**字符串常见操作**

**<1>find**

检测 str 是否包含在 mystr中，如果是返回开始的索引 值，否则返回-1

mystr.find(str, start=0, end=len(mystr))

**<2>index**

跟find()方法一样，只不过如果str不在 mystr中会报 一个异常

mystr.index(str, start=0, end=len(mystr))

**<3>count**

返回 str在start和end之间 在 mystr里面出现的次数

mystr.count(str, start=0, end=len(mystr))

**<4>replace**

把 mystr 中的 str1 替换成 str2,如果 count 指定，则 替换不超过 count 次

mystr.replace(str1, str2, mystr.count(str1))

**<5>split**

以 str 为分隔符切片 mystr，如果 maxsplit有指定值，则仅分隔 maxsplit 个子字符串

mystr.split(str=" ", 2)

**<6>capitalize**

把字符串的第一个字符大写

mystr.capitalize()

**<7>title**

把字符串的每个单词首字母大写

>>>a = "hello itcast"

>>>a.title()

'Hello Itcast'

**<8>startswith**

检查字符串是否是以 obj 开头, 是则返回 True，否则返 回 False

mystr.startswith(obj)

**<9>endswith**

检查字符串是否以obj结束，如果是返回True,否则返回 False

mystr.endswith(obj)

**<10>lower**

转换 mystr 中所有大写字符为小写

mystr.lower()

**<11>upper**

转换 mystr 中的小写字母为大写

mystr.upper()

**<12>ljust**

返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串

mystr.ljust(width)

**<13>rjust**

返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串

mystr.rjust(width)

**<14>center**

返回一个原字符串居中,并使用空格填充至长度 width 的新字符串

mystr.center(width)

**<15>lstrip**

删除 mystr 左边的空白字符

mystr.lstrip()

**<16>rstrip**

删除 mystr 字符串末尾的空白字符

mystr.rstrip()

**<17>strip**

删除mystr字符串两端的空白字符

>>>a = "\n\t itcast \t\n"

>>>a.strip()

'itcast'

**<18>rfind**

类似于 find()函数，不过是从右边开始查找

mystr.rfind(str, start=0,end=len(mystr) )

**<19>rindex**

类似于 index()，不过是从右边开始

mystr.rindex( str, start=0,end=len(mystr))

**<20>partition**

把mystr以str分割成三部分,str前，str和str后

mystr.partition(str)

**<21>rpartition**

类似于 partition()函数,不过是从右边开始

mystr.rpartition(str)

**<22>splitlines**

按照行分隔，返回一个包含各行作为元素的列表

mystr.splitlines()

**<23>isalpha**

如果 mystr 所有字符都是字母 则返回 True,否则返回False

mystr.isalpha()

**<24>isdigit**

如果 mystr 只包含数字则返回 True 否则返回 False

mystr.isdigit()

**<25>isalnum**

如果 mystr 所有字符都是字母或数字则返回 True,否则 返回 False

mystr.isalnum()

**<26>isspace**

如果 mystr 中只包含空格，则返回 True，否则返回 False

mystr.isspace()

**<27>join**

mystr 中每个字符后面插入str,构造出一个新的字符串

mystr.join(str)

**列表的相关操作**

**<1>添加元素("增"append, extend, insert)**

**append**

通过append可以向列表添加元素

**extend**

通过extend可以将另一个集合中的元素逐一添加到列表 中

**insert**

insert(index, object) 在指定位置index前插入元素 objec

**<2>修改元素("改")**

修改元素的时候，要通过下标来确定要修改的是哪个元素，然后才能进行修改

**<3>查找元素("查"in, not in, index, count)**

所谓的查找，就是看看指定的元素是否存在

**in, not in**

in（存在）,如果存在那么结果为true，否则为false

not in（不存在），如果不存在那么结果为true，否则false

**index, count**

index和count与字符串中的用法相同

**<4>删除元素("删"del, pop, remove)**

**del**：根据下标进行删除

**pop**：删除最后一个元素

**remove**：根据元素的值进行删除

**<5>排序(sort, reverse)**

**sort**方法是将list按特定顺序重新排列，默认为由小到大，参数reverse=True可改为倒序，由大到小

**reverse**方法是将list逆置

**字典的常见操作**

**<1>修改元素**

字典的每个元素中的数据是可以修改的，只要通过key找到，即可修改

**<2>添加元素**

如果在使用 **变量名['键'] = 数据**时，这个“键”在字典中，不存在，那么就会新增这个元素

**<3>删除元素**

del

clear()

del可以删除指定的元素，也可以删除整个字典

clear清空整个字典

**<4>len()**

测量字典中，键值对的个数

**<5>keys**

返回一个包含字典所有KEY的列表

**<6>values**

返回一个包含字典所有value的列表

**<7>items**

返回一个包含所有（键，值）元祖的列表

**<8>has\_key**

dict.has\_key(key)如果key在字典中，返回True，否则返回False

**公共方法**

| **运算符** | **Python 表达式** | **结果** | **描述** | **支持的数据类型** |
| --- | --- | --- | --- | --- |
| + | [1, 2] + [3, 4] | [1, 2, 3, 4] | 合并 | 字符串、列表、元组 |
| \* | 'Hi!' \* 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | 复制 | 字符串、列表、元组 |
| in | 3 in (1, 2, 3) | True | 元素是否存在 | 字符串、列表、元组、字典 |
| not in | 4 not in (1, 2, 3) | True | 元素是否不存在 | 字符串、列表、元组、字典 |

## 运算符

## python内置函数

| **序号** | **方法** | **描述** |
| --- | --- | --- |
| 1 | cmp(item1, item2) | 比较两个值 |
| 2 | len(item) | 计算容器中元素个数 |
| 3 | max(item) | 返回容器中元素最大值 |
| 4 | min(item) | 返回容器中元素最小值 |
| 5 | del(item) | 删除变量 |

