
Table of Contents

传智播客—python高级进阶	1.1
[函数进阶]	1.2
程序运行原理	1.2.1
闭包	1.2.2
装饰器	1.2.3
迭代器	1.2.4
生成器	1.2.5
内建函数	1.2.6
functools函数	1.2.7
类进阶	1.3

python高级课程

传智播客python学院

- 函数进阶
- 类进阶
- 模块进阶
- 调试
- 编码风格
- 其它

程序运行原理

一个程序是如何运行起来的？比如下面的代码

```
#othermodule.py
def add(a, b):
    return a + b

#mainrun.py

import othermodule

a = ['itcast', 1, 'python']
a = 'other string'

def func():
    a = 1
    b = 257
    print(a + b)

print(a)

if __name__ == '__main__':
    func()
    res = othermodule.add(1, 2)
    print(res)
```

运行上述程序

```
python@ubuntu:~/workspace/test/code$ tree
.
└── mainrun.py
    └── othermodule.py

0 directories, 2 files
python@ubuntu:~/workspace/test/code$ python mainrun.py
other string
258
3
python@ubuntu:~/workspace/test/code$ tree
.
└── mainrun.py
    └── othermodule.py
        └── othermodule.pyc

0 directories, 3 files
```

分析

```
#othermodule.py
def add(a, b):
    return a + b

#mainrun.py
5.搜索路径(从哪个范围)和路径搜索(依次查找)
import othermodule

a = ['itcast', 1, 'python']
a = 'other string'

def func():
    a = 1
    b = 257
    print(a + b)

print(a)

if __name__ == '__main__':
    func()
    res = othermodule.add(1, 2)
    print(res)
```

4.def 创建一个函数对象，然后将 add 这个名称符号绑定到这个函数上
3.变量赋值与垃圾回收(动态类型)
 小整数[-5,257]共用对象，常驻内存
 单个字符共用对象，常驻内存
 单个单词，不可修改，默认开启intern机制，共用对象，引用计数为0，则销毁
 字符串(含有空格)，不可修改，没开启intern机制，不共用对象，引用计数为0，销毁
 大整数不共用内存，引用计数为0，销毁

2.查询变量顺序LEG_B
 Local：本地
 Enclosing function：外部嵌套函数的命名空间
 Global：全局
 Builtins：内建模块命名空间

1.模块的内建__name__属性，主模块其值为__main__，导入模块其值为模块名
6.pyc文件
 创建时间，py文件比pyc文件新，则从新生成pyc
 magic num，做运行前版本检测，版本不同从新生成pyc
 PyCodeObject 对象，源代码中的字符串，常量值，字节码指令，原始代码行号的对应关系等

运行上述程序

```
python@ubuntu:~/workspace/test/code$ tree
.
└── mainrun.py
    └── othermodule.py

0 directories, 2 files
python@ubuntu:~/workspace/test/code$ python mainrun.py
other string
258
3
python@ubuntu:~/workspace/test/code$ tree
.
└── mainrun.py
    └── othermodule.py
        └── othermodule.pyc
```

7. import作用
 执行一遍导入模块的代码创建模块内函数和类
 生成pyc文件，下次不需编译
 module中符号与当前模块符号链接
 缓存module，下次再import时只需返回此对象

8.如何生成pyc文件
 import过的模块
 python -m py_compile xx_module.py
 使用dis模块(略)

LEG_B 规则

Python 使用 LEGB 的顺序来查找一个符号对应的对象

```
locals -> enclosing function -> globals -> builtins
```

- locals，当前所在命名空间(如函数、模块)，函数的参数也属于命名空间内的变量
- enclosing，外部嵌套函数的命名空间(闭包中常见)

```
def fun1():
    a = 10
def fun2():
    # a 位于外部嵌套函数的命名空间
    print(a)
```

- globals，全局变量，函数定义所在模块的命名空间

```
a = 1
def fun():
    # 需要通过 global 指令来声明全局变量
    global a
    # 修改全局变量，而不是创建一个新的 local 变量
    a = 2
```

