Python 系列分享会——第二讲 函数&类与对象&魔法方法

金融科技协会 2020 年 10 月 23 日

目录

一、函数

1.0 什么是函数 callable()

callable() 函数用于检查一个对象是否是可调用的。可调用返回 True，否则返回 False。对于函数、lambda 函式、 类以及实现了 **call** 方法的类实例，它都返回 True。（如图 1）

```
#callable()方法语法:
callable(object)  #object为对象
```

图表 1 callable()方法用法

类对象都是可被调用对象，类的实例对象是否可调用对象，取决于类是否定义了 **call** 方法。（如图 2-5）

```
def add(a,b):
    return a+b

callable(add)
```

```
True
```

图表 2 add 函数可以被成功调用

```
class A: #定义类A
    pass

callable(A) #类A是可调用对象
```

```
True
```

图表 3 类 A 是可调用对象

```
a = A() #调用变量a
callable(a) #在没有实现__call__的条件下,变量a不可调用,返回false
```

```
False
```

图表 4 没有实现 call,变量不可被调用

```
class B: #定义类B
    def __call__(self):   #参数self指向对象本身
        print('instances are callable now.')

callable(B) #类B是可调用对象

b = B()  #想去调用变量b
callable(b)
```

```
True
```

图表 5 实现 call 可以调用变量 b

1.1 函数定义

运行这些代码后，将有一个名为 hello 的新函数。它返回一个字符串，其中包含向唯一参数指定的人发出的问候语。你可像使用内置函数那样使用这个函数。（如图 6-7）

```
def hello(name):
    return "hello,"+name+"!"

print (hello("world"))
print (hello("lucy"))
```

```
hello,world!
hello,lucy!
```

图表 6 自定义函数 hello

```
def sum_2_sum(num1,num2): #形参
    result = num1+num2
    return result #返回值

end = sum_2_sum(2,3)
print(end)
```

```
5
```

图表 7 自定义函数 sum_2_sum

即，定义函数包括以下要素：

1.1.1 函数头：包括函数名和参数（形式参数）

1. 函数名

def 是英文 define 的缩写 函数名称应该能够表达函数封装代码的功能，方便后续的调用注意，在创建函数时，即使函数不需要参数，也必须保留一对空的“()”，否则 Python 解释 器将提示“invaild syntax”错误。

2. 参数（形参和实参）

形参：定义函数时，小括号中的参数，是用来接收参数用的，在函数内部作为变量使用

实参：调用函数时，小括号中的参数，是用来把数据传递到函数内部用的

3. 函数的返回值

return 的作用：

在函数中使用 return 可以从函数返回结果 数学意义上的函数总是返回根据参数计算得到的结果。

函数体中 return 语句有指定返回值时返回的就是其值。在 Python 中，有些函数什么都不返回。

函数体中没有 return 语句，或有 return 语句，但没有在后面指定值，函数运行结束会隐含返回一个 None 作为返回值。（如图 8）

```
def test():
    print('This is printed')
    return #这里使用return语句只是为了结束函数
    print('This is not')

x = test() #调用函数

print(x) #test是个空函数，什么都不返回。但所有的函数都返回值，如果你没有告诉他们该返回什么，将返回None
```

```
This is printed
None
```

图表 8 为了结束函数的 return 用法

1.1.2 调用函数

调用函数也就是执行函数，如果把创建的函数理解为一个具有某种用途的工具，那么调用函数就相当于使用该工具。

函数调用的基本语法格式如下所示：（如图 9）

```
#[返回值] = 函数名([形参值])
#x,y为实际参数

z = max(x,y)
```

图表 9 函数调用的基本语法格式

1.1.3 返回多个值

函数可以返回多个值吗？答案是肯定的。但其实这只是一种假象，Python 函数返回的仍然是单一值，返回多值其实就是返回一个 tuple，但写起来更方便。

无论定义的是返回什么类型，return 只能返回单值，但值可以存在多个元素；

return [1,3,5] 是指返回一个列表，是一个列表对象，1,3,5 分别是这个列表的元素；

return 1,3,5 看似返回多个值，隐式地被 Python 封装成了一个元祖返回。（如图 10）

```
def test():
    a=11
    b=22
    c=33
    return (a,b,c)

num=test()
print (num)
```

```
(11, 22, 33)
```

图表 10 函数返回多个值

1.1.4 定义空函数与 pass 的使用

Python pass 是空语句，是为了保持程序结构的完整性。

pass 不做任何事情，一般用做占位语句。 如果想定义一个什么事也不做的空函数，可以用 pass 语句。（如图 11）

```
def nop():
    pass
```

