目录：

**一、命名规则**

**二、迭代器和列表解析**

**三、匿名函数和 lambda**

**四、yield 和生成器**

**五、函数式编程工具：map、filter、reduce**

**六、**[**Python编码介绍——encode和decode**](http://blog.chinaunix.net/uid-27838438-id-4227131.html)

**七、if、while、for、with**

**八、函数闭包及装饰器**

**九、递归函数**

**十、Python中的类**

**一、命名规则**

命名规范1 ：模块与函数

**1 包名 & 模块名**：package\_name、module\_name.py

小写字母，单词之间用\_分割

**2 函数 & 方法**：function\_name()、method\_name()

小写字母，单词之间用\_分割

**注意**：混合大小写仅被允许用于这种风格已经占据优势的时候，以便保持向后兼容。

**3 私有函数**（外部访问会报错）： \_\_get\_name()

以\_\_开头（2个下划线），其他和普通函数一样

**4 类名**：ClassName

单词首字母大写

内部类可以使用额外的前导下划线。

命名规范 2：变量

**1 普通变量**：this\_is\_a\_var

小写字母，单词之间用\_分割

**2 全局变量**：GLOBAL\_VAR\_NAME

写字母，单词之间用\_分割

不论是类成员变量还是全局变量，均不使用 m 或 g 前缀。

**注意**:应避免使用全局变量

对于from M import\*导入语句，如果想阻止导入模块内的全局变量可以使用旧有的规范，在全局变量上加一个前导的下划线。

**3 实例变量**：\_instance\_var\_name

以\_开头，其他和普通变量一样

**4 私有实例变量**（外部访问会报错）：\_\_private\_var\_name

以\_\_开头（2个下划线），其他和普通变量一样

**5 专有变**量：\_\_doc\_\_、\_\_class\_\_

\_\_开头，\_\_结尾，一般为python的自有变量，不要以这种方式命名

命名规范 3：其他

**1 常量** ：MAX\_OVERFLOW

常量名所有字母大写，由下划线连接各个单词如，TOTAL。

**2 异常** ：ExceptionName

以“Error”作为后缀，如 ImportError

**3 缩写** ：命名应当尽量使用全拼写的单词，缩写的情况有如下两种：

常用的缩写，如XML、ID等，在命名时也应只大写首字母，如XmlParser。

命名中含有长单词，对某个单词进行缩写。这时应使用约定成俗的缩写方式。

例如： function 缩写为 fn 、text 缩写为 txt

**函数和方法的参数**

总使用“self”作为实例方法的第一个参数。总使用“cls”作为类方法的第一个参数。

如果一个函数的参数名称和保留的关键字冲突，通常使用一个后缀下划线好于使用缩写或奇怪的拼写。

**变量**

变量名全部小写，由下划线连接各个单词。如color = WHITE，this\_is\_a\_variable = 1

\*注意\*：

1.不论是类成员变量还是全局变量，均不使用 m 或 g 前缀。

2.私有类成员使用单一下划线前缀标识，多定义公开成员，少定义私有成员。

3.变量名不应带有类型信息，因为Python是动态类型语言。如 iValue、names\_list、dict\_obj 等都是不好的命名。

**前导后缀下划线**

一个前导下划线：表示非公有。

一个后缀下划线：避免关键字冲突。

两个前导下划线：当命名一个类属性引起名称冲突时使用。

两个前导和后缀下划线：“魔”（有特殊用途）对象或者属性，例如\_\_init\_\_或者\_\_file\_\_。绝对不要创造这样的名字，而只是使用它们。

\*注意\*：关于下划线的使用存在一些争议。

Python 用下划线作为变量前缀和后缀指定特殊变量。

\_xxx      不能用'from module import \*'导入

\_\_xxx\_\_ 系统定义名字

\_\_xxx    类中的私有变量名

核心风格：避免用下划线作为变量名的开始。

因为下划线对解释器有特殊的意义，而且是内建标识符所使用的符号，我们建议程序员避免用下划线作为变量名的开始。一般来讲，变量名\_xxx被看作是“私有的”，在模块或类外不可以使用。当变量是私有的时候，用\_xxx 来表示变量是很好的习惯。

"单下划线" 开始的成员变量叫做保护变量，意思是只有类对象和子类对象自己能访问到这些变量；

"双下划线" 开始的是私有成员，意思是只有类对象自己能访问，连子类对象也不能访问到这个数据。

以单下划线开头（\_foo）的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用“from xxx import \*”而导入；

以双下划线开头的（\_\_foo）代表类的私有成员；

以双下划线开头和结尾的（\_\_foo\_\_）代表python里特殊方法专用的标识，如 \_\_init\_\_（）代表类的构造函数。

**特定命名方式**

主要是指 \_\_xxx\_\_ 形式的系统保留字命名法。

项目中也可以使用这种命名，它的意义在于这种形式的变量是只读的，这种形式的类成员函数尽量不要重载。如

class Base(object):

def \_\_init\_\_(self, id, parent = None):

self.\_\_id\_\_ = id

self.\_\_parent\_\_ = parent

def \_\_message\_\_(self, msgid):

# …略

其中 \_\_id\_\_、\_\_parent\_\_ 和 \_\_message\_\_ 都采用了系统保留字命名法。

**附:Google Python命名规范**

module\_name, package\_name, ClassName, method\_name, ExceptionName, function\_name, GLOBAL\_VAR\_NAME, instance\_var\_name, function\_parameter\_name, local\_var\_name.

**二、迭代器和列表解析**

<http://blog.csdn.net/wangyezi19930928/article/details/31378819>

循环：

>>> L = [1,2,3,4,5,6]

>>> **for** i **in** range(len(L)):

...     L[i] += 10

...

>>> L

[11, 12, 13, 14, 15, 16]

列表解析：

>>> L = [i+10 **for** i **in** L ]

>>> L

[21, 22, 23, 24, 25, 26]

直接结果是相同的，但是它需要较少的代码，并且可能会运行的更快，但是列表解析并不是必须的，因为我们总是可以用一个for循环手动的构建一个表达式结果的列表：

>>> res = []

>>> **for** x **in** L:

...     res.append(x+10)

...

>>> res

[31, 32, 33, 34, 35, 36]

在文件上使用列表解析：

>>> f = open("myfile.txt")

>>> lines = f.readlines()

>>> lines

['hello\n', 'i\n', 'love\n', 'python\n']

文件对象中有一个readlines()方法，它能一次性的把文件载入到行字符串的一个列表中。

在结果中的行在末尾都包含了一个换行符号（\n）。对于很多程序来说，换行符很讨厌，我们必须小心避免打印的时候留下双倍的空白

>>>> lines = [line.rstrip() **for** line **in** lines]

>>> lines

['hello', 'i', 'love', 'python']

>>> lines = [line.rstrip() **for** line **in** open("myfile.txt")]

>>> lines

['hello', 'i', 'love', 'python']

>>> lines = [ line.strip('\n') **for** line **in** open("myfile.txt")]

>>> lines

['hello', 'i', 'love', 'python']

在文件上使用列表解析不仅具有高效性而且其表现能力也很强。我们可以在迭代时在一个文件的行上运行任何字符串操作：

>>> f = open("myfile.txt")

>>> lines = f.readlines()

>>> lines

['hello\n', 'i\n', 'love\n', 'python\n']

>>> lines = [line.upper() **for** line **in** lines]

>>> lines

['HELLO\n', 'I\n', 'LOVE\n', 'PYTHON\n']

>>> lines = [line.upper().rstrip() **for** line **in** lines]

>>> lines

['HELLO', 'I', 'LOVE', 'PYTHON']

>>> [line.split() **for** line **in** lines]

[['HELLO'], ['I'], ['LOVE'], ['PYTHON']]

>>> [line.replace('','!') **for** line **in** lines]

['!H!E!L!L!O!', '!I!', '!L!O!V!E!', '!P!Y!T!H!O!N!']