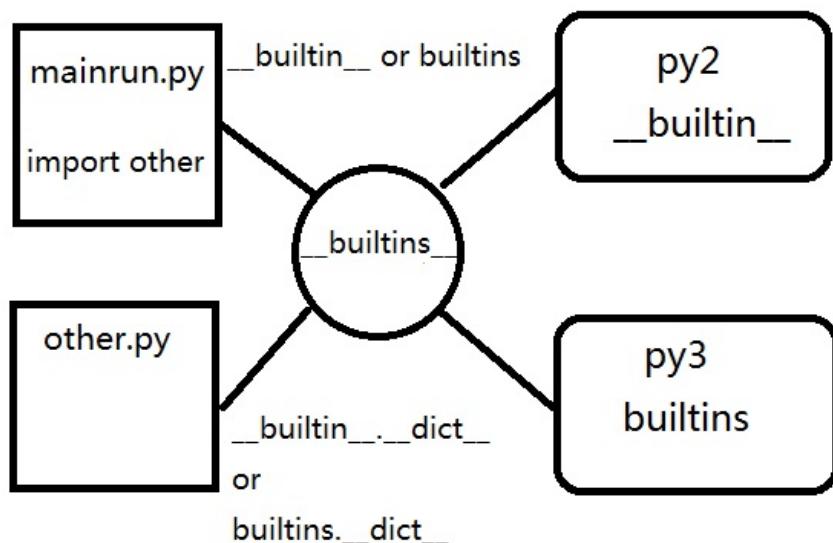
- builtins，内建模块的命名空间。

Python 在启动的时候会自动为我们载入很多内建的函数、类，
比如 dict, list, type, print, 这些都位于 `__builtins__` 模块中，
可以使用 `dir(__builtins__)` 来查看。
这也是为什么我们在没有 import 任何模块的情况下，
就能使用这么多丰富的函数和功能了。

在Python中，有一个内建模块，该模块中有一些常用函数；在Python启动后，
且没有执行程序员所写的任何代码前，Python会首先加载该内建函数到内存。
另外，该内建模块中的功能可以直接使用，不用在其前添加内建模块前缀，
其原因是对函数、变量、类等标识符的查找是按LEGB法则，其中B即代表内建模块。
比如：内建模块中有一个abs()函数，其功能求绝对值，如abs(-20)将返回20。

Python2.X版本中，内建模块被命名为 `__builtin__`
Python3.X版本中，更名为 `builtins`
Python2.X和Python3.X 是对内建模块一个引用 `__builtins__`

运行 `python mainrun.py`



dir 函数

Python 的内置函数 `dir` 可以用来查看一个命名空间下的所有名字符号。一个用处是查看一个命名空间的所有属性和方法（这里的命名空间就是指类、函数、module）。

比如，查看当前的命名空间，可以使用 `dir()`，查看 `sys` 模块，可以使用 `dir(sys)`

垃圾回收

小整数对象池

整数在程序中的使用非常广泛，Python为了优化速度，使用了小整数对象池，避免为整数频繁申请和销毁内存空间。

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。在一个 Python 的程序中，所有位于这个范围内的整数使用的都是同一个对象。

同理，单个字母也是这样的。

但是当定义2个相同的字符串时，引用计数为0，触发垃圾回收

大整数对象池

每一个大整数，均创建一个新的对象。

```
In [15]: b=1356
In [16]: id(b)
Out[16]: 39430264
In [17]: a=1356
In [18]: id(a)
Out[18]: 38895736
In [19]: type(a)
Out[19]: int
In [20]: b=a
In [21]: id(b)
Out[21]: 38895736
```

intern机制

```
a1="HelloWorld"
a2="HelloWorld"
a3="HelloWorld"
a4="HelloWorld"
a5="HelloWorld"
a6="HelloWorld"
a7="HelloWorld"
a8="HelloWorld"
a9="HelloWorld"
```

python会不会创建9个对象呢？在内存中会不会开辟9个”HelloWorld”的内存空间呢？想一下，如果是这样的话，我们写10000个对象，比如a1=”HelloWorld”.....a1000=”HelloWorld”，那他岂不是开辟了1000个”HelloWorld”所占的内存空间了呢？如果真这样，内存不就爆了吗？所以python中有这样一个机制——intern机制，让他只占用一个”HelloWorld”所占的内存空间。靠引用计数去维护何时释放。

```
In [22]: a='abcde'  
In [23]: b='abcde'  
In [24]: id(b)  
Out[24]: 139929937277168  
  
In [25]: id(a)  
Out[25]: 139929937277168  
  
In [26]: del(a)  
  
In [27]: del(b)  
  
In [28]: e='abcde'  
  
In [29]: id(e)  
Out[29]: 139929937277360
```