图表 11 空函数的使用

1.2 函数参数

1.2.1 参数可变吗？（考虑字符串 str 和列表 list）

为何要修改参数

在提高程序的抽象程度方面，使用函数来修改数据结构（如列表或字典）是一种不错的方式。

可以将 python 中常见数据类型按照可变与不可变大致分为两类

不可变数据（四个）：Number（数字）、String（字符串）、Tuple（元组）、Sets（集合）；

可变数据（两个）：List（列表）、Dictionary（字典）

字符串（以及数和元组）是不可变的（immutable），这意味着你不能修改它们（即只能替换为新值），也就是说丢弃原来的存储空间，将变量名链接到新的空间中。而像 list 和 dict 是支持增删改的。（如图 12-14）

```
#在函数内部给参数赋值对外部没有任何影响.
def try_to_change(n):
    n = 'Mr. Gumby'

name = 'Mrs. Entity'
try_to_change(name)  #在try_to_change里，将新值赋予给了参数n，这对变量name没有影响，这是一个完全不同的变量
print (name)
```

```
Mrs. Entity
```

图表 12 字符串替换为新值

```
#如果参数为可变的数据结构（如列表）呢？
def change(n):
    n[0] = 'Mr. Gumby'

names = ['Mrs. Entity', 'Mrs. Thing']
change(names)
names    #在这个示例中，也在函数内修改了参数，但这个示例与前一个示例之间存在一个重要的不同。在前一个示例中，
```

```
['Mr. Gumby', 'Mrs. Thing']
```

图表 13 可变数据列表

```
names = ['Mrs. Entity', 'Mrs. Thing']
n = names # 再次假装传递名字作为参数
n[0] = 'Mr. Gumby' # 修改列表
names
```

```
['Mr. Gumby', 'Mrs. Thing']
```

图表 14 可变数据列表

1.2.2 位置参数

位置参数要求参数的顺序和个数要和函数定义中一致，比如 funcB(100, 99)和 funcB(99, 100)的执行结果是不一样的。

在下图调用 printN 时，我们可以按照位置参数的规则传递实参，即实参必须和函数头中定义的形参在顺序、个数、类型上匹配。（如图 15）

```
#将一个字符串（strA）打印多次
def printN(strA,N):
    for _ in range(N):  #只想for循环，而不需要引用具体数值，不想给数值起名字，那么就叫它"_"丢弃变量
        print(strA)

printN("hello world",10)
print (printN)
```

```
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
<function printN at 0x000001A1844C2048>
```

图表 15 位置参数实参必须和形参在顺序个数类型上匹配

1.2.3 关键字参数

除了使用位置参数的规则传递参数外，也可以使用关键字参数的规则传递参数。也就是在调用函数的时候，明确指定参数值付给那个形参。

有时候，参数的排列顺序可能难以记住，尤其是参数很多时。为了简化调用工作，可指定参数的名称。（如图 16）

```
def printN(strA,N):
    for _ in range(N): #_在python中的应用
        print(strA)

printN(N=10,strA = 'helloworld') #指定参数名称
print(printN)
```

```
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
<function printN at 0x000001A184405378>
```

图表 16 关键字参数的应用

另外，在函数调用中，可以混合使用基于位置匹配的参数和关键字参数，前提是先给出固定位置的参数，否则解释器将不知道它们是哪个参数（即不知道参数对应的位置）。比如图 17-18。

```
def hello_(name, greeting='Hello', punctuation='!'):
    print('{}, {}{}'.format(greeting, name, punctuation))

hello_('Mars', 'Howdy', '...')

print(hello_)
```

```
Howdy, Mars...
<function hello_ at 0x000001A1844C6268>
```

图表 17 混合使用位置参数和关键字参数

```
def hello_(name, greeting='Hello', punctuation='!'):  #hello是关键字参数的默认值
    print('{}, {}{}'.format(greeting, name, punctuation))

hello_('Mars', greeting='top of the morning to you')

print(hello_)
```

```
top of the morning to you, Mars!
<function hello_ at 0x000001A1844C2EA0>
```

图表 18 混合使用位置参数和关键字参数

1.2.4 含默认值的参数

关键字参数最大的优点在于，可以指定默认值。 带有默认值的参数一定是右连续的，有默认值的参数的右边的参数不能没有默认值。（如图 18）

```
def mypow(x, N=1):
    return x ** N

print(mypow(3))  #因为没有传入N的值，因此默认N=1,即求3的一次方
print(mypow(3,5))  #求3的5次方
```

```
3
243
```