>>> [line.replace(' ','!') **for** line **in** lines]

['HELLO', 'I', 'LOVE', 'PYTHON']

扩展的列表解析语法：

作为一个特别有用的扩展，表达式中嵌套的for循环可以有一个相关的if子句来过滤那些[**测试**](http://lib.csdn.net/base/softwaretest)不为真的结果。

>>> line = [line.rstrip() **for** line **in** open("myfile.txt") **if** line[0] == 'l']

>>> line

['love']

这条if语句检查从文件读取的每一行，检查第一个字符是否为'l'，，如果不是从结果列表中省略该行

总是可以把一个列表解析转换为for循环语句（但是这种表达方式运行起来要慢很多，而且占据行数也比较多）

>>> res = []

>>> **for** line **in** open("myfile.txt"):

...     **if** line[0] == 'l':

...             res.append(line.rstrip())

...     **else**:

...             **pass**

...

>>> res

['love']

如果我们需要的话，列表解析可以变得更复杂：

>>> [x+y **for** x **in** 'abc' **for** y **in** 'cde']

['ac', 'ad', 'ae', 'bc', 'bd', 'be', 'cc', 'cd', 'ce']

再次理解这个表达式的一种方式是通过缩进其各个部分将它转换为语句的形式，下面是其等价形式，但可能会慢一些，这是实现相同效果的一种代方式：

>>> res = []

>>> **for** x **in** 'abc':

...     **for** y **in** 'edc':

...             res.append(x+y)

...

>>> res

['ae', 'ad', 'ac', 'be', 'bd', 'bc', 'ce', 'cd', 'cc']

**三、匿名函数和 lambda**

总的来说，lambda 函数可以接收任意多个参数 (包括可选参数) 并且返回单个表达式的值。lambda 函数不能包含命令，包含的表达式不能超过一个。不要试图向 lambda 函数中塞入太多的东西；如果你需要更复杂的东西，应该定义一个普通函数，然后想让它多长就多长。

lambda 函数是一种风格问题。不一定非要使用它们；任何能够使用它们的地方，都可以定义一个单独的普通函数来进行替换。我将它们用在需要封装特殊的、非重用代码上，避免令我的代码充斥着大量单行函数。

lambda语句中，冒号前是参数，可以有多个，用逗号隔开，冒号右边的为表达式。其实lambda返回值是一个函数的地址，也就是函数对象。

g = lambda x:x\*x

**匿名函数lambda**，它是Python一种生成函数对象的表达式形式。匿名函数通常是创建了可以被调用的函数，它返回了函数，而并没有将这个函数命名。lambda有时被叫做匿名函数也就是这个原因，需要一个函数，又不想动脑筋去想名字，这就是匿名函数。

[?](http://www.iplaypython.com/wenda/lambda.html#)

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | #-\*- coding:utf-8 -\*- #\_\_author\_\_ = "www.iplaypython.com" # 普通python函数 def func(a,b,c): return a+b+c print func(1,2,3) # 返回值为6 # lambda匿名函数 f = lambda a,b,c:a+b+c print f(1,2,3) # 返回结果为6 |

# 大家注意观察上面的Python示例代码，f = lambda a,b,c:a+b+c 中的关键字lambda表示匿名函数，

# 冒号:之前的a,b,c表示它们是这个函数的参数。

# 匿名函数不需要[return](http://www.iplaypython.com/jinjie/return.html)来返回值，表达式本身结果就是返回值。

**四、yield 和生成器**

**列表推导与生成器表达式**

当我们创建了一个列表的时候，就创建了一个可以迭代的对象：

>>> squares=[n\*n for n in range(3)]

这种创建列表的操作很常见，称为列表推导。但是像列表这样的迭代器，比如str、file等，虽然用起来很方便，但有一点，它们是储存在内存中的，如果值很大，会很麻烦。

而生成器表达式不同，它执行的计算与列表包含相同，但会迭代的生成结果。它的语法与列表推导一样，只是要用小括号来代替中括号：

>>> squares=(n\*n for n in range(3))

生成器表达式不会创建序列形式的对象，不会把所有的值都读取到内存中，而是会创建一个通过迭代并按照需求生成值的生成器对象(Generator)。

**生成器Generator**

python中生成器的定义很简单，使用了**yield关键字**的函数就可以称之为**生成器**，它生成一个值的序列：

def countdown(n):

    while n>0:

        yield n

        n-=1

if \_\_name\_\_=='\_\_main\_\_':

    for i in countdown(10):

        print i

**生成器函数返回生成器。**要注意的是生成器就是一类**特殊的迭代器**。作为一个迭代器，生成器必须要定义一些方法，其中一个就是\_\_next\_\_()。如同迭代器一样，我们可以使用next()函数(Python3是\_\_next\_\_() )来获取下一个值：

>>> c=countdown(10)

>>> c.next()

10

>>> c.next()

9

每当生成器被调用的时候，它会返回一个值给调用者。在生成器内部使用yield来完成这个动作。为了记住yield到底干了什么，最简单的方法是把它当作专门给生成器函数用的特殊的return。调用next()时，生成器函数不断的执行语句，直至遇到yield为止，此时生成器函数的”状态”会被冻结，所有的变量的值会被保留下来，下一行要执行的代码的位置也会被记录，直到再次调用next()继续执行yield之后的语句。

next()不能无限执行，当迭代结束时，会抛出StopIteration异常。迭代未结束时，如果你想结束生成器，可以使用**close()方法**

>>> c.next()

1

>>> c.next()

StopIteration

>>> c=countdown(10)

>>> c.next()

10

>>> c.close()

>>> c.next()

StopIteration

**协程与yield表达式**

yield语句还有更给力的功能，作为一个语句出现在赋值运算符的右边，接受一个值，或同时生成一个值并接受一个值。

def recv():

    print 'Ready'

    while True:

        n=yield

        print 'Go %s'%n

>>> c=recv()

>>> c.next()

Ready

>>> c.send(1)

Go 1

>>> c.send(2)

Go 2

以这种方式使用yield语句的函数称为协程。

在这个例子中，对于next()的初始调用是必不可少的，这样协程才能执行可通向第一个yield表达式的语句。在这里协程会挂起，等待相关生成器对象send()方法给它发送一个值。传递给send()的值由协程中的yield表达式返回。

**协程的运行一般是无限期的，使用方法close()可以显式的关闭它。**

如果yield表达式中提供了值，协程可以**使用yield语句同时接收和发出返回值**。

def split\_line():

    print 'ready to split'

    result=None

    while True:

        line=yield result

        result=line.split()

>>> s=split\_line()

>>> s.next()

ready to split

>>> s.send('1 2 3')

['1', '2', '3']

