的程序利用pdb 来调试程序。不过看一下pdb 的帮助还有很有用的。

### §2.3.1 使用DDD 和pydb 调试python 程序

DDD 是一个著名的图形界面的调试程序接口，他使用底层的调试工具，典型的就是GDB 。

为了和DDD 配合，pydb 是一个的pdb 改进版本。

按照下面的步骤可以安装pydb 和使用DDD 。

1. 安装DDD ，如果你的Unix 或者Linux 的开发包装得很全的话，应该已经装好了DDD 。如果没有，查看DDD 的安装手册。

2. 下载pydb 。我是从

   ```
   http://ftp.debian.org/debian/pool/main/p/pydb/pydb_1.01.orig.tar.gz
   ```

   下载的。如果这个链接失败的话，但愿你能找到更好的地方下载。

3. 拷贝pydb.py 到`$PATH` 中的一个目录。也就是说，Shell 可以找到并且运行这个命令。例如:

   ```
   $cp pydb.py /usr/bin/pydb
   ```

   注意要改名字，去掉`.py` 扩展名称。

4. 拷贝`pydbsupt.py` 和`pydbcmd.py` ，到一个目录，确保`import` 命令可以找得到他们。一般是

   ```
   $cp pydbsupt.py pydbcmd.py /usr/local/lib/python/
   ```

5. 运行

   ```
   $ddd --pydb
   ```

   就可以打开图形界面了。

   可以选择Edit/Preference.../Source 菜单，把“Refer to Program Source” 设置成为“by Full Path Name”，还可以更改缩进值和是否显示行号等等。

6. 选择File/Open 菜单，打开你要的源程序，就可以调试了。

因为是图形界面，具体的调试方法，摸索一会儿也就会了。

# 第三章

# Python 的常用模块

模块的接口不像核心语言那么稳定，可能会变，就像自然语言的语法是比较稳定的，但是写出来的文章确实千差万别的。但是有一些常用的模块是比较稳定的。如果这里介绍的和实际的版本不一致，还是需要查看最新的手册的。

## §3.1 内置模块

内置模块是不用import 就可以直接使用的。它含有很多有用的函数。列举如下，具体可以查手册，或者`print obj.__doc__` 察看帮助。

### §3.1.1 常用函数

常用的内置函数在表3.1-1中。

例如:

```
>>> help(dir)
Help on built-in function dir:

dir(...)
    dir([object]) -> list of strings

    Return an alphabetized list of names comprising (some of)
    the attributes of the given object, and of attributes
    reachable from it:

    No argument:  the names in the current scope.
    Module object:  the module attributes.
    Type or class object:  its attributes, and recursively the
```

| | |
|---|---|
| help(obj) | 在线帮助，obj 可以是等等任何类型。 |
| callable(obj) | 看看一个obj是不是可以像函数一样调用他。 |
| repr(obj)<br>eval(str) | 得到obj 的表示字符串，可以利用这个字符串用eval 函数重建该对象的一个拷贝<br>str是一个字符串，表示合法的python 表达式。返回这个表达式 |
| dir(obj)<br>hasattr(obj，name)<br>getattr(obj，name)<br>setattr(obj，name，value)<br>delattr(obj，name)<br>vars(obj)<br>locals()<br>globals() | 查看obj 的name space 中可见的name 。<br>看一看一个obj 的name space 中是否有name<br>得到一个obj的name space 中的一个name。<br>为一个obj 的name space 中的name 指向value这个object<br>从obj 的name space 中删除一个的name<br>返回一个object 的name space。用dictionary 表示<br>返回一个局部name space，用dictionary 表示<br>返回一个全局name space，用dictionary 表示 |
| type(obj)<br>isinstance(obj，cls)<br>issubclass(subcls，supcls) | 查看一个obj 的类型<br>看看obj是不是cls的instance<br>看看subcls 是不是supcls 的子类 |

表 3.1-1: Python 的重用内置函数

```
    attributes of its bases.
    Otherwise:  its attributes, its class's attributes, and
    recursively the attributes of its class's base classes.
>>> callable(dir)  #可调用的
1
>>> a=1
>>> callable(a)   #不可调用
0
>>> a=dir         #函数 aliase 也是可调用的
>>> callable(a)
1
```

```
>>> a(" ")       #和dir(" ")一样。
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
 '__eq__', '__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__repr__', '__rmul__', '__setattr__', '__str__', 'capitalize',
 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'replace', 'rfind', 'rindex', 'rjust', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper']
>>> repr({"one":1,"two":2})  # repr 用于得到字符串，用来表示的对象
"{'two': 2, 'one': 1}"
>>> a=[1,2]
>>> eval(repr(a))          # eval 和 repr 可以得到一个对象的拷贝。
[1, 2]
>>> a
[1, 2]
>>> hasattr(a,"append")
1
>>> getattr(a,"append")(3)
>>> getattr(a,"append")(4)
>>> a
[1, 2, 3, 4]
>>> type(a)
<type 'list'>
```

### §3.1.2 类型转换函数

常用类型转换函数如表3.1-2

举例:

```
>>> chr(65)
'A'
>>> [ord(k) for k in "ABC"]
[65, 66, 67]
>>> oct(65) , oct(65000),oct(650000000)
('0101', '0176750', '04657433200')
```

| | |
|---|---|
| chr(i) | 把一个ASCII 数值，变成字符 |
| ord(i) | 把一个字符或者unicode字符，变成ASCII 数值 |
| oct(x) | 把整数x 变成八进制表示的字符串 |
| hex(x) | 把整数x 变成十六进制表示的字符串 |
| str(obj) | 得到obj 的字符串描述 |
| list(seq) | 把一个sequence 转换成一个list |
| tuple(seq) | 把一个sequence 转换成一个tuple |
| dict()，dict(list) | 转换成一个dictionary |
| int (x) | 转换成一个integer |
| long(x) | 转换成一个long integer |
| float(x) | 转换成一个浮点数 |
| complex(x) | 转换成复数 |
| max(... | 求最大值 |
| min(...) | 求最小值 |

表 3.1-2: 类型转换函数

```
>>> hex(65) , hex(65000),hex(650000000)
('0x41', '0xfde8', '0x26be3680')
>>> str(10),str([1,2]),str({"one":1})
('10', '[1, 2]', "{'one': 1}")
>>> list("abc"),list((1,2,3))
(['a', 'b', 'c'], [1, 2, 3])
>>> tuple("abc"),tuple([1,2,3])
(('a', 'b', 'c'), (1, 2, 3))
>>> dict(),dict([("one",1),("two",2)])
({}, {'two': 2, 'one': 1})
>>> dict([ ( str(x), x) for x in [1,2] ])
{'1': 1, '2': 2}
>>> int(1),int("1"),int(1.4),int(1.6)
(1, 1, 1, 1)
>>> int("1.2")          #不行, 1.2 看起来不像整数
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 1.2
>>> long(1),long("1"),long(1.4),long(1.6)
(1L, 1L, 1L, 1L)
```

```
>>> float(1),float("1.2"),float("1e-4")
(1.0, 1.2, 0.0001)
>>> min(1,2,3,4,5)
1
>>> min([1,2,3,4,5])
1
>>> max([1,2,3,4,5])
5
>>> max(1,2,3,4,5)
5
```

### §3.1.3 用于执行程序的内置函数

#### 导入模块

import，导入一个module。

#### 执行代码

exec code [ in globaldict[,localdict]]。

```
>>> exec "a=1"    #在当前 namespace 上执行代码.
>>> a
1
>>> mynamespace={}  #创建一个空的dictionary,用于存储namespace
>>> exec "a=1" in mynamespace #在 mynamespace 上执行代码.
>>> print mynamespace.keys()  #查看结果
['__builtins__', 'a']         #引入两个name
>>> type(mynamespace["__builtins__"])
<type 'dict'>
#下一个更惊人的例子
>>> class A: #一个空的 class
...     pass
...
>>> exec "a=1" in  vars(A)  #增加一个属性
>>> A.a                     #成功
1
>>> exec "def abc(self): print 'ok' " in vars(A)
# 还能增加一个 methord
>>> dir(A)                #注意,有 abc 了
```

```
['__builtins__', '__doc__', '__module__', 'a', 'abc']
>>> x=A()
>>> x.abc()                #成功
ok
>>>
```

这种方式是很灵活和强大的，但是他很难翻译成为C++ 或者Java 语言。除非，你确定用Python 作为程序开发的主体语言，而且对Python 的性能很有信心，不想用其它语言如C++ 重写某些功能模块。为什么会用C++ 重写，而不是直接用C++ 写呢? 这涉及到一个软件工程的课题。简单地说，用C++ 可能写错，风险较大。Python 也可能写错，但是风险小。C++ 上来就精雕细刻，结果的作品可能不是用户要的。Python 是素描，用户可以先看一个轮廓，然后再精雕细琢，用C++ 重写某些模块。

一种是瀑布模型，会引起风险的累计，一种是迭代模型，逐步细化，释放风险。用C++重写是手段，不是目的，目的是满足客户的需求。如果可以满足用户的需求，就不用重写。

就算不能满足用户的需求，也不要急着修改，先测量，看看性能瓶颈在哪里，有的放矢，别瞎猜，费了很多力气，性能没有改变多少，磨刀不误砍柴工，python 有profile module 用于计算python 程序中每段代码的性能。找到“瓶颈”，确认“瓶颈”，然后优化。

### 编译代码

compile，如果一段代码经常要使用，那么先编译，再运行会更快。

```
>>> c=compile("print 'hello'","abc","single")
>>> exec c
hello
>>> c=compile("raise 1 ","abc","single")
>>> exec c
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "abc", line 1, in ?            # 注意文件名是 abc
TypeError: exceptions must be strings, classes, or instances,
not int
```

compile有三个参数，第一个代码字符串。是对应的Python源程序的文件名称，他用于在执行代码，产生异常的时候，打印出错信息。第三个参数是“single”，“exec” “eval”三者之一，这取决于代码是立即打印结果的语句，还是一组语句，还是一个表达式。下面的例子比较编译和不编译的性能差别。

```
>>> code="print 'Hello World'"
```

```
>>> c=compile(code,"abc","exec")
>>> profile.run("for i in range(10000): exec c")
        10002 function calls in 0.250 CPU seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.090    0.090    0.250    0.250 <string>:1(?)
 10000    0.160    0.000    0.160    0.000 abc:1(?)
     1    0.000    0.000    0.250    0.250 profile:0(for i in
                                            range(10000): exec c)
     0    0.000             0.000          profile:0(profiler)

>>> profile.run("for i in range(10000): exec code")
  10002 function calls (2 primitive calls) in 0.720 CPU seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
10001/1    0.720    0.000    0.720    0.720 <string>:1(?)
      1    0.000    0.000    0.720    0.720 profile:0(for i in
                                             range(10000): exec code)
      0    0.000             0.000          profile:0(profiler)
```

可以看到，一个是0.250个CPU seconds，一个是0.720 CPU second。可以提高性能大约3 倍。

### 计算表达式

eval(str) 用于返回一个表达式。注意到表达式的语法和语句的语法是不同的。

```
>>> a=2
>>> eval("a==1")
0
>>> eval("a=1")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<string>", line 1
    a=1
     ^
```

```
SyntaxError: invalid syntax
```

## §3.2 和操作系统相关的调用

和系统相关的信息都在sys 中，用`import sys`使用。

例如，得到命令行参数(command line arguments)。`sys.argv`是一个list，包含所有的命令行参数。

得到标准输入输出。`sys.stdout`，`sys.stdin`，`sys.stderr` 分别表示标准输入输出，错误输出的文件对象。

退出程序。`sys.exit(exit_code)`

得到系统中的modules，`sys.modules`是一个dictionary ，表示系统中所有可用的module ，例如`sys.modules["os"]` 表示module `os`，也就是说他们指向同一个对象。

得到得到运行的操作系统环境`sys.platform`

`sys.path` 是一个list，指明所有查找module ，package 的路径。

和操作系统相关的调用和操作os 中，用`import os`使用。

| | |
|---|---|
| `os.environ` | 一个dictionary，包含环境变量的映射关系。`os.environ["HOME"]` 可以得到环境变量HOME的值。 |
| `os.chdir(dir)` | 改变当前目录 |
| `os.cwd()` | 得到当前目录 |
| `os.getegid()` | 得到有效组id |
| `os.getgid()` | 得到组id |
| `os.getuid()` | 得到用户id |
| `os.geteuid()` | 得到有效用户id |
| `os.setegid()` | 设置有效组id |
| `os.setgid()` | 设置组id |
| `os.setuid()` | 设置用户id |
| `os.seteuid()` | 设置有效用户id |
| `os.getgroups()` | 得到用户组名称列表 |
| `os.getlogin()` | 得到用户登录名称 |
| `os.getenv` | 得到环境变量 |
| `os.putenv` | 设置环境变量 |
| `os.umask` | 设置umask |

表 3.2-3: 系统调用举例

太多了，表3.2-3 仅仅列出一小部分。只要能想到的系统调用，在python中都能找到。

下面有几个例子。

**一个简单的**shell

利用os.system() 编写一个简单的shell。

```
#!/usr/bin/python
import os,sys
cmd=sys.stdin.readline()
while cmd:
    os.system(cmd)
    cmd=sys.stdin.readline()
```

os.system(cmd)利用系统调用，运行从标准输入读来的命令。

**基本文件操作功能**

利用os 中的基本文件操作功能。有点类似shell，当然输入不够简练。

```
>>> os.mkdir('/tmp/xx') #创建目录
>>> os.system("echo 'hello' > /tmp/xx/a.txt")
0
>>> os.listdir('/tmp/xx')
>>> os.listdir('/tmp/xx')
['a.txt']
>>> os.rename('/tmp/xx/a.txt','/tmp/xx/b.txt')
>>> os.listdir('/tmp/xx')
['b.txt']
>>> os.remove('/tmp/xx/b.txt')
>>> os.listdir('/tmp/xx')
[]
>>> os.rmdir('/tmp/xx')
>>> os.listdir('/tmp/xx')  #成功删除了目录
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
OSError: [Errno 2] No such file or directory: '/tmp/xx'
```

还有chown，chmod，unlink，symlink，link等等。熟悉Unix ，应该也熟悉这些操作。

os 中还有很多函数，事实上，只要是POSIX 标准函数以及在大多数Unix 平台广泛使用的函数在Python 的Unix 版本里面都是支持的。这些都是遵循POSIX 的惯例。有本书叫做《Advanced Programming In Unix Environments 》很好的介绍了Unix 下的编程。

包括pipe，dup，open 等等操作管道，文件描述符，进程，终端，任务，用户，用户组等等。

### 用os.path 编写平台无关的程序

用os.path 编写独立于平台的程序。我们知道Windows(DOS)，MAC 和Unix 关于文件名称，目录名称，路径名称的用法是不同的。一个简单的例子是，DOS下我们用C:\temp\a.txt 表示一个文件Unix 下是/tmp/a.txt。下面的例子可以看到一些用法:

```
>>> os.getcwd()
'/home/Charles'
>>> os.path.split(os.getcwd())
('/home', 'Charles')
#用于分开一个目录名称中的目录部分和文件名称部分.
>>> print os.path.join(os.getcwd(),os.pardir,'a',"a.doc")
/home/Charles/../backup/a.doc
#和split相反,合成路径名称.
#os.pardir 表示但前平台下表示上一级目录的字符 ..
>>> print os.path.exists(os.getcwd())
1
#判断文件是否存在.
>>> os.path.expanduser('~/dir')
'/home/Charles/dir'
#把 ~ 扩展成用户根目录.
#在 Windows 下可能是 H:\Charles\dir
>>> os.path.expandvars('$TMP')
'/tmp'
#扩展环境变量,Windows 下可能是 c:\temp
>>> os.path.isfile(os.getcwd())
0
#是文件名吗？0，不是
>>> os.path.isfile(os.path.expanduser("~/tmp/a.c"))
1
# 我的目录下面有 /home/Charles/tmp/a.c这个文件
>>> os.path.isdir(os.getcwd())
1
#是目录吗？ 1，是的
>>> os.path.islink(os.getcwd())
0
```

```
#是符号连接吗？Windows 下可能不能用这个函数
>>> os.path.ismount(os.getcwd())
1
#是文件系统安装点吗？Windows 下可能不能用这个函数
>>> os.path.samefile(os.getcwd(),'/home/Charles')
1
#看看两个文件名称是不是指的同一个文件.
>>> def test_find(filename,dirname,names):
...     if filename in names:
...             print os.path.join(dirname,filename)
...
>>> os.path.walk('/home/Charles',test_find,"a.c")
/home/Charles/tmp/a.c
```

最后一个例子很有意思，os.path.walk(p,func,arg) 的第一个参数是一个目录名字，第二个参数是一个函数的名字，第三个参数随便是什么。os.path.walk 会把遍历p 下的所有子目录，包括p，对于每一个目录，都会调用func，func(arg,dirname,names) 的第一个参数就是，walk 的最后一个参数arg，dirname是访问的目录名称。names 是一个list，包含dirname目录下的所有内容。这样上面那最后4行程序的意思是，在某个目录中，和他所有的子目录中查找名称是a.c 的文件或者目录。我是在Unix 环境下，Windows 下应该类似。

### §3.2.1 打开文件

file 对象是一个和string，integer，float 一样的内置对象类型。他有以下方法。

```
f=open("filename","r")
```

filename 指定文件名称，“r”指用只读方式打开，“w”表示写方式，“rw”表示读写方式，“rb”表示读二进制方式，“wb”表示写二进制方式。

### §3.2.2 读写文件

open返回一个file 对象，可以用一下方法读写一个文件。

```
 >>> f=open("/tmp/x.txt","w")
>>> f.write("a")
>>> f.close()
>>> f=open("/tmp/x.txt","r")
>>> f.read()
'a'
>>> f.read()
```