小结

- 小整数[-5,257)共用对象，常驻内存
- 单个字符共用对象，常驻内存
- 单个单词，不可修改，默认开启intern机制，共用对象，引用计数为0，则销毁

```
In [53]: a='hello'  
  
In [54]: b='hello'  
  
In [55]: id(a)  
Out[55]: 140086726350768  
  
In [56]: id(b)  
Out[56]: 140086726350768  
  
In [57]: del(a)  
  
In [58]: del(b)  
  
In [59]: c='hello'  
  
In [60]: id(c)  
Out[60]: 140086726350336
```

- 字符串（含有空格），不可修改，没开启intern机制，不共用对象，引用计数为0，销毁

```
In [44]: s1='hello  itcast'  
In [45]: s2='hello  itcast'  
In [46]: id(s1)  
Out[46]: 140086705709216  
  
In [47]: id(s2)  
Out[47]: 140086705709440  
  
In [48]: s3=intern(s1)  
  
In [49]: print s3  
hello  itcast  
  
In [50]: id(s3)  
Out[50]: 140086705709216
```

- 大整数不共用内存，引用计数为0，销毁

```
In [15]: b=1356  
  
In [16]: id(b)  
Out[16]: 39430264  
  
In [17]: a=1356  
  
In [18]: id(a)  
Out[18]: 38895736  
  
In [19]: type(a)  
Out[19]: int  
  
In [20]: b=a  
  
In [21]: id(b)  
Out[21]: 38895736
```

备注：数值类型和字符串类型在 Python 中都是不可变的，这意味着你无法修改这个对象的值，每次对变量的修改，实际上是创建一个新的对象。

```
In [62]: id(a)
Out[62]: 37122392

In [63]: a+=1

In [64]: id(a)
Out[64]: 37122368

In [65]: b='hello'

In [66]: id(b)
Out[66]: 140086726350336

In [67]: b='itcat'

In [68]: id(b)
Out[68]: 140086726351392
```

def 指令

`def func()` 在字节码指令中就是 `MAKE_FUNCTION`。Python 是动态语言，`def` 实际上是执行一条指令，用来创建函数（`class` 则是创建类的指令），而不仅仅是个语法关键字。函数并不是事先创建好的，而是执行到的时候才创建的。

`def func()` 将会创建一个名称为 `func` 的函数对象。实际上是先创建一个函数对象，然后将 `func` 这个名称符号绑定到这个函数上。

import 搜索路径

```
import sys
sys.path
```

```
In [2]: sys.path
Out[2]:
['',
 '/usr/bin',
 '/usr/local/lib/python2.7/dist-packages/setuptools-21.0.0-py2.7.egg',
 '/usr/local/lib/python2.7/dist-packages/virtualenvwrapper-4.7.1-py2.7.egg',
 '/usr/local/lib/python2.7/dist-packages/stevedore-1.13.0-py2.7.egg',
 '/usr/local/lib/python2.7/dist-packages/virtualenv_clone-0.2.6-py2.7.egg',
 '/usr/local/lib/python2.7/dist-packages/six-1.10.0-py2.7.egg',
 '/usr/local/lib/python2.7/dist-packages/pbr-1.9.1-py2.7.egg',
 '/usr/lib/python2.7',
 '/usr/lib/python2.7/plat-x86_64-linux-gnu',
 '/usr/lib/python2.7/lib-tk',
 '/usr/lib/python2.7/lib-old',
 '/usr/lib/python2.7/lib-dynload',
 '/usr/local/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages/gtk-2.0',
 '/usr/lib/python2.7/dist-packages/IPython/extensions',
 '/home/python/.ipython']
```

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

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
```

设置Linux下导入模块路径

```
echo $PYTHONPATH
export PYTHONPATH=$PYTHONPATH:/home/itcast/xxx'
```

重新导入模块

模块被导入后，`import module` 不能重新导入模块，重新导入需用

```
reload(module)
```

pyc文件

pyc 文件是 PyCodeObject 对象在硬盘上的表现形式。生成pyc文件:

```
python -m py_compile xx_module.py
```

pyc文件三大作用

- 创建时间，py文件比pyc文件新，则从新生成pyc
- magic num做运行前版本检测，版本不同从新生成pyc
- PyCodeObject 对象

在运行期间，编译结果也就是 PyCodeObject 对象，只会存在于内存中，而当这个模块的 Python 代码执行完后，就会将编译结果保存到了 pyc 文件中，这样下次就不用编译，直接加载到内存中。

这个 PyCodeObject 对象包含了 Python 源代码中的字符串，常量值，以及通过语法解析后编译生成的字节码指令。PyCodeObject 对象还会存储这些字节码指令与原始代码行号的对应关系，这样当出现异常时，就能指明位于哪一行的代码。

import 指令

import 指令是用来载入 module 的，如果需要，也会顺道做编译的事。但 import 指令，还会做一件重要的事情就是把 import 的那个 module 的代码执行一遍，这件事情很重要。Python 是解释执行的，连函数都是执行的时候才创建的。如果不把那个 module 的代码执行一遍，那么 module 里面的函数都没法创建，更别提去调用这些函数了。

执行代码的另外一个重要作用，就是在这个 module 的命名空间中，创建模块内定义的函数和各种对象的符号名称（也就是变量名），并将其绑定到对象上，这样其他 module 才能通过变量名来引用这些对象。

Python 虚拟机还会将已经 import 过的 module 缓存起来，放到一个全局 module 集合 `sys.modules` 中。这样做有一个好处，即如果程序的在另一个地方再次 import 这个模块，Python 虚拟机只需要将全局 module 集合中缓存的那个 module 对象返回即可。

备注：如果一个正在运行的程序，模块内容修改，重新import无法更新，需要reload模块

闭包

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

def counter(start=0):
    count=[start]
    def incr():
        count[0] += 1
        return count[0]
    return incr
```

启动python解释器

```
>>>import closure
>>>c1=closure.counter(5)
>>>print c1()
6
>>>print c1()
7
>>>c2=closure.counter(100)
>>>print c2()
101
>>>print c2()
102
```

按值传递参数和按引用传递参数

- 1.按值传递,单个变量
- 2.按引用传递,列表

闭包思考：

- 1.闭包似有化了变量，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

装饰器

装饰器(decorator)里引入通用功能处理

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

例1:无参数的函数

```
#decorator.py

from time import ctime, sleep

def timefun(func):
    def wrappedfunc():
        print("%s called at %s"%(func.__name__, ctime()))
        return func()
    return wrappedfunc

@timefun
def foo():
    print("I am foo")

foo()
sleep(2)
foo()
```

上面代码理解装饰器执行行为可理解成

```
foo = timefun(foo)
#foo先作为参数赋值给func后, foo接收指向timefun返回的wrappedfunc
foo()
#调用foo(), 即等价调用wrappedfunc()
#内部函数wrappedfunc被引用, 所以外部函数的func变量(自由变量)并没有释放
#func里保存的是原foo函数对象
```

例2:被装饰的函数有参数

```
#decorator2.py
from time import ctime, sleep

def timefun(func):
    def wrappedfunc(a, b):
        print("%s called at %s"%(func.__name__, ctime()))
        print(a, b)
        return func(a, b)
    return wrappedfunc

@timefun
def foo(a, b):
    print(a+b)

foo(3,5)
sleep(2)
foo(2,4)
```

例3: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

def timefun_arg(pre="hello"):
    def timefun(func):
        def wrappedfunc():
            print("%s called at %s %s"%(func.__name__, ctime(), pre))
            return func()
        return wrappedfunc
    return timefun

@timefun_arg("itcast")
def foo():
    print("I am foo")

@timefun_arg("xwp")
def too():
    print("I am too")

foo()
sleep(2)
foo()

too()
sleep(2)
too()
```

可以理解为

```
foo() == timefun_arg("itcast")(foo)()
```

例4: 装饰器和闭包混用(难)