>>> s.send('a b c')

['a', 'b', 'c']

注意：理解这个例子中的先后顺序非常重要。首个next()方法让协程执行到yield result，这将返回result的值None。在接下来的send()调用中，接收到的值被放到line中并拆分到result中。send()方法的返回值就是下一条yield语句的值。也就是说，send()方法可以将一个值传递给yield表达式，但是其返回值来自下一个yield表达式，而不是接收send()传递的值的yield表达式。

**如果你想用send()方法来开启协程的执行，必须先send一个None值，因为这时候是没有yield语句来接受值的，否则就会抛出异常。**

>>> s=split\_line()

>>> s.send('1 2 3')

TypeError: can't send non-None value to a just-started generator

>>> s=split\_line()

>>> s.send(None)

ready to split

考虑一个简单的例子，文件的遍历。要遍历一个目录下的所有文件需要递归的操作。如果我们只是单纯的打印文件名，我们可以在递归的过程中完成，每当发现一个非目录就可以打印文件名。代码如下：

class TraverseDirectory(object):

    @staticmethod

    def traverse(dir):

        if os.path.isdir(dir):

            files=os.listdir(dir);

            for file in files[::-1]:

                full\_name=os.path.join(dir,file);

                TraverseDirectory.traverse(full\_name):

        else:

            print dir;

但是如果我们想保存文件名或者对每一个文件执行更复杂的操作返回一个结果，情况就稍微有些复杂。问题在于我们需要一个全局变量保存访问的结果。例如在遍历到一个文件时获取其文件大小，则我们还需要一个dictionary结构：

class TraverseDirectory(object):

    fileSize=dict();

    @staticmethod

    def traverse(dir):

        if os.path.isdir(dir):

            files=os.listdir(dir);

            for file in files[::-1]:

                full\_name=os.path.join(dir,file);

                TraverseDirectory.traverse(full\_name);

        else:

            TraverseDirectory.fileSize[dir]=os.path.getsize(dir);

初看代码感觉其实也不复杂，只是多了一个变量而已。在多数情况下确实如此，但是从应用角度来看上述代码稍有不足：为了访问遍历的结果，我们需要访问该类的全局变量或者一个静态变量，正如我们在其他语言中一样，为此我们还需要一个get函数。但是，这还不是最严重的问题，fileSize占用的空间才是问题！

       上述递归函数只有在遍历完所有的文件之后我们才能访问fileSize结构。这就意味着如果目录很大，则fileSize也会非常大，如果要控制内存占用，上述方式会很不好。此时，就可以采用yield生成器完成遍历。

       何谓生成器？其实很简单，概念和迭代器类似，都是为了遍历而存在。迭代器是为了遍历变量，例如列表、元组、字典等等。生成器不是遍历变量，而是函数。简而言之，包含yield语句的函数会被特地编译成生成器。当函数被调用时，他们返回一个生成器对象，这个对象支持迭代器接口。不像一般的函数会生成值后退出，生成器函数在生成值后会自动挂起并暂停他们的执行和状态，他的本地变量将保存状态信息，这些信息在函数恢复时将再度有效。我们可以将生成器理解成可以随时获得函数返回值的迭代器。可以将生成器与gdb调试器对比，我们在每一步单步调试的过程中，系统中总会有许多变量和其运行时值，yield就对应每一次的单步调试，并获取当前的变量值，不过其获取返回值是在每一次递归结束时。将上述遍历代码改成yield方式之后为：

class TraverseDirectory(object):

    @staticmethod

    def traverse(dir):

        if os.path.isdir(dir):

            files=os.listdir(dir);

            for file in files[::-1]:

                full\_name=os.path.join(dir,file);

                for results in TraverseDirectory.traverse(full\_name):

                    yield results;

        else:

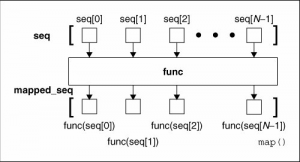
           yield  {dir:os.path.getsize(dir)};

与之前的代码相比，就是少了一个变量，然后多了两个yield。通过这个微小的变化，我们去掉了函数的内存大小限制，也使代码更简洁。需要注意的一点是，调用过程也很简单，因为函数是一个生成器，具有迭代器的功能，我们就可以利用for循环去遍历函数的返回值。

**五、函数式编程工具：map、filter、reduce**

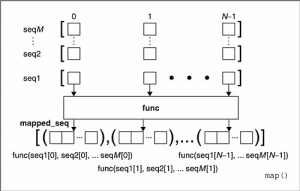
**1 map**

**3.1 当seq只有一个时**，将func函数作用于这个seq的每个元素上，得到一个新的seq。



>>> def add100(x):
... return x+100
...
>>> hh = [11,22,33]
>>> map(add100,hh)
[111, 122, 133]

**3.2 当seq多于一个时**，map可以并行地对每个seq执行如下图所示的过程：



也就是说每个seq的同一位置的元素在执行过一个多元的func函数之后，得到一个返回值，这些返回值放在一个结果列表中。**需要注意的是，不同长度的多个seq是无法执行map函数的，会出现类型错误。**

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

# [4, 10, 18]

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

# [(4, 5), (10, 7), (18, 9)]

>>> def abc(a, b, c):
... return a\*10000 + b\*100 + c
...
>>> list1 = [11,22,33]
>>> list2 = [44,55,66]
>>> list3 = [77,88,99]
>>> map(abc,list1,list2,list3)
[114477, 225588, 336699]

3.3 如果'function'给出的是‘None’，自动假定一个‘identity’函数

print map( None, [1, 2, 3], [4, 5, 6] )

# [(1, 4), (2, 5), (3, 6)]

 print zip( [1, 2, 3], [4, 5, 6] )

# [(1, 4), (2, 5), (3, 6)]

>>> list1 = [11,22,33]
>>> map(None,list1)
[11, 22, 33]
>>> list1 = [11,22,33]
>>> list2 = [44,55,66]
>>> list3 = [77,88,99]
>>> map(None,list1,list2,list3)
[(11, 44, 77), (22, 55, 88), (33, 66, 99)]

**2 filter**

filter函数会对指定序列执行过滤操作。

filter函数的定义：

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

function是一个谓词函数，接受一个参数，返回布尔值True或False。

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

比如返回序列中的所有偶数：

def is\_even(x):

return x & 1 != 0

filter(is\_even, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# [1, 3, 5, 7, 9]

如果function参数为None，返回结果和sequence参数相同。

**3 reduce**

在**Python 3**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**

reduce函数，reduce函数会对参数序列中元素进行累积。

reduce函数的定义：

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

function参数是一个有两个参数的函数，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。

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

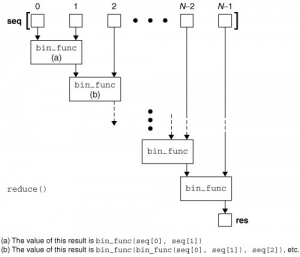
结果为21(  (((((1+2)+3)+4)+5)+6)  )

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

结果为20

注意function函数不能为None。

reduce函数即为化简，它是这样一个过程：每次迭代，将上一次的迭代结果（第一次时为init的元素，如没有init则为seq的第一个元素）与下一个元素一同执行一个二元的func函数。在reduce函数中，init是可选的，如果使用，则作为第一次迭代的第一个元素使用。