,,

这段代码先打开一个文件，然后写了一个字符，然后关闭，再用读方式打开，读出来的刚刚写进去的字符。f.read(size)表示从文件中读取size 个字符，如果忽略size 那么将把所有的内容读出来。如果到了文件的结尾，f.read(size) 会返回一个空的字符串。注意到空的字符串表示逻辑假，可以用下面的方式一个一个的字符读取。

```
f.open("/tmp/x.txt","r")
c=f.read(1)
while c:
    print c
    c=f.read(1)
```

f.readline() 读取一行，适用于行操作。如果到了文件结尾，就返回空的字符串。

f.readlines() 读取文件的所有内容，返回一个list ，list 中的每一个元素表示一行，包含“\n”。

f.write(str)把一个字符串写入到文件中去。

f.tell() 返回当前文件读取的位置。

f.seek(off,where) 用来定位文件读取的位置。where= 0 表示从开算起，1表示从当前位置算起，2表示从结尾算起。off表示移动的偏移量，正数向文件结尾移动，负数表示向文件开头移动。注意到sys 中的sys.stdout，sys.stderr ，sys.stdin 是三个十分有用的文件对象。

```
>>> import sys
>>> a=sys.stdin.readline()
Hello           #输入一个字符串
>>> sys.stdout.write(a)
Hello           #输出
>>>
```

和读操作对应的写操作是writeline，write，writelines。

### §3.2.3 关闭文件

虽然Python 会自动把没有关闭的文件都关掉，但是明显的指明关闭文件总是一个好注意，用完了东西要放会原位，完玩的玩具要记得收拾整齐，吃完饭要记得刷碗。比刷碗简单，f.close()结束。

## §3.3 regular expression

regular expression (正则表达式)，简称regexp，是编译原理中的一个概念，在Unix的世界中无所不在，vi、emacs、sed、awk、perl 都广泛的使用。python 中的re，是一

个regexp 的module 通过他可以像perl 中一样使用regexp。regexp 就是一种表示一个字符串集合(set)的方法。说某个字符串和某个regexp 匹配(match），意思就是说这个字符串是regexp 所表达的集合中的一个元素。在Python 中，match 还表示某个字符串的开头的一部分和某个regexp match。

### §3.3.1 简单的regexp

最简单的regexp 表示有若干字符组成，表示字符本身。

```
>>> p=re.compile("abc")
>>> if p.match("abc") : print "match"
...
match
>>>
```

这里介绍了一个简单的用法，首先要生成用`re.compile("abc")` 一个pattern (模式)，pattern 的一个`match` 方法，如果和某个字符串匹配，就返回一个match object，否则就返回None。除了某些特殊字符metacharacter (元字符) ，大多数字符都和自身匹配。这些特殊字符是

。 ^ $ * + ? { [ ] \ | ( )

### §3.3.2 字符集合

用`[]`可以表示字符的集合。有以下几种用法

- 列出字符，如`[abc]` 表示匹配`a` 或者`b` 或者`c`，或者说字母`a`，`b`，`c` 都在`[abc]`表示的集合中。大多数metacharacter 在`[]` 中会失去metacharacter 本身的特殊意义，只表示和本身匹配。例如:

```
>>> a=".^$*+?{\\|()"
>>> p = re.compile("["+a+"]")
>>> for i in a:
...     if p.match(i):
...         print "[%s] is match"%i
...     else:
...         print "[%s] is not match"%i
...
[.] is match
[^] is match
[$] is match
[*] is match
```

```
[+] is match
[?] is match
[{] is match
[\] is not match
[|] is match
[(] is match
[)] is match
```

可见大多数metacharacter 在[]中都和本身匹配，但是有四个metacharacter 不和本身匹配，那就是“^[]\”，后面会一个一个说明，^ 表示取反，后面介绍，[]如果出现在[]中，就会分不清楚哪里是开头哪里是结尾了。\ 的作用是用来表示无法用打印的字符等等。

- 如何在[]中包含[]本身，表示和“[” 或者“]” 匹配?

```
>>> p=re.compile("[\]\[]")
>>> p.match("]")
<_sre.SRE_Match object at 0x815de60>
>>> p.match("]")
<_sre.SRE_Match object at 0x81585e8>
>>> p.match("[")
<_sre.SRE_Match object at 0x8158330>
```

用\[ 和\] 表示

- ^ 出现在[]的开头，表示取反，和[]中的字符都不匹配，没有出现的就匹配。[^abc]表示匹配除了a，b，c之外的所有字符。^ 出现在[] 中间的话，就没有这个意思了，表示^字符本身。看上面的例子，^没有出现在开头，就和自身匹配。
- 可以用-表示范围。[a-zA-Z]匹配任何一个英文字母。[0-9]匹配任何一个数字。
- \ 的在[] 中的其他妙用。

| | |
|---|---|
| \d | [0-9] |
| \D | [^0-9] |
| \s | [ \t\n\r\f\v] |
| \S | [^ \t\n\r\f\v] |
| \w | [a-zA-Z0-9_] |
| \W | [^a-zA-Z0-9_] |
| \t | 表示和tab 匹配，其他的都和字符串的表示法一致 |
| \x20 | 表示和十六进制ascii 0x20 匹配 |

有了\ ，可以在[] 中表示任何字符。注意，单独的一个“.” 如果没有出现[]中，表示除了换行\n 以外的匹配任何字符。类似[^\n]。

### §3.3.3 重复

可以用一下方法表示一个regexp 的重复。

- {m,n} 表示出现m 个以上(包含m个)，n个以下(包含n个)。如ab{1,3}c 就会和abc，abbc，abbbc 匹配，但是不会和ac，abbbbc 匹配。m是下界，n是上界。m可以省略，表示下界是0，n可以省略，表示上界是无限大。
- * 表示{,}，没有出现(0次)，重复出现1次以上。
- + 表示{1,}，重复出现1次以上
- ? 表示{0,1}，重复出现0次或者1次。

最大匹配和最小匹配的问题。为什么会有这个问题？看下面的例子。[ab]* 和ababab 进行匹配，那么是和a 匹配还是和ababab 匹配呢？如果和a 匹配，叫做最小匹配，和ababab 匹配，叫做最大匹配。python 中都是最大匹配。

```
>>> re.compile("a*").match('aaaa').end()
4
>>> re.compile("a*?").match('aaaa').end()
0
```

match object 的end可以得到匹配的最后一个字符的位置。如果要最小匹配，那么就在*，+，?，{m,n} 后面加一个?，变成*?，+?，??，{m,n}?表示最小匹配。

### §3.3.4 使用原始字符串

我们知道，字符串表示的方法中用\\ 表示字符\，而在regexp 中大量的使用\，所以在regexp中出现大量\\这样影响可读性，要在regexp 中匹配\ 需要这样写re.compile("\\\\") 这样会很麻烦。解决这个问题的办法是：在字符串前面加一个r表示raw 格式，\的特殊作用就不见了。所以在python 表示regexp一般都用加r的方法。例如

```
>>> a=r"\a"
>>> print a
\a
>>> a=r"\"a"
>>> print a
\"a
>>>
```

### §3.3.5 使用re 模块

使用regexp，通常用以下几个步骤：

- 先用re.compile 得到一个RegexObject，表示一个regexp。
- 然后用pattern 的match，search的方法，得到MatchObject。
- 再用match object 得到匹配的位置，匹配的字符串等信息。

RegxObject 的常用函数:

- RE_obj.match(str,[pos[,endpos]]), 如果str的开头和RE_obj匹配，得到一个MatchObject 否则返回None。注意到是从str的开头开始进行匹配。

```
>>> re.compile("a").match("abab")
<_sre.SRE_Match object at 0x81d43c8>
>>> print re.compile("a").match("bbab")
None
```

- RE_obj.search(str,[pos[,endpos]])，在str中搜索第一个和RE_obj匹配的部分。

```
>>> re.compile("a").search("abab")
<_sre.SRE_Match object at 0x81d43c8>
>>> print re.compile("a").search("bbab")
<_sre.SRE_Match object at 0x8184e18>
```

  注意和match()的不同。不必从开头就匹配。
- RE_obj.findall(str)，返回str中搜索所有和RE_obj匹配的部分。返回一个tuple，其中元素是匹配的字符串。

MatchObject的常用函数

- m.start() 返回起始位置，对于RE_obj.match()返回的MatchObject，start()总是返回0
- m.end() 返回结束位置(不包含该位置的字符)。

```
>>> s="I say:helloworld"
>>> m=re.compile("hello").search(s)
>>> for i in range(m.start(),m.end()):
...     print s[i]
...
h
e
l
l
o
>>>
```

- `m.span()` 返回一个tuple 表示`(m.start(),m.end())`。
- `m.pos()`，`m.endpos()`，`m.re()`，`m.string()` ，这些分别表示生成这个MatchObject函数中的参数。

  `RE_obj.match(str,[pos[,endpos]]),RE_obj.search(str,[pos[,endpos]])`

  中的`pos`，`endpos`，`RE_obj`，`str`。也就是说

  `m.re().search(m.string(),m.pos(),m.endpos())`

  会得到`m`本身。
- `m.finditer()` 可以返回一个iterator，用来遍历所有找到的MatchObject。

```
>>> for m in re.compile("[ab]").finditer("tatbxaxb"):
...     print m.span()
...
(1, 2)
(3, 4)
(5, 6)
(7, 8)
```

### §3.3.6 高级regexp

除了前面介绍的，还有很多有用的表达regexp 的方式。

- `|` 表示联合多个regexp 。如果A 和B 是两个regexp ，那么`A|B` 表示和A 匹配或者和B匹配。但是注意到`|` 的计算优先级最低，`Good|Bad` 会和“Good” 或者“Bad” 匹配，不会和“Goodad” 或者“GooBad”匹配。
- `^` 表示只匹配一行的开始行首。例如“`^abc`”会和“abc”匹配，但是不会匹配“xabc”，尤其是用`search` 的时候。`^` 只在开头才有这种特殊意义，就是是说，只有在开头，`^`才是metacharacter ，否则`^` 就是一个普通字符。
- `$` 表示只匹配一行的结尾。和`^` 类似，只有在最后一个字符的时候，`$` 才有这种特殊意思。
- `\A` 表示只匹配字符串的开头。如果没有在多行模式(MULTILINE) 下的时候，`\A` 和`^`是一个意思，但是在多行模式下`\A` 只匹配第一行的行首。而`^`匹配每一行的行首。
- `\Z` 表示只匹配字符串的结尾。如果没有在多行模式(MULTILINE) 下的时候，`\Z` 和`$`是一个意思，但是在多行模式下`\Z` 只匹配最后一行的行尾。而`^`匹配每一行的行尾。
- `\b` 只匹配词的边界，在汉语中，词的边界是很难界定的，但是在英语中标点符号，空格等等都可以表示一个单词的边界，其他语言也有类似，具体如何界定，由locale 决定。例如`\binfo\b` 只会匹配“info” 而不会匹配“information”。但是要注意下面的例子。

```
>>> print re.compile(r"\binfo\b").match("info ") #使用 raw 格式
<_sre.SRE_Match object at 0x817aa98>             #\b 表示单词边界
>>> print re.compile("\binfo\b").match("info ")  #没有使用 raw 格式
None                                             #\b 表示退格符号
>>> print re.compile("\binfo\b").match("\binfo\b ")
<_sre.SRE_Match object at 0x8174948>
```

注意双引号前面的r。在没有是用raw 格式的时候，\b表示退格符号，0x8，使用raw格式的时候\b表示单词的边界。

- \B 表示匹配非单词边界。

以上所有的metacharacter 并不和真正的字符匹配，只是一种条件检查，不会使用匹配的部分增加长度。例如: ^abc 匹配“abc”，但是匹配长度还是3。[1]

### §3.3.7 分组(Group)

通常情况，你真正想要的只是匹配字符串中的一部分，而不是全部。regexp 经常用来把一个字符串分解成几个部分，这样的例子很多，如email 中分解email header(信头)和email body (正文)，email header 中又要分解发信人，收信人等等，email 地址中又要分解用户名称，和用户域名。例如，你用下面的格式存储一个通信录。

```
name: Charles
Address: BUPT
telephone: 62281234
email:  annCharles@tom.com
name: Ann
Address: BUPT
telephone: 62284321
email:  CharlesWang@peoplemail.com.cn
```

用空行分开记录。那就要得到Charles ， Ann 等名称部分，BUPT 的地址部分等等。如何解决这个问题呢? Group ！看下面的例子。

```
>>> x="""
... name: Charles
... Address: BUPT
... telephone: 62281234
... email:  annCharles@tom.com
...
```

[1]在lex 中，“âbc” 的匹配长度比“abc” 的大

```
... name: Ann
... Address: BUPT
... telephone: 62284321
... email:  CharlesWang@peoplemail.com.cn
... """
>>> p=re.compile(
...   r"^name:(.*)\n^Address:(.*)\n^telephone:(.*)\n^email:(.*)\n",re.M)
>>> for m in p.finditer(x):
...     print "here is your friends list"
...     print "%s,%s,%s,%s"%m.groups()
...
here is your friends list
 Charles, BUPT, 62281234,  annCharles@tom.com
here is your friends list
 Ann, BUPT, 62284321,  CharlesWang@peoplemail.com.cn
```

这个例子有点复杂，但是应该没有什么语法方面的问题，多数情况可以查手册解决问题。关键在`p.compile()`处，其中`(.*)`中多了一个括号，表示group，也就是表示匹配的一部分。在MatchObject 中的`groups()` 返回所有匹配的group，本例子中分别表示`Charles` 等等。MatchObject 中的`start()` 、`end()`、`span()` 、`group()` 都可以传递参数，表示group 的号码，0总是存在的，而且是缺省参数，表示匹配的总的字符串。从1 开始计数，一直到regexp 中的最大的group 数，所以，本例子中`m.start(1)`表示name ，`m.start(2)` 表示Address 等等。括号是可以嵌套的，这时计算group 的号码，就数数`(` 的位置，例如:

```
>>> re.compile("(a(b)c)d").match("abcd").groups()
('abc', 'b')
>>>
```

有两组括号，数数`(` 的位置，先是`abc` ，然后是`b`。`p.finditer(x)` 在前面介绍过，可以用来遍历所有找到的MatchObject。后面的`print` 语句用`m.groups()` 来格式化一个字符串。因为`m.groups()` 是一个tuple，其中的每一个元素是一个string。group 除了这种用法，还可以表示替换，如: `(\b\w+)\s+\1` 就表示匹配连续出现的两个同样的单词。

从上面的例子中，会看到有这样一个问题，用`m.group(1)` 表示name，很不清楚，而且如果在增加一个group 的话，就需要调整后面所有的group 的号码。如果能够用`m.group('name')` 来表示name 的话，那么就清晰多了。这个问题是这样解决的，看下面的例子:

```
>>> p=re.compile(r"^name:(?P<name>.*)\n^Address:(?P<address>.*)\n" \
       r"^telephone:(?P<telephone>.*)\n^email:(?P<email>.*)\n",re.M)
>>> for m in p.finditer(x):
```

```
...     print m.groups()
(' Charles', ' BUPT', ' 62281234', '  annCharles@tom.com')
(' Ann', ' BUPT', ' 62284321', '  CharlesWang@peoplemail.com.cn')
>>> matchs=[m for m in p.finditer(x)]
>>> matchs[0].group('name')
' Charles'
>>> matchs[0].group('address')
' BUPT'
>>> matchs[0].group('telephone')
' 62281234'
>>> matchs[0].group('email')
'  annCharles@tom.com'
>>> matchs[1].group('name')
' Ann'
>>> matchs[1].group('address')
' BUPT'
>>> matchs[1].group('telephone')
' 62284321'
>>> matchs[1].group('address')
' BUPT'
>>> matchs[1].group('email')
'  CharlesWang@peoplemail.com.cn'
```

在这个例子中，用`(?P<xxx>.*)` 代替了`(.*)`，就可以用`m.group('xxx')`的索引匹配的group ，而不是用group 的号码了，在写很复杂的regexp 的时候，显得特别有用。这种方法是python 自己的扩展，和其他方法不兼容，听说Perl 5 提供了类似的机制，但是为了保持向后兼容Python 还是保留了这种扩展。在表示替换的时候，就用`(?=<xxx>)` 注意没有了regexp 的部分。如: `(\b\w+)\s+\1` 就表示匹配连续出现的两个同样的单词。可以改写成为，`(?P<n>\b\w+)\s+(?=<n>)`，是一个意思。还有一种方案，适合于已经用group 的号码了，只是想增加一个group 但是又不想影响原来的号码顺序。术语叫做Non-capture group。增加了一个group，但是不为他分配号码，但是没有办法在MatchObject 得到这个group 的匹配部分了。用`(?:...)`来表示一个Non-capture group。其中: 表示这种解决方案的扩展方法。这不是个好办法。

### §3.3.8 Compile Flag

在用`re.compile` 得到RegxObject 的时候，可以有一些flag 用来调整RegxObject 的详细特征。

| | |
|---|---|
| DOTALL，S | 让. 匹配任意字符，包括换行符\n |
| IGNORECASE，I | 忽略大小写 |
| LOCALES，L | 让\w、\W、\b、\B 和当前locale一致 |
| MULTILINE，M | 多行模式，只影响^和$ |
| VERBOSE，X | verbose 模式 |

## §3.4 用struct模块处理二进制数据

C 语言的程序员可能有这样的经历，定义一个struct ，然后直接用下面的方式处理。

```
struct{
...
} header;
write(fd,(void*) &header,sizeof(struct));
read(fd,(void*) &header,sizeof(struct));
```