图表 19 含默认值的关键字参数

1.2.5 收集参数

收集位置参数的*

在参数前加了一个 * 号，函数可以接收零个或多个值作为参数。返回结果是一个元组。传递零个参数时函数并不报错，而是返回一个空元组。但以上这种方法也有局限性，它不能收集关键字参数。（如图 20）

```
def print_params_2(title, *params):   #此星号意味着收集余下的位置参数。
    print(title)
    print(params)

print_params_2('Params:', 1, 2, 3)


print_params_2('Nothing:') #如果没有可供收集的参数，params将是一个空元组
```

```
Params:
(1, 2, 3)
Nothing:
()
```

图表 20 收集位置参数的*

收集关键字参数的**

对关键字参数进行收集的另一种 收集参数 机制：使用两个星号（**），用法同上。最后返回一个以参数名为键、参数值为键值的字典。（如图 21）

```
def print_params_3(**params):   #**可以自定义参数
    print(params)

print_params_3(x=1, y=2, z=3)   #这样得到的是一个字典而不是元组
```

```
{'x': 1, 'y': 2, 'z': 3}
```

图表 21 收集关键字参数的**

和 *是可以一起使用的，返回特定的结果。（如图 22）

```
def print_params_4(x, y, z=3, *pospar, **keypar):
    print(x, y, z)
    print(pospar)
    print(keypar)

print_params_4(1, 2, 3, 5, 6, 7, foo=1, bar=2)
```

```
1 2 3
(5, 6, 7)
{'foo': 1, 'bar': 2}
```

图表 22 混合使用收集位置参数*和收集关键字参数**

1.3 函数文档

1.3.1 help()

只要是函数，都可以用 help（）来查看函数的文档，也就是函数的说明书

如果是自定义函数，如何查看它的文档？在 def 后面首行使用''' '''三个单引号对自定义函数进行说明，之后使用 help() 就可以查看自定义文档的自定义说明了。

说明要包括的内容:1. 对函数的具体描述 2. 对函数的参数的说明（如图 23-24）

```
def test(a,b):
    print("%d"%(a+b))

test(11,22)

help(test) #能够看到test函数的相关说明 用来完成对两个数求和
```

```
33
Help on function test in module __main__:

test(a, b)
```

图表 23 help()查看函数文档

```
def sum(num1,num2):
    '''这个函数实现了对两个数的和的打印''' #自定义函数的文档说明必须放在第一行

    print(num1+num2)

help(sum)
```

```
Help on function sum in module __main__:

sum(num1, num2)
    这个函数实现了对两个数的和的打印
```

图表 24 自定义函数的文档说明必须放在定义后的第一行

1.3.2 *doc*

doc 是每个对象都有的属性，其存放了对象文档。（如图 25）

```
import struct

struct.__doc__
```

```
"Functions to convert between Python values and C structs.\nPython bytes objects are used to hold the data repre
senting the C struct\nand also as format strings (explained below) to describe the layout of data\nin the C stru
ct.\n\nThe optional first format char indicates byte order, size and alignment:\n  @: native order, size & align
ment (default)\n  =: native order, std. size & alignment\n  <: little-endian, std. size & alignment\n  >: big-en
dian, std. size & alignment\n  !: same as >\n\nThe remaining chars indicate types of args and must match exactl
y;\nthese can be preceded by a decimal repeat count:\n  x: pad byte (no data); c:char; b:signed byte; B:unsigned
byte;\n  ?: _Bool (requires C99; if not available, char is used instead)\n  h:short; H:unsigned short; i:int; I:
unsigned int;\n  l:long; L:unsigned long; f:float; d:double; e:half-float.\nSpecial cases (preceding decimal cou
nt indicates length):\n  s:string (array of char); p: pascal string (with count byte).\nSpecial cases (only avai
lable in native format):\n  n:ssize_t; N:size_t;\n  P:an integer type that is wide enough to hold a pointer.\nSp
ecial case (not in native mode unless 'long long' in platform C):\n  q:long long; Q:unsigned long long\nWhitespa
ce between formats is ignored.\n\nThe variable struct.error is an exception raised on errors.\n"
```

图表 25 doc 查看对象文档

1.4 函数作用域

1.4.1 什么是函数作用域

变量到底是什么呢？可将其视为指向值的名称。因此，执行赋值语句 x=1 后，名称 x 指向值 1。这几乎与使用字典一样(字典中的键指向值)，只是你使用的是"看不见"的字典。实际上，这种解释已经离真相不远。有一个名为 vars 的内置函数，它返回这个不可见的字典

Python 中，程序的变量并不是在哪个位置都可以访问的，访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序可以访问哪个特定的变量名称。

1.4.2 全局作用域与局部作用域的区别

局部变量

在函数内部被定义的变量，只能在函数内部访问。

全局变量

在所有函数之外创建的变量，可以被所有的函数访问。

```
globalvar = 1 #全局变量

def f1():
    localvar = 2 #局部变量
    print(globalvar)
    print(localvar)

f1()
print(globalvar)
print(localvar)
```

```
1
2
1

---------------------------------------------------------------------------
```