**阶乘**是一个常见的数学方法，Python中并没有给出一个阶乘的内建函数，我们可以使用reduce实现一个阶乘的代码。

n = 5

print reduce(lambda x, y: x \* y, range(1, n + 1))

# 120

得到2倍阶乘的值

m = 2

n = 5

print reduce( lambda x, y: x \* y, range( 1, n + 1 ), m )

# 240

**六、**[**Python编码介绍——encode和decode**](http://blog.chinaunix.net/uid-27838438-id-4227131.html)

<http://blog.chinaunix.net/uid-27838438-id-4227131.html>

在 python 源代码文件中，**如果你有用到非ASCII字符，则需要在文件头部进行字符编码的声明**，声明如下：

**# code: UTF-8**

因为python 只检查 #、coding 和编码字符串，所以你可能回见到下面的声明方式，这是有些人为了美观等原因才这样写的：

#-\*- coding: UTF-8 -\*-

**常见编码介绍：**

GB2312编码：适用于汉字处理、汉字通信等系统之间的信息交换

GBK编码：是汉字编码标准之一，是在 GB2312-80 标准基础上的内码扩展规范，使用了双字节编码

ASCII编码：是对英语字符和二进制之间的关系做的统一规定

Unicode编码：这是一种世界上所有字符的编码。当然了它没有规定的存储方式。

UTF-8编码：是 Unicode Transformation Format - 8 bit 的缩写， **UTF-8 是 Unicode 的一种实现方式**。它是可变长的编码方式，可以使用 1~4 个字节表示一个字符，可根据不同的符号而变化字节长度。

**编码转换：**

Python内部的字符串一般都是 Unicode编码。**代码中字符串的默认编码与代码文件本身的编码是一致的。**所以要做一些编码转换通常是要以Unicode作为中间编码进行转换的，即先将其他编码的字符串解码（decode）成 Unicode，再从 Unicode编码（encode）成另一种编码。

**decode**的作用是将其他编码的字符串转换成 Unicode 编码，eg name.decode(“GB2312”)，表示将GB2312编码的字符串name转换成Unicode编码

**encode**的作用是将Unicode编码转换成其他编码的字符串，eg name.encode(”GB2312“)，表示将GB2312编码的字符串name转换成GB2312编码

所以在进行编码转换的时候必须先知道 name 是那种编码，然后 decode 成 Unicode 编码，最后载 encode 成需要编码的编码。当然了，如果 name 已经就是 Unicode 编码了，那么就不需要进行 decode 进行解码转换了，直接用 encode 就可以编码成你所需要的编码。**值得注意的是：对 Unicode 进行编码和对 str 进行编码都是错误的。**

具体的说就是：如果在UTF-8文件中，则这个字符串就是 UTF-8编码的。它的编码取决于当前的文本编码。当然了，GB2312文本的编码就是GB2312。**要在同一个文本中进行两种编码的输出等操作就必须进行编码的转换**，先用decode将文本原来的编码转换成Unicode，再用encode将编码转换成需要转换成的编码。

eg：

由于内置函数 open() 打开文件时，read() 读取的是 str，读取后需要使用正确的编码格式进行 decode()。write() 写入时，如果参数是 Unicode，则需要使用你希望写入的编码进行 encode()，如果是其他编码格式的 str，则需要先用该 str 的编码进行 decode()，转成 Unicode 后再使用写入的编码进行 encode()。如果直接将 Unicode 作为参数传入 write() ，python 将先使用源代码文件声明的字符编码进行编码然后写入。

# coding: UTF-8

fp1 = open('test.txt', 'r')

info1 = fp1.read()

# 已知是 GBK 编码，解码成 Unicode

tmp = info1.decode('GBK')

fp2 = open('test.txt', 'w')

# 编码成 UTF-8 编码的 str

info2 = tmp.encode('UTF-8')

fp2.write(info2)

fp2.close()

**获取编码的方式：**

判断是 s 字符串否为Unicode，如果是返回True，不是返回False ：

isinstance(s, unicode)

下面代码可以获取系统默认编码：

#!/usr/bin/env python

#coding=utf-8

import sys

print sys.getdefaultencoding()

**七、if、while、for、with**

**1 if**

**2 while**

**3 for**

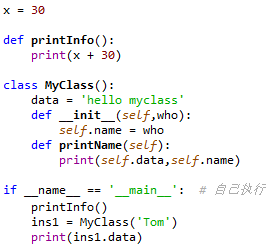
**4 with**

**八、函数闭包及装饰器**

**九、递归函数**

**十、Python中的类**

module name: mymod



调用过程：

1 加入路径

import sys

sys.path.append(...)

2 调用

import mymod

mymod.x

mymod.printInfo()

# 类要先实例化

ins1 = mymod.MyClass('Tom')

ins1.data # 'hello myclass'

ins1.name # Tom

ins1.printName() # hello myclass Tom

**reload(mymod)**

<http://python.jobbole.com/82297/>

在Python中，可以通过class关键字定义自己的类，然后通过自定义的类对象类创建实例对象。

例如，下面创建了一个Student的类，并且实现了这个类的**初始化函数”\_\_init\_\_”**:

class Student(object):

count = 0

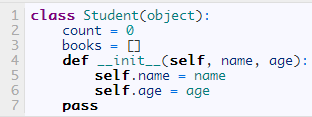
books = []

def \_\_init\_\_(self,name,age):

self.name = name

self.age = age

pass



**1 数据属性**

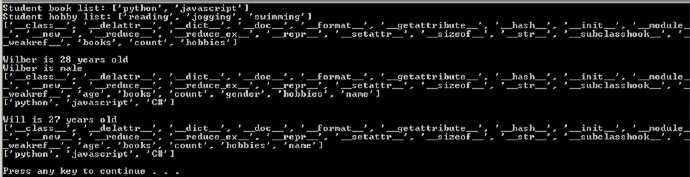
在上面的Student类中，”count””books””name”和”age”都被称为类的数据属性，但是它们又分为类数据属性和实例数据属性。

**1.1 类数据属性和实例数据属性**

首先看一段代码，代码中分别展示了对类数据属性和实例数据属性的访问：



通过内建函数dir()，或者访问类的字典属性\_\_dict\_\_，这两种方式都可以查看类有哪些属性，代码的输出为：



**对于类数据属性和实例数据属性，可以总结为：**

类数据属性属于类本身，可以通过类名进行访问/修改

类数据属性也可以被类的所有实例访问/修改

在类定义之后，可以通过类名动态添加类数据属性，新增的类属性也被类和所有实例共有

实例数据属性只能通过实例访问

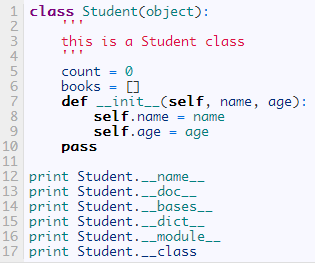
在实例生成后，还可以动态添加实例数据属性，但是这些实例数据属性只属于该实例

**1.2 特殊的类属性**

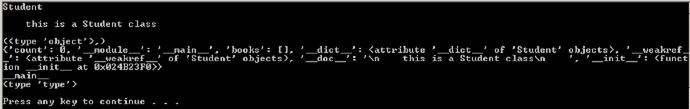
对于所有的类，都有一组特殊的属性：

|  |  |
| --- | --- |
| 类属性 | **含义** |
| \_\_name\_\_ | 类的名字（字符串） |
| \_\_doc\_\_ | 类的文档字符串 |
| \_\_bases\_\_ | 类的所有父类组成的元组 |
| \_\_dict\_\_ | 类的属性组成的字典 |
| \_\_module\_\_ | 类所属的模块 |
| \_\_class\_\_ | 类对象的类型 |