```
#coding=utf-8

from time import time

def logged(when):
    def log(f, *args, **kargs):
        print("fun:%s args:%r kargs:%r" %(f, args, kargs))
        #%r字符串的同时，显示原有对象类型

    def pre_logged(f):
        def wrapper(*args, **kargs):
            log(f, *args, **kargs)
            return f(*args, **kargs)
        return wrapper

    def post_logged(f):
        def wrapper(*args, **kargs):
            now=time()
            try:
                return f(*args, **kargs)
            finally:
                log(f, *args, **kargs)
                print("time delta: %s" %(time()-now))
        return wrapper
    try:
        return {"pre":pre_logged, "post":post_logged}[when]
    except KeyError, e:
        raise ValueError(e), 'must be "pre" or "post"'

@logged("post")
def fun(name):
    print("Hello, %s" %(name))

fun("itcastcpp!")
```

例5: 类装饰器 (扩展, 非重点)

要定义类型的时候, 实现**call**函数, 这个类型就成为可调用的。可以把这个类的对象当作函数来使用。

```
class Itcast(object):
    def __init__(self, func):
        super(Itcast, self).__init__()
        self._func = func

    def __call__(self):
        print 'class Itcast'
        self._func()

@Itcast
def showpy():
    print 'showpy'

showpy()
```

迭代器

在Python中，很多对象都是可以通过for语句来直接遍历的，例如list、string、dict等等，这些对象都可以被称为可迭代对象。至于说哪些对象是可以被迭代访问的，就要了解一下迭代器相关的知识了。

迭代器仅是一容器对象，它实现了迭代器协议。它有两个基本方法：

1. `next`方法, 返回容器的下一个元素
2. `iter`方法,返回迭代器自身

```
In [11]: num=[1,2,3,4,5]
In [12]: num.next()
-----
AttributeError                                 Traceback (most recent call last)
<ipython-input-12-c3cc5caa8523> in <module>()
----> 1 num.next()

AttributeError: 'list' object has no attribute 'next'

In [13]: n=iter(num)

In [14]: n.next()
Out[14]: 1

In [15]: n.next()
Out[15]: 2
```

`next()`方法返回容器的下一个元素，在结尾时引发`StopIteration`异常。

对于可迭代对象，可以使用内建函数`iter()`来获取它的迭代器对象.

生成器

生成器是这样一个函数，它记住上一次返回时在函数体中的位置。对生成器函数的第二次（或第 n 次）调用跳转至该函数中间，而上次调用的所有局部变量都保持不变。

生成器不仅“记住”了它数据状态；生成器还“记住”了它在流控制构造（在命令式编程中，这种构造不只是数据值）中的位置。生成器的特点：

1. 生成器是一个函数，而且函数的参数都会保留。
2. 迭代到下一次的调用时，所使用的参数都是第一次所保留下的，即是说，在整个所有函数调用的参数都是第一次所调用时保留的，而不是新创建的
3. 节约内存

例子：执行到yield时，gen函数作用暂时保存，返回x的值；tmp接收下次c.send("python")，send发送过来的值，c.next()等价c.send(None)

```
#generation.py
def gen():
    for x in xrange(4):
        tmp = yield x
        if tmp == "hello":
            print "world"
        else:
            print "itcastcpp ", str(tmp)
```

执行如下

```
>>>from generation import gen
>>>c=gen()
>>>c.next()
0
>>>c.next()
itcastcpp None
1
>>>c.send("python")
itcastcpp python
2
```

对比

当需要一个非常大的列表时，为了减少内存消耗，可以使用生成器

```
class A(object):
    def __init__(self, i):
        from time import sleep, time
        sleep(i)
        print (time())
```

- []是通过遍历可迭代对象生成一个list
- ()是直接返回可迭代对象

列表形式

```
In [18]: for c in [A(i) for i in range(5)]:  
    ....:     print c  
    ....:  
1476498605.22  
1476498606.22  
1476498608.23  
1476498611.23  
1476498615.24  
<t3.A object at 0x7f8003a57f90>  
<t3.A object at 0x7f8003a57b90>  
<t3.A object at 0x7f8003af0a50>  
<t3.A object at 0x7f8002720050>  
<t3.A object at 0x7f8002720090>
```

生成器(元祖)形式