Python包含了以下内置函数

eval( )方法：将字符串str当成有效的表达式来求值并返回计算结果。

**文件和文件夹的常用操作（with关键字）**

**关键字with 在不再需要访问文件后将其关闭。要删除显示文件末尾多出来的空行，可在print 语句中使用rstrip()**

**<1>打开文件**

在python，使用open函数，可以打开一个已经存在的文件，或者创建一个新文件

open(文件名，访问模式)

f = open('test.txt', 'w')

| **访问模式** | **说明** |
| --- | --- |
| r | 以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。 |
| w | 打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| a | 打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。 |
| rb | 以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。 |
| wb | 以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| ab | 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。 |
| r+ | 打开一个文件用于读写。文件指针将会放在文件的开头。 |
| w+ | 打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| a+ | 打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。 |
| rb+ | 以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。 |
| wb+ | 以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| ab+ | 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。 |

**<2>关闭文件**

close()

# 新建一个文件，文件名为:test.txt

f = open('test.txt', 'w')

# 关闭这个文件

f.close()

**<3>写数据(write)**

使用write()可以完成向文件写入数据

f = open('test.txt', 'w')

f.write('hello world, i am here!')

f.close()

**※ 如果文件不存在那么创建，如果存在那么就先清空，然后写入数据**

**<2>读数据(read)**

使用read(num)可以从文件中读取数据，num表示要从文件中读取的数据的长度（单位是字节），如果没有传入num，那么就表示读取文件中所有的数据

f = open('test.txt', 'r')

content = f.read(5)

print(content)

print("-"\*30)

content = f.read()

print(content)

f.close()

**※ 如果open是打开一个文件，那么可以不用写打开的模式，即只写 open('test.txt')**

**※ 如果使用读了多次，那么后面读取的数据是从上次读完后的位置开始的**

**<3>读数据（readlines）**

就像read没有参数时一样，readlines可以按照行的方式把整个文件中的内容进行一次性读取，并且返回的是一个列表，其中每一行的数据为一个元素

#coding=utf-8

f = open('test.txt', 'r')

content = f.readlines()

print(type(content))

i=1

for temp in content:

print("%d:%s"%(i, temp))

i+=1

f.close()

**<4>读数据（readline）**

#coding=utf-8

f = open('test.txt', 'r')

content = f.readline()

print("1:%s"%content)

content = f.readline()

print("2:%s"%content)

f.close()

**<5>获取当前读写的位置**

在读写文件的过程中，如果想知道当前的位置，可以使用**tell()**来获取

# 打开一个已经存在的文件

f = open("test.txt", "r")

str = f.read(3)

print"读取的数据是 : ", str

# 查找当前位置

position = f.tell()

print"当前文件位置 : ", position

str = f.read(3)

print"读取的数据是 : ", str

# 查找当前位置

position = f.tell()

print"当前文件位置 : ", position

f.close()

**<6>定位到某个位置**

如果在读写文件的过程中，需要从另外一个位置进行操作的话，可以使用seek()

seek(offset, from)有2个参数

offset:偏移量

from:方向

0:表示文件开头

1:表示当前位置

2:表示文件末尾

# 打开一个已经存在的文件

f = open("test.txt", "r")

str = f.read(30)

print"读取的数据是 : ", str

# 查找当前位置

position = f.tell()

print"当前文件位置 : ", position

# 重新设置位置

f.seek(5,0)

# 查找当前位置

position = f.tell()

print"当前文件位置 : ", position

f.close()

# 打开一个已经存在的文件

f = open("test.txt", "r")

# 查找当前位置

position = f.tell()

print"当前文件位置 : ", position

# 重新设置位置

f.seek(-3,2)

# 读取到的数据为：文件最后3个字节数据

str = f.read()

print"读取的数据是 : ", str

f.close()

**<7>文件重命名**

os模块中的rename()可以完成对文件的重命名操作

rename(需要修改的文件名, 新的文件名)

import os

os.rename("毕业论文.txt", "毕业论文-最终版.txt")

**<8>删除文件**

os模块中的remove()可以完成对文件的删除操作

remove(待删除的文件名)

import os

os.remove("毕业论文.txt")

**<9>创建文件夹**

import os

os.mkdir("张三")

**<10>获取当前目录**

import os

os.getcwd()

**<11>改变默认目录**

import os

os.chdir("../")

**<12>获取目录列表**

import os

os.listdir("./")

**<13>删除文件夹**

import os

os.rmdir("张三")

**魔法方法**

**<1>\_\_new\_\_方法**

1、\_\_new\_\_至少要有一个参数cls，代表要实例化的类，此参数在实例化时由Python解释器自动提供

2、\_\_new\_\_必须要有返回值，返回实例化出来的实例，这点在自己实现\_\_new\_\_时要特别注意，可以return父类\_\_new\_\_出来的实例，或者直接是object的\_\_new\_\_出来的实例

def\_\_new\_\_(cls):

print("这是 new 方法")

returnobject.\_\_new\_\_(cls)

**<2>\_\_init\_\_()方法**

1、\_\_init\_\_()方法，在创建一个对象时默认被调用，不需要手动调用

2、\_\_init\_\_(self)中，默认有1个参数名字为self，如果在创建对象时传递了2个实参，那么\_\_init\_\_(self)中除了self作为第一个形参外还需要2个形参，例如\_\_init\_\_(self,x,y)

3、\_\_init\_\_(self)中的self参数，不需要开发者传递，python解释器会自动把当前的对象引用传递进去

class类名:

#初始化函数，用来完成一些默认的设定

def\_\_init\_\_():

pass

**<3>\_\_str\_\_()方法**

1、在python中方法名如果是\_\_xxxx\_\_()的，那么就有特殊的功能，因此叫做“魔法”方法

2、当使用print输出对象的时候，只要自己定义了\_\_str\_\_(self)方法，那么就会打印从在这个方法中return的数据

def\_\_str\_\_(self):

msg = "需要打印出的内容！"

return msg

**<3>\_\_del\_\_()方法**

当删除一个对象时，python解释器也会默认调用一个方法，这个方法为\_\_del\_\_()方法

1、当有1个变量保存了对象的引用时，此对象的引用计数就会加1

2、当使用del删除变量指向的对象时，如果对象的引用计数不会1，比如3，那么此时只会让这个引用计数减1，即变为2，当再次调用del时，变为1，如果再调用1次del，此时会真的把对象进行删除

def\_\_del\_\_(self):

print("\_\_del\_\_方法被调用")

print("%s对象马上被干掉了..."%self.\_\_name)