通过这些属性，可以得到 Student类的一些信息：



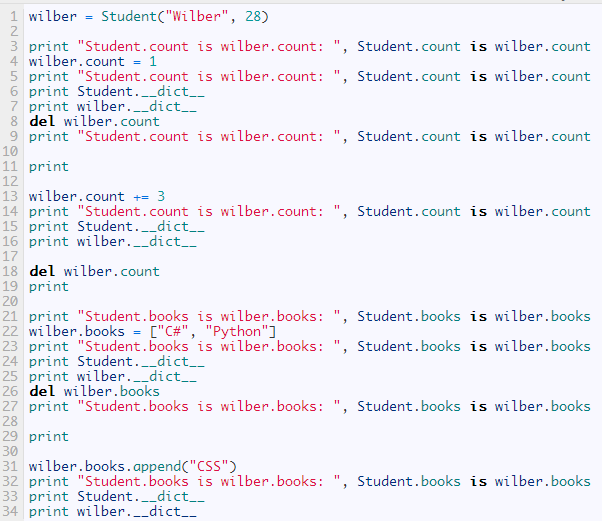
代码输出为：



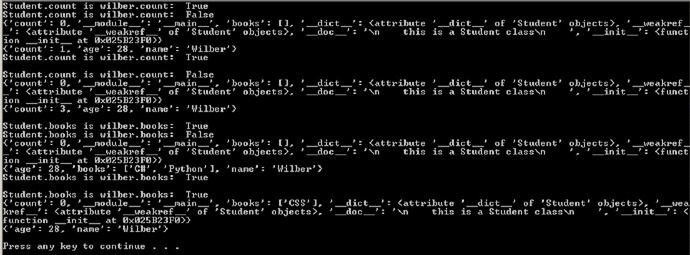
**1.3 属性隐藏**

从上面的介绍了解到，类数据属性属于类本身，被所有该类的实例共享；并且，通过实例可以去访问/修改类属性。但是，**在通过实例中访问类属性的时候一定要谨慎，因为可能出现属性”隐藏”的情况**。

继续使用上面的Student类，来看看属性隐藏：



代码的输出为：



分析一下上面代码的输出：

对于不可变类型的类属性Student.count，可以通过实例wilber进行访问，并且”Student.count is wilber.count”

当通过实例赋值/修改count属性的时候，都将为实例wilber新建一个count实例属性，这时，”Student.count is not wilber.count”

当通过”del wilber.count”语句删除实例的count属性后，再次成为”Student.count is wilber.count”

同样对于可变类型的类属性Student.books，可以通过实例wilber进行访问，并且”Student. books is wilber. books”

当通过实例赋值books属性的时候，都将为实例wilber新建一个books实例属性，这时，”Student. Books is not wilber. books”

当通过”del wilber. books”语句删除实例的books属性后，再次成为”Student. books is wilber. books”

当通过实例修改books属性的时候，将修改wilber.books指向的内存地址（即Student.books），此时，”Student. Books is wilber. books”

注意，虽然通过实例可以访问类属性，但是，不建议这么做，最好还是通过类名来访问类属性，从而避免属性隐藏带来的不必要麻烦。

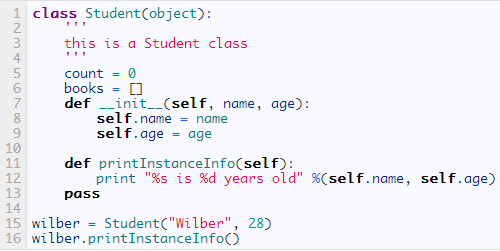
**2 方法**

在一个类中，可能出现三种方法，实例方法、静态方法和类方法，下面来看看三种方法的不同。

**2.1 实例方法**

实例方法的第一个参数必须是”self”，”self”类似于C++中的”this”。

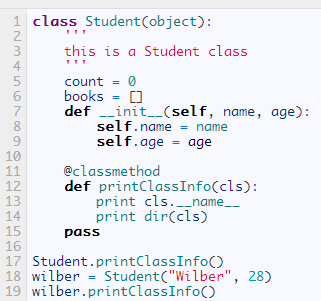
**实例方法只能通过类实例进行调用，这时候”self”就代表这个类实例本身。通过”self”可以直接访问实例的属性。**



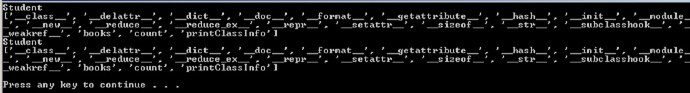
|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | class Student(object): ''' this is a Student class ''' count = 0 books = [] def \_\_init\_\_(self, name, age): self.name = name self.age = age def printInstanceInfo(self): print "%s is %d years old" %(self.name, self.age) pass wilber = Student("Wilber", 28) wilber.printInstanceInfo() |

**2.2 类方法**

类方法以cls作为第一个参数，cls表示类本身，定义时使用@classmethod装饰器。通过cls可以访问类的相关属性。



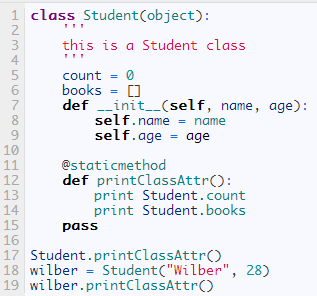
代码的输出为，从这段代码可以看到，类方法可以通过类名访问，也可以通过实例访问。



**2.3 静态方法**

与实例方法和类方法不同，静态方法没有参数限制，既不需要实例参数，也不需要类参数，定义的时候使用@staticmethod装饰器。

同类方法一样，静态法可以通过类名访问，也可以通过实例访问。



这三种方法的主要区别在于参数，实例方法被绑定到一个实例，只能通过实例进行调用；但是对于静态方法和类方法，可以通过类名和实例两种方式进行调用。

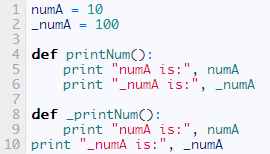
**3 访问控制**

Python中没有访问控制的关键字，例如private、protected等等。但是，在Python编码中，有一些约定来进行访问控制。

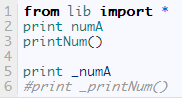
**3.1 单下划线”\_”**

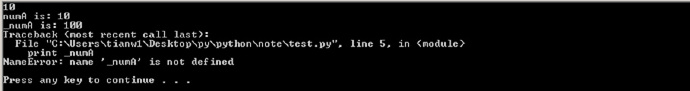
在Python中，通过单下划线”\_”来实现模块级别的私有化，一般约定以单下划线”\_”开头的变量、函数为模块私有的，也就是说”from moduleName import \*”将不会引入以单下划线”\_”开头的变量、函数。