```
In [20]: for c in (A(i) for i in range(5)):  
    print c  
    ....:  
1476498629.23  
<t3.A object at 0x7f80027200d0>  
1476498630.23  
<t3.A object at 0x7f8002720110>  
1476498632.23  
<t3.A object at 0x7f80027200d0>  
1476498635.24  
<t3.A object at 0x7f8002720110>  
1476498639.25  
<t3.A object at 0x7f80027200d0>
```

小结

- 无限递归成为可能
- 极大的降低了线程或进程间上下文切换的开销
- 用户手工指定纤程调用，避免了锁开销

内建函数

Build-in Function, 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。

这些函数数量众多，不宜记忆，开发时不是都用到的，待用到时再`help(function)`, 查看如何使用，或结合百度查询即可，在这里介绍些常用的内建函数。

range

```
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)`

python2中`range`返回列表，python3中`range`返回一个迭代值。如果想得到列表,可通过`list`函数

```
a = range(5)
list(a)
```

map

```
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`

参数序列中的每一个元素分别调用`function`函数，返回包含每次`function`函数返回值的`list`。

```
#函数需要一个参数
map(lambda x: x*x, [1, 2, 3])
[1, 4, 9]

#函数需要两个参数
map(lambda x, y: x+y, [1, 2, 3], [4, 5, 6])
[5, 7, 9]

#函数为None,相当于合并参数为元祖
map(None, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])
[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]

#两个序列参数个数不一致时,个数少的补None
map(None, [1, 3, 5, 7, 9], [2, 4, 6])
[(1, 2), (3, 4), (5, 6), (7, None), (9, None)]
```

filter

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

    Return those items of sequence for which function(item) is true. If
    function is None, return the items that are true. If sequence is a tuple
    or string, return the same type, else return a list.
```

- **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。

返回值的类型和参数sequence的类型相同

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

filter(None, "she")
'she'
```

reduce

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

    Apply a function of two arguments cumulatively to the items of a sequence,
    from left to right, so as to reduce the sequence to a single value.
    For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates
    (((1+2)+3)+4)+5). If initial is present, it is placed before the items
    of the sequence in the calculation, and serves as a default when the
    sequence is empty.
```

- **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
```

sorted函数

```
sorted(...)
    sorted(iterable, cmp=None, key=None, reverse=False) --> new sorted list
```

```
In [14]: sorted([1,4,2,6,3,5])
Out[14]: [1, 2, 3, 4, 5, 6]

In [15]: sorted([1,4,2,6,3,5],reverse=1)
Out[15]: [6, 5, 4, 3, 2, 1]

In [16]: sorted(['dd','aa','cc','bb'])
Out[16]: ['aa', 'bb', 'cc', 'dd']

In [17]: sorted(['dd','aa','cc','bb'],reverse=1)
Out[17]: ['dd', 'cc', 'bb', 'aa']
```

自定义cmp比较函数，返回三种情况:

- $x < y$ 返回 -1
- $x > y$ 返回 1
- $x == y$ 返回 0

```
def cmp_ignore_case(s1, s2):  
    u1 = s1.upper()  
    u2 = s2.upper()  
    if u1 < u2:  
        return -1  
    if u1 > u2:  
        return 1  
    return 0
```

functools函数

functools 是python2.5被引入的,一些工具函数放在此包里。

python2.7中

```
In [2]: dir(functools)
Out[2]:
['WRAPPER_ASSIGNMENTS',
 'WRAPPER_UPDATES',
 '__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 'cmp_to_key',
 'partial',
 'reduce',
 'total_ordering',
 'update_wrapper',
 'wraps']
```

python3.5中

```
import functools
dir(functools)
```

python3中增加了更多工具函数，做业务开发时大多情况下用不到，此处介绍使用频率较高的2个函数。

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

def showarg(*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')
```

```
python@ubuntu:~/workspace/test$ python t6.py
(1, 2, 3)
{}
(1, 2, 3, 4, 5, 6)
{}
(1, 2, 3)
{'a': 'python', 'b': 'itcast'}
()
{'a': 3, 'b': 'linux'}
(1, 2)
{'a': 3, 'b': 'linux'}
()
{'a': 'python', 'b': 'itcast'}
```

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）。

添加后由于函数名和函数的doc发生了改变，对测试结果有一些影响，例如：

```
def note(func):
    "note function"
    def wrapper():
        "wrapper function"
        print('note something')
        return func()
    return wrapper

@note
def test():
    "test function"
    print('I am test')

test()
print(test.__doc__)
```