在读写二进制文件，如声音，图片，图像等等，他们一般都有一个文件头格式。这种办法极为适用。还有在处理网络通信程序的时候，也是很有用的。Python 也有类似的处理办法。而且很简单有效，只要struct 模块中的pack，unpack，和cacsize 就搞定了。使用这个模块的时候，多少需要对C 语言有些了解。看个例子:

```
$cat a.c
#include <stdio.h>
struct testheader{
    int version;
    char tag[4];
} ;
struct testheader header={0x00010002,"WAVE"};
int main(int argc, char **argv)
{
    FILE * fp = fopen("test.dat","wb");
    if(!fp) return 1;
    fwrite((void*)&header,1,sizeof(header),fp);
    fclose(fp);
    return 0;
}
$gcc -o x a.c
$./x
$xxd test.dat
0000000: 0200 0100 5741 5645                      ....WAVE
```

一个C 的程序，写了一个文件头。包含版本信息和一个字符常量。xxd 可以显示二进制文件。little endian 格式。

```
>>> import struct
>>> data = open('test.dat','rb').read()
>>> start,stop = 0 , struct.calcsize('hh4s')
>>> ver1,ver2,tag= struct.unpack('hh4s',data[start:stop])
>>> ver
2
>>> subver
1
>>> tag
'WAVE'
```

这个例子中，可以得到二进制的文件头信息。unpack 的第一个参数用于描述struct 的结构，第二个参数是输入的二进制数据。

格式定义如表3.4-4。 表3.4-4有几个值得注意的地方。

| 格式 | C 语言类型 | Python 类型 |
|---|---|---|
| x | char | 无(表示填充字节) |
| c | char | 长度为1的字符串 |
| b | signed char | Integer |
| B | unsigned char | Integer |
| h | signed short | Integer |
| H | unsigned short | Integer |
| i | signed int | Integer |
| I | unsigned int | Integer |
| l | signed long | Integer |
| L | unsigned long | Integer |
| q | signed long | Integer |
| Q | unsigned long | Integer |
| f | float | Float |
| d | double | Float |
| s | char[] | String |
| p | char[] | String |
| P | void* | Integer |

表 3.4-4: struct 模块中的格式定义

| 字符 | 字节顺序 | 长度和对齐方式 |
|---|---|---|
| @ | native | native |
| = | native | standard |
| < | little-endian | standard |
| > | big-endian | standard |
| ! | network (= big-endian) | standard |

表 3.4-5: 字节顺序定义

- q，Q 只有在机器支持64 位操作的时候才有意义。
- 每个格式前面可以有一个数字，表示多个。如上面例子中可以写成。

```
>>> struct.unpack('2h4s',data[start:stop])
(2, 1, 'WAVE')
```

- x 表示填充位，没有输出结果。如

```
>>> struct.unpack('xxh4s',data[start:stop])
(1, 'WAVE')
```

- s 格式就是表示一定长度的字符串，4s表示长度为4的字符串。但是p 不同，他表示“Pascal String”，二进制数据中，而一个字节表示字符串长度，最大256，后面才是该长度的字符串。
- P 用和机器相关的长度，转换一个指针。64 位机器用long integer，32 位的用integer。

在缺省的情况下，struct 模块根据本地机器的字节顺序转换，也就是说，Intel 的CPU用little-endian，SPARC 的 CPU 用big-endian。字节对齐的方式和本地的C 语言编译器中的对齐方式是一致的。还可以用格式中的第一个字符来改变这种方式，如下表3.4-5: 例如:

```
>>> data
'\x02\x00\x01\x00WAVE'
>>> struct.unpack('!hh4s',data)
(512, 256, 'WAVE')
>>> struct.unpack('<hh4s',data)
(2, 1, 'WAVE')
```

pack 和unpack 的操作相反。

## §3.5 用Cmd 模块编写简单的命令行接口

可以在/usr/lib/python2.2 下找到cmd.py，用他可以很简单的编写一个命令行接口，用于大型程序的测试，或者直接就使用这种接口，作为最终产品。

为了快速开发程序的原型，用于以后开发的基础，这样的简单接口强大而且适用，例如GDB，Python 交互解释器，Bash ，Octave ，Gnuplot ，等等等等就是这样接口。

Cmd 使用了readline 库，这样所有以上工具中常用的命令行编辑命令都是起作用的，还有历史编辑功能。如表3.5-6中包含了常用的一些功能键。这些在所有使用readline 的工具都是通用的，包括上面提到的gnuplot 等程序。

| | |
|---|---|
| C-a | 移动到行首 |
| C-e | 移动到行尾 |
| C-u | 删除到行首 |
| C-w | 删除上一个词语 |
| C-y | 粘贴删除的内容 |
| C-f | 向左移动一个格 |
| C-b | 向右移动一个格 |
| M-f | 向左移动一个单词 |
| M-b | 向右移动一个单词 |

表 3.5-6: 命令行编辑的快捷键

Cmd 是如此的短小，让我大吃一惊，cmd.py共有269 行，还包括47 行的文档。参考cmd.py 的源代码会得到更多的信息。

### §3.5.1 简单的例子

源代码demo_cmd.py 如下：

```
# 从 cmd.py 导入 class Cmd
from cmd import *
# 定义 Cmd 的子类
class CmdInterface(Cmd):
    # 定义一个命令
    # argc 是这个命令的命令行参数
    def do_say_hello(self,argc):
        print "hello! " + argc

if __name__=="__main__" :
    # welcome 定义系统的欢迎信息
    welcome ="welcome a simple command interface demo demo"
    # 进入命令解释的循环，读一个命令并且解释执行。
    CmdInterface().cmdloop(welcome)
```

运行结果:

```
%python demo_cmd.py
welcome a simple command interface demo demo
(Cmd) help

Undocumented commands:
======================
help  say_hello

(Cmd) say_hello
hello!
(Cmd) say_hello Charles
hello! Charles
```

每一个命令都是一个成员函数，如果定义了`do_foo` 这样一个成员函数，那么碰到`foo` 这个命令的时候，就会调用`do_foo`。`argc` 就是`foo xxx` 后面跟的所有命令行参数，`xxx`。

`cmd_loop`表示进入命令循环，读取一个命令行，解释命令行，找到合适的成员函数，执行成员函数。`cmd_loop` 还做很多其他的事情。例如调用readline 库等等。

Cmd 缺省的提供help 命令，用于提供在线帮助。

### §3.5.2 定义默认命令

如果输入了一个命令，找不到对应的命令处理函数，就会执行`default(self,line)` 其中`line` 表示输入的命令行。

Cmd 提供的默认动作是

```
print '*** Unknown syntax:', line
```

### §3.5.3 处理EOF

上面的程序有一个毛病，就是不能正常退出。一般程序碰到文件结束符号EOF (Control-D)，表示程序应该退出了。如果

```
% echo "say_hello Charles" | python demo_cmd.py
```

就会有死循环。

加上EOF 的处理就可以正常退出了。

```
def do_EOF(self,arc):
    return 1
```

`do_EOF` 会在碰到文件结束的时候会被执行。在从标准输入读入的时候，`Control-D`表示文件结束。任何一个`do_xxx` 函数如果返回真值，就表示要退出`cmd_loop`。

还可以输入“ `exit`” 退出。

```
def do_exit(self,arc):
    return 1
```

### §3.5.4 处理空命令

在缺省的情况下，简单的按一个回车，输入一个空行，表示执行上一条命令。

如果想改变这种处理，那么定义下面的函数:

```
def emptyline(self):
    pass
```

输入空行的时候，会执行emptyline。

### §3.5.5 命令行自动补齐

这个功能是最吸引人的功能，使用Bash 的人常常使用tab 键，用于自动补齐文件名称，和其他参数。

缺省的补齐功能是补齐当前目录中的文件名称。

加入下面的函数可以改变缺省动作:

```
def complete_say_hello(self,text,line,begingidx,endidx):
    x=["Charles","Smith","Tom"]
    return [ k for k in x if k.find(text)==0]
```

如果输入“say_hello” 后按两次tab ，就会出现

```
Charles  Smith    Tom
```

用于提示用户补齐的命令。如果输入“C” 然后按tab 键，就会自动输入“Charles”。

complete_say_hello 仅仅用于say_hello 的命令的自动补齐功能。text 表示用户输入的单词的开头，line表示当前命令行，beginidx 表示text 在line 中出现的位置。endidx 表示text 在line中的结束位置。这些参数为用户提供的足够的信息，用于提供命令行补齐的功能。

如果不希望用tab 键来补齐命令，那么可以通过init函数改变这个键。

```
x=CmdInterface("?")
```

改为“?” 用于命令行补齐。

### §3.5.6 改变标准输入输出

init 还可以提供改变标准输入和标准输出的机会。缺省的情况下，他们分别是sys.stdin 和sys.stdout。

### §3.5.7 改变提示符

Cmd 的缺省提示符是“Cmd” ，可以通过改变prompt 的属性，改变命令行提示符。例如:

```
x=CmdInterface()
x.prompt="?"
x.cmdloop(welcome)
```

### §3.5.8 提供在线帮助功能

细心的读者已经发现，help 命令的输出中有“Undocumented commands:”。

Cmd 的在线帮助把命令分为三种。

- Undocumented commands:
  就是没有提供任何帮助信息的命令。
- Documented commands:
  就是提供帮助信息的命令。
- Miscellaneous help topics:
  就是提供了帮助信息，但是没有与之对应的命令。用于提供对一组命令，或者叫做一个主题topic 的帮助。

因为我们没有为say_hello 提供任何帮助信息，那么他就是Undocumented commands. 如果我们定义成员函数:

```
def help_say_hello(self):
    print """
    say_hello [<person>]
    say hello to somebody
    """
```

say_hello 就变成了Documented commands. 输入“ help say_hello” 就会执行help_say_hello。

如果我们定义:

```
def help_test(self):
    print """
    this is only a test topic
    """
```

“test” 就成为Miscellaneous help topics。可以输入“help test” 执行help_test

可以通过“?test” “?say_hello” 得到一样的效果。

### §3.5.9 运行Shell 的功能

我们定义:

```
def do_shell(self,argc):
  import os
  os.system(arg)
```

可以输入“shell ls” 或者“!ls”，执行shell 成员函数。

### §3.5.10 getattr 功能的使用

getattr 函数的使用，是Cmd 模块中的关键技术。

我们知道一般的命令行解释的程序，都是要做一个表，用来对应每一个命令，和相应的命令处理函数，但是Cmd 模块中，只要定义命令处理函数就可以了，是怎么实现的呢？

```
def onecmd(self, line):
    # line 是用户输入的命令行。
    # 把 line 分解成为命令部分和参数部分。
    cmd, arg, line = self.parseline(line)
    try:
        # 得到以 do_cmd 格式开头的函数对象。
        func = getattr(self, 'do_' + cmd)
    except AttributeError:
        return self.default(line)
    # 调用 func 对象。
    return func(arg)
```

由上面的一段代码，可以看到getattr 的使用是关键。这种特性使得程序可以动态知道对象的所有成员。

这段代码有一个小的问题，就是没有对得到的func 作检查，如果能判断callable(func) 就会更好，不然，子类定义的do_cmd 的是成员函数，而是成员变量，那么就会产生异常。

同样，帮助信息的生成，也使查询help_cmd 格式的成员函数。

而帮助的分类也是和getattr 有关系。

- Undocumented commands:
  有do_cmd 格式的成员函数，但是没有help_cmd 格式的成员函数。
- Documented commands:
  有do_cmd 格式的成员函数，也有help_cmd 格式的成员函数。

- Miscellaneous help topics:
  没有do_cmd 格式的成员函数，但是有help_cmd 格式的成员函数。

这样，写程序的时候不需要为各种类型的帮助函数做分类，程序本身就提供了足够的信息，用于分类。

## §3.6 处理命令行选项

熟悉Unix 的人都知道，Unix 下有很多命令都是可以处理很多很多命令行选项，用于控制程序的行为，和指明程序要处理的文件名称。如:

```
ls -l -a
grep -i a somefile
```

getopt 就是方便程序员处理命令行参数的一个module 。getopt.py 在/usr/lib/python2.2/ 下

### §3.6.1 一个简单的例子

源代码demo_getopt.py

```
#导入 getopt
from getopt import *
import sys
opts,args = getopts(sys.argv[1:],"a","--all")
print opts
print args
```

运行结果:

```
%python demo_getopt.py -a f
[('-a', '')]
['f']
%python demo_getopt.py
[]
[]
%python demo_getopt.py --all
[('--all', '')]
[]
%python demo_getopt.py -x
Traceback (most recent call last):
  File "demo_getopt.py", line 4, in ?
```

```
    opts,args = getopt(sys.argv[1:],"a","all")
  File "/usr/local/lib/python2.3/getopt.py", line 91,
  in getopt
  opts, args = do_shorts(opts, args[0][1:], shortopts,args[1:])
  File "/usr/local/lib/python2.3/getopt.py", line 191,
  in do_shorts
    if short_has_arg(opt, shortopts):
  File "/usr/local/lib/python2.3/getopt.py", line 207,
  in short_has_arg
    raise GetoptError('option -%s not recognized' % opt, opt)
getopt.GetoptError: option -x not recognized
```

从上面的例子可以看出，getopt 需要有3 个参数输入。

- sys.argv[1:] 要解释的命令行选项。不包括命令本身。
- "a" 短选项
  短选项是用一个“-” 分隔的选项名称，就象ls -a 中a 一样。只用一个字母表示。
- ["all"] 表示长选项。
  如果只有一个选项，那么就直接写成一个字符串，如果不只一个长选项，那么就要写成字符串的list。

无论长短选项，都不包括前面的“-” 或者“–”。

getopt 返回一个tuple，含有两个元素，opts,args，args 表示没有解释过的sys.argv[1:] 的部分，一般表示需要处理的文件名称。

opts 表示发现的选项，是一个list，list 中的每一个元素是一个tuple，表示一个pair,(opt,val)，opt 表示在命令行中发现的选项的名称。val 表示选项所对应的参数。如果是空字符串，表示该选项没有参数。opts 中选项出现的顺序和命令行上出现的顺序是一样的。允许选项重复出现多次。

如果在命令行上发现不能解释的参数，则抛出GetoptError 的异常。GetoptError 中的msg属性表示出错的描述，opt属性表示不能解释的选项的。

### §3.6.2 带有参数的命令行选项

对于短选项，在选项的后面跟一个冒号表示这个选项需要参数。例如：上个一个例子中的getopt 改为。

```
getopt(sys.argv[1:],"a:","all")
```

如果命令行中没有提供选项，一样会抛出GetoptError异常。运行结果:

```
%python demo_getopt.py -a
Traceback (most recent call last):
  File "demo_getopt.py", line 4, in ?
    opts,args = getopt(sys.argv[1:],"a:","all")
  File "/usr/local/lib/python2.3/getopt.py", line 91, in getopt
    opts, args = do_shorts(opts, args[0][1:], shortopts, args[1:])
  File "/usr/local/lib/python2.3/getopt.py", line 195, in do_shorts
    opt)
getopt.GetoptError: option -a requires argument
%python demo_getopt.py -a b
[('-a', 'b')]
[]
```

对于长选项，则要在选项名称后面加一个等号。例如:

```
getopt(sys.argv[1:],"a:","all=")
```

结果是:

```
%python demo_getopt.py -a b --all xx file1 file2
[('-a', 'b'), ('--all', 'xx')]
['file1', 'file2']
```

### §3.6.3 optparser 模块

optparser 模块是Python 2.3 带的一个强大、灵活、易用、易扩展的命令行解释器。使用optparser 模块，只用很少的代码，就可以为你的程序增加流行的，专业的命令行接口。他比getopt 模块更好用。

这里有一个简单的例子

```
#!/usr/bin/env python
# simplyOptparser1.py
from optparse import OptionParser

parser = OptionParser(version="%prog 0.1 ")
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")
```

```
(options, args) = parser.parse_args()

D = vars(options)
for i in D:
     print i,"=",D[i]
```

这样，你的程序就像所有GNU 下的工具一样，有很多的option 可以使用。

```
%./simplyOptparser1.py -f output --quiet
verbose = False
filename = output
%./simplyOptparser1.py -qf output
verbose = False
filename = output
%./simplyOptparser1.py -q --file=output
verbose = False
filename = output
%./simplyOptparser1.py --quiet --file=output
verbose = False
filename = output
%./simplyOptparser1.py --help
usage: simplyOptparser1.py [options]

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -fFILE, --file=FILE   write report to FILE
  -q, --quiet           don't print status messages to stdout
%./simplyOptparser1.py --version
simplyOptparser1.py 0.1
%
```

程序看起来很专业。

### optparser 的设计理念

optparser 模块的设计理念很大程度上受了Unix 和GNU 下的工具集的影响。现介绍一些术语:

- argument (命令行参数)

```
%ls -l /usr
```

其中`-l usr` 就是命令行参数。`sys.argv` 是一个list ，用于表示命令行。例子中，`sys.argv` 就是

```
['ls','-l','/usr']
```

命令行参数就是`sys.argv[1:]` ，因为`sys.argv[0]` 表示命令本身。

- option (命令行选项)
  程序在执行的时候，从argument 得到信息，用来调整程序的执行。传统的Unix和GNU 的命令行语法是:

  ```
  -q -f  --file --quiet
  ```

  一个减号，后面跟一个短选项，或者两个减号，后面跟一个长选项。多个短选项可以合并在一起，如`-qf`
  optionparser 不支持下面的语法。
  - 一个减号后面跟超过两个字符以上的短选项，如`-pf`，否则就分不清楚是`-p -f` 两个选项，还是`-pf` 一个选项。
  - 一个减号后面跟一个长选项，如`-file` ，跟上面类似，就没有办法分清楚了`-file` 是一个选项还是`-f -i -l -e` 四个选项?
  - 用加号，代替减号，如`+f` ，`+rgb` ，不合习惯。
  - 用斜杠代替减号，例如`/l` ，`/rgb` ，这是Windows 和VMS 的用法，但是在Unix下斜杠表示目录，所以ls /l— 就分不清楚是`/l` 是选项，还是目录名称了。