**异常**

**<1>捕获异常 try...except...**

try:

print('-----test--1---')

open('123.txt','r')

print('-----test--2---')

except IOError:

pass

1、把可能出现问题的代码，放在try中

2、把处理异常的代码，放在except中

**<2> except捕获多个异常**

#coding=utf-8

try:

print('-----test--1---')

open('123.txt','r') # 如果123.txt文件不存在，那么会产生 IOError 异常

print('-----test--2---')

print(num)# 如果num变量没有定义，那么会产生 NameError 异常

#如果想通过一次except捕获到多个异常可以用一个元组的方式

except (IOError,NameError):

# errorMsg里会保存捕获到的错误信息

print(errorMsg)

1、当捕获多个异常时，可以把要捕获的异常的名字，放到except 后，并使用元组的方式仅进行存储

**<3>获取异常的信息描述**

try:

print(a)

except (NameError,IOError) as result

print(result)

name ‘a’ is not defined

**<4>捕获所有异常**

try:

open(“a.txt”)

except Exception as result:

print(result)

[Error 2] No such file or directory: ‘a.txt’

**<5> else**

即如果没有捕获到异常，那么就执行else中的事情

try:

num = 100

print num

except NameError as errorMsg:

print('产生错误了:%s'%errorMsg)

else:

print('没有捕获到异常，真高兴')

**<6> try...finally...**

在程序中，如果一个段代码必须要执行，即无论异常是否产生都要执行，那么此时就需要使用finally

import time

try:

f = open('test.txt')

try:

whileTrue:

content = f.readline()

if len(content) == 0:

break

time.sleep(2)

print(content)

except:

#如果在读取文件的过程中，产生了异常，那么就会捕获到

#比如按下了 ctrl+c

pass

finally:

f.close()

print('关闭文件')

except:

print("没有这个文件")

**模块**

**<1>import**

在Python中用关键字import来引入某个模块，比如要引用模块math，就可以在文件最开始的地方用import math来引入。

import module1,mudule2...

**<2>from…import**

Python的from语句让你从模块中导入一个指定的部分到当前命名空间中

from modname import name1[, name2[, ... nameN]]

**<3>from … import \***

把一个模块的所有内容全都导入到当前的命名空间

from modname import \*

**<4> as**

In [1]: import time as tt

In [2]: time.sleep(1)

------------------------------------------------------------------------

NameError: name 'time' is not defined

In [3]: tt.sleep(1)

**<6>定位模块**

当你导入一个模块，Python解析器对模块位置的搜索顺序是：

1.当前目录

2.如果不在当前目录，Python则搜索在shell变量PYTHONPATH下的每个目录。

3.如果都找不到，Python会察看默认路径。UNIX下，默认路径一般为/usr/local/lib/python/

4.模块搜索路径存储在system模块的sys.path变量中。变量里包含当前目录，PYTHONPATH和由安装过程决定的默认目录。

**<7>\_\_all\_\_()方法**

如果一个文件中有\_\_all\_\_变量，那么也就意味着不在这个变量中的元素，不会被from xxx import \*时导入

\_\_all\_\_ = [“test”, “test1”]

**<8>\_\_init\_\_.py文件（包）**

将有联系的模块组织在一起，即放到同一个文件夹下，并且在这个文件夹创建一个名字为\_\_init\_\_.py 文件，那么这个文件夹就称之为包

\_\_init\_\_.py文件中编写内容：

\_\_all\_\_ = [“test”, “test1”]

**<8>给程序传参数**

import sys

print(sys.argv)

>>>python3 01-传递参数.py haha 1 2 3

>>>[‘01-传递参数.py’, ‘haha’, ‘1’, ‘2’, ‘3’]

**<8>列表推导式**

**1、基本方式**

a = [x for i in range(3,19,2)]

>>>a = [3, 5, 7, 9, 11, 13, 15, 17]

**2、在循环的过程中使用if**

a = [x for x in range(3, 10) if x%2==0]

>>>a = [4, 6, 8]

**3、2个for循环**

a = [(x,y) for x in range(1,3) for y in range(3)]

>>>a = [(1, 0), (1, 1), (1, 2), (2, 0), (2, 1),

(2, 2)]

**4、3个for循环**

a = [(x,y,z) for x in range(1,3) for y in range(3) for z in range(4,6)]

>>>a = [(1, 0, 4), (1, 0, 5), (1, 1, 4), (1, 1, 5),

(1, 2, 4), (1, 2, 5), (2, 0, 4), (2, 0 ,5),

(2, 1, 4), (2, 1, 5), (2, 2, 4), (2, 2, 5)]

list = [“列表”] tuple=(元组) set={集合}

**Python高级**

**<1>import搜索路径：**

importsys

sys.path

**路径搜索**

从上面列出的目录里依次查找要导入的模块文件

' ' 表示当前路径

**程序执行时导入模块路径**

sys.path.append('/home/itcast/xxx')

sys.path.insert(0, '/home/itcast/xxx') #可以确保先搜 索这个路径

**<2>重新导入模块**

from imp import \*

reload(xxx)

**<3>==、is**

is 是比较两个引用是否指向了同一个对象（引用比较）。

== 是比较两个对象是否相等。

**<4>深拷贝、浅拷贝**

**1. 浅拷贝（ =）**

浅拷贝是对于一个对象的顶层拷贝

通俗的理解是：拷贝了引用，并没有拷贝内容

**2. 深拷贝（copy.deepcopy(xxx)）**

深拷贝是对于一个对象所有层次的拷贝(递归)

**3. 拷贝的其他方式（copy.copy(xxx)）**

对于可变类型，第一层为深拷贝，内部层为浅拷贝。

对于不可变类型，全为浅拷贝。

**<5>进制、位运算**、

**1.进制**

十进制 int（xxx）

二进制 bin（xxx）

十六进制 hex（xxx）

八进制 oct（xxx）

**2.位运算**

**& 按位与 全1才1否则为0**

**| 按位或 有1就1全0为0**

**^ 按位异或 不同为1相同为0**

**~ 按位取反 按位取反 0为11为0**

**<< 按位左移   左移1位相当于 乘以2**

**>> 按位右移 右移1位相当于 除以2**

**<6>私有化**

xx: 公有变量

\_x: 单前置下划线,私有化属性或方法，from somemodule import \*禁止导入,类对象和子类可以访问

\_\_xx：双前置下划线,避免与子类中的属性命名冲突，无法 在外部直接访问(名字重整所以访问不到<\_类名\_\_xx>)