所以，Python的functools包中提供了一个叫wraps的装饰器来消除这样的副作用。例如：

```
import functools
def note(func):
    "note function"
    @functools.wraps(func)
    def wrapper():
        "wrapper function"
        print('note something')
        return func()
    return wrapper

@note
def test():
    "test function"
    print('I am test')

test()
print(test.__doc__)
```

类进阶

实例属性,实例对象独有的属性

类属性,类名访问类属性

实例中无同名属性时, 可访问到类属性, 当定义同名实例属性时, 则无法访问

使用实例属性去修改类属性十分危险, 原因在于实例拥有自己的属性集, 修改类属性需要使用类名, 而不是实例名。

`vars` : 查看实例内属性(自定义的属性) `dir` : 显示类属性和所有实例属性 `type` : 显示类型

类的方法

而在Python中, 方法分为三类实例方法、类方法、静态方法。代码如下:

```
class Test(object):
    def instancefun(self):
        print("InstanceFun")
        print(self)

    @classmethod
    def classfun(cls):
        print("ClassFun")
        print(cls)

    @staticmethod
    def staticfun():
        print("StaticFun")
```

- 实例方法隐含的参数为类实例`self`
- 类方法隐含的参数为类本身`cls`
- 静态方法无隐含参数, 主要为了类实例也可以直接调用静态方法。
- 类名可以调用类方法和静态方法, 但不可以调用实例方法

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)
- `__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

通过name mangling (名称改编(目的就是以防子类意外重写基类的方法或者属性), 即前面加上“单下划线”+类名,eg: `_Class__object`) 机制就可以访问private了。

```
#coding=utf-8

class Person(object):
    def __init__(self, name, age, taste):
        self.name = name
        self._age = age
        self.__taste = taste

    def showperson(self):
        print(self.name)
        print(self._age)
        print(self.__taste)

    def dowork(self):
        self._work()
        self.__away()

    def _work(self):
        print('my _work')

    def __away(self):
        print('my __away')

class Student(Person):
    def construction(self, name, age, taste):
        self.name = name
        self._age = age
        self.__taste = taste

    def showstudent(self):
        print(self.name)
        print(self._age)
        print(self.__taste)

    @staticmethod
    def testbug():
        _Bug.showbug()

#模块内可以访问，当from cur_module import *时，不导入
class _Bug(object):
    @staticmethod
    def showbug():
        print("showbug")

s1 = Student('jack', 25, 'football')
s1.showperson()
print('*'*20)

#无法访问__taste，导致报错
#s1.showstudent()
s1.construction('rose', 30, 'basketball')
```

```
s1.showperson()
print('*'*20)

s1.showstudent()
print('*'*20)

Student.testbug()
```

```
python@ubuntu:~/workspace/test$ python t10.py
jack
25
football
*****
rose
30
football
*****
rose
30
basketball
*****
showbug
python@ubuntu:~/workspace/test$
```

分析一个类

```
class Person(object):
    pass
```

python2.7中类的内建属性和方法

```
In [3]: from teachclass import Person
```

```
In [4]: dir(Person)
```

```
Out[4]:
```

```
[('__class__',  
 '__delattr__',  
 '__dict__',  
 '__doc__',  
 '__format__',  
 '__getattribute__',  
 '__hash__',  
 '__init__',  
 '__module__',  
 '__new__',  
 '__reduce__',  
 '__reduce_ex__',  
 '__repr__',  
 '__setattr__',  
 '__sizeof__',  
 '__str__',  
 '__subclasshook__',  
 '__weakref__']
```

python3.5中类的内建属性和方法

```
In [1]: from teachclass import Person

In [2]: dir(Person)
Out[2]:
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

python2.7中经典类(旧式类)的内建属性和方法

```
In [1]: from teachclass import Person

In [2]: dir(Person)
Out[2]: ['__doc__', '__module__']

In [3]: []
```

经典类(旧式类),早期如果没有要继承的父类,继承里空着不写的类

```
#py2中无继承父类,称之为经典类,py3中已默认继承object
class Person:
    pass
```

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, 优先级高于__dict__访问

__getattribute__例子:

```
#coding=utf-8
class Itcast(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
            return object.__getattribute__(self,obj)

    def show(self):
        print 'this is Itcast'

s = Itcast('python')
print s.subject1
print s.subject2
```