PCS108　函式

概述

函式是完成特定功能的一些语句块，可以作为一个独立单位使用，若再给它取一个名字，就可以通过这个名字在程序的不同地方多次调用，这样就不须要重复地编写这些语句了。在 Python 语言中，函式也是对象。

应用

函式定义

在 Python 中，通过 def 关键字定义函式。def 关键字后跟一个函式名，然后跟一对圆括号，圆括号之中可以包括一些变量名，该行以冒号结尾。接下来，是一块语句，称为函式体。例如：

```
In [1]: def add(a, b):
   ...:     """返回 a 和 b 的和"""
   ...:     return a+b
   ...:
```

这样就定义了函式 add，接受两个参数：a 和 b，接下来是文档字符串，加入一些关于该函式功能的描述，函式体只有一个 return 语句返回结果，通过函式名来调用它：

```
In [3]: add(1, 2)
Out[3]: 3
```

函式参数

参数可以默认，即当调用时没有指定某个参数值，就可用该参数的默认值。

```
In [4]: def add(a, b=1):
   ...:     """返回 a 和 b 的和"""
   ...:     return a+b
   ...:

In [6]: add(5)
Out[6]: 6
```

在 Python 语言中，类似于参数 b 的称为关键字参数，而类似于参数 a 的称为位置参数，因为它是通过所处的位置来进行参数匹配的。

```
In [6]: add(5)
Out[6]: 6

In [7]: add(5, 2)
Out[7]: 7

In [8]: add(5, b=3)
Out[8]: 8
```

传递关键字参数时无须考虑顺序，而位置参数，则在要传参数的时候须按照参数定义的位置一一对应。也可以以关键字参数的形式传递位置参数，还可以以位置参数的形式传递关键字参数（不推荐，因为这样很容易导致混乱），唯一的规则就是关键字参数一定要在位置参数的右边，并且在指定参数时必须保证其右边的参数值已经指定，否则会引起歧义。

传递参数时，可以使用*和**。*的作用是把序列 args 中的每个元素当作位置参数传进去。**的作用则是把字典 kwargs 变成关键字参数传递。

```
In [9]: def add(*args):
   ...:     print args
   ...:

In [10]: add(1, 2, 3)
(1, 2, 3)

In [11]: def add(**args):
   ...:     print args
   ...:

In [12]: add(a=1, b=2, c=3)
```

```
{'a': 1, 'c': 3, 'b': 2}
```

普通的参数定义和传递方式与 * 可以和平共处，但 * 必须放在所有位置参数的最后，而**必须放在所有关键字参数的最后，否则就要产生歧义。

```
In [13]: def add(a=1,**arg):
   ....:     print a
   ....:     print arg

In [14]: add(a=1,b=2,c=3)
1
{'c': 3, 'b': 2}
```

函式返回

每一个函式都有返回值，即 return 后面的值。若函式中没有 return，则该函式返回为 None。

lambda 函式

lambda 函式是匿名函式，用来定义没有名字的函式对象。在 Python 中，lambda 只能包含表达式！

```
lambda arg1, arg2 ... : expression
```

lambda 关键字后就是逗号分隔的形参列表，冒号后面是一个表达式，表达式求值的结果为 lambda 的返回值。虽然 lambda 的滥用会严重影响代码可读性，不过在适当的时候使用一下 lambda 来减少键盘的敲击还是有其实际意义的，比如做排序的时候，使用 data.sort(key=lambda o:o.year)显然比

```
def get_year(o):
   return o.year
func=get_year(o)
data.sort(key=func)
```

要方便许多！

闭包(closure)

闭包其实就是通常所说的函式嵌套。在嵌套函式中内部函式对象本身包含了外部函式对象的名称空间。

```
In [15]: def Outter(n):
   ....:     def inner(m):
   ....:         return n+m
   ....:     return inner
   ....:

In [16]: Outter(1)(2)
```

```
Out[16]: 3

In [17]: Outter(2)(2)
Out[17]: 4
```

装饰器(decorator)

```
@log
def test(a, b):
    pass
```

其中 log 是接受一个函式作为参数同时返回一个新函式的函式，其作用是在调用 test 的前后分别输出 enter test 和 exit test，使用符号@来应用这个装饰器。

```
def log(func):
    def wrapper(*args, **kw):
        print 'enter', func.__name__
        func(*args, **kw)
        print 'exit', func.__name__
    wrapper.__name__ = func.__name__
    wrapper.__globals__ = func.__globals__
    return wrapper
```

log 函式定义另一个叫 wrapper 的嵌套函式，它把所有接受的参数简单地全部传给 func，并在调用前后输出一些信息。最后在对 wrapper 的一些属性进行偷梁换柱之后，就将它返回了，于是这个 wrapper 就变成了一个被包装过的如假包换的 func 了。

生成器(generator)

```
>>> def number_generator():
...     i = 0
...     while True:
...         yield i
...         i += 1
...
>>> for item in number_generator():
...     print item
...
0
1
2
# 省略后面输出的无穷个数字 Ctrl+c 停止
```

Python 的生成器可以有两种讲法，简单讲，它就是方便地实现迭代器的方法，就像上面代码使用的那样。复杂来讲呢，Python 的生成器其实正是一个神秘的 continuation，实际

上大部分时候我们都只需要把 yield 当成是快速实现迭代器的工具来用就行了。神秘的 continuation 已经超出了本书的范畴。

小结

和许多语言不同，在 Python 中，函式是一等公民，函式也是对象，这也是在 Python 中进行编程的基础，理解这一点非常重要。越深入学习 Python，就越能感受到函式在 Python 中的重要地位。