\_\_xx\_\_:双前后下划线,用户名字空间的魔法对象或属性。 例如:\_\_init\_\_ , \_\_ 不要自己发明这样的名字

xx\_:单后置下划线,用于避免与Python关键词的冲突

1、父类中属性名为\_\_名字的，子类不继承，子类不能访问

2、如果在子类中向\_\_名字赋值，那么会在子类中定义的一个与父类相同名字的属性

3、\_名的变量、函数、类在使用from xxx import \*时都不会被导入

**<7>属性property**

**1. 私有属性添加getter和setter方法**

classMoney(object):

def\_\_init\_\_(self):

self.\_\_money = 0

defgetMoney(self):

returnself.\_\_money

defsetMoney(self, value):

ifisinstance(value, int):

self.\_\_money = value

else:

print("error:不是整型数字")

**2. 使用property升级getter和setter方法**

classMoney(object):

def\_\_init\_\_(self):

self.\_\_money = 0

defgetMoney(self):

returnself.\_\_money

defsetMoney(self, value):

ifisinstance(value, int):

self.\_\_money = value

else:

print("error:不是整型数字")

money = property(getMoney, setMoney)

>>>from get\_set import Money >>>a = Money() >>>a.money >>>a.money = 100 >>>a.get\_money

>>>0 >>>100 >>>100

**3. 使用property取代getter和setter方法**

classMoney(object):

def\_\_init\_\_(self):

self.\_\_money = 0

@property

defmoney(self):

returnself.\_\_money

@money.setter

defmoney(self, value):

ifisinstance(value, int):

self.\_\_money = value

else:

print("error:不是整型数字")

>>>a = Money()

>>>a.money >>>a.money = 100 >>>a.money

>>>0 >>>100 >>>100

**<8>生成器（generator）---（协程）**

**1、创建生成器方法<1>**

**只要把一个列表生成式的 [ ] 改成 ( )**

L = [x\*2for x inrange(5)]

>>> [0, 2, 4, 6, 8]

G = (x\*2for x inrange(5))

>>><generator object <genexpr> at 0x7f626c132db0>

※***尽量不用 next(),而是用for循环迭代它***

2、**创建生成器方法<2>**

**要把函数变成generator，只需要把print(b)改为yield b就可以了**

def fib(times):

n = 0

a,b = 0,1

while n<times:

yield b

a,b = b,a+b

n+=1

return 'done'

※***尽量不用 next(),而是用for循环迭代它***

**用for循环调用generator时，拿不到generator的return 语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误,返回值包含在StopIteration的value中**

g = fib(5)

whileTrue:

try:

x = next(g)

print("value:%d"%x)

except StopIteration as e:

print("生成器返回值:%s"%e.value)

break

3、**send**

defgen():

i = 0

while i<5:

temp = yield i

print(temp)

i+=1

执行到yield时，gen函数作用暂时保存，返回i的值;temp 接收下次c.send("python")，send发送过来的值，

**c.next() 等价c.send(None)**

**<9>迭代器**

**1. 可迭代对象**

可以直接作用于 for 循环的对象统称为可迭代对象： Iterable。如 list 、 tuple 、 dict 、 set 、 str 等。

**2. 判断是否可以迭代**

使用 isinstance() 判断一个对象是否是Iterable对象

from collections import Iterable

isinstance([], Iterable)

>>>True

isinstance({}, Iterable)

>>>True

isinstance('abc', Iterable)

>>>True

isinstance((x for x in range(10)), Iterable)

>>>True

isinstance(100, Iterable)

>>>False

**3.迭代器**

可以被next()函数调用并不断返回下一个值的对象称为 迭代器：Iterator

使用 isinstance() 判断一个对象是否是Iterator对象

from collections import Iterator

isinstance((x for x in range(10)), Iterator)

>>>True

isinstance([], Iterator)

>>>False

isinstance({}, Iterator)

>>>False

isinstance('abc', Iterator)

>>>False

isinstance(100, Iterator)

>>>False

**4.iter()函数**

生成器都是 Iterator 对象，但 list 、 dict 、 str 虽 然是 Iterable ，却不是 Iterator 。

把 list 、 dict 、 str 等 Iterable 变成 Iterator 可以使用 iter() 函数:

isinstance(iter([]), Iterator)

>>>True

isinstance(iter('abc'), Iterator)

>>>True

1、凡是可作用于 for 循环的对象都是 Iterable 类型；

2、凡是可作用于 next() 函数的对象都是 Iterator 类型

3、集合数据类型如 list 、 dict 、 str 等是 Iterable 但不是 Iterator ，不过可以通过 iter() 函数获得一个 Iterator 对象。

**<10>闭包**

**1. 函数引用**

deftest1():

print("--- in test1 func----")

#调用函数

test1()

#引用函数

ret = test1

print(id(ret))

print(id(test1))

#通过引用调用函数

ret()

运行结果:

--- in test1 func----

140212571149040

140212571149040

--- in test1 func----

**2. 什么是闭包**

#定义一个函数

deftest(number):

#在函数内部再定义一个函数，并且这个函数用到了外边函数的变量，那么将

这个函数以及用到的一些变量称之为闭包

deftest\_in(number\_in):

print("in test\_in 函数, number\_in is %d"%number\_in)

return number+number\_in

#其实这里返回的就是闭包的结果

return test\_in

#给test函数赋值，这个20就是给参数number

ret = test(20)

#注意这里的100其实给参数number\_in

print(ret(100))

#注意这里的200其实给参数number\_in

print(ret(200))

in test\_in 函数, number\_in is100

120

in test\_in 函数, number\_in is200

220

**3. 闭包再理解**

内部函数对外部函数作用域里变量的引用（非全局变量）， 则称内部函数为闭包。

# closure.py

defcounter(start=0):

count=[start]

defincr():

count[0] += 1

returncount[0]

return incr

nonlocal访问外部函数的局部变量(python3)

defcounter(start=0):

defincr():

nonlocal start

start += 1

return start

return incr

闭包的实际例子

defline\_conf(a, b):

defline(x):

return a\*x + b

return line

line1 = line\_conf(1, 1)

line2 = line\_conf(4, 5)

print(line1(5))

print(line2(5))