现在有一个模块lib.py，内容用如下，模块中一个变量名和一个函数名分别以”\_”开头：



当通过下面代码引入lib.py这个模块后，所有以”\_”开头的变量和函数都没有被引入，如果访问将会抛出异常：

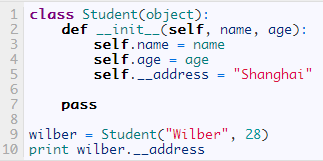




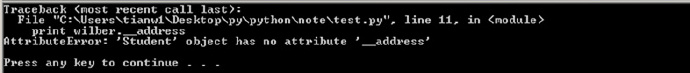
**3.2 双下划线”\_\_”**

对于Python中的类属性，可以通过双下划线”\_\_”来实现一定程度的私有化，因为双下划线开头的属性在运行时会被”混淆”（mangling）。

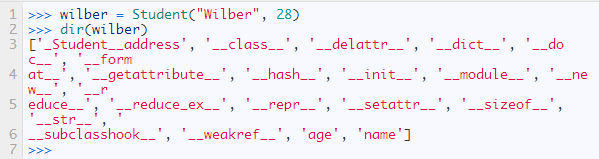
在Student类中，加入了一个”\_\_address”属性：



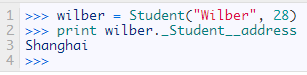
当通过实例wilber访问这个属性的时候，就会得到一个异常，提示属性”\_\_address”不存在。



其实，通过内建函数dir()就可以看到其中的一些原由，”\_\_address”属性在运行时，属性名被改为了”\_Student\_\_address”（属性名前增加了单下划线和类名）

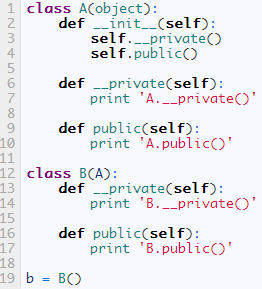


所以说，即使是双下划线，也没有实现属性的私有化，因为通过下面的方式还是可以直接访问”\_\_address”属性：



双下划线的另一个重要的目地是，避免子类对父类同名属性的冲突。

看下面一个例子：



当实例化B的时候，由于没有定义\_\_init\_\_函数，将调用父类的\_\_init\_\_，但是由于双下划线的”混淆”效果，”self.\_\_private()”将变成 “self.\_A\_\_private()”。

看到这里，就清楚为什么会有如下输出了：

j20gv01qmx2.jpeg

“\_”和” \_\_”的使用 更多的是一种规范/约定，不没有真正达到限制的目的：

“\_”：以单下划线开头的表示的是protected类型的变量，即**只能允许其本身与子类进行访问；同时表示弱内部变量标示**，如，当使用”from moduleNmae import \*”时，不会将以一个下划线开头的对象引入。

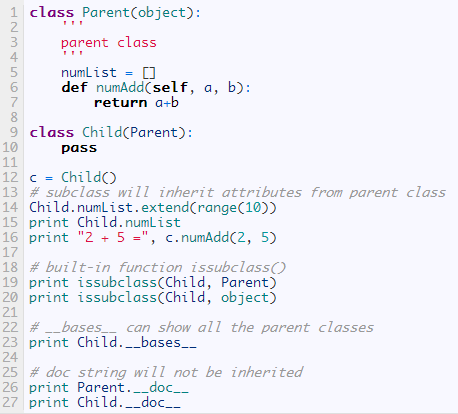
“\_\_”：双下划线的表示的是私有类型的变量。**只能是允许这个类本身进行访问了，连子类也不可以**，这类属性在运行时属性名会加上单下划线和类名。

**4 继承**

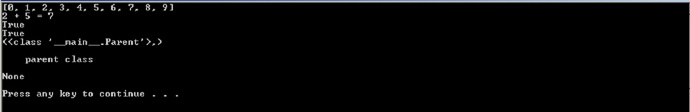
在Python中，同时支持单继承与多继承，一般语法如下：

clipboard.png

实现继承之后，子类将继承父类的属性，也可以使用内建函数issubclass()来判断一个类是不是另一个类的子孙类：



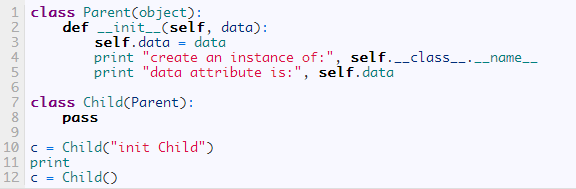
代码的输出为，例子中唯一特别的地方是文档字符串。文档字符串对于类，函数/方法，以及模块来说是唯一的，也就是说\_\_doc\_\_属性是不能从父类中继承来的。



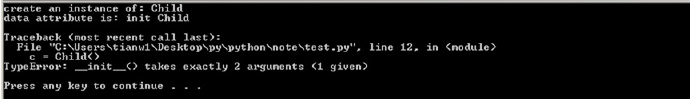
**4.1 继承中的\_\_init\_\_**

当在Python中出现继承的情况时，一定要注意初始化函数\_\_init\_\_的行为。

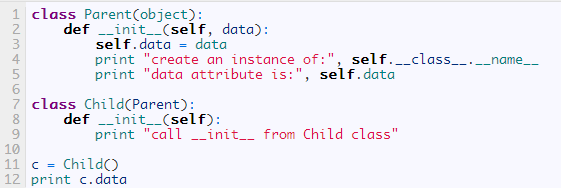
1. 如果子类没有定义自己的初始化函数，父类的初始化函数会被默认调用；但是如果要实例化子类的对象，则只能传入父类的初始化函数对应的参数，否则会出错。



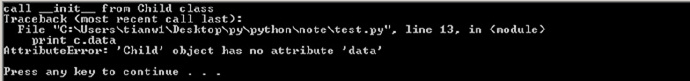
代码的输出为：



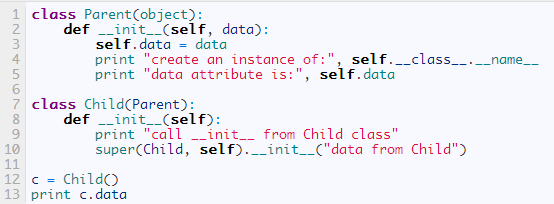
2. 如果子类定义了自己的初始化函数，而没有显示调用父类的初始化函数，则父类的属性不会被初始化



代码的输出为：



3. 如果子类定义了自己的初始化函数，显示调用父类，子类和父类的属性都会被初始化



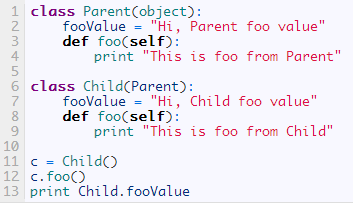
代码的输出为：



**4.2 super**

前面一个例子中，已经看到了通过super来调用父类\_\_init\_\_方法的例子，下面看看super的使用。

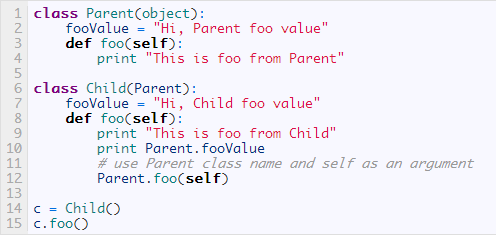
在子类中，一般会定义与父类相同的属性（数据属性，方法），从而来实现子类特有的行为。也就是说，子类会继承父类的所有的属性和方法，子类也可以覆盖父类同名的属性和方法。



