Python函数

- 函数的定义与调用
- 调用函数的形式
- 函数的参数
- 局部变量和全局变量
- 函数的注释说明
- 常用函数

函数的定义与调用

- 函数是一个能完成特定功能的代码块，可在程序中重复使用，减少程序的代码量和提高程序的执行效率。
- 在python中函数定义语法如下：
  - def function_name(arg1,arg2[,...]):
    - statement
    - [return value]
- 返回值不是必须的，如果没有return语句，则Python默认返回值None

函数的定义与调用

- 定义函数，通常使用def语句。
- 函数名可以是任何有效的Python标识符。
- 参数列表可以由多个、一个或零个参数组成。
- 圆括号是必不可少的，即使没有参数也不能没有它；不要忘记圆括号后面的冒号。
- 函数体一定要注意缩进。
- “形参”和“实参”。
- return语句的作用是结束函数调用，可以出现在函数体的任意位置。

```
def 函数名（参数列表）：
    函数体
```

```
def add1(x):
    x = x + 1
    return x
```

```
>>>def add1(x):
        x = x + 1
        return x


>>>add1(1)
2
```

调用函数的形式

- 调用函数的一般形式是：

函数名（参数表）

- 上例：

```
add1（1）
```

- 对于没有使用return语句的函数，它实际上也向调用者返回一个值，那就是None。

```
>>>def myadd():
        sum=1+1

>>>a=myadd()

>>>a

>>>print a
None
```

- 标准调用方式，传递的值按照形参定义的顺序相应地赋给它们。

调用函数的形式

- “关键字调用”方式，即在调用函数时同时给出形式参数和实际参数。
- “关键字调用”方式在函数具有多个参数是非常有用，因为解释器能通过给出的关键字来匹配参数的值，所以这样就允许参数缺失或者不按定义函数时的形式参数的顺序提供实际参数。

```
def select(x, y):
    让x年级y班的学生打扫卫生
```

```
select(3, 6)
```

```
select(6, 3)
```

```
select(x=3, y=6)
```

```
select(y=6,x=3 )
```

函数的参数

- 在定义函数时，我们可以用赋值符号给某些形参指定默认值，这样当调用该函数的时候，如果调用方没有为该参数提供值的话，则使用默认值。
- 如果调用该函数的时候为该参数提供了值的话，则使用调用方提供的值——像这样的参数我们称之为缺省参数。
- 默认参数必须在所有标准参数之后定义。

```
def f(arg1,arg2=2,arg3=3):

    print 'arg1 = ', arg1

    print 'arg2 = ', arg2

    print 'arg3 = ', arg3
```

函数的参数

- 带有缺省参数的函数：

```
>>>def f(arg1,arg2=2,arg3=3):
        print 'arg1 = ', arg1
        print 'arg2 = ', arg2
        print 'arg3 = ', arg3

>>>f(10)
arg1 =  10
arg2 =  2
arg3 =  3

>>>f(10,10)
arg1 =  10
arg2 =  10
arg3 =  3

>>>f(10,10,10)
arg1 =  10
arg2 =  10
arg3 =  10
```

- 用“关键字调用”方式调用带有缺省参数的函数：

```
>>>f(10,arg3=10)
arg1 =  10
arg2 =  2
arg3 =  10

>>>f(arg3=10,arg1=10)
arg1 =  10
arg2 =  2
arg3 =  10

>>>f(10,arg2=10)
arg1 =  10
arg2 =  10
arg3 =  3

>>>f(arg2=8,arg1=10)
arg1 =  10
arg2 =  8
arg3 =  3
```

函数的参数

- 需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数.