1、闭包似优化了变量，原来需要类对象完成的工作，闭包 也可以完成

2、由于闭包引用了外部函数的局部变量，则外部函数的局 部变量没有及时释放，消耗内存

**<11>装饰器**

**1. 什么是装饰器**

写代码要遵循**开放封闭**原则，虽然在这个原则是用的面 向对象开发，但是也适用于函数式编程，简单来说，它规定 已经实现的功能代码不允许被修改，但可以被扩展，即：

**封闭：已实现的功能代码块**

**开放：对扩展开发**

defw1(func):

definner():

# 验证1

# 验证2

# 验证3

func()

return inner

@w1

deff1():

print('f1')

**@函数名 是python的一种语法糖**

@w1内部会执行一下操作

执行w1函数 ，并将 @w1 下面的函数作为w1函数的 参数，即：**@w1 等价于 w1(f1)**所以，内部就会去执行：

definner():

#验证 1

#验证 2

#验证 3

f1() # func是参数，此时 func 等于 f1

return inner # 返回的 inner，inner代表的是函数，非执 行函数,其实就是将原来的 f1 函数塞进另外一个函数中

将执行完的w1函数返回值 赋值 给@w1下面的函数的 函数名f1 即将w1的返回值再重新赋值给 f1，即：

新f1 = definner():

#验证 1

#验证 2

#验证 3

原来f1()

return inner

所以，以后业务部门想要执行 f1 函数时，就会执行 新 f1 函数，在新f1 函数内部先执行验证，再执行原来的f1函 数，然后将原来f1 函数的返回值返回给了业务调用者。

**2. 再议装饰器**

#定义函数：完成包裹数据

defmakeBold(fn):

defwrapped():

return"<b>" + fn() + "</b>"

return wrapped

#定义函数：完成包裹数据

defmakeItalic(fn):

defwrapped():

return"<i>" + fn() + "</i>"

return wrapped

@makeBold

deftest1():

return"hello world-1"

@makeItalic

deftest2():

return"hello world-2"

@makeBold

@makeItalic

deftest3():

return"hello world-3"

print(test1()) >>><b>hello world-1</b>

print(test2()) >>><i>hello world-2</i>

print(test3()) >>><b><i>hello world-3</i></b>

**3. 装饰器(decorator)功能**

1、引入日志

2、函数执行时间统计

3、执行函数前预备处理

4、执行函数后清理功能

5、权限校验等场景

6、缓存

**4. 装饰器示例**

**1、无参数的函数**

from time import ctime, sleep

deftimefun(func):

defwrappedfunc():

print("%s called at %s"%(func.\_\_name\_\_, ctime()))

func()

return wrappedfunc

@timefun

deffoo():

print("I am foo")

foo()

sleep(2)

foo()

**2、被装饰的函数有参数**

from time import ctime, sleep

deftimefun(func):

defwrappedfunc(a, b):

print("%s called at %s"%(func.\_\_name\_\_, ctime()))

print(a, b)

func(a, b)

return wrappedfunc

@timefun

deffoo(a, b):

print(a+b)

foo(3,5)

sleep(2)

foo(2,4)

**3、被装饰的函数有不定长参数**

from time import ctime, sleep

deftimefun(func):

defwrappedfunc(\*args, \*\*kwargs):

print("%s called at %s"%(func.\_\_name\_\_, ctime()))

func(\*args, \*\*kwargs)

return wrappedfunc

@timefun

deffoo(a, b, c):

print(a+b+c)

foo(3,5,7)

sleep(2)

foo(2,4,9)

**4、装饰器中的return**

from time import ctime, sleep

deftimefun(func):

defwrappedfunc():

print("%s called at %s"%(func.\_\_name\_\_, ctime()))

func()

return wrappedfunc

@timefun

deffoo():

print("I am foo")

@timefun

defgetInfo():

return'----hahah---'

foo()

sleep(2)

foo()

print(getInfo())

**一般情况下为了让装饰器更通用，可以有return**

**5、装饰器带参数,在原有装饰器的基础上，设置外部变量**

#decorator2.py

from time import ctime, sleep

deftimefun\_arg(pre="hello"):

deftimefun(func):

defwrappedfunc():

print("%s called at %s %s"%(func.\_\_name\_\_, ctime(), pre))

returnfunc()

return wrappedfunc

return timefun

@timefun\_arg("itcast")

deffoo():

print("I am foo")

@timefun\_arg("python")

deftoo():

print("I am too")

foo()

sleep(2)

foo()

too()

sleep(2)

too()

**<12>python是动态语言**

**1、运行的过程中给对象绑定(添加)属性**

>>>classPerson(object):

def\_\_init\_\_(self, name = None, age = None):

self.name = name

self.age = age

>>>P = Person("小明", "24")

>>>P.sex = "male"

>>>P.sex

'male'

**2、运行的过程中给类绑定(添加)属性**

>>> Person.sex = None#给类Person添加一个属性

>>>P1 = Person("小丽", "25")

>>>print(P1.sex) #如果P1这个实例对象中没有sex属性的 话，那么就会访问它的类属性

None#可以看到没有出现异常

**3、运行的过程中给类绑定(添加)方法**

import types

#定义了一个类

classPerson(object):

num = 0

def\_\_init\_\_(self, name = None, age = None):

self.name = name

self.age = age

defeat(self):

print("eat food")

#定义一个实例方法

defrun(self, speed):

print("%s在移动, 速度是 %d km/h"%(self.name, speed))

#定义一个类方法

@classmethod

deftestClass(cls):

cls.num = 100

#定义一个静态方法

@staticmethod

deftestStatic():

print("---static method----")

#创建一个实例对象

P = Person("老王", 24)

#调用在class中的方法

P.eat()

#给这个对象添加实例方法

P.run = types.MethodType(run, P)

#调用实例方法

P.run(180)

#给Person类绑定类方法

Person.testClass = testClass

#调用类方法

print(Person.num)

Person.testClass()

print(Person.num)

#给Person类绑定静态方法

Person.testStatic = testStatic

#调用静态方法

Person.testStatic()

**4、运行的过程中删除属性、方法**

删除的方法:

1、del 对象.属性名

2、delattr(对象, "属性名")