- option augument(选项参数)
  像`-f outputfile` 这样的选项，后面跟一个参数，`outputfile` ，表示选项的参数。`-foutputfile` 和`-f outputfile` 都一样。有的选项不需要参数，有的选项必须给出参数。
  是否需要optional option argument (可选的选项参数)呢？也就是说，如果后面跟了参数就认为他是选项参数，如果没有，就认为选项没有参数。例如，假如`-a` 选项后面有optional option argument:

  ```
  ls -a xxx -b
  ```

  表示`-a` 带有参数`xxx` ，

  ```
  |ls -a -b|
  ```

  表示`-a` 没有参数。但是，如果是`ls -ab` 那么我们该怎么认为呢？分不清楚了，所以optionparser 模块不支持optional option argument 。

- positional argument (位置参数)
  在命令行参数中，除了选项以外，剩下的就是positional argument ，一般用于表示文件名称。例如：

  ```
  ls -l /usr /lib
  ```

  其中中`['/usr', '/lib']` 就是positional argument 。
- required option (必备选项)
  是指用户必须提供的选项。这是个坏主意。如果你的程序要求用户必须提供一些option ，那么这是一个很不好的接口设计，很少有用户会耐心的输入所有你要option 。当然你如果一定要这么做，也是又办法的。

例子:

```
命令行 : prog -v --report /tmp/report.txt foo bar
命令名称        : prog
argument 命令行参数: -v --report /tmp/report.txt foo bar
option : -v --report
option argument : /tmp/report.txt
positional argument : foo bar
```

**设计命令行接口忌讳**

- 要求用户提供的德必须提供的信息越少越好。如果没有option ，程序也可以正常运行。大多数的Unix 和GNU 工具都是这样的。除了find, tar, dd 等等，很多人批评这些程序的命令行接口不标准，容易引起混淆。
- 不要有必备选项required option 。如果用户必须提供这些信息，那么他们就不是可有可无的选项了，而是positional argument 。
- 每个option 都应该有合理的默认值。
- 长短option 相互配合，一一对应。短的易于交互式输入方便，长的可以提高脚本程序的可读性。所以，交互式的shell 中，输入`ls -l` 比较好。但是在批处理的脚本程序中，最好写成`--long` 。

例如: cp 命令

```
cp SOURCE DEST
cp SOURCE ... DESTDIR
```

你不用任何选项，cp 总能完成他的主要任务，把一个文件拷贝成为另一个文件，或者把一堆文件拷贝到一个目录中去。当然，你也可以指定很多选项，用于指定是否保留文件的权限，是否保留修改时间，是否跟随链接，是否递归拷贝目录等等，但是他们都不影响cp的运行。

positional argument 一般是程序必需的信息。当然，有些很多程序不需要任何信息，也可以很好的运行，例如cat, sort, cut, grep, sed, awk, head, tail, ed, uniq, fold, wc, fmt,split, od, xxd, sum, 等等，他们都接收positional argument 中的参数作为输入的文件名称，但是如果没有指定输入文件的名称，那么就从标准输入读。这样的命令非常好，和管道机制配合起来，威力无穷。而有些程序，则必须要给出这些信息，如diff, join, tr, 因为这些程序缺少了必备的信息，就不能完成它的任务。如: diff 必须要给出两个文件名称，否则他比较什么呢? join 是把两个文件合并，所以必须给出两个文件名称。tr 是翻译功能，必须说明把什么翻译成什么，他才能工作。所以这些必备的参数都作为positional argument。

cut 在这一点上做得不好，

```
%cut
cut: you must specify a list of bytes, characters, or fields
Try ‘cut --help’ for more information.
```

这是一个例外，cut 一定要用户提供信息，怎样提出从输入中提出若干列。用户有多种选择，指明按字节byte，字符character 还是字段field 选择一部分。我认为应该有一个默认动作`cut -f 1` ，也许这不是很好的一个选择，或者默认动作是`cut -b1` ，`cut -c1` ，但是无所谓，总之应该提供一个默认的动作，别伤了用户的心，别让用户失去耐心。

假设，你的程序要求用户提供17 的不同的信息，程序才能成功的工作，无论你提供什么样的界面，是command line, GUI 都一样，那么几乎没有人有耐心运行你的程序，甚至包括你自己。

当然，允许用户提供的option 越多，你的程序的灵活性就越好，程序也就越难实现。提供option 的方式很多，可以是配置文件的方式，可以是在菜单上选择Preferences ，然后选择一个东西。还可以是命令行参数的方式，无论哪种方式也好，要小心设计，否则就容易晕头，包括你自己。想象有的程序给出1000 个option ，然后让你修改其中的一个，光找到那个option 就很费事了。提供帮助信息也会令人头大。

### 基本用法

1. 首先

   ```
   import optparse
   parser = optparse.OptionParser()
   ```

   这个不难理解，先创建一个OptionParser 的对象。

2. 其次

   ```
   parser.add_option("-f","--file",...)
   ```

   增加option 。

3. 解释命令行

```
(options, args) = parser.parse_args()
```

options 中含有所有命令行提供的option 的信息。args 是positional argument 的信息。

现在，重点讨论`add_option` 。其中头两个参数不言而喻，一个表示短option，一个表示长option 。二者一一对应。

后面的参数提供各种功能:

1. 需要参数的option 。
   例如:

```
parser.add_option("-f", "--file",
     action="store", type="string", dest="filename")
```

其中`action` 表示怎样处理option ，上面的意思就是说: 把`-f` 或`--file` 选项后面的选项参数保存为`filename` 变量。例如:

```
args = ["-f", "foo.txt"]
parser.add_option("-f", "--file",
      action="store", type="string", dest="filename")
(options, args) = parser.parse_args(args)
print options.filename
```

就会打印出来foo.txt 。类型是字符串。

`type` 指明类型，还可以是int 或者float 。默认值是string 。如果发现参数不符合指定的类型，那么就打印出错信息，然后退出程序。

`dest` 指明存储变量的名称。默认值是和长option 的名称相对应的。把长option 的开头两个减号去掉，把其中的减号变成下划线，就是变量的名称。上个例子中就是把相应的变量作为`options.filename` 保存。

2. 不需要参数的option 。
   把

```
action = "store"
```

改成为

`action = "store_false"` 或者 `action = "store_true"`

例如:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

如果命令行中有`-v` 或者`-q` ，就会分别把`verbose` 设置成为True 和False。

3. option 参数的默认值。
   如果没有设置option 参数的默认值，而且也没有在命令行中发现该选项，那么对应的变量的值就是None 。这满足大多数情况下的需求，但是如果你要自己设置默认值也是可以的。下面的例子可以改变默认值。

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose",
                default=True)
```

下面的例子和前一个例子的效果一样。

```
parser.add_option("-v", action="store_true", dest="verbose",
            default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

如果是

```
parser.add_option("-v", action="store_true", dest="verbose",
           default=False)
parser.add_option("-q", action="store_false", dest="verbose",
          default=True)
```

那么`verbose` 的默认值是True 。因为后面的语句会覆盖前面语句的默认值。

4. 生成帮助
   下面的例子会自动生成帮助信息。 运行结果:

```
% ./optparser.help.py  --help
usage: optparser.help.py [options] arg1 arg2

options:
```

```
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -fFILE, --file=FILE   write output to FILE
  -mMODE, --mode=MODE   interaction mode: one of 'novice',
                        'intermediate'[default], 'expert'
%
```

说明:

(a) 自动打印使用方法usage。其中%prog 会自动变成sys.argv[0] ，表示命令名称。usage 的默认值是

```
'usage : %prog [options]'
```

(b) 增加每一个option 的时候，指定help ，那么生成的帮助中就会打印相应的信息。help 的默认值是空字符串。

(c) meta-variable 是指在帮助信息中提供的参数的代名词。例如帮助信息中的-mMODE -fFILE 等等。可以用metavar 来指定。metavar 的默认值是长option 的名字去掉头两个减号，然后变成大写的来的。

5. 为option 分组
当你的程序要接受很多option 的时候，最好把他们分成若干组，这样便于管理，无论对程序开发者还是最终用户，这都是一个好主意。

例子:

运行结果:

```
$./opt.group1.py --help
usage: opt.group1.py [options] arg1 arg2

options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -fFILE, --file=FILE   write output to FILE
  -mMODE, --mode=MODE   interaction mode: one of 'novice',
                       'intermediate'[default], 'expert'

  Dangerous Options:
```

```
    Caution: use these options at your own risk.  It is
believed that some  of them bite.
    -g                Group option.
$
```

说明:

(a) OptionGroup 函数接收三个参数，所属的parser 对象，group 的简短描述，和group 的详细描述。返回一个option group 的对象。

(b) 然后像parser 对象一样，为group 对象增加option 。

(c) 用`add_option_group`函数把一个group 增加到一个parser 对象中。

6. 打印版本信息。和usage 类似，在创建parser 对象的时候，指定version 参数，就可以打印版本信息了，同时自动增加了`-v --version` 这个option 。

   例如：

```
parser = OptionParser(usage="%prog [-f] [-q]",
                      version="%prog 1.0")
```

   同样`%prog` 会变成sys.argv[0] 。

## §3.7 关于时间的模块

time 这个模块在不同的平台上有所不同，所以移植性不是太好，但是只有少数函数的移植性不好，大多数还是可以的。

模块中的函数名称和标准的C 语言的函数名称是一样的，所以参考平台上的相应的C 语言参考手册是非常有用的。

现有几个术语值的解释:

- epoch (时间原点)，相对于epoch 时间是指从epoch 开始，经过的秒数。epoch 的时间是0 。因为epoch 相对于epoch 经过了0 秒。在UNIX 系统中，epoch 的时间是1970 年1 月1 日0 点0 分0 秒。
  模块中的函数值能处理一段时间内的时间，即从epoch 开始，到一个时间点，这个时间点，与机器上用于表示时间的数据位数有关系，并且和C 语言的库函数有关系。这是因为计算机总是使用有限的比特数来表示一个数值，他总是有限制的，不能表示无限大的范围内的所有数。在UNIX 系统中，这个最大值一般是2038 年。
- 2000 年问题。Python 处理时间的函数，是使用C 语言库，所以一般不会有2000 年问题，因为，时间的表示使用相对于epoch 经过的秒数来表示的。如果为了向后兼容的话，可以查看python-lib 的帮助。

- UTC (Coordiated Universal Time) ，全球统一时间，或者说GMT (Greenwich Mean Time) 格林威治时间，
- DST( Daylight Saving Time) ，表示夏令时，C 的库函数可以处理夏令时，具体怎么做，是根据不同地区的设置有关系，一般有一个系统文件，用于描述夏令时的规则。一般系统管理员应该已经做好这个文件了。
  gmtime() , localtime(), strptime() 的返回值，还有asctime(), mktime() ,strftime() 的参数都是一个tuple ，其中含有9 个整数。这个9 个整数的含义如表**??**

| 索引值 | 属性 | 范围 |
|---|---|---|
| 0 | `tm_year` | 如1993 |
| 1 | `tm_mon` | 1 – 12 |
| 2 | `tm_mday` | 1 – 31 |
| 3 | `tm_hour` | 0 – 23 |
| 4 | `tm_min` | 0 – 59 |
| 5 | `tm_sec` | 0 – 61 |
| 6 | `tm_wday` | 0 – 6, Monday is 0 |
| 7 | `tm_yday` | 1 – 366 |
| 8 | `tm_isdst` | 0, 1 or -1 |

表 3.7-7: 表示时间的

各个函数之间的关系。图3-1说明了各个函数之间的关系。

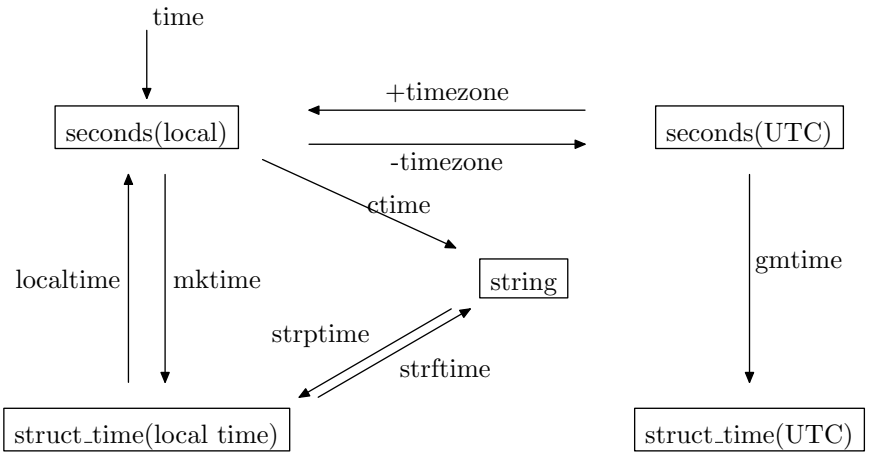


图 3-1: 时间函数之间的关系

# 第四章

# Tkinter 编程

## §4.1 Tkinter介绍

### §4.1.1 什么是Tkinter

Tkinter 是一个python 的模块，是一个调用Tcl/Tk 的接口。Tcl/Tk 是为数不多的(也许是唯一的)跨平台的脚本图形界面接口。

在大多数Unix 平台上都有Tcl/Tk。现在Windows 和MacOS 上也可以使用Tcl/Tk

Tk 的底层C 语言接口在动态连接库_tkinter 中，Tkinter.py 是利用_tkinter 的一个Python 的包装，用于为Python 用户提供接口。

系统层次如图4-1下，以X windows 为例：

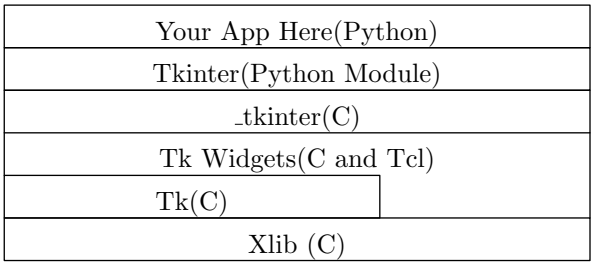


图 4-1: Tkinter 系统层次图

**Your App Here (Python)** Python 的应用程序，调用Tkinter.py gv

**Tkinter (Python Module)** Tkinter.py 用于把Python 中的表达方式转变成为Tk 的命令和参数。

**tkinter (C)** 用于把Tkinter.py 生成的Tk 的命令和参数，传递给Tk 解释器，让其解释执行。

**Tk Widgets (C and Tcl)** Tk 是由C 和一部分的Tcl 脚本写成的。

**Tk (C)** Tk 的底层C 语言部分。

**Xlib (C)** 调用X 的库，和X Server 通信。

Python 用户不会直接调用_tkinter ，而是通过Tkinter.py 间接使用他。

Tkinter 不是唯一的Python 图形编程接口，是其中比较流行的一个。最大的特点是跨平台。明显的缺点是性能不太好，因为Tcl/Tk 也是解释性语言，Python 也是，用Python 生成Tcl/Tk 代码，然后再用Tcl/Tk 解释执行，速度当然比直接调用底层图形接口的要慢，这也是他为跨平台所付出的代价。但是大多数情况下，你不会建立成千上万个按钮，窗口吧？所以一般是可以满足要求的。

一般使用Tkinter 的方法是

```
from Tkinter import *
```

也可以是

```
import Tkinter
```

但是Tkinter 的每个函数和类的前面都要有`Tkinter.somefunc()`，写起来很费时间。

什么是Widget ？ Widget 是Button(按钮)，Canvas(画布)，Menu(菜单)，Label(单行标签)，Message(多行标签)，Listbox(列表)，Entry (单行输入框)，Frame(框架)，Radiobutton(多种选一的单选框)，CheckButton(复选框)，Scrollbar( 滚动条)，Text(多行文本输入框)，Scale(在一定范围内选择一个数值的滚动条) TopLevel(窗口) 等等的统称。

什么是option ？ 所有的Widget 都有很多options ，包括fg(前景色)，bg(背景色)，font(字体)，command(事件处理函数)，text(内容) 等等。

## §4.2 Hello Tkinter 程序

### §4.2.1 简单的例子

学习语言都是从“Hello World” 开始，我也不破例。

```
# File: hello1.py

# 导入 Tkinter
from Tkinter import *

# 创建一个 root Widget
# 通常每一个 Tkinter 的程序都有一个，而且只有一个
# root widget
```

```
# 他是一个简单的窗口，带有标题栏，和边框和其他 Windows
# Manage 提供的装饰。
# 必须在创建其他 Widget之前，创建 root widget。
# 也可以不创建，Tkinter 会自动为你创建一个。
root = Tk()

# 创建一个 Label，叫 w，
# root 是 w 的 master(父窗口)，
# w 是 root 的 slave (子窗口)
# 这里的术语有点不同于 Windows， Windows 下
# 是 parent window， 和 child window。
# text 是 w 的一个 option (选项)，表示 w 中
# 要显示的内容，可以是文字或者图片。
w = Label(root, text="Hello, world!")
# w 创建之后，并没有真正的显示在屏幕中，
# 只有在 pack 之后，才能根据 Label 中的内容，计算好
# Label 的大小，放的位置，然后显示在屏幕上。
w.pack()

# 进入事件循环，接收来自用户的事件，执行对应的
# 事件处理，直到用户调用 quit() 或者 窗口被关闭了。
# mainloop() 还要处理内部的 widget 的更新，来自
# 和 Windows Manager 的通信。
root.mainloop()
```

运行程序

```
%python hello1.py
```

运行结果如图4-2

图 4-2: hello1.py 的运行结果

顺便介绍一下WM(Windows Manager)，熟悉X Windows 的用户也许对WM不会陌生。但是Windows 用户就会很陌生了。在X 的体系结构中，窗口的边框，标题栏，关闭按钮，最大化，最小化，窗口的移动，放大缩小，等等都是WM 来做的，WM 不过是一个普通

的X Client，通过标准协议和X Server 通信。任何一个X client 程序都可以和WM 来通信。在Windows 系统中，WM 和整个操作系统整合在一起，很难分的清楚，但是我从Win XP和Win 2000 看到，有一些Windows 程序可以通过某种机制完成WM 的功能，让Windows的外观看起来像MacOS ，或者X 下的其他WM 的效果。

### §4.2.2 另一个简单程序

在一个比较大的程序中，最好把一个你的程序写在一个类中，如下

```
# File: hello2.py
from Tkinter import *
class App:
    def __init__(self, master):
        # 创建一个 Frame 用于包含其他 widget
        frame = Frame(master)
        frame.pack()
        #创建一个按钮
        self.button = Button ( frame, # master widget
                               text="QUIT",
                               fg="red",
                               command=frame.quit
                               )
        self.button.pack(side=LEFT)
        self.hi_there = Button(frame,
                               text="Hello",
                               command=self.say_hi
                               )
        self.hi_there.pack(side=LEFT)
    def say_hi(self):
        print "hi there, everyone!"
root = Tk()
app = App(root)
root.mainloop()
```

运行结果如图§4.2.2

如果你按”Hello” 的按钮，会在控制台上打印“hi there, everyone!”。

如果w 是一个widget，w.pack(side=LEFT) 表示摆放w 的方式。


## §4.3 Widget 的配置

Tk 中的每一个widget 都有很多option ，通过改变这些option 可以改变widget 的外观。典型的option 有控制显示的内容，颜色，大小，事件处理函数等等，Tkinter 提供了统一的界面用于处理所有的option.