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

- 加了星号（*）的变量名会存放所有未命名的变量参数。选择不多传参数也可。

函数的参数

```
>>># 可写函数说明
def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print "输出: "
   print arg1
   for var in vartuple:
      print var
   return

>>>printinfo( 10 )        # 调用printinfo 函数
输出:
10

>>>printinfo( 70, 60, 50 )        #以上实例输出结果:
输出:
70
60
50
```

函数的参数

- 在一个函数中对参数名赋值不影响调用者。
- 在一个函数中改变一个可变的对象参数会影响调用者，如列表，字典，数组等。

```
>>> a=1
>>> def test(a):
        a=a+1
        print a
>>> test(a)
2
>>> a
 1 # a值不变
```

```
>>> a=1
>>> b=[1,2]
>>> def test(a,b):
        a=5
        b[0]=4
        print a,b
>>> test(a,b)
 5 [4, 2]
>>> a
 1
>>> b
 [4, 2] # b值已被更改
```

函数的参数

- 参数是对象指针，无需定义传递的对象类型
  -

```
>>> def test(a,b):
        return a+b

>>> test(1,2) #数值型
3

>>> test("a","b") #字符型
 'ab'

>>> test([12],[11]) #列表
 [12, 11]

>>> test((12,),(11,))
(12, 11)
```

匿名函数

- 用lambda关键词能创建小型匿名函数。
- lambda函数能接收任何数量的参数但只能返回一个表达式的值，不能有return 。
- 为什么要用匿名函数? 程序一次使用，不需要定义函数名，节省内存中变量定义空间。能让程序更加简洁时。
- 匿名函数调用：直接赋值给一个变量,然后再像一般函数调用

```
lambda [arg1 [,arg2,.....argn]]:expression
```

匿名函数

```
>>>#直接赋值给一个变量,然后再像一般函数调用
   sum = lambda arg1, arg2: arg1 + arg2
>>>print "Value of total : ", sum( 10, 20 )
Value of total :  30


>>> c = lambda x,y=2: x+y #使用了默认值
>>> c(10) #不输的话，使用默认值2
12


#lambda代码简洁，常和map,reduce,filter等函数结合使用。
>>> filter(lambda x:x%3==0,[1,2,3,4,5,6])
[3, 6]
```

局部变量和全局变量

- 在一个函数中定义的变量一般只能在该函数内部使用，这些只能在程序的特定部分使用的变量我们称之为局部变量；
- 在一个文件顶部定义的变量可以供该文件中的任何函数调用，这些可以为整个程序所使用的变量称为全局变量。
- 如想在局部作用域中改变全局作用域的对象，必须使用global关键字。

```
# coding=utf-8
globalInt = 9

#定义一个函数
def myAdd():
    localInt = 3
    global gi
    gi =7
#在函数中定义一个局部变量
    return globalInt + localInt

#测试变量的局部性和全局性
print myAdd()
print globalInt
print gi
print localInt
```

局部变量和全局变量

```
>>>globalInt = 9

>>>def myAdd():
        localInt = 3
        global gi
        gi =7
        return globalInt + localInt

>>>print myAdd()
12
>>>print globalInt
9
>>>print gi
7
>>>print localInt
NameError
NameError: name 'localInt' is not defined
```

函数的注释说明—文档字符串

- 在函数定义后紧跟的字符串会被认为是函数的说明，使用help(函数名)可显示出来。

```
def add2(a,b):
        "add two item together"
        return (a+b)

if __name__ == "__main__":
        print add2(1,2)
        print add2("123","45")
```

- import test
- help(add2)
- 显示__doc__属性

```
>>> import test
>>> dir(test)
['__builtins__', '__doc__', '__file__', '__name__', 'add2']
>>> help(test.add2)
Help on function add2 in module test:
add2(a, b)
    add two item together

>>>test.add2(3, 7)

>>>python test.py  (cmd环境)
3
12345
```

help()
help (__builtin__)