通过以上例子可以得出一个结论：相对于动态语言，静 态 语言具有严谨性！所以，玩动态语言的时候，小心动态 的坑！那么怎么避免这种情况呢？ 请使用\_\_slots\_\_

**5、\_\_slots\_\_**

**动态语言：可以在运行的过程中，修改代码**

**静态语言：编译时已经确定好代码，运行过程中不能修改**

在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来 限制该class实例能添加的属性：

>>>classPerson(object):

\_\_slots\_\_ = ("name", "age")

>>>P = Person()

>>>P.name = "老王"

>>>P.age = 20

>>>P.score = 100

Traceback (most recent call last):

File "<pyshell#3>", line 1, in<module>

AttributeError: Person instance has no attribute 'score'

**使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类 实例起作用，对继承的子类是不起作用的**

**<13>元类**

**1、类也是对象**

你可以将它赋值给一个变量

你可以拷贝它

你可以为它增加属性

你可以将它作为函数参数进行传递

**2. 动态地创建类**

def choose\_class(name):

if name == 'foo':

class Foo(object):

pass

return Foo # 返回的是类，不是类的实例

else:

class Bar(object):

pass

return Bar

MyClass = choose\_class('foo')

print MyClass # 函数返回的是类，不是类的实例

<class '\_\_main\_\_'.Foo>

print MyClass() # 你可以通过这个类创建类实例

<\_\_main\_\_.Foo object at 0x89c6d4c>

**3、使用type创建类**

type(类名, 由父类名称组成的元组（针对继承的情况，可 以为空），包含属性的字典（名称和值）)

Test2 = type("Test2",(),{}) #定了一个Test2类

print(Test2())#创建了一个Test2类的实例对象

<\_\_main\_\_.Test2 at 0x10d406b38>

**4. 使用type创建带有属性的类**

>>>Foo = type('Foo', (), {'bar':True})

***等同于：***

>>>classFoo(object):

… bar = True

>>> print Foo

<class '\_\_main\_\_.Foo'>

>>> print Foo.bar

True

>>> f = Foo()

>>> print f

<\_\_main\_\_.Foo object at 0x8a9b84c>

>>> print f.bar

True

>>>classFooChild(Foo):

… pass

可以写成：

>>> FooChild = type('FooChild', (Foo,),{})

>>> print FooChild

<class '\_\_main\_\_.FooChild'>

>>> print FooChild.bar # bar属性是由Foo继承而来

True

**type的第2个参数，元组中是父类的名字，而不是字符串**

**添加的属性是类属性，并不是实例属性**

**5、使用type创建带有方法的类**

defecho\_bar(self):#定义了一个普通的函数

print(self.bar)

#让FooChild类中的echo\_bar属性，指向了上面定义的函数

FooChild = type('FooChild', (Foo,), {'echo\_bar': echo\_bar})

hasattr(Foo, 'echo\_bar') #判断Foo类中，是否有echo\_bar这个属性

False

#判断FooChild类中，是否有echo\_bar这个属性 hasattr(FooChild, 'echo\_bar')

True

my\_foo = FooChild()

my\_foo.echo\_bar()

True

**添加静态方法**

@staticmethod

deftestStatic():

print("static method ....")

Foochild = type('Foochild', (Foo,), {"echo\_bar":echo\_bar, "testStatic":testStatic})

fooclid = Foochild()

fooclid.testStatic

>>><function \_\_main\_\_.testStatic>

fooclid.testStatic()

>>>static method ....

fooclid.echo\_bar()

>>>True

**添加类方法**

@classmethod

deftestClass(cls):

print(cls.bar)

Foochild = type('Foochild', (Foo,), {"echo\_bar":echo\_bar, "testStatic":testStatic, "testClass":testClass})

fooclid = Foochild()

fooclid.testClass()

>>>True

**6、到底什么是元类**

元类就是用来创建这些类（对象）的，元类就是类的类， Python中所有的东西，注意，我是指所有的东西——都是对 象。这包括整数、字符串、函数以及类。它们全部都是对象， 而且它们都是从一个类创建而来，这个类就是type。

>>> age = 35

>>>age.\_\_class\_\_

<type 'int'>

>>> name = 'bob'

>>>name.\_\_class\_\_

<type 'str'>

>>> def foo(): pass

>>>foo.\_\_class\_\_

<type 'function'>

>>> class Bar(object): pass

>>> b = Bar()

>>> b.\_\_class\_\_

<class '\_\_main\_\_.Bar'>

**7. \_\_metaclass\_\_属性，自定义元类**

**python2中：**

#-\*- coding:utf-8 -\*-

defupper\_attr(future\_class\_name, future\_class\_parents, future\_class\_attr):

#遍历属性字典，把不是\_\_开头的属性名字变为大写

newAttr = {}

forname,valuein future\_class\_attr.items():

ifnotname.startswith("\_\_"):

newAttr[name.upper()] = value

#调用type来创建一个类

returntype(future\_class\_name, future\_class\_parents, newAttr)

classFoo(object):

\_\_metaclass\_\_ = upper\_attr #设置Foo类的元类为upper\_attr

bar = 'bip'

print(hasattr(Foo, 'bar'))

print(hasattr(Foo, 'BAR'))

f = Foo()

print(f.BAR)

**python3中:**

#-\*- coding:utf-8 -\*-

defupper\_attr(future\_class\_name, future\_class\_parents, future\_class\_attr):

#遍历属性字典，把不是\_\_开头的属性名字变为大写

newAttr = {}

forname,valuein future\_class\_attr.items():

ifnotname.startswith("\_\_"):

newAttr[name.upper()] = value

#调用type来创建一个类

returntype(future\_class\_name, future\_class\_parents, newAttr)

classFoo(object, metaclass=upper\_attr):

bar = 'bip'

print(hasattr(Foo, 'bar'))

print(hasattr(Foo, 'BAR'))

f = Foo()

print(f.BAR)

就元类本身而言，它们其实是很简单:

1、拦截类的创建

2、修改类

3、返回修改之后的类

**<14>垃圾回收**

**1. 小整数对象池**

Python 对小整数的定义是 [-5, 257) 这些整数对象是 提前建立好的，不会被垃圾回收

**2. 大整数对象池**

每一个大整数，均创建一个新的对象。

**3. intern机制**

intern机制让字符串只占用一个”HelloWorld”所占的内存 空间。靠引用计数去维护何时释放。

**4. 垃圾回收机制**

**1、导致引用计数+1的情况**

**1.**对象被创建，例如a=23

**2.**对象被引用，例如b=a

**3.**对象被作为参数，传入到一个函数中，例如func(a)

**4.**对象作为一个元素，存储在容器中，例如list1=[a,a]

**2、导致引用计数-1的情况**

**1.**对象的别名被显式销毁，例如del a

**2.**对象的别名被赋予新的对象，例如a=24

**3.**一个对象离开它的作用域，例如f函数执行完毕时， func函数中的局部变量（全局变量不会）

**4.**对象所在的容器被销毁，或从容器中删除对象

**3、查看一个对象的引用计数**

import sys

a = "hello world"

sys.getrefcount(a)

可以查看a对象的引用计数，但是比正常计数大1，因为 调用函数的时候传入a，这会让a的引用计数+1

**<15>内建属性**

子类没有实现\_\_init\_\_方法时，默认自动调用父类的。 如定义\_\_init\_\_方法时，需自己手动调用父类的\_\_init\_\_方法

| **常用专有属性** | **说明** | **触发方式** |
| --- | --- | --- |
| \_\_init\_\_ | 构造初始化函数 | 创建实例后,赋值时使用,在\_\_new\_\_后 |
| \_\_new\_\_ | 生成实例所需属性 | 创建实例时 |
| \_\_class\_\_ | 实例所在的类 | 实例.\_\_class\_\_ |
| \_\_str\_\_ | 实例字符串表示,可读性 | print(类实例),如没实现，使用repr结果 |
| \_\_repr\_\_ | 实例字符串表示,准确性 | 类实例回车或者 print(repr(类实例)) |
| \_\_del\_\_ | 析构 | del删除实例 |
| \_\_dict\_\_ | 实例自定义属性 | vars(实例.\_\_dict\_\_) |
| \_\_doc\_\_ | 类文档,子类不继承 | help(类或实例) |
| \_\_getattribute\_\_ | 属性访问拦截器 | 访问实例属性时 |
| \_\_bases\_\_ | 类的所有父类构成元素 | 类名.\_\_bases\_\_ |

\_\_getattribute\_\_例子:

classItcast(object):

def\_\_init\_\_(self,subject1):

self.subject1 = subject1

self.subject2 = 'cpp'

#属性访问时拦截器，打log

def\_\_getattribute\_\_(self,obj):

if obj == 'subject1':

print('log subject1')

return'redirect python'

else: #测试时注释掉这2行，将找不到subject2

returnobject.\_\_getattribute\_\_(self,obj)

defshow(self):

print('this is Itcast')

s = Itcast("python")

print(s.subject1)

print(s.subject2)

**\_\_getattribute\_\_的坑**

classPerson(object):

def\_\_getattribute\_\_(self,obj):

print("---test---")

ifobj.startswith("a"):

return"hahha"

else:

return self.test

deftest(self):

print("heihei")

t.Person()

t.a #返回hahha

t.b #会让程序死掉

#原因是：当t.b执行时，会调用Person类中定义的 \_\_getattribute\_\_方法，但是在这个方法的执行过程中if条件不 满足，所以程序执行else里面的代码，即returnself.test 问题就在这，因为return 需要把self.test的值返回，那 么首先要获取self.test的值，因为self此时就是t这个对 象，所以self.test就是t.test 此时要获取t这个对象的test 属性，那么就会跳转到 \_\_getattribute\_\_方法去执行，即此时产 生了递归调用，由于这个递归过程中没有判断什么时候退出，所 以这个程序会永无休止的运行下去，又因为每次调用函数，就需要保 存一些数据，那么随着调用的次数越来越 多，最终内存吃光，所以 程序崩溃

# 注意：以后不要在\_\_getattribute\_\_方法中调用self.xxxx

**<16>内建函数**

启动python解释器，输入dir(\_\_builtins\_\_), 可以看到很多python解释器启动后默认加载的属性和函数，这些函数称之为内建函数， 这些函数因为在编程时使用较多，cpython解释器用c语言实现了这些函数，启动解释器 时默认加载。

**1、range函数:**

range(stop) -> list of integers

range(start, stop[, step]) -> list of integers

start:计数从start开始。默认是从0开始。例如range （5）等价于range（0， 5）;

stop:到stop结束，但不包括stop.例如：range（0， 5） 是[0, 1, 2, 3, 4]没有5

step:每次跳跃的间距，默认为1。例如：range（0， 5） 等价于 range(0, 5, 1)

**2、map函数：**

map函数会根据提供的函数对指定序列做映射

map(...)

map(function, sequence[, sequence, ...]) -> list

function:是一个函数

sequence:是一个或多个序列,取决于function需要几个 参数

返回值是一个list

#函数需要一个参数

map(lambda x: x\*x, [1, 2, 3])

#结果为:[1, 4, 9]

#函数需要两个参数

map(lambda x, y: x+y, [1, 2, 3], [4, 5, 6])

#结果为:[5, 7, 9]

deff1( x, y ):

return (x,y)

l1 = [ 0, 1, 2, 3, 4, 5, 6 ]

l2 = [ 'Sun', 'M', 'T', 'W', 'T', 'F', 'S' ]

l3 = map( f1, l1, l2 )

print(list(l3))

**3、filter函数：**

filter函数会对指定序列执行过滤操作

filter(...)

filter(function or None, sequence) -> list, tuple, or string

function:接受一个参数，返回布尔值True或False

sequence:序列可以是str，tuple，list

filter(lambda x: x%2, [1, 2, 3, 4])

[1, 3]

filter(None, "she")

'she'

filter函数会对序列参数sequence中的每个元素调用 function函数，最后返回的结果包含调用结果为True的元 素。

**4、reduce函数：**

reduce函数会对参数序列中元素进行累积

reduce(...)

reduce(function, sequence[, initial]) -> value

function:该函数有两个参数

sequence:序列可以是str，tuple，list

initial:固定初始值

reduce依次从sequence中取一个元素，和上一次调用 function的结果做参数再次调用function。 第一次调用 function时，如果提供initial参数，会以sequence中的第 一个元素和initial 作为参数调用function，否则会以序列 sequence中的前两个元素做参数调用function。 注意 function函数不能为None。

reduce(lambda x, y: x+y, [1,2,3,4])