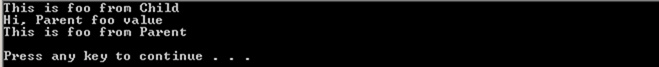
在这段代码中，子类的属性”fooValue”和”foo”覆盖了父类的属性，所以子类有了自己的行为。

j20jq01pmx4.jpeg

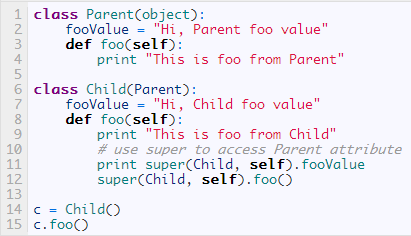
但是，有时候可能需要在子类中访问父类的一些属性：



这时候，可以通过父类名直接访问父类的属性，当调用父类的方法是，需要将”self”显示的传递进去的方式。



这种方式有一个不好的地方就是，需要经父类名硬编码到子类中，为了解决这个问题，可以使用Python中的super关键字：

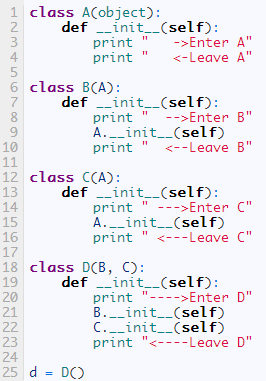


对于”super(Child, self).foo()”可以理解为，首先找到Child的父类Parent，然后调用父类的foo方法，同时将Child的实例self传递给foo方法。

但是，如果当一个子类有多个父类的时候，super会如何工作呢？这是就需要看看MRO的概念了。

**4.3 MRO**

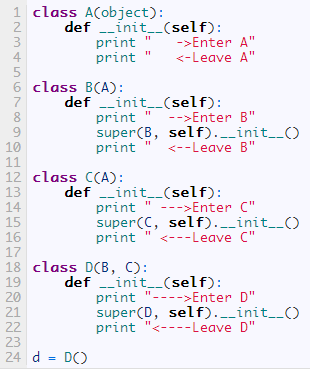
假设现在有一个如下的继承结构，首先通过类名显示调用的方式来调用父类的初始化函数：



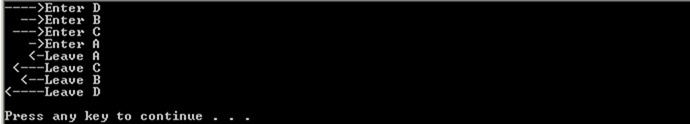
从输出中可以看到，类A的初始化函数被调用了两次，这不是我们想要的结果：



下面，我们通过super方式来调用父类的初始化函数：



通过输出可以看到，当使用super后，A的初始化函数只能调用了一次：



为什么super会有这种效果？下面就开始看看Python中的方法解析顺序MRO（Method Resolution Order）。

Python的类有一个\_\_mro\_\_属性，这个属性中就保存着方法解析顺序。结合上面的例子来看看类D的\_\_mro\_\_:

clipboard.png

看到这里，对于上面使用super例子的输出就应该比较清楚了。

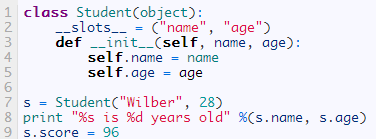
Python的多继承类是通过MRO的方式来保证各个父类的函数被逐一调用，而且保证每个父类函数只调用一次（如果每个类都使用super）

混用super类和非绑定的函数是一个危险行为，这可能导致应该调用的父类函数没有调用或者一个父类函数被调用多次

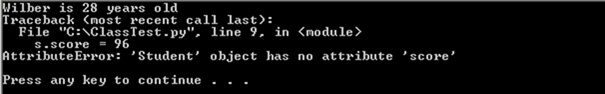
**4.4 \_\_slots\_\_**

从前面的介绍可以看到，当我们通过一个类创建了实例之后，仍然可以给实例添加属性，但是这些属性只属于这个实例。

有些时候，我们可以需要限制类实例对象的属性，这时就要用到类中的\_\_slots\_\_属性了。\_\_slots\_\_属性对于一个tuple，只有这个tuple中出现的属性可以被类实例使用。

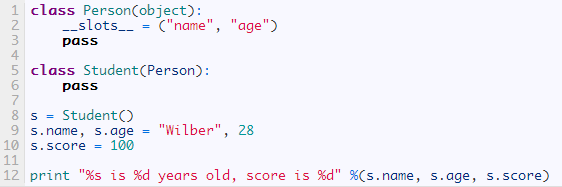


在这个例子中，当场是给Student的实例s添加一个score属性的时候，就会遇到下面的异常：



**4.5 子类没有\_\_slots\_\_属性**

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

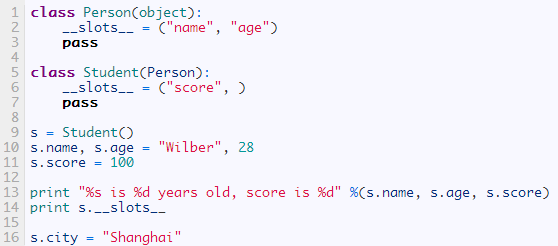


从代码的输出可以看到，子类Student的实例并不受父类中\_\_slots\_\_属性的限制：

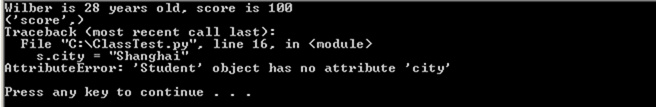
j20hv019dfr.jpeg

**4.6 子类拥有\_\_slots\_\_属性**

但是，如果子类本身也有\_\_slots\_\_属性，子类的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。



代码的输出为：



所以说，对于\_\_slots\_\_属性：

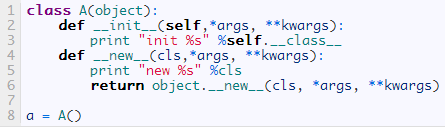
如果父类包含对\_\_slots\_\_的定义，子类**不包含**对\_\_slots\_\_的定义，解释器忽略\_\_slots\_\_的作用

如果父类包含对\_\_slots\_\_的定义，子类**包含**对\_\_slots\_\_的定义，并且无论元组的的元素个数，解释器都会按照父类的\_\_slots\_\_和子类的\_\_slots\_\_的并集来检查

**5 类构造和初始化**

在前面的文章中，经常使用初始化函数”\_\_init\_\_”，下面看看”\_\_init\_\_”和”\_\_new\_\_”的联系和差别。

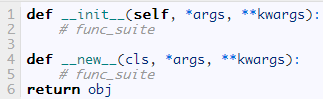
下面先通过一段代码看看这两个方法的调用顺序：



从代码的输出可以看到，当通过类实例化一个对象的时候，”\_\_new\_\_”方法首先被调用，然后是”\_\_init\_\_”方法。

j20hh01omx4.jpeg

一般来说，”\_\_init\_\_”和”\_\_new\_\_”函数都会有下面的形式：



对于”\_\_new\_\_”和”\_\_init\_\_”可以概括为：

“\_\_new\_\_”方法在Python中是真正的构造方法（创建并返回实例），通过这个方法可以产生一个”cls”对应的实例对象，所以说”\_\_new\_\_”方法一定要有返回