常用函数 1/3

- abs(x):abs()返回一个数字的绝对值。如果给出复数，返回值就是该复数的模。
- callable(object)：callable() 函数用于检查一个对象是否是可调用的。如果返回True，object仍然可能调用失败；但如果返回False，调用对象object绝对不会成功。对于函数, 方法, lambda 函式, 类, 以及实现了 __call__ 方法的类实例, 它都返回 True。

```
>>>print abs(-100)
 100

 >>>print abs(1+2j)
2.2360679775
```

```
a = 'abc'

def f(a):
    pass

print(callable(a), callable(f))
```

常用函数 1/3

- cmp(x,y) :cmp()函数比较x和y两个对象，并根据比较结果返回一个整数，如果x<y，则返回-1；如果x>y，则返回1,如果x==y则返回0。
- isinstance(object,class-or-type-or-tuple) -> bool测试对象类型 isinstance(a,str)

```
>>>a=1
>>>b=2
>>>c=2
>>> print cmp(a,b)
-1
>>> print cmp(b,a)
1
>>> print cmp(b,c)
0
```

```
>>> a='isinstance test'
>>> b=1234
>>> isinstance(a,str)
True
>>> isinstance(a,int)
False
>>> isinstance(b,str)
False
>>> isinstance(b,int)
True
```

```
def displayNumType(num):
    print num, 'is',
    if isinstance(num, (int, long, float, complex)):
        print 'a number of type:', type(num).__name__
    else: print 'not a number at all!!!'

displayNumType(-69)
displayNumType(9999999999999999999999999L)
displayNumType(565.8)
displayNumType(-344.3+34.4j)
displayNumType('xxx')
```

常用函数 2/3

- divmod(x,y): divmod(x,y)函数完成除法运算，返回商和余数。
- pow(x,y[,z]) :pow()函数返回以x为底，y为指数的幂。如果给出z值，该函数就计算x的y次幂值被z取模的值。
- len(object) -> integer :len()函数返回字符串和序列的长度。
- min(x[,y,z...]) :返回序列或参数的最小值
- max(x[,y,z...]) ： 返回序列或参数的最大值

```
>>> divmod(10,3)
(3, 1)
>>> divmod(9,3)
(3, 0)
```

```
>>> print pow(2,4)
16
>>> print pow(2,4,2)
0
>>> print pow(2.4,3)
 13.824
```

常用函数 3/3

- range([lower,]stop[,step]) :range()函数可按参数生成连续的有序整数列表。
- round(x[,n]) :round()函数返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。
- type(obj):type()函数可返回对象的数据类型。
- xrange([lower,]stop[,step]):xrange()函数与range()类似，但xrnage()并不创建列表，而是返回一个xrange对象，它的行为与列表相似，但是只在需要时才计算列表值，当列表很大时，这个特性能节省内存。

```
>>> a=xrange(10)
>>> print a[0]
 0
>>> print a[2]
2
```

类型转换函数—数值型

- float(x)：把一个数字或字符串转换成浮点数。
- hex(x) :把整数转换成十六进制数。
- oct(x)：把整数转换成八进制数。
- int(x[,base])：把数字和字符串转换成一个整数，base为可选的基数。
- complex(real[,imaginary])：complex()函数可把字符串或数字转换为复数。
  - complex("2+1j")、complex(2,1)
- long(x[,base]) long()函数把数字和字符串转换成长整数，base为可选的基数。

```
>>> complex(2L,1)
(2+1j)
```

```
>>> long("123")
123L
>>>?long()
```

类型转换函数—字符串

- chr(i)：chr()函数返回ASCII码对应的字符串
- ord(x):ord()函数返回一个字符串参数的ASCII码或Unicode值 。
- str(obj)：str()函数把对象转换成可打印字符串。

```
>>> print chr(65)+chr(66)
AB
>>> ord("a")
97
>>> str(3+2j)
 '(3+2j)'
```

类型转换函数—序列对象

- list(x) :list()函数可将序列对象转换成列表
- tuple(x): tuple()函数把序列对象转换成tuple

```
>>> list("hello world")
 ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> list((1,2,3,4))
[1, 2, 3, 4]
```

```
>>> tuple("hello world")
('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')
>>> tuple([1,2,3,4])
(1, 2, 3, 4)
```