10

reduce(lambda x, y: x+y, [1,2,3,4], 5)

15

reduce(lambda x, y: x+y, ['aa', 'bb', 'cc'], 'dd')

'ddaabbcc'

在Python3里,reduce函数已经被从全局名字空间里移除 了, 它现在被放置在fucntools模块里用的话要先引 入： from functools import reduce

**5、sorted函数**

sorted(...)

sorted(iterable, cmp=None, key=None, reverse=False)

--> newsorted list

sorted函数会对传入的iterable排序，并返回一个排序 完成的列表，参数reverse=True时，则为逆序。

**<17>集合**

集合与之前列表、元组类似，可以存储多个数据，但是这些数据是不重复的，集合对象还支持union(联合),intersection(交), difference(差)和sysmmetric\_difference(对称差集)等数学运算。

>>> x = set('abcd')

>>> x

{'c', 'a', 'b', 'd'}

>>> type(x)

<class 'set'>

>>>

>>> y = set(['h','e','l','l','o'])

>>> y

{'h', 'e', 'o', 'l'}

>>>

>>> z = set('spam')

>>> z

{'s', 'a', 'm', 'p'}

>>>

>>> y&z #交集

set()

>>> x&z #交集

{'a'}

>>>

>>> x|y #并集

{'a', 'e', 'd', 'l', 'c', 'h', 'o', 'b'}

>>>

>>> x-y #差集

{'c', 'a', 'b', 'd'}

>>>

>>> x^z #对称差集(在x或z中，但不会同时出现在二者中)

{'m', 'd', 's', 'c', 'b', 'p'}

>>>

>>> len(x)

4

>>> len(y)

4

>>> len(z)

4

**<18>functools**

functools，一些工具函数放在此包里。

import functools

dir(functools)

**1、partial函数(偏函数)**

把一个函数的某些参数设置默认值，返回一个新的函 数，调用这个新函数会更简单。

import functools

defshowarg(\*args, \*\*kw):

print(args)

print(kw)

p1=functools.partial(showarg, 1,2,3)

p1()

p1(4,5,6)

p1(a='python', b='itcast')

p2=functools.partial(showarg, a=3,b='linux')

p2()

p2(1,2)

p2(a='python', b='itcast')

**2、wraps函数**

使用装饰器时，有一些细节需要被注意。例如，被装饰 后的函数其实已经是另外一个函数了（函数名等函数属性 会发生改变）。

defnote(func):

defwrapper():

print('note something')

returnfunc()

return wrapper

@note

deftest():

print('I am test')

test()

print(test.\_\_doc\_\_)

运行结果：

note something

I am test

wrapper function

import functools

defnote(func):

@functools.wraps(func)

defwrapper():

print('note something')

returnfunc()

return wrapper

@note

deftest():

print('I am test')

test()

print(test.\_\_doc\_\_)

运行结果：

note something

I am test

test function

**<19>模块进阶**

**常用标准库：**

| **标准库** | **说明** |
| --- | --- |
| builtins | 内建函数默认加载 |
| os | 操作系统接口 |
| sys | Python自身的运行环境 |
| functools | 常用的工具 |
| json | 编码和解码 JSON 对象 |
| logging | 记录日志，调试 |
| multiprocessing | 多进程 |
| threading | 多线程 |
| copy | 拷贝 |
| time | 时间 |
| datetime | 日期和时间 |
| calendar | 日历 |
| hashlib | 加密算法 |
| random | 生成随机数 |
| re | 字符串正则匹配 |
| socket | 标准的 BSD Sockets API |
| shutil | 文件和目录管理 |
| glob | 基于文件通配符搜索 |

**hashlib：（哈希）**

import hashlib

import datetime

KEY\_VALUE = 'Itcast'

now = datetime.datetime.now()

m = hashlib.md5()#创建hash对象，md5:(message-Digest Algorithm 5)消息摘要算法,得出一个128位的密文

str = '%s%s' % (KEY\_VALUE,now.strftime("%Y%m%d"))

m.update(str.encode('utf-8'))#更新哈希对象以字符串参数

value = m.hexdigest()#返回十六进制数字字符串

print(value)

**常用扩展库：**

| **扩展库** | **说明** |
| --- | --- |
| requests | 使用的是 urllib3，继承了urllib2的所有特性 |
| urllib | 基于http的高层库 |
| scrapy | 爬虫 |
| beautifulsoup4 | HTML/XML的解析器 |
| celery | 分布式任务调度模块 |
| redis | 缓存 |
| Pillow(PIL) | 图像处理 |
| xlsxwriter | 仅写excle功能,支持xlsx |
| xlwt | 仅写excle功能,支持xls ,2013或更早版office |
| xlrd | 仅读excle功能 |
| elasticsearch | 全文搜索引擎 |
| pymysql | 数据库连接库 |
| mongoengine/pymongo | mongodbpython接口 |
| matplotlib | 画图 |
| numpy/scipy | 科学计算 |
| django/tornado/flask | web框架 |
| xmltodict | xml 转 dict |
| SimpleHTTPServer | 简单地HTTP Server,不使用Web框架 |
| gevent | 基于协程的Python网络库 |
| fabric | 系统管理 |
| pandas | 数据处理库 |
| scikit-learn | 机器学习库 |

**<20>调试**

**pdb是基于命令行的调试工具**

| **命令** | **简写命令** | **作用** |
| --- | --- | --- |
| break | b | 设置断点 |
| continue | c | 继续执行程序 |
| list | l | 查看当前行的代码段 |
| step | s | 进入函数 |
| return | r | 执行代码直到从当前函数返回 |
| quit | q | 中止并退出 |
| next | n | 执行下一行 |
| print | p | 打印变量的值 |
| help | h | 帮助 |
| args | a | 查看传入参数 |
|  | 回车 | 重复上一条命令 |
| break | b | 显示所有断点 |
| break lineno | b lineno | 在指定行设置断点 |
| break file:lineno | b file:lineno | 在指定文件的行设置断点 |
| clear num |  | 删除指定断点 |
| bt |  | 查看函数调用栈帧 |

**执行时调试：**

python -m pdb some.py