通常有以下方法处理widget 的option:

- 可以使用下面的办法在创建widget 的时候设置widget 的options。

  ```
  widgetclass (master , option=<value> ...)
  ```

  例如:

  ```
  w = label(root,text="hello",fg="red")
  ```

  创建一个widget ，叫做`w` ，他的master widget 是`root` ，他的option 使用后面的参数来给定的。所有的widget 的option 都有默认值，所以最简单的创建一个widget 的方法是所有的option 都使用默认值，也就是不指定任何option，只需指定widget 的master widget，如果你很懒的话，甚至连master widget 也可以不用指定，Tkinter 会用最近使用过的master widget 作为默认的master widget 。
- `w.cget(option)` 得到一个widget 的option 的值。`w` 是一个widget 。所有的widget class 都有把`__getitem__` 映射到了`w.cget` 上，所以他们两个没有什么区别。所以用w[option] 一样可以得到一个option 。例如:

  ```
  w = Label(text="abc")
  print w.cget("text")
  print w["text"]  # 没有区别，都可以得到 abc
  ```

  我觉得`w['text']` 的方式更好看一些。
- 用`w.configure` 或者`w.config` 设置option 的值。w.config 是w.configure 的缩写，所以功能上没有区别。用法和创建widget 的时候一样。

  ```
  w.config(option=<value>, ... )
  ```

  例如:

  ```
  w.config(text="xxx")
  ```

所有的widget class 把__setitem__ 映射到了w.config 上，所以

```
w["text"] = 'xxx'
```

效果一样。

如果有很多的option 要设置，那么用config 更好看一些。如果只有一个options 要设置，那么用后面的方式更好看[1]。

- config 成员函数如果不带任何参数，那么会得到一个widget 的所有可以设置的option的名称。w.keys() 也一样。我更喜欢使用w.keys() ，他的意义更清楚，简单。例如:

```
for k in w.keys():
    print "%s = %s"%(k,w[k])
```

可以打印一个widget 的所有的option 。

综上所述，访问一个widget 的option 有两套办法。

| | |
|---|---|
| w.config() | w.keys() |
| w.config(optionname=optionvalue,...) | w['optionname']=optionvalue |
| w.cget('optionname') | w['optionname'] |

具体使用那种由个人习惯决定，我觉得第二种方式更好。

## §4.4 Geometry Manager(几何管理器)

什么是Geometry Manager?

编写过GUI 的人有这样的经历，摆放好每一个Widget 是一件很繁琐的事情，这要调整每一个Widget 自身的大小，让他能够很漂亮的包含和显示他内部的内容，还要放好和其他兄弟Widget 之间的距离，让他们只之间的距离尽量均匀，还要设置好他在Master 的中位置，如果Master 的大小改变了，应该如何调整Widget 所有。

繁琐的事情就交给计算机去做，Tk 提供了几个Geemetry Manager。可以帮助你完成这些功能，只用很少的代码，就可以让你的界面看起来很专业。

这和Delphi 或者VB 等可视化编程环境，有很大不同。VB 是一种WYSIWYG，What you see is what you get (所见即所得)的方式，Geometry Manager 是一种WYTIWYG，What you think is what you get (所想即所得)。有点像TeX(LaTeX) 和Word 的区别。

在VB 中，你把Widget 摆放在哪，你看得到，运行的结果就和你看到的一样，但是你要不断的选择"格式/对齐" 等菜单，把他们对齐，放在合适的位置，但这需要艺术家的审美观念和人体工程学的观念，才能让他们看起来很美。

用Geometry Manager ，你需要用一些参数告诉他你想要的效果就可以了，一定是你想要的。

一般Tk 有三种Geometry Manager :

[1]也许你的审美观点和我不一样。

- Pack
- Grid
- Place

下面分别介绍。

### §4.4.1 Pack 管理器

有几个术语现介绍一下。

cavity : master widget 上的空间，表示分配一部分空间给一部分slave widget 之后，master widget 上还剩下的空间，用于分配给其他还没有摆放好的slave widget 。

parcel : 从cavity 中分配出的一个空间，用于摆放一个slave widget 。

每次给一个slave widget 分配空间的时候，都是从cavity 中得到一个parcel，然后再把slave widget 摆放在parcel中。然后从cavity 中减去parcel 的大小，得到一个新的cavity ，继续分配，直到所有的slave widget 都摆放完毕。

Pack 管理器的使用方法很简单`w.pack(options)` 。`w` 是一个slave widget。常用的options 有:

Pack管理器常用的option

**side** 表示把`w` 放在那个边上。可以是TOP (上) ，BOTTOM( 下)，LEFT( 左)，RIGHT( 右)。

**padx 和pady** 表示parcel 的每一个边和`w` 之间预留的空间。

**ipadx 和ipady** 表示`w`的每一个边和`w` 内包含的内容之间的预留空间。`w` 会因此变大。

**fill** 可以是None, x, y , both。如果parcel 太大了，那么就会根据fill 的值，增加slave widget 的大小。

- None: 表示维持slave widget 原来的大小。
- X : 表示只扩大slave widget 的宽度。
- Y : 表示只扩大slave widget 的高度。
- BOTH: 表示同时扩大slave widget 的宽度和高度。

值的注意的，fill 是指slave widget 的大小和parcel 的大小的之间关系，决定slave widget 怎样改变自己的大小，来适应parcel 的大小。

**anchor** 表示在parcel 中放置slave widget 的方式。可以是以下的值。缺省值是CENTER。

| | | | |
|---|---|---|---|
| W | 靠左放置 | E | 靠右放置 |
| N | 靠上放置 | S | 靠下放置 |
| NW | 靠左上放置 | NE | 靠右上放置 |
| SW | 靠左下放置 | SE | 靠右下放置 |
| CENTER | 放在正中间 | | |

这里值得注意的是，如果fill 的值是X ，那么anchor 是W 和E 就无所谓了，因为slave widget 的宽度已经是最大，靠左放还是靠右放就没有什么意义了。同样道理，如果fill 的值是Y ，那么anchor 是N 和S 就无所谓了。如果fill 的值是BOTH ，那么anchor option 就没有意义了。这很不好，grid 管理器在这一点上有改进。

**expand** 可以是0 或者1。如果是1 ，表示slave widget 会根据下面所说的方法，把master 的剩余空间分配个每一个slave widget 的parcel。如果是0 ，让剩余空间空着。

值的注意的是，expand 是只master 剩余空间和所有slave widget 的parcel 之间的关系如果把所有的slave widget 都摆放好了，但是还有剩余空间，他会决定是不是把master 剩余空间给所有parcel 。

注意，以上所有的option 如side，padx , pady , ipadx , ipady , expand, fill 都是调用pack() 时指定的option，而不是创建widget 的时候指定的option。

Pack 的算法

每一个master widget 都有一个packing list 用于管理他内部的所有slave widget。创建一个新的slave widget，会在master widget 的Packing list 的结尾追加一个slave widget。还可以用in , after , before 这三个option 改变顺序。

Pack 管理器按照packing list 总的widget 的顺序，一一摆放每一个widget。在刚开始，master widget 内部所有的空间都是空的，就是cavity (空洞)。

Pack 按照一下步骤意义摆放每一个slave widget。

1. 沿着一个cavity 的一个边切下一个矩形，叫做parcel (包裹)，具体沿哪一个边是由side option 指定的。

   如果side = TOP 或者side = BOTTOM ，那么parcel 的宽度就是cavity 的宽度，高度就是slave widget 要求的高度，再加上pady 和ipady 这两个options 所指定的数值。

   如果side = LEFT 或者side = RIGHT ，那么parcel 的高度就是cavity 的高度，宽度就是slave widget 要求的宽度，再加上padx 和ipadx 这两个options 所指定的数值。

   以后parcel 还可能因为expand option 所指定的值而变大。如果所有的slave widget 都放好的，发现master widget 还有多余的空间没有使用，那么就会根据expand option 来改变所有parcel 的大小。如果expand option 是0 ，那么就算了，就让那些空的

地方空着好了。如果expand option 是1 ，那么就把剩下的空间均匀的分配给所有的parcel。如果一个可以扩大的slave widget 的side option 是left 或者right ，那么把多余的水平距离会平均分配他们。如果一个可以扩大的slave widget 的side option 是top 或者bottom ，那么把多余的垂直距离会平均分配他们。这样也很缺乏灵活性，grid 管理器增加了权重的概念，是有所改进的。

2. pack 管理器slave widget 选择合适的高度和宽度。宽度一般是widget 所需宽度加上两倍的ipadx ，高度是一般是widget 所需高度加上两倍的ipady ，为什么是两倍? 是因为左右都需要留出ipadx 的距离，同样上下也都需要多留出来ipady 的距离。但是如果fill option 的值如果是X 或者BOTH，那么扩大宽度，slave widget 的宽度会是parcel 的宽度减去两倍的padx 的值，同样如果fill option 的值是Y 或者BOTH ，那么slave widget 的宽度会是parcel 的宽度减去两倍的pady 的值，

3. pack 管理器把slave widget 放在parcel 中。如果slave widget 比parcel 小，那么就根据anchor option 来放置slave widget，parcel 剩下的空间也不会给其他slave widget 使用了。如果padx option 或者pady option 的值不是0 ，那么在parcel 的边界和slave widget 的边界之间总是要留出来padx 和pady 那么多的空白空间。

放置好一个slave widget 后，pack 管理器会减去parcel ，剩下一个比较小的cavity 用来放置下一个slave widget。如果cavity 的空间不够放置一个slave widget ，那么这个slave widget 会占据cavity 所有的空间。当cavity 缩小至0 的时候，其他的剩下的slave widget 就看不到了，除非master widget 变大。

一般情况下master widget 会自动变大，术语叫做Geometry Propragation (几何传播，大小传播)。Geometry propragation 的意思就是说，自动调整master widget 的宽度和高度，用来正好的放置所有的slave widget ，并且会在widget 的层次结构中传递大小。什么叫做widget 的层次结构? 我们知道master widget 和slave widget 的关系。而master widget 又是另一个widget 的slave widget。这样就像形成了像爷爷，父亲，儿子，孙子这样的层次关系。也可以说是树叶，树茎，树根的关系。

首先计算树叶widget 的大小，树叶widget 没有slave widget ，他们会自己计算自己的大小，足以容纳下所有slave widget 的所有内容，然后包含这个树茎widget 的大小就可以根据他所slave widget 的大小改变，这样一直向上传递，直到TopLevel。

这样最终的结果就是所有的Widget 摆放得十分合理。当然，有的时候这不是我们想要的结果，那么我们可以通过调用下面的函数停止Geometry Propagation。

```
w.pack_propagate('true')
```

使用Pack 管理器的时候，经常会使用frame widget ，用于把一组widget 看成一个widget。

### §4.4.2 Grid 管理器

尽管Pack 很不错，但是，很多程序员认为Grid 管理器更好用。而且偏激的人会只用Grid ，而不用Pack。也许他是对的。我认为各有各的优点。

我认为Grid 管理器的概念简单，功能强大，灵活，确实比Pack 强一些。

使用Grid 的方法也很简单，和Pack 类似，调用w.grid(options) ，w 是一个slave widget。

Grid 管理器把一个master widget 的所有可用空间分成格状的很多小份。每一小份叫做一个Cell ，和Pack 管理器中的parcel 的概念很相像。用row ，column ，rowspan ，columnspan 就可以确定一个cell 。有下常用option。

**常用的**options

**row** 指明在那一行

**column** 指明在那一列

**rowspan** 指明占了几行

**columnspan** 指明占了几列

**ipadx ipady** 和Pack 管理器的意义相同。表示slave widget 内部增加的填充距离。

**padx pady** 表示slave widget 的边界和cell 的边界之间的填充距离。

**sticky** 可以是N ，W ，S ，E ，包含了是Pack 管理器中的fill ，anchor 的功能。表示当Cell 比slave widget 大的时候，如何摆放slave widget，必要的时候，增加slave widget 的高度或者宽度。

如果是单个的N 表示，向上靠，类似Pack 管理器中的anchor= N 。

如果是两个的组合表示，如N+S ，表示增加高度，类似于Pack 管理器中的fill = Y 。如E+W ，表示增加宽度，类似于Pack 管理器中的fill = X 。

如果两个的组合是N+E 表示不增加高度或宽度，向右上方摆放，相当于fill=None，anchor=NE 其它类似。

如果是多个组合N+W+S+E 表示放在中间。类似于Pack 管理器中的fill=BOTH 。

如果不设置sticky 的值，表示不增加高度也不增加宽度，放在中间，类似于Pack 管理器中的anchor=CENTER，fill=None 。

### Grid 管理器的算法

有三个步骤:

1. 计算至少需要多大的地方，就可以摆放下所有的slave widget 。
2. 用上一步计算的结果和master widget 的实际大小比较，如果不一样大，那么就增加或者减少master widget 的大小。
3. 根据每一个slave widget 的sticky options ， 摆放所有的slave widget 。

在第一步中，首先计算所有只占一行(或列)的cell 的宽度(或高度)，也就是rowspan=1(或colspan=1)的cell 。每一行(或列) 的宽度(或者高度)就是这一行(或列)中所需要的最大宽度(或者高度)，以保证放得下所有slave widget 。每一行，每一列的大小都知道了，这样每一个cell 的大小就确定了。

然后在考虑占多行(或者多列)的cell ，cell 所占的连续的行(或列)叫做一个cell group ，cell group 的宽度(或者高度)是组内所有cell 的宽度(或者高度)的和。如果cell group 够大，足够放的下相对应的slave widget ，那么最好了，维持组内的所有行(或列)的宽度(或高度)不变。如果放不下，就需要增加组内Cell 的每一行(或列)度(或高度)。增加的方法就是根据行(或列)的weight(权重) 按比例来增加cell group 中每一行(者列)的宽度(者高度)，weight 大的增加的多，weight 小的增加的小，weight 是0 的就不增。如果cell group 中所有的cell 的weight 都是0，那么就平均增加组内所有的cell 的宽度(或高度)。

依次下去，可以确定每一列(或行)的宽度(或高度)，这样每一个cell 和cell group 的大小就确定了。

设置一行的weight，可以使用grid_rowconfgure 函数，例如

```
m.grid_rowconfigure(0,weigtht=2)
```

把第0行的weight 设置为2 。

设置一列的weight，可以使用grid_columnconfgure 函数，例如

```
m.grid_columnfigure(0,weigtht=2)
```

把第0列的weight 设置为2 。其中m 是所有slave widget 的master widget。

在第二步中，如果master widget 非常大，还有很多的剩余空间，那么就会把垂直(或者水平)的剩余空间分配给所有的行(或者列)，根据每一行(或者列)的weight 的值按比例分配，weight 大的多得，weight 小的少得，weight 为0 的不得。如果所有行(或者列)的weight 都是0 ，那么就把垂直(或者水平)的剩余空间平均分配在上下(或者左右)两端，也就是居中放置。这样，用weight 的概念完成了Pack 管理器中的expand option 的功能。

综上所述，Grid 管理器中的option 的要比Pack 管理器中的option 要少得多。但是功能却比之更加灵活，更加强大，Pack 能做的，Grid 几乎都能做，Grid 能做的，Pack 需要放置很多的frame 和很多的option 才能完成。

#### Place 管理器

Place 可以精确的放置一个slave widget 的位置和大小，类似于VB 中的top，left，bottom，right 的属性。这样很麻烦，一般不用。如果你真的需要这么做，那么就查手册好了。至少我暂时不会用他。

## §4.5 Widget 的样式

由许多的widget 都支持同样的option，这些option 定义了widget 的基本的样式，包括颜色，字体等等。

### §4.5.1 颜色

大多数的widget 允许用户设置background option 和foreground option 来改变一个widget 的前景色和背景色。前景色一般是widget 上面文字的颜色。

bg 和fg 是background 和foreground 的简称。如果你打字比较慢或者比较懒，那么就用bg 和fg 代替background 和foreground。

在指定颜色的时候，可以直接指定RGB 三个分量的值，也可以指定颜色的名字，如red, green 等等。

#### 颜色的名字

Tkinter 含有一个颜色的数据库，用于包含所有颜色的名称和颜色实际RGB 值的对应关系。数据库中包含常用的颜色，如：red , yellow , blue , green , lightblue 等等，也包括一些不常用的但是很漂亮的颜色，如Moccasin, PeachPuff, etc.