序列操作函数

- 常用函数中的len()、max()和min()同样可用于序列.
- filter(function,list):调用filter()时，它会把一个函数应用于序列中的每个项，并返回该函数返回真值时的所有项，从而过滤掉返回假值的所有项。
- map(function,list[,list])：map()函数把一个函数应用于序列中所有项，并返回一个列表。

```
def is_odd(n):
  return n % 2 == 1

filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15])
# 结果: [1, 5, 9, 15]
```

```
>> def f(x):
...   return x * x
...
>>> map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

序列操作函数

- reduce(function,seq[,init]) reduce()函数获得序列中前两个项，并把它传递给提供的函数，获得结果后再取序列中的下一项，连同结果再传递给函数，以此类推，直到处理完所有项为止。
- zip(seq[,seq,...]) zip()函数可把两个或多个序列中的相应项合并在一起，并以元组的格式返回它们，在处理完最短序列中的所有项后就停止。

```
import operator
>>> reduce(operator.mul,[2,3,4,5]) # ((2*3)*4)*5
120
>>> reduce(operator.mul,[2,3,4,5],2) # (((2*2)*3)*4)*5
240
```

```
>>> zip([1,2,3],[4,5],[7,8,9])
[(1, 4, 7), (2, 5, 8)]
>>> zip([1,2,3,4,5]) #如果参数是一个序列，则zip()会以一元组的格式返回每个项
[(1,), (2,), (3,), (4,), (5,)]
```

Python模块

- 模块简介
- 模块的__name__
- 创建模块
- dir() 函数
- 包（package）

模块简介

- 模块是最高级别的程序组织单元，它将程序代码和数据封装起来以便重用。
- 模块是一个包含所有你定义的函数和变量的文件，其后缀名是.py。
- 模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用python标准库的方法。

模块简介

```
# Filename: using_sys.py
import sys
print 'The command line
arguments are:`
for i in sys.argv:
    print i
print '\n\nThe PYTHONPATH
is',sys.path,'\n'
```

```
python using_sys.py we are
arguments
The command line arguments are:
using_sys.py
we
are
arguments
The PYTHONPATH is
['/home/swaroop/byte/code', …
```

- import sys引入python标准库中的sys.py模块；这是引入某一模块的方法。
- 2、sys.argv是一个包含命令行参数的列表。
- 3、sys.path包含了一个Python解释器自动查找所需模块的路径的列表。

模块简介

- 搜索路径被存储在sys模块中的path变量，做一个简单的实验，在交互式解释器中

```
import sys
sys.path
```

- 作为环境变量，PYTHONPATH由装在一个列表里的许多目录组成。PYTHONPATH的语法和shell变量PATH的一样。在Windows，典型的PYTHONPATH如下：

```
sys.path.append('E:\\pythontry')
```

模块简介

- from...import语句
  - 如果你想要直接输入argv变量到你的程序中（避免在每次使用它时打sys.），那么你可以使用from sys import argv语句。
  - 如果你想要输入所有sys模块使用的名字，那么你可以使用from sys import *语句。这对于所有模块都适用。
  - 一般说来，应该避免使用from...import而使用import语句，因为这样可以使你的程序更加易读，也可以避免名称的冲突。

模块的__name__

- 当一个模块被第一次输入的时候，这个模块的主块将被运行。假如只想在程序本身被使用的时候运行主块，而在它被别的模块输入的时候不运行主块，我们该怎么做呢？这可以通过模块的__name__属性完成。

```
# Filename: using_name.py

if __name__ == '__main__':
        print 'This program is being run by itself'
else:
        print 'I am being imported from another
module'
```

模块的__name__

```
! Python using_name.py /run using_name.py
This program is being run by itself