对于”\_\_init\_\_”方法，是一个初始化的方法，”self”代表由类产生出来的实例对象，”\_\_init\_\_”将对这个对象进行相应的初始化操作

前面文章中已经介绍过了”\_\_init\_\_”的一些行为，包括继承情况中”\_\_init\_\_”的表现。下面就重点看看”\_\_new\_\_”方法。

**5.1 \_\_new\_\_特性**

“\_\_new\_\_”是在新式类中新出现的方法，它有以下行为特性：

“\_\_new\_\_” 方法是在类实例化对象时第一个调用的方法，将返回实例对象

“\_\_new\_\_” 方法始终都是类的静态方法（即第一个参数为cls），即使没有被加上静态方法装饰器

第一个参数cls是当前正在实例化的类，如果要得到当前类的实例，应当在当前类中的 “\_\_new\_\_” 方法语句中调用当前类的父类的” \_\_new\_\_” 方法

对于上面的第三点，如果当前类是直接继承自 object，那当前类的 “\_\_new\_\_” 方法返回的对象应该为：

clipboard.png

**5.2 重写\_\_new\_\_**

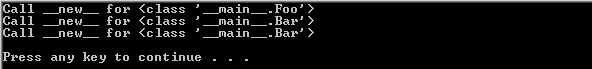
如果（新式）类中没有重写”\_\_new\_\_”方法，Python默认是调用该类的直接父类的”\_\_new\_\_”方法来构造该类的实例，如果该类的父类也没有重写”\_\_new\_\_”，那么将一直按照同样的规则追溯至object的”\_\_new\_\_”方法，因为object是所有新式类的基类。

而如果新式类中重写了”\_\_new\_\_”方法，那么可以选择任意一个其他的新式类（必须是新式类，只有新式类有”\_\_new\_\_”，因为所有新式类都是从object派生）的”\_\_new\_\_”方法来创建实例，包括这个新式类的所有前代类和后代类，只要它们不会造成递归死循环。

看一段例子代码：



代码的输出为：

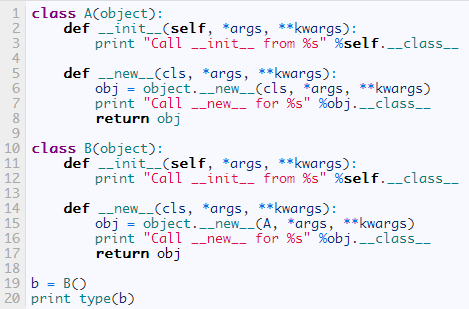


**5.3 \_\_init\_\_的调用**

“\_\_new\_\_”决定是否要使用该类的”\_\_init\_\_”方法，因为”\_\_new\_\_” 可以调用其他类的构造方法或者直接返回别的类创建的对象来作为本类的实例。

通常来说，新式类开始实例化时，”\_\_new\_\_”方法会返回cls（cls指代当前类）的实例，然后调用该类的”\_\_init\_\_”方法作为初始化方法，该方法接收这个实例（即self）作为自己的第一个参数，然后依次传入”\_\_new\_\_”方法中接收的位置参数和命名参数。

但是，如果”\_\_new\_\_”没有返回cls（即当前类）的实例，那么当前类的”\_\_init\_\_”方法是不会被调用的。看下面的例子：



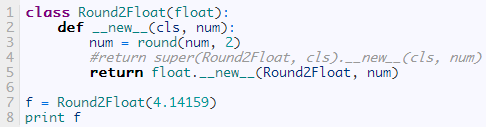
代码中，在B的”\_\_new\_\_”方法中，通过”obj = object.\_\_new\_\_(A, \*args, \*\*kwargs)”创建了一个A的实例，在这种情况下，B的”\_\_init\_\_”函数就不会被调用到。

j20eu01l3yh.jpeg

**5.4 派生不可变类型**

关于”\_\_new\_\_”方法还有一个重要的用途就是用来派生不可变类型。

例如，Python中float是不可变类型，如果想要从float中派生一个子类，就要实现”\_\_new\_\_”方法：



代码中从float派生出了一个Round2Float类，该类的实例就是保留小数点后两位的浮点数。

**6 定制一个类**

在Python中，我们可以通过”魔术方法”使自定义的class变得强大、易用。

例如，前面的文章中介绍过Python迭代器，当我们想定义一个可迭代的类对象的时候，就可以去实现”\_\_iter\_\_(self)”这个魔术方法；

又例如，前面文章介绍的上下文管理器，当需要建立一个上下文管理器类对象的时候，就可以去实现”\_\_enter\_\_(self)”和”\_\_exit\_\_(self)”方法。

所以，建议参考 “魔术方法” 的文档，通过魔术方法来定制自定义的类。

**6.1 调用魔术方法**

一些魔术方法直接和内建函数相对应的，在这种情况下，调用他们的方法很简单，下面给出了一些对应表。

|  |  |  |
| --- | --- | --- |
| **魔术方法** | **调用方式** | **解释** |
| **\_\_new\_\_(cls [,…])** | instance = MyClass(arg1, arg2) | \_\_new\_\_ 在创建实例的时候被调用 |
| **\_\_init\_\_(self [,…])** | instance = MyClass(arg1, arg2) | \_\_init\_\_ 在创建实例的时候被调用 |
| **\_\_cmp\_\_(self, other)** | self == other, self > other, 等。 | 在比较的时候调用 |
| **\_\_pos\_\_(self)** | +self | 一元加运算符 |
| **\_\_neg\_\_(self)** | -self | 一元减运算符 |
| **\_\_invert\_\_(self)** | ~self | 取反运算符 |
| **\_\_index\_\_(self)** | x[self] | 对象被作为索引使用的时候 |
| **\_\_nonzero\_\_(self)** | bool(self) | 对象的布尔值 |
| **\_\_getattr\_\_(self, name)** | self.name # name 不存在 | 访问一个不存在的属性时 |
| **\_\_setattr\_\_(self, name, val)** | self.name = val | 对一个属性赋值时 |
| **\_\_delattr\_\_(self, name)** | del self.name | 删除一个属性时 |
| **\_\_getattribute(self, name)** | self.name | 访问任何属性时 |
| **\_\_getitem\_\_(self, key)** | self[key] | 使用索引访问元素时 |
| **\_\_setitem\_\_(self, key, val)** | self[key] = val | 对某个索引值赋值时 |
| **\_\_delitem\_\_(self, key)** | del self[key] | 删除某个索引值时 |
| **\_\_iter\_\_(self)** | for x in self | 迭代时 |
| **\_\_contains\_\_(self, value)** | value in self, value not in self | 使用 in 操作测试关系时 |
| **\_\_concat\_\_(self, value)** | self + other | 连接两个对象时 |
| **\_\_call\_\_(self [,…])** | self(args) | “调用”对象时 |
| **\_\_enter\_\_(self)** | with self as x: | with 语句环境管理 |
| **\_\_exit\_\_(self, exc, val, trace)** | with self as x: | with 语句环境管理 |
| **\_\_getstate\_\_(self)** | pickle.dump(pkl\_file, self) | 序列化 |
| **\_\_setstate\_\_(self)** | data = pickle.load(pkl\_file) | 序列化 |