在X windows 的系统中，颜色的名字由X windows 定义，可以在找到rgb.txt ，一般在/usr/X11R6/lib/X11/ 目录下这个文件中包含了所有颜色名字和颜色的RGB 的值的对应关系。

在Windows 系统下，可以使用Windows 的系统颜色(可以通过控制面板改变)

SystemActiveBorder, SystemActiveCaption, SystemAppWorkspace, SystemBackground, SystemButtonFace, SystemButtonHighlight, SystemButtonShadow, SystemButtonText, SystemCaptionText, SystemDisabledText, SystemHighlight, SystemHighlightText, SystemInactiveBorder, SystemInactiveCaption, SystemInactiveCaptionText, SystemMenu, SystemMenuText, SystemScrollbar, SystemWindow, SystemWindowFrame, SystemWindowText.

在Macintosh 中，可以使用下面的系统颜色:

SystemButtonFace, SystemButtonFrame, SystemButtonText, SystemHighlight, SystemHighlightText, SystemMenu, SystemMenuActive, SystemMenuActiveText, SystemMenuDisabled, SystemMenuText, SystemWindowBody.

颜色的名称是不分大小写的。而且有些颜色的名字可以在单词之间加一个空格。如”light blue” ”LightBlue” 是一个颜色。

RGB 颜色值

还可以指定RGB 三个分量的值，来指定颜色。格式是

```
#RGB,#RRGGBB, #RRRRGGGGBBBB
```

例如:

```
w["fg"]="#f00"
w["fg"]="#ff0000"
w["fg"]="#ffff00000000"
```

都是把前景色改成红色。RGB 三个分量都应该是十六进制。下面简单的例子，说明了如何设置RGB 分量。

```
tk_rgb = "#%02x%02x%02x" % (172, 182, 220)
tk_rgb = "#%04x%04x%04x" % (172, 182, 220)
```

`w.wininfo_rgb()` 可以把颜色名称翻译称为颜色的RGB 分量。如

```
rgb = widget.winfo_rgb("red")
red, green, blue = rgb[0]/256, rgb[1]/256, rgb[2]/256
```

注意`winfo_rgb` 返回的是范围是0 – 65535 ，如果要得到0 – 256 范围内的值，要除以256 才可以，也就是右移8 位。

### §4.5.2 字体

Tkinter 中的widget 允许用户指定字体。一般不需要指定字体，因为Tk 中大多数的简单的widget ，如Label ，Button 等等，都有很好的默认字体。

在指定字体的时候，用font option ，Tkinter 可以使用不同方式描述字体。

- 字体描述符(font descriptor )
- 用户自定义字体名称，使用tkFont 模块。
- 系统字体名称
- X 字体描述符( X font descriptor)

字体描述符(font descriptor)

从Tk 8.0 开始，Tkinter 可以使用独立于具体系统的字体描述，有助于跨平台编程。可以通过指定字体族，字体高度，和其他一些可选的字体风格(font style)来指定字体。如

```
("Times", 10, "bold")
("Helvetica", 10, "bold italic")
("Symbol", 8)
```

字体风格(font style) 是指字体是不是加粗，是不是斜体。

如果仅仅给出了字体族的名称，那么就会得到默认的字体高度，和字体风格。

如果字体族的名称中不含有空格，也可以仅仅给出来一个字符串用来描述字体。还可以增加字体高度和字体风格的定义。

```
"Times 10 bold"
"Helvetica 10 bold italic"
"Symbol 8"
```

在Windows 中常用的字体包括:

```
Arial (corresponds to Helvetica),
Courier New (Courier),
Comic Sans MS,
Fixedsys,
MS Sans Serif,
MS Serif,
Symbol,
System,
Times New Roman (Times),
and Verdana:
```

如果字体族名城中含有空格，那么就必须用第一种表达方式来表示字体。

常用的字体风格有: normal, bold, roman, italic, underline, 和overstrike.

在不同的平台上，Tk 8.0，自动把这些常用的字体，如，Courier, Helvetica, 和Times 映射到平台本身的字体表达方式上，而且，如果字体映射失败的话，也会找一个最贴近的字体来代替。所以在指定字体的时候Tk 8.0 从来都不会报告错误的。如果找不到最贴近的字体，那么就是用默认字体。但是Tk 认为最贴近的字体，有的时候并不是你想要的，所以不要过多的依靠这个特性。

Windows 下的Tk 4.2 也支持这种字体的表达方式。

#### 使用tkFont 的Font 类来指定字体

除了用字体描述符(font descriptor) 描述字体之外，还可以通过tkFont 来指定字体。Font 模块中的tkFont 类，可以用来指定字体。任何Tkinter 需要字体的参数，你都可以用tkFont 来指定。例如:

```
tkFont.Font(family="Times", size=10, weight=tkFont.BOLD)
tkFont.Font(family="Helvetica", size=10,
            weight=tkFont.BOLD,
            slant=tkFont.ITALIC)
tkFont.Font(family="Symbol", size=8)
```

`myfont=tkFont.Font(...)` 返回一个对象叫做`myfont` ，那么就可以用`w.config(font=myfont)` 来指定widget `w` 的字体是`myfont` 所描述的字体。这样有一个好处是，如果使用`myfont.config(size=40)` ，那么所有使用`myfont` 作为字体的widget 的字体都自动更新了，术语叫做font propagate 。很酷。这样在程序中，可以给一组类似的widget 指定字体`myfont` ，然后只要修改了`myfont` 那么这一组widget 的字体都会更新。

tkFont.Font 支持option 如表4.5-1:

| Option | 类型 | 描述 |
|---|---|---|
| family | string | 字体族名称 |
| size | integer | 字体大小，单位point ，如果是负数，单位是pixel |
| weight | constant | NORMAL 还是BOLD ，默认是NORMAL. |
| slant | constant | NORMAL 还是ITALIC ，默认是NORMAL. |
| underline | flag | 1 带下划线。0 不带下划线。 |
| overstrike | flag | 1 中间有横线，0中间不带横线。 |

表 4.5-1: tkFont 支持的option

### 系统字体名称

Tk 还支持系统定义的字体名称，在X Windows 下，一般是指字体别名，如fixed, 6x10 等等。

Windows 下，可以是ansi, ansifixed, device, oemfixed, system, and systemfixed。

在Macintosh 下，系统字体名称可以是application and system。

注意到系统字体名称是字体的全称，包括字体族名称和字体大小，字体风格。如果从可移植性考虑的话，最好不要使用这种方式。

X Font Descriptor

X Font Descriptor 是使用下面格式的字符串。星号表示用默认值。

```
-*-family-weight-slant-*--*-size-*-*-*-*-charset
```

family 表示字体族的名称。

weight 可以是Bold 或者Normal。

slant 可以是R 表示Roman ， 正常。或者I 表示斜体。或者O 表示oblique ， 斜体一样。

size 是表示字体大小，单位是point 的十分之一，通常72 point = 1 inch

charset 表示字符集，通常是ISO8859-1。

```
-*-Times-Bold-R-*--*-120-*-*-*-*-ISO8859-1
```

表示的字体是12 point 大小的，加粗的Times 字体，使用ISO Latin 1 编码的字符集。

如果你不关心某些值，那么就用星号代替。

一个典型的X Server 一般都支持Times, Helvetica, Courier, 字体，大小是8 , 10 , 12, 14, 18 , 24 . 可以是加粗，斜体。可以使用xlsfonts 和xfontsel Unix 命令来显示字体。

Windows 和Macintosh 也支持大部分这些字体，他们会把这种表达方式翻译成Windows 下或Macintosh 合适的字体。

### §4.5.3 文字格式化

一般来说，Label 和Button 不仅支持单行文字，还支持多行文字。多行文字就是用"\n" 分割的多个单行文字。默认情况下，文字是居中对齐的，可以改变justify option 为LEFT 或者RIGHT 表示左或者右对齐。

还可以设置wraplength option 来指定最大长度，文字的长度如果超过了这个最大长度就自动换行。

### §4.5.4 边框

尽管有些Widget的边框在默认的情况下是看不到的，但是所有的Widget 都有一个边框。边框包括3D 的Relief( 浮雕) 效果和Focus Highlight( 焦点高显)区域。

relief option 用来表示如何绘制一个Widget 的边框。可以是SUNKEN RAISED GROOVE RIDGE 和FLAT borderwidth 或者bd option 表示widget 的边框的大小，一般是2 或者1 ，太大了就不好看了。

Focus Highlight Region (焦点高亮区)也是一种边框，是在边框之外的另一个边框。在widget 得到输入焦点的时候，也就是widget 可以接受用户的键盘输入事件的时候，绘画这个Focus Highlight Region ，也就是在边框的外面再画一个边框，表示这个widget 得到了焦点，可以接受键盘的输入。

highlightcolor option 表示widget 得到焦点的时候，Focus Highlight Region 的颜色，一般是黑色。highlightbackground option 表示widget 没有得到焦点的时候，Focus Highlight Region 的颜色，一般和widget 一样的背景色。highlightthickness 表示Focus Highlight Region 的宽度。一般是1 或者2 ，太大了就不好看了。

这里有一个例子，用来说明边框和Focus Highlight Region 的用法。

```
# border1.py
```

```
from Tkinter import *
bd = range(1,9,3)
t = [ "border = " + str(k) for k in bd ]
relief= [ RAISED, SUNKEN,GROOVE ,RIDGE,FLAT]
for i in range(len(bd)):
    for j in range(len(relief)) :
        Button(text=t[i]+" " + relief[j],
               relief=relief[j],
               highlightcolor = 'blue',
               highlightbackground = 'white',
               highlightthickness = bd[i],
               bd=bd[i]).grid( row=j,
                               column=i,
                               padx=2,
                               sticky=W)
mainloop()
```

运行结果如图4-3。

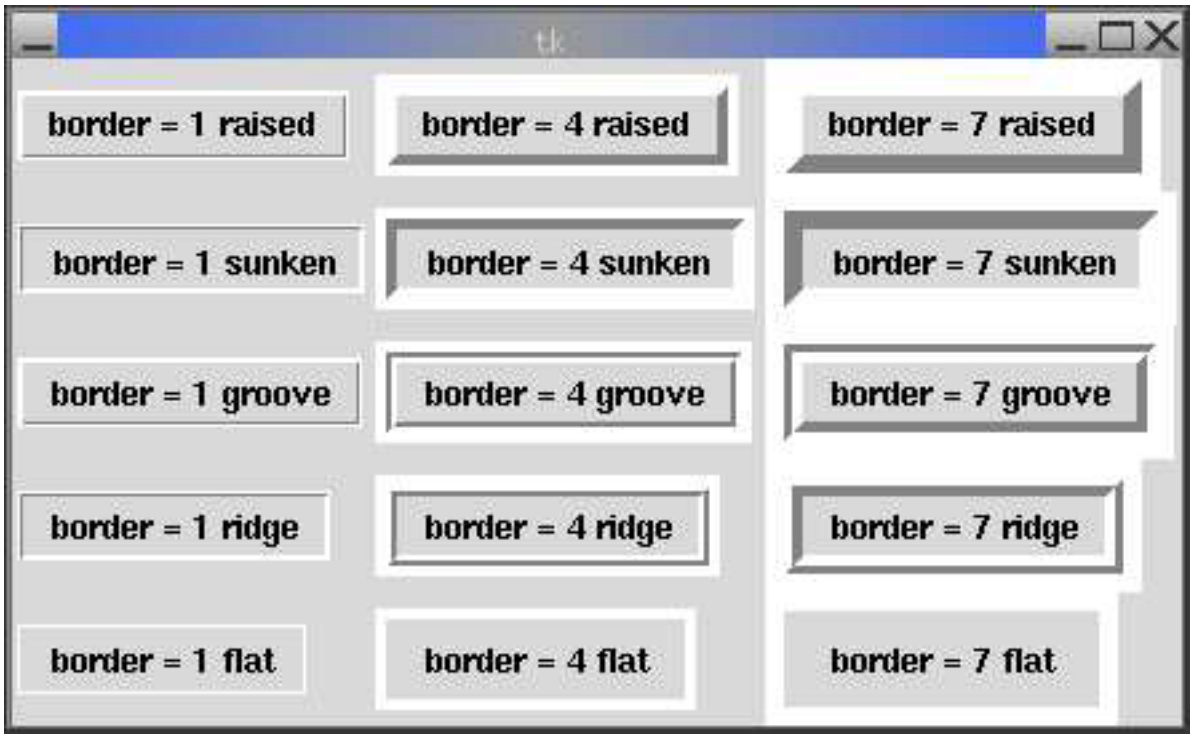


图 4-3: border1.py 的运行结果

highlightcolor 和highlightbackground 故意做成不是特别好看的颜色，让读者注意到他们的变化。

### §4.5.5 鼠标

cursor option 可以指定一个鼠标形状，当鼠标移动到这个widget 上的时候，显示这个的鼠标形状。如果不指定这个option ，那么当鼠标移动到widget 上的时候，显示和他的master widget 的cursor option 所指定的鼠标形状。

像Text 和Entry 这样的widget ，默认的改变了鼠标形状，表示需要输入文字。

下面的这个例子演示了cursor option 的效果。

```
#!/usr/bin/env python
cursors =["arrow", "based_arrow_down", "based_arrow_up", "boat",
"bogosity", "bottom_left_corner", "bottom_right_corner",
"bottom_side", "bottom_tee", "box_spiral", "center_ptr",
"circle", "clock", "coffee_mug", "cross", "cross_reverse",
"crosshair", "diamond_cross", "dot", "dotbox", "double_arrow",
"draft_large", "draft_small", "draped_box", "exchange", "fleur",
"gobbler", "gumby", "hand1", "hand2", "heart", "icon",
"iron_cross", "left_ptr", "left_side", "left_tee", "leftbutton",
"ll_angle", "lr_angle", "man", "middlebutton", "mouse",
"pencil", "pirate", "plus", "question_arrow", "right_ptr",
"right_side", "right_tee", "rightbutton", "rtl_logo",
"sailboat", "sb_down_arrow", "sb_h_double_arrow",
"sb_left_arrow", "sb_right_arrow", "sb_up_arrow",
"sb_v_double_arrow", "shuttle", "sizing", "spider", "spraycan",
"star", "target", "tcross", "top_left_arrow",
"top_left_corner", "top_right_corner", "top_side", "top_tee",
"trek", "ul_angle", "umbrella", "ur_angle", "watch", "xterm"]
from Tkinter import *
root = Tk()
col=0; row = 0
for i in cursors:
    b=Button(text=i)
    b.grid( column=col,row=row, sticky=E+S+W+N)
    b.bind("<Motion>",lambda e,cur=i:root.config(cursor=cur))
    col = col + 1
    if col == 4 : row = row + 1 ; col = 0
```

```
mainloop()
```

运行结果如图4-4。


图 4-4: cursor1.py 的运行结果

当鼠标移动到了每一个不同的按钮上，鼠标的形状就会变成不同的形状。

## §4.6 事件和事件的帮定

什么是事件，用户可能移动鼠标，或者按了一个键，这些都是事件，事件还包括WM要求widget 重画自己的事件。很多widget 都可以接受事件，并且调用相应的处理函数。将事件和是事件处理函数联系到一起的方法就是调用

```
w.bind(event,handler)
```

w 是一个widget, event 是一个字符串，用于表述事件，handler 是一个callable 的对象，用于处理事件，也就是说，在事件发生的时候，会调用这个对象，同时会传递一个参数，用于描述事件。

### §4.6.1 一个简单的例子

```
# File: bind1.py
from Tkinter import *
root = Tk()
def callback(event):
    print "clicked at", event.x, event.y
def test(e):
    D = vars(e);
    for k in D:
        print "%-10s:%s"%(k,D[k])
frame = Frame(root, width=100, height=100)
frame.bind("<Button-1>", callback)
root.bind("<Control-Alt-Key-colon>",test);
frame.pack(expand=1,fill=BOTH)
frame.focus_set()
root.mainloop()
```

这个程序把单击鼠标左键的事件和callback 这个函数绑定在一起，绑定的意思就是说当产生这个某个事件的时候，就调用相应的处理函数。

如果某个事件发生了，同时执行了相应的处理函数，我们称为捕获了这个事件。

运行这个程序，如果在Frame 上单击鼠标，就会在控制台上打印鼠标的坐标。如果同时按下Control ， Alt ， 和冒号，就会打印事件的所有信息。

#### 描述事件的方式

上一个例子中，”¡Button-1¿” 描述鼠标单击左键的事件，描述事件的格式如下:

```
<modifier-modifier-type-detail>
```

介绍鼠标的键定义在X windows 中一共有5 个鼠标键的定义，一般是

| | | | |
|---|---|---|---|
| 1 | 左键 | 2 | 中键 |
| 3 | 右键 | 4 | 滚轴上滚 |
| 5 | 滚轴下滚 | | |

一般只用1 , 2 , 3 三个键。

但是不完全是一成不变的，可以用xmodmap 命令改变。

```
xmodmap -pp
```

可以看到当前鼠标按键的定义。

在查看发生什么事件的时候，以下工具是很有用的。

| | |
|---|---|
| xmodmap | 查看和修改X 的键盘映射关系和鼠标映射关系。 |
| xev | 用来查看事件的。 |

modifier 表示发生鼠标或者键盘事件的时候，同时按下了其他键，如control ，alt ，win(super) 键等等，或者一个事件在短时间内重复多次。

modifier 可以是

| | |
|---|---|
| Control | Mod2, M2 |
| Shift | Mod3, M3 |
| Lock | Mod4, M4 |
| Button1, B1 | Mod5, M5 |
| Button2, B2 | Meta, M |
| Button3, B3 | Alt |
| Button4, B4 | Double |
| Button5, B5 | Triple |
| Mod1, M1 | Quadruple |