#在python环境中
>>> import using_name
I am being imported from another module
```

- 每个Python模块都有它的__name__属性，如果它是‘__main__’，这说明这个模块被用户单独运行，可以进行相应的恰当操作。

创建模块

- 每个Python程序也是一个模块。确保它具有.py扩展名了。

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Filename: mymodule.py

def sayhi():
        print'Hi, this is mymodule speaking.'
version = '0.1’

# End of mymodule.py
```

创建模块

```
#!/usr/bin/python
# Filename: mymodule_demo.py

import mymodule

mymodule.sayhi()
print 'Version', mymodule.version
```

```
!python mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

注意使用了相同的点号来使用模块的成员。

创建模块

- 下面是一个使用from...import语法的版本。

输出与mymodule_demo.py完全相同。

```
#!/usr/bin/python
# Filename: mymodule_demo2.py

from mymodule import sayhi,version
# Alternative:
# from mymodule import *

sayhi()
print 'Version', version
```

```
!python mymodule_demo2.py
Hi, this is mymodule speaking.
Version 0.1
```

dir() 函数

- 内置的函数 dir() 可以找到模块内定义的所有名称。以一个字符串列表的形式返回:

```
>>>import mymodule

>>>dir(mymodule)
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 'sayhi',
 'version']
```

- 如果没有给定参数，那么 dir() 函数会罗列出当前定义的所有名称。

包（package）

- package包是一组module的集合，一个文件夹下面只要有个__init__.py 文件，这个文件夹就可以看作是一个包，用以下方法创建一个包
  - 先在当前目录创建一个目录testpackage
  - 在testpackage下创建一个空文件__init__.py
  - 在testpackage中创建一个testmodule.py，里面编写任意代码。
  - 启动Python，运行：

```
>>> import testpackage.testmodule
>>> testpackage.testmodule.sayhi()
Hi, this is mymodule speaking.
```

包（package）

- 包是一种组织模块的方法，提供了一个命名空间，防止发生名字冲突。
- 包中还可以有包，所以这种方式可以很好的组织一个树状结构，用来管理多个模块。

输入输出

- input和print
- 输出格式美化
- 读和写文件

input和print

- 在很多时候，程序会需要与用户交互。程序会从用户那里得到输入，然后打印一些结果。简单的可以分别使用input 、raw_input和print语句来完成这些功能。

```
>>> name =raw_input("Please input your name:\n")
print("Hello,", name)

>>> name =input("Please input your name:\n")
print("Hello,", name)
```

```
>>> raw_input_B = raw_input("raw_input: ")
raw_input: 123
>>> type(raw_input_B)
<type 'str'>
>>> input_B = input("input: ")
input: 123
>>> type(input_B)
<type 'int'>
```

输出格式美化

Python两种输出值的方式：表达式语句和 print() 函数。(第三种方式是使用文件对象的 write() 方法)

- 如果你希望输出的形式更加多样，可以使用 str.format() 函数来格式化输出值。
- 如果你希望将输出的值转成字符串，可以使用 repr() 或 str() 函数来实现。
  - str() 函数返回一个用户易读的表达形式。
  - repr() 产生一个解释器易读的表达形式。

读和写文件

- open() 将会返回一个 file 对象，基本语法格式如下：

```
open(filename, mode)
```

  - 第一个参数为要打开的文件名。
  - 第二个参数描述文件如何使用的字符。 mode 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写。 mode 参数是可选的; 'r' 将是默认值。

读和写文件

- f.write(string) 将 string 写入到文件中

```
#  f = file("foo.txt", "w")

>>>f = open("foo.txt", "w")

>>>f.write( "Python 是一个非常好的语言。\n是的，的确非常好!!\n" )

>>>f.close()

>>>f = open("foo.txt", "r")

>>>str1 = f.read()

>>>print(str1)