B1 表示发生键盘事件的时候，鼠标的左键是处于按下的状态。

Meta 和Alt 都是可以通过xmodmap 改变映射关系的。具体是哪一个键和X 系统的设置有关系。

可以指定多个modifier ，用减号分割。如

```
<Control-Alt-Key-colon>
```

其中Control ，Alt 是modifier ，Key 是type ，colon 是detail ，整个事件描述表示同时按下Control Alt 和冒号，也就是说同时按下Control Alt Shift 和分号，因为同时按下Shift 和分号，表示冒号。

Double 用于描述鼠标双击事件。

type 是最重要的，它定义了我们绑定哪一种事件。一般是ButtonPress ，ButtonRelease 或者是KeyPress ，KeyRelease ，也可以是其他的值，可以通过

```
man -S n bind
```

查看所有事件类型。

当type 是ButtonPress 和Button Release 的时候，detail 可以是1, 2, 3, 4, 5 用于表示鼠标键，如果不指定detail ，表示按任何键。

注意鼠标按键作为modifier 和type 的区别，也就是<Button1> 和<Button-1> 的区别。前者表示当某种事件发生的时候，鼠标的某个按键处在按下的状态。后者表示发生了鼠标按键的事件。

当type 是Motion 的时候，表示鼠标移动事件。一般指定B1 等modifier。如<B1-Motion> 表示鼠标左键按下并且移动鼠标。如果不指定modifier ，那么表示所有的鼠标移动事件。

如果type 是KeyPress 或者KeyRelease ，那么detail 用来指定到底是哪一个键盘按键。具体的值可以查看keysyms 的帮助。

```
man keysyms
```

keysyms 是一个可打印的ASCII 的字符，或者一个不可打印的ASCII 字符的描述，还可以是非ASCII 字符的描述。

两个有用的缩写。<1> <2> <3> 分别表示鼠标左键，中键，右键的按键的事件。<a> <b> <c> ... 表示按下a, b, c 等字符的事件。

事件描述举例，如表4.6-2：

### §4.6.2 事件处理函数

发生事件的时候，会调用事件的处理函数，传递一个参数event ，用于描述事件发生的时候的信息。用下面的代码可以打印所有信息。

```
def test(event):
    D = vars(event);
    for k in D:
            print "%-10s:%s"%(k,D[k])
```

具体参数如下表4.6-3

### §4.6.3 事件的层次

同一个事件可以有多个事件处理函数，这些处理函数属于不同的层次。每一个层次还可以有多个事件处理函数处理同一个事件。

可以在四个不同层次中绑定事件。

- widget instance 级
  指定某一个widget instance 的事件绑定。只有在事件发生在某一个指定的widget 的时候，才会捕获事件。绑定方法是w.bind(...) ，w 是一个widget instance 。

| | |
|---|---|
| <Button-1> 或<1> | 按下鼠标左键 |
| <B1-Motion> | 按下鼠标左键同时移动鼠标。 |
| <ButtonRelease-1> | 释放鼠标左键。 |
| <Double-Button-1> | 在很短的时间内(非常快速的)，鼠标移动也在很小的范围内(几乎没有移动)，连续按下两次鼠标左键。 |
| <Enter> | 鼠标移动到了Widget 的范围之内。注意不是按下了键盘的回车键，Enter 表示事件的type 。 |
| <Leave> | 鼠标离开Widget 的范围之内。 |
| <Return> | 按下了键盘的回车键。是一个简写，表示<KeyPress-Return> 。在keysyms 的帮助中可以查到几乎所有的非打印字符的描述。 |
| <Key> | 按下了任何的键盘。 |
| a | 表示按下了键盘的“a” 键。 |
| <Shift-Up> | 表示按住键盘的Shift 键不放的同时，按下了方向键中的上键。 |
| <Configure> | 表示Widget 的大小发生改变的时候的事件。 |

表 4.6-2: 事件描述举例

| | |
|---|---|
| widget | 产生事件的widget ，所有的事件event 都会设置这个属性。 |
| x,y | 表示发生事件的时候，鼠标在widget 中的相对位置。单位是pixel |
| x_root, y_root | 表示鼠标在屏幕中的相对位置。 |
| char | 类型是string ，表示和键盘有关的事件中，具体按下的是哪一个的字符码。 |
| keysym | 在和键盘有关的事件中，表示按键的文字描述。 |
| num | 在和鼠标有关的事件中，表示鼠标按键。 |
| wight, height | 表示widget 新的大小。只在configure 事件中有效。 |
| type | 表示事件的类型。 |

表 4.6-3: 描述事件的参数

- 窗口级
  widget 所属的Toplevel 对象或者root 对象，或者说窗口级，也绑定方法是`root.bind(...)` 或者`top.bind(...)` `root` 和`top` 分别表示Toplevel 对象和root 对象。
- widget class 级
  使用`bind_class` ，表示所有指定class 的instance 都会有一个事件绑定。
- 应用程序级
  任何地方发生事件，都会捕获。使用`bind_all` 。

例如使用`bind_all` 绑定F1 键，这样无论在任何地方，只要按了F1 键就会执行帮助程序。

但是如果同一个事件在不同层次上绑顶多次的时候，该怎么办呢? 或者说一个事件重叠的绑定的时候，该怎么办？什么是重叠绑定呢？例如，我同时绑定了`<Key>` 和`<Return>` 事件，一个表示按下任意键，一个表示按下回车键的事件，`<Return>` 事件同时也是一个`<Key>` 事件，这就是重叠绑定。

首先，一个事件发生的时候，Tkinter 在每一个的层次上，寻找一个最相近的事件绑定。例如，如果按下了回车键，`<Return>` 事件比`<Key>` 事件更近。`<Return>` 事件会被捕获，`<Key>` 事件则不会被捕获。

也就是说，重叠事件相排斥，最近的事件被捕获。

用类似数学的语言描述就是，每一个事件描述都一个事件的集合，集合中的元素是一个事件，如果一个集合A 是另一个集合B 的子集，那么说这个A 比B 小。当事件发生的时候，那么就在每一个层次上寻找包含该事件的最小集合，然后再每一个层次上捕获该事件。

如果在widget instance 层次绑定了`<key>` 事件，在toplevel 上绑定了`<Return>` 事件，那么当按下回车键的时候，这个事件在两个层次上都会被捕获。

这是因为这两个事件不是重叠事件，而是在不同层次上的事件，重叠事件是定义在同一层次上的概念，也就是说，重叠事件都是在同一层次上的。

Tkinter 首先在widget instance 的层次上捕获事件，执行事件处理函数，然后依次在widget class 的层次上，Toplevel 的层次上，应用程序级上，分别捕获事件。这个捕获事件的顺序很重要。

也就是说，不同层次上的事件按顺序捕获。

程序bind_level.py 很好的说明了这一点。

```
# File: bind_level.py
from Tkinter import *
root = Tk()
def callback(event):
```

```
    print "clicked at", event.x, event.y
def test(e):
    print "you press: " + e.keysym
    return
def application_level_bind(e):
    print "application level binding"
    test(e)
def class_level_bind(e):
    print "class level bind "
    test(e)
def toplevel_bind(e):
    print "top level bind"
    test(e)
def widget_level_bind(e):
    print "widget instance level bind"
    test(e)
def myprint(a):
    print a
frame = Frame(root, width=100, height=100)
root.bind("<Key>", toplevel_bind)
root.bind_class("Frame","<Key>",class_level_bind)
root.bind_all("<Key>",application_level_bind)
frame.bind("<Key>", widget_level_bind)
frame.bind("<Return>", lambda e: myprint("RETURN PRESSED") )
frame.pack(expand=1,fill=BOTH)
frame.focus_set()
root.mainloop()
```

### §4.6.4 同一个事件的多个处理函数

同一个事件可以有两个以上的事件处理函数。`bind(e,func)` 函数默认的动作是用指定的事件处理函数`func` 代替所有的原有的事件处理函数。但是如果指定第三个参数为“add”例如:

```
bind(e,func,"add")
```

就可以保留原来的事件处理函数，也就是在一个事件处理函数链的最后增加一个处理函数。当一个事件发生的时候，就会依此执行事件处理函数链上的每一个事件处理函数。如果哪一个函数返回了一个字符串“break” 那么就会停止执行后面的事件处理函数，同时也停止

调用其他层次的事件处理函数。

例子change_text_bind.py 说明了这个问题。

```
# change_text_bind.py
from Tkinter import *
def ignore (e):
    return "break"
def ignore2(e):
    print "you can only input one line"
root = Tk()
t = Text()
t.bind("<Return>",ignore2)
t.bind("<Return>",ignore,"add")
t.pack(fill=BOTH,expand=1)
root.mainloop()
```

这个例子想修改一个Text widget 默认的事件绑定，在默认的情况下，按下回车键表示输入新的一行，如果仅仅绑定ignore2 是不行的，尽管替代了instance 级别上的绑定关系，但是<Return> 事件的处理是class 级别上的，所以还不足以改变默认操作，用户还是可以输入新的一行。所以增加ignore 的绑定，让他切断事件处理函数的继续执行，这样才能达到目的。当然，在ignore2 中返回“break” 也是可以的。

如果事件处理函数链中有一个函数返回了“break” 那么再增加处理函数是没有意义的。

还有一种方法可以做到，

```
root.bind_class("Text","<Return",ignore)
```

但是这样做有一个缺点，你程序中所有的Text widget 都不能都输入新的一行了。解决的办法就是，创建一个新的类，继承Text ，然后在修改新类的事件绑定关系。

### §4.6.5 和WM 相关的事件绑定

当发生和WM 相关的事件的时候，例如程序被最小化(又叫图标化)，最大化，关闭，移动等等。最常用的事件就是WM_DELETE_WINDOW 表示用户使用WM 来关闭窗口的事件。

要绑定这一类事件的方法是

```
w.protocol("WM_DELETE_WINDOW",handler)
```

w 是一个widget 对象，handle 是事件处理函数，不传递任何参数。

看下面的例子:

```
from Tkinter import *
from tkMessageBox import askokcancel
```

```
def handler():
    if askokcancel("Quit","Do you really want to exit?"):
        root.destroy()

root=Tk()
root.protocol("WM_DELETE_WINDOW",handler)
root.mainloop()
```

一般情况下是不用这样绑定的，因为，每一个窗口处理WM_DELETE_WINDOW 事件的默认动作就是调用destroy 函数来关闭自己。

其他的类似WM_DELETE_WINDOW 的事件可以查看ICCCM(Inter-Client Communication Conventions Manual) 的文档，

## §4.7 常用的应用程序使用的Widget

### §4.7.1 基本窗口

我们已经用过了，root=Tk() 就会创建一个root window ，这个窗口被关闭了，那么就关闭所有的其它窗口，程序退出。

创建一个Toplevel 对象，就可以创建其他的窗口，这些窗口关闭，不会退出程序，其它方面Toplevel 和root window 是十分类似的。

Toplevel 不像其他的Widget ，不用pack 或者grid ，他就可以显示出来，因为它是一个窗口，WM 会显示他的。

每个窗口的默认标题是“Tk” ，改变窗口的标题用

```
root.Title("My title")
```

得到一个窗口的标题的方法是

```
print root.Title()
```

### §4.7.2 菜单

例子menubar2.py 制作菜单栏的一个例子。

```
# menubar2.py
from Tkinter import *
import sys

root = Tk()
```

```
mainmenu=Menu(root)
root["menu"]=mainmenu

filemenu = Menu(mainmenu)
mainmenu.add_cascade(label='file',menu=filemenu)
m = [ "new","open","save"]
for i in m :
    filemenu.add_command( label=i,
          command=lambda t = i: sys.stdout.write(t+"\n") )
filemenu.add_command(label="exit", command= root.quit)
mainloop()
```

运行结果如图4-5 注意，事件处理函数的小技巧，

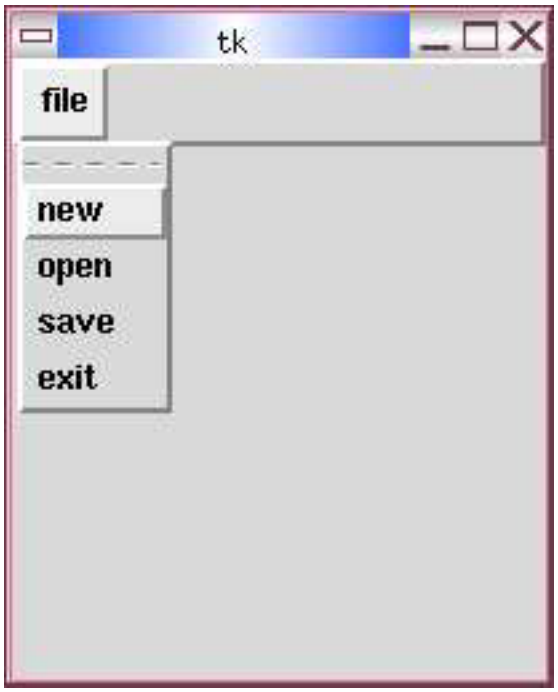


图 4-5: menubar2.py 的运行结果

```
lambda : sys.stdout.write(i)
```

是不行的。

因为每一个菜单的处理函数都是一样的，都是打印i 的值，什么时候你改变了i 的值，打印的结果就会变。

```
lambda t = i : sys.stdout.write(i)
```

则不同，每一个菜单的处理函数是不一样的，他们的默认参数不同，默认参数的赋值是在函数创建的时候。

可以看到lambda 创建函数的方便之处，可以用这种办法，为某些事件处理函数增加默认参数。这样一个事件处理函数可以处理多个widget产生的事件，减少函数的数量，可以简化设计，方便维护。

能用更简单更少的语句完成同样的功能，这样的程序才是好程序。

还有一种做法，使用Menubutton

例子menubar3.py 制作菜单栏的一个例子。

```
# menubar3.py
from Tkinter import *
import sys
root = Tk()
# create a frame to contain the menu bar
menuframe = Frame()
# create a menu button
filemenuButton = Menubutton(text="File")
filemenuButton.pack(side=LEFT,anchor=N)
# create the "file" menu
filemenu = Menu(filemenuButton)
# connect the filemenu with hte filemenuButton
filemenuButton['menu']=filemenu
m = [ "new","open","save"]
for i in m :
    filemenu.add_command(label=i,
            command= lambda t = i: sys.stdout.write(t+"\n") )
filemenu.add_command(label="exit", command= root.quit)
menuframe.pack(side=TOP,fill=X,expand=1)
mainloop()
```

运行结果如图4-6

两个例子还是有区别的。Menubutton 是一个和Button 类似的东西，但是他默认的command 动作是弹除一个菜单。

使用Menubutton 有以下几个步骤:

1. 建立Menubutton.

```
mb = Menubutton()
```

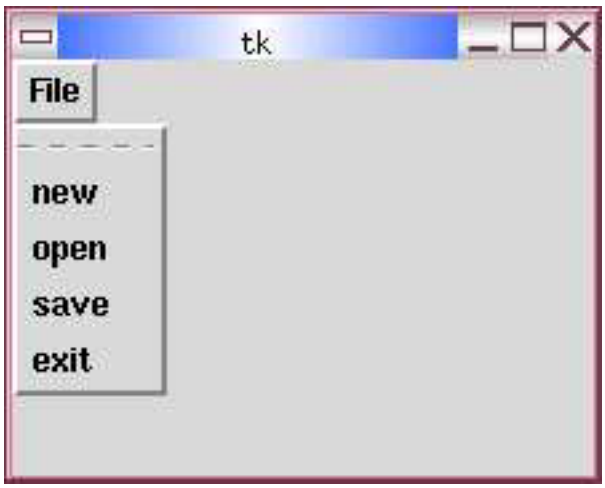


图 4-6: menubar3.py 的运行结果

2. 在他上面显示一个好的内容，提示用户。

```
mb["text"]="press me"
```

3. 创建一个Menu ， 他的master widget 是mb

```
m = Menu(mb)
```

4. 把创建的Menu 和Menubutton 联系在一起。

```
mb["menu"] = m
```

5. 增加menu 的内容

```
m.add_command(label="new",
    command=lambda : sys.stdout.write("Hello Menu\n")
```

6. 显示Menubutton

```
mb.pack()
```

如果要在menu 上增加一个分割线，用

```
m.add_separator()
```

如果要在menu 上增加一个子菜单，用

```
submenu = Menu(m) # 注意 master widget 一定要是父菜单。
m.add_cascade(menu=submenu,label="a submenu")
```

### §4.7.3 工具栏

很多应用程序都有一个工具栏。其实一个工具栏就是一个Frame ，其中包含了很多的Button。

请看例子toolbar.py 。

```
# toolbar1.py
from Tkinter import *
root = Tk()
def callback():
    print "called the callback!"
x = ["error",
"gray12",
"gray25",
"gray50",
"gray75",
"hourglass",
"info",
"question",
"warning"]
# create a toolbar
toolbar = Frame(root)
for i in x :
    b = Button(toolbar, text=i, bitmap=i,
               command=lambda t=i: sys.stdout.write("%s\n"%t) )
    b.pack(side=LEFT, padx=2, pady=2,fill=BOTH,expand=1)
toolbar.pack(side=TOP, fill=X)
mainloop()
```

运行结果如图4-7

每一个Button 都上面有一个图标，放置的方式是：靠左放，自动改变大小，改变的大小的时候在X Y 方向上都跟着一起改变Button 的大小，上下左右都留出来2 个像素的空隙。

toolbar 的放置方式是: 靠上放，在X 方向上自动改变大小。

除了使用bitmap otption 放置图标之外，还可以改变image option 来显示图标，这需要使用PhotoImage 创建一个图像。

### §4.7.4 状态栏

很多程序的下方都有一个状态栏，其实也是一个Frame 其中含有一个或者多个Label ，

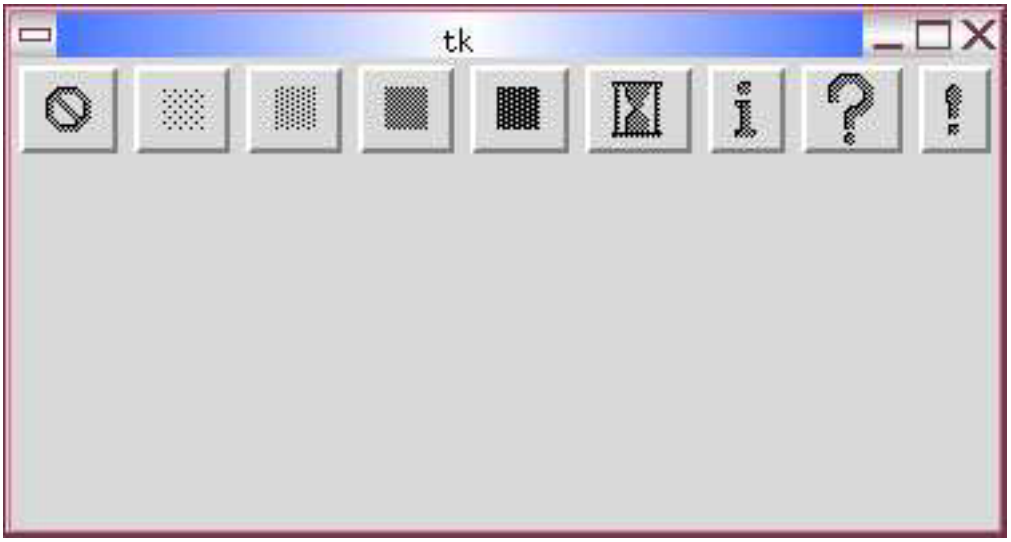


图 4-7: toolbar1.py 的运行结果

可以自己写一个程序，也不难。

请看例子statusbar.py 。

```
# toolbar1.py
from Tkinter import *
root = Tk()
def callback():
    print "called the callback!"
x = ["error",
"gray12",
"gray25",
"gray50",
"gray75",
"hourglass",
"info",
"question",
"warning"]
# create a toolbar
toolbar = Frame(root)
for i in x :
    b = Button(toolbar, text=i, bitmap=i,
               command=lambda t=i: sys.stdout.write("%s\n"%t) )
```

```
    b.pack(side=LEFT, padx=2, pady=2,fill=BOTH,expand=1)
toolbar.pack(side=TOP, fill=X)
mainloop()
```

使用的tkSimpleStatus.py 。

```
# File: tkSimpleStatusBar.py
from Tkinter import *
class StatusBar(Frame):

    def __init__(self, master):
        Frame.__init__(self, master)
        self.label = Label(self, bd=1, relief=SUNKEN, anchor=W)
        self.label.pack(fill=X)

    def set(self, format):
        self.label.config(text=format)
        self.label.update_idletasks()

    def clear(self):
        self.label.config(text="")
        self.label.update_idletasks()
```

运行结果如图4-8

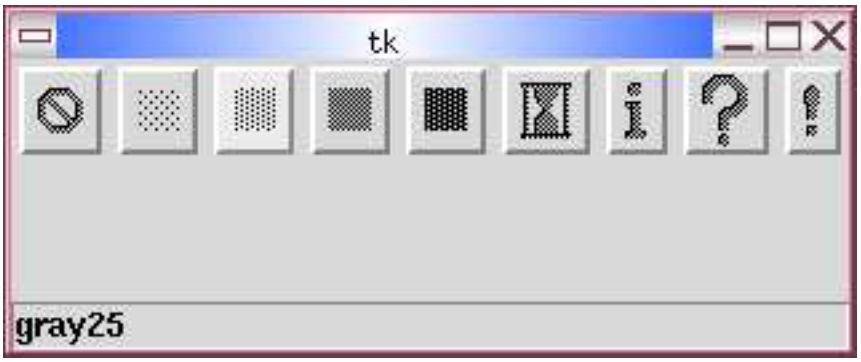


图 4-8: statusbar1.py 的运行结果

这个例子中只有一个显示栏，很容易增加一个显示栏，无非就是增加一个Label 。

### §4.7.5 标准对话框

如果就是得到简单的输入信息，如一个数字，一个字符串，回答是还是不是，得到一个

文件名称，得到一个字体名称，得到一个颜色等等，有现成的简单标准输入框。很好用。

### 简单的消息框

tkMessageBox 模块提供了很多函数，用来创建一个简单的消息框。

但是要注意，不能滥用消息框，Windows 下的程序就经常弹出一个消息框，问我要不要保存，要不要真的退出，完全把用户当成傻瓜，一个好的的办法是第一次弹出来对话框，用户知道了应该小心，同时用户可以选择以后不要再弹出来这样的对话框了。

可以看一下tkMessageBox.py 的源程序，短的吓人，没想到这么短的程序可以完成这么多的事情，尽管大多数事情都是Tk 做的。

可以运行

```
python  /usr/lib/python2.2/lib-tk/tkMessageBox.py
```

看到他的测试程序。

这个模块提供了一下函数:

`showinfo, showwarning, showerror,`
`askquestion, askokcancel, askyesno,` 和
`askretryignore.`

他们的调用方法都一样。

```
tkMessageBox.function(title, message [, options]).
```

title 是标题。message 是要显示给用户的信息。

option 有以下的选择:

| | |
|---|---|
| default | 可以是ABORT, RETRY, IGNORE, OK, CANCEL, YES, 或者NO 表示默认的条件下，哪一个按钮得到焦点。 |
| icon | 表示使用什么样的标准图标。可以是ERROR, INFO, QUESTION, WARNING |
| message | 和第二个参数message 是一个意思。 |
| parent | 表示对话框结束之后，哪一个窗口变成了当前窗口。 |
| title | 和第一个参数title 是一个意思。 |
| type | 显示哪一类按钮，可以是ABORTRETRYIGNORE, OK, OKCANCEL, RETRYCANCEL, YESNO, YESNOCANCEL。从源代码可以看到，每一个函数已经默认的写好了type ，这样的好处是提高程序的可读性。 |

### 简单数据输入

在tkSimpleDialg 模块中，提供了一个输入简单数据的简单对话框。

```
tkSimpleDialog.askstring(title, prompt,[options])
```

可以返回一个字符串，如按了Cancel 按钮，那么返回None

其中：

| | |
|---|---|
| title | 表示窗口的标题。 |
| prompt | 表示提示用户输入信息的文字。 |
| options | 可以是以下选择。 |
| initialvalue | 表示初始值是什么。 |
| parent | 表示用户输入完毕后，显示哪一个窗口。 |

还有类似的tkSimpleDialog.askinteger 和tkSimpleDialog.askfloat 用法和上面的一样，如果用户按了Cancel ，那么返回None 这两个函数还有两个option 可以选择。minvalue 和maxvalue 分别表示最大最小值，用于限制用户在一定范围内输入数值。

### 打开文件的对话框

tkFileDialog 模块提供了两个函数:

```
tkFileDialog.askopenfilename
tkFileDialog.asksavefilename
```

可以返回一个文件对象，如果用户按了Cancel 那么返回None 。

使用方法是:

```
tkFileDialog.askopenfilename(options)
tkFileDialog.asksavefilename(options)
```

可以使用的option 有:

| 名称 | 类型 | 描述 |
| --- | --- | --- |
| defaultextension | string | 如果用户没有给出文件的扩展名，那么就自动加上这个扩展名称。 |
| filetypes | list | list 中的每一个元素是一个(label,pattern) 的tuple 。前面的label 用于提示用户是什么样的类型的文件，pattern 用于匹配文件名称。 |
| initialdir | string | 初始目录。 |
| initialfile | string | 默认文件名称。 |
| parent | widget | 同上。 |
| title | string | 窗口标题。 |

例如:

```
openfile(filetypes=[("text file","*.txt")])
```

打开一个对话框，提示用户打开文本文件，也就是以txt 结尾的文件。

### 选择颜色的对话框

tkColorChooser 模块提供了一个askcolor 的函数用于提示用户输入一个颜色值。使用方法是:

```
askcolor([color[,options]])
```

可用的options 如下:

| | |
| --- | --- |
| color | 表示一个颜色值，可以是一个字符串，遵循颜色定义的格式，必须是“#rgb” “#rrggbb” “#rrrrggggbbbb”的格式。也可以是一个tuple 其中含有三个元素，分别表示RGB 的三个不同分量。这个颜色值和第一个参数是一个意思，指明了初始颜色值。 |
| parent | 同上。 |
| title | 窗口标题。 |

返回值是一个tuple，其中有两个元素，第一个是用tuple 表达颜色的方式，第二个是用字符串表达颜色的方式。

### 自定义对话框

x 当以上对话框不满足要求的时候，可以使用tkSimpleDialog 模块中提供的Dialog 类。可以查看tkSimpleDialog.py 的源代码，也不是特别复杂。

使用方法很简单。Dialog 类继承了Toplevel ， 所以他也是一个窗口对象。

看下面的例子inputpassword.py

```
#inputpassword.py
from Tkinter import *
from tkSimpleDialog import *

class DlgInputPassword(Dialog):
    def body(self,master):
        Label(master,text="user name:").grid(row=0,column=0)
        Label(master,text="password :").grid(row=1,column=0)
        self.username = Entry(master)
        self.username.grid(row=0,column=1)
        self.password = Entry(master)
        self.password.grid(row=1,column=1)
    def apply(self):
        self.result = self.username.get() , self.password.get()
    def validate(self):
        return 0
root = Tk()
d=DlgInputPassword(root)

print d.result
```

运行结果如图4-9

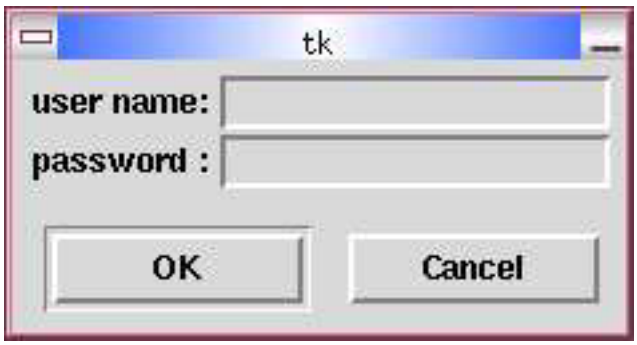


图 4-9: inputpassword.py 的运行结果

其中重写了body 函数，和apply 函数。有了这两个函数就可以完成一般功能的操作了。body 函数用来创建对话框中的每一个widget ， 但是不包含OK 和Cancel 按钮。

apply 函数用于返回有用的值，一般设置self.result 属性，用于返回信息。

在用户按了OK 按钮的时候，就会调用apply 函数设置self.result 。如果按了Cancel 按钮，self.result 就是None 。

还有一个函数validate()可以重写。没有任何参数。validate() 这个函数用来判断用户输入的信息是否满足要求，当用户按了OK 按钮的时候，就会自动调用这个函数，如果这个函数返回1 ，表示用户输入是满足逻辑要求的，就调用apply 设置self.result 然后返回。如果这个函数返回0，按了OK 也不起作用。最好给用户一个提示，告诉用户为什么不能正常的退出。

还有一个buttonbox 函数可以重写。也是没有多余的参数。这个函数的默认动作是创建两个按钮，一个是OK 一个是Cancel 。改写这个函数可以创建用户自己想要的按钮形式，而默认的两个按钮。同时，把OK 按钮的处理函数和self.ok 绑定，Cancel 按钮的处理函数和self.cancel 绑定。self.ok 和self.cancel 是两个已经写好的默认函数。

查看tkSimpleDialog.py 会得到更多的信息。

# 第五章

# Python的扩展

## §5.1 用C编写扩展模块

用C 语言编写扩展是很大的一个题目。先举个例子。

```
$ls                    #当前目录只有一个 demo.c
spammodule.c
$cat spammodule.c
#include <Python.h> /*必须要加入的头文件*/
static PyObject *   /*所有的Python对象都是这个类型.*/
spam_system(PyObject * self, PyObject * args)
{
    char *command;
    int sts;
    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
static PyMethodDef
SpamMethods[] = {
    {"system", spam_system, METH_VARARGS,
     "Execute a shell command."},
    {NULL, NULL, 0, NULL}       /* 表示表格的结束 */
};
void initspam(void)
{
```

```
    Py_InitModule("spam", SpamMethods);
}
```

我们希望建立一个module 名字叫作spam，我可以通过spam.system(cmd) 的方式，完成和os.system类似的工作。

首先分析一下这个程序，一共有两个函数，initspam(...) ，spam_system(...) ，一个全局变量，SpamMethords 。initspam是模块初始化函数，当import的时候会调用，它的是唯一的非静态函数。名字是init + module_name的形式，我们的module 叫做spam。他相当于main 函数。他的主要任务是通知python 解释器，我们的spam 中有多少个函数。Py_InitModule 就是干这个事情的。第一个参数是"spam" ，表示module 的名称。第二个参数是SpamMethords 是一个表格，通知python ，我们的spam 有多少个函数。SpamMethors 一定要是全局的静态变量，类型是PyMethodDef。

PyMethordDef 以全空表示表格的结束。每一行有四个字段：

- 第一个表示python 中函数的名字，这里是 system，在python 中可以通过spam.system 来调用这个函数。
- 第二个字段是一个函数指针，spam_system，这个函数的原型一定要是

  ```
  static PyObject *
  spam_system(PyObject * self, PyObject * args);
  ```

  为了和METH_VARARGS相对应。
- 第三个字段是doc string，在python 中可以通过spam.system.__doc__ 得到。
- 最后一字段是一个宏(macro) ，表示函数类型。从C语言的观点来看，Python 中的函数有两种，一种是常用的用tuple 来传递参数，一般是METH_VARARGS表示，一种是用dictionary 来传递参数，叫做 keyword parameter，用METH_VARARGS|METH_KEYWORDS表示。这两个Macro 就分别对应这两种函数。两种方式在C 语言中对应的函数原型是不一样的。

看看我们真正干活的函数spam_system， 他的返回值表示Python 中函数的返回值，“一切都是对象” 这句话在C 语言中的表现形式就是PyObject* ，一切都是PyObject*。如果出错，那么在C 语言中有函数，可以用来抛出异常，返回值就是NULL 。

spam_system的参数有两个，看到他们的类型是PyObject*就一点也不奇怪了吧。

- self 一般不用，如果是不同函数的话，他总是NULL，如果他是一个class 的methord 的话，self 表示class 的instance。
- args表示传进来的参数，是一个tuple ，注意tuple 在C 内部也是PyObject*

PyArg_ParseTuple用来得到参数。其中args 是传进来的参数。"s"表示格式转换的类型字符串，s表示string 类型，还有其它类型。后面的参数就要和"s"所指定的类型相适应。和"s"的相适应的类型是char**，是一个字符指针的指针。也就是说，不用我们自己来编写内存管理的函数，如果Python库系统函数PyArg_ParseTuple正常返回，那么command就指向一个有效的字符串。如果是失败，那么就返回NULL。当PyArg_ParseTuple 失败的时候，他会自动抛出异常的，不用我们自己在编写抛出异常的代码。这也看出异常机制的好处。

下面就是调用system ，这些和C 语言编程没有什么两样。我们把system的返回值 sts 作为我们函数的返回值。Py_BuildValue()是用来创建一个python 对象用的，返回一个PyObject* ，他的第一个参数用来指明对象的类型。"i"表示integer。后面的参数和参数类型相一致。否则会抛出异常，返回NULL。

我发现扩展Python 比用Python 编程需要难得多，就像编写驱动程序要比普通的应用程序难，因为不但要对系统调用了解，还要对操作系统的内部有些了解。扩展Python 也是一样，需要对Python 的实现机制有所了解。好了，现在我们要生成我们的module 了。用C 语言写的module 是有两种办法加到Python 中。一种是把直接和Python 的源代码一起编译。在Unix平台下，很简单，只要在Modules/Setup.local中加上一行，

```
spam spammodule.o
```

然后，把spammodule。c 拷贝到Modules 目录下面就可以了。重新编译Python 。这个方法有显而易见的缺点，重新编译Python 可没有什么好玩的。

第二个方法，把C 程序编译成为动态连接库。但是有个问题，不同平台下对动态连接库的支持方式是不一样的，区别很大。幸好，我们有一个distutils 的package 可以帮我们做这件事情。编写一个setup.py 的文件，文件内容如下。

```
$ls
setup.py  spammodule.c
$cat setup.py
from distutils.core import setup, Extension
module1 = Extension('spam',
                    sources = ['spammodule.c'])
setup (name = 'spam package',
    version = '1.0',
    description = 'This is a spam package',
    ext_modules = [module1])
```

具体的意思，可以参考distulis 的手册。

```
$python setup.py build      #编译
running build
running build_ext
```

```
building 'spam' extension
creating build
creating build/temp.linux-i686-2.2
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC \
-I/usr/include/python2.2 -c spammodule.c           \
-o build/temp.linux-i686-2.2/spammodule.o
creating build/lib.linux-i686-2.2
gcc -shared build/temp.linux-i686-2.2/spammodule.o \
-o build/lib.linux-i686-2.2/spam.so
$
$ls             #多了一个build 的子目录
build  setup.py  spammodule.c
$cp build/lib.linux-i686-2.2/spam.so . #拷贝到当前目录
$python                                #运行python 解释器
Python 2.2 (#1, Apr 12 2002, 15:29:57)
[GCC 2.96 20000731 (Red Hat Linux 7.2 2.96-109)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import spam                        #引入我们的 module
>>> print spam.system.__doc__          #查看 DOC string
Execute a shell command.
>>> print spam.system("echo 'hello'")  #看看成功了吗?
hello                                    #ok,正常输出
0                                 # spam.system 的返回值
>>>
```