>>>f.close()
```

读和写文件

- f.read(size), 读取数据, 然后作为字符串或字节对象返回。size 是一个可选的数字类型的参数。 当 size 被忽略了或者为负, 那么该文件的所有内容都将被读取并且返回。
- f.readline() 会从文件中读取单独的一行。换行符为 '\n'。f.readline() 如果返回一个空字符串, 说明已经已经读取到最后一行。

```
>>> f = open("foo.txt", "r")

>>> str2 = f.readline()

>>> print(str2)

>>> f.close()
```

读和写文件

- f.tell() 返回文件对象当前所处的位置，它是从文件开头开始算起的字节数。
- f.seek()

  如果要改变文件当前的位置，可以使用 f.seek(offset, from_what) 函数。

  - from_what 的值，如果是 0 表示开头，如果是 1 表示当前位置，2 表示文件的结尾，例如：

    seek(x,0) ： 从起始位置即文件首行首字符开始移动 x 个字符

    seek(x,1) ： 表示从当前位置往后移动x个字符

    seek(x,2)：表示从文件的结尾往前移动x个字符

    from_what 值为默认为0，即文件开头

读和写文件

- pickle 模块

python的pickle模块实现了基本的数据序列和反序列化。

通过pickle模块的序列化操作能够将程序中运行的对象信息保存到文件中去，永久存储。通过pickle模块的反序列化操作，能够从文件中创建上一次程序保存的对象。

```
pickle.dump(obj, file, [,protocol])
```

有了 pickle 这个对象, 就能对 file 以读取的形式打开:

```
x = pickle.load(file)
```

读和写文件

- 使用pickle模块将数据对象保存到文件

```
import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]

selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

pickle.dump(data1, output)   # Pickle dictionary using protocol 0.

pickle.dump(selfref_list, output, -1) # Pickle the list using the
highest protocol available.

output.close()
```

读和写文件

- 使用pickle模块从文件中重构python对象

```
import pprint, pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)

pprint.pprint(data1)
{'a': [1, 2.0, 3, (4+6j)], 'b': ('string', u'Unicode string'), 'c': None}

data2 = pickle.load(pkl_file)

pprint.pprint(data2)
[1, 2, 3, <Recursion on list with id=167991560>]

data2[3][1]

pkl_file.close()
```

异常处理

- Python 错误和异常
- 异常处理
- 异常使用

Python 错误和异常

Python有两种错误很容易辨认：语法错误和异常。

- **语法错误**

Python 的语法错误或者称之为解析错。

```
>>>while True print('Hello world')
  File "<ipython-input-1-614901b0e5ee>", line 1
    while True print('Hello world')
                    ^
SyntaxError: invalid syntax
```

这个例子中，函数 print() 被检查到有错误，是它前面缺少了一个冒号（:）。语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的箭头。

Python 错误和异常

异常

即便Python程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里：

```
>>>10 * (1/0)
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-9ce172bd90a7> in <module>()
----> 1 10 * (1/0)

ZeroDivisionError: integer division or modulo by zero
```

Python 错误和异常

```
>>>4 + spam*3
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-6b1dfe582d2e> in <module>()
----> 1 4 + spam*3

NameError: name 'spam' is not defined

>>>'2' + 2
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-4c6dd5170204> in <module>()
----> 1 '2' + 2

TypeError: cannot concatenate 'str' and 'int' objects
```

异常以不同的类型出现，这些类型都作为信息的一部分打印出来。错误信息的前面部分显示了异常发生的上下文，并以调用栈的形式显示具体信息。

异常处理

Python的异常处理能力是很强大的，可向用户准确反馈出错信息。

- 方式一：try语句：
  - 1、使用try和except语句来捕获异常

```
try:
   block
except [exception,[data…]]:
   block

try:
   block
except [exception,[data...]]:
   block
else:
   block
```

异常处理

该种异常处理语法的规则是：

- 执行try下的语句，如果引发异常，则执行过程会跳到第一个except语句。
- 如果第一个except中定义的异常与引发的异常匹配，则执行该except中的语句。
- 如果引发的异常不匹配第一个except，则会搜索第二个except，允许编写的except数量没有限制。
- 如果所有的except都不匹配，则异常会传递到下一个调用本代码的最高层try代码中。
- 如果没有发生异常，则执行else块代码。

异常处理

```
fname = raw_input('Enter filename: ')
try:
        fobj = open(fname, 'r')
except IOError, e:
        print "*** file open error:", e
else:
        for eachLine in fobj:
            print eachLine,
        fobj.close()
```

捕获到的IOError错误的详细原因会被放置在对象e中,然后运行该异常的except代码块。

使用except子句需要注意的事情，就是多个except子句截获异常时，如果各个异常类之间具有继承关系，则子类应该写在前面，否则父类将会直接截获子类异常。放在后面的子类异常也就不会执行到了。

异常处理

- 2、使用try跟finally：

```
try:
    block
finally:
    block
```

该语句的执行规则是：

执行try下的代码。如果发生异常，在该异常传递到下一级try时，执行finally中的代码。如果没有发生异常，则执行finally中的代码。

第二种try语法在无论有没有发生异常都要执行代码的情况下是很有用的。例如在python中打开一个文件进行读写操作，在操作过程中不管是否出现异常，最终都是要把该文件关闭的。

异常处理

try语句分句形式

分句形式	说明
`except:`	捕获所有异常类型
`except name:`	只捕获指定类型异常
`except name,value:`	捕获所列异常，并获得抛出的异常对象
`except (name1,name2):`	捕获任何列出类型的异常
`except (name1,name2),value:`	捕获任何列出类型的异常，并获得抛出的异常对象
`else:`	如何没有异常发生，则运行
`finally:`	不管有没有异常，都运行此代码块

异常处理

- 方式二：**抛出异常**

Python 使用 raise 语句抛出一个指定的异常。

```
>>> raise NameError('HiThere')
---------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
<ipython-input-15-93385ba972b1> in <module>()
----> 1 raise NameError('HiThere‘)
NameError: HiThere
```

raise 唯一的一个参数指定了要被抛出的异常。它必须是一个异常的实例或者是异常的类（也就是 Exception 的子类）。

异常处理

如果你只想知道这是否抛出了一个异常，并不想去处理它，那么一个简单的 raise 语句就可以再次把它抛出。

```
>>>try:
        raise NameError('HiThere')
    except NameError:
        print('An exception flew by!')
        raise

An exception flew by!
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-5d1f4d009277> in <module>()
     1 try:
----> 2     raise NameError('HiThere')
    .....

NameError: HiThere
```

异常处理

- 可见，异常是另一种函数返回方式，C语言或者其他没有异常机制的语言，函数只有通过返回值指明函数出错，所以每一级函数都要检查函数的返回值。

- 异常机制就是提供除了返回值之外的另一种出错处理方法。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。大部分库中的模块的出错处理大多数都是抛出异常。

异常处理

- 在抛出异常时，可以是任何对象，用于详细描述错误类型；但一般要求是Exception类的子类

```
class ShortInputException:
    '''A user-defined exception class.'''
    def __init__(self, length, atleast):
        self.length = length
        self.atleast = atleast
try:
    s = raw_input('Enter something --> ')
    if len(s) < 3:
        raise ShortInputException(len(s), 3)
except ShortInputException, x:
    print 'ShortInputException: The input was of length %d, \
          was expecting at least %d' % (x.length, x.atleast)
else:
    print 'No exception was raised.'
```

异常处理

- 与**Python**异常相关的关键字：

关键字	关键字说明
raise	抛出/引发异常
try/except	捕获异常并处理
pass	忽略异常
as	定义异常实例(except IOError as e)
finally	无论是否出现异常，都执行的代码
else	如果try中的语句没有引发异常，则执行else中的语句