图表 26 局部变量 lovalvar 不能在函数外访问

globalvar 是全局变量，localvar 是局部变量，globalvar 可以在函数中访问，但是 localvar 却不能在函数外访问，会报错。（如图 26）

1.4.3 在局部作用域中，全局变量遮盖的问题

如果有一个局部变量或参数与你要访问的全局变量同名，就无法直接访问全局变量，因为它被局部变量遮住了。

全局变量与局部变量同名：如果局部变量与全局变量同名，那么函数中将优先使用的是局部变量的值，而不是使用全局变量。（如图 27）

```
firstValue = "Hello World"

def printStr():

    global firstValue
    firstValue = "hi man"
    print(firstValue) #注意：这里的firstValue调用的是局部变量firstValue，在方法中直接覆盖掉同名的全局变量firstValue

printStr()
print(firstValue)
```

```
hi man
hi man
```

图表 27 局部变量与全局变量同名，优先访问局部变量的值

python 访问变量的顺序： 当前作用域局部变量->外层作用域变量->当前模块中的全局变量->python 内置变量.

global

global 可以将局部变量的作用域改为全局。 如果需要，可使用函数 global 来访问全局变量。这个函数类似于 vars，返回一个包含全局变量的字典。(locals 返回一个包含局部变量的字典。)（如图 28-30）

```
x = 123

def testGlobal():
    global x

    x = 100
    print(x)

testGlobal()
x
```

```
100

100
```

图表 28 global 实现在函数内部访问全局变量

```
x = 123

def testGlobal():

    x = 100
    print(x)

testGlobal()
print(x)
```

```
100
123
```

图表 29 没有实现 global 依然返回全局变量的值

```
count = 5

def func():
    global count
    count = 10
    print(count)

func()
print(count)
```

```
10
10
```

图表 30 使用 global 在函数内部修改全局变量 count 的值

1.4.4 全局变量、外部非全局变量，在局部作用域中的重新关联

全局变量在局部作用域中的重新关联。（如图 31）

```
gcount = 0#第一行定义了一个全局变量

def global_test():
    global gcount #在内部函数中声明gcount为全局变量
    gcount +=1
    print(gcount) #打印的值为全局变量值为1

global_test()
print(gcount)  #打印的值为全局变量值为1
```

```
1
1
```

图表 31 全局变量在局部作用域中的重新关联

外部非全局变量在局部作用域中的重新关联

nonlocal 关键字用来在函数或其他作用域中使用外层(非全局)变量

nonlocal 适用于在局部函数中调用另一个局部函数的局部变量， 把最内层的局部变量设置成外层局部或者其它局部可用，但不可在全局使用

简而言之，nonlocal 的作用：在一个函数中调用另一个函数的私有化变量

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter

def make_counter_test():
    mc = make_counter()
    print(mc())
    print(mc())
    print(mc())

make_counter_test()
```

```
1
2
3
```

图表 32 外部非全局变量在局部作用域中的重新关联

1.5 递归调用

1.5.1 递归是什么

递归意味着引用(这里是调用)自身。可使用递归完成的任何任务都可使用循环来完成，但有时使用递归函数的可读性更高。

基线条件（针对最小的问题）：满足这种条件时函数将直接返回一个值。

递归条件：包含一个或多个调用，这些调用旨在解决问题的一部分。（如图 33）

```
#计算幂
def power(x, n):
    if n == 0:
        return 1  #对于任何数字x，power(x, 0)都为1

    else:
        return (x * power(x, n-1))    #n>0时，power(x, n)为power(x, n-1)与x的乘积

print (power(2,3))
```

```
8
```

图表 33 计算幂的递归调用

1.5.2 递归在二分查找中的应用

如果上限和下限相同，就说明它们都指向数字所在的位置，因此将这个数字返回。

否则，找出区间的中间位置（上限和下限的平均值），再确定数字在左半部分还是右半部分。然后在继续在数字所在的那部分中查找。（如图 34）

```
#对方心里想着一个1～100的数字，你必须猜出是哪个，不断将可能的区间减半，直到猜对为止
def search(sequence, number, lower, upper):  #在列表中找数字的位置
    if lower == upper:
        assert number == sequence[upper]
        return upper    #如果上限和下限相同，就说明它们都指向数字所在的位置，因此将这个数字返回

    else:
        middle = (lower + upper) // 2  #否则，找出区间的中间位置（上限和下限的平均值）


        if number > sequence[middle]:   #再确定数字在左半部分还是右半部分，然后在继续在数字所在的那部分中查找。
            return search(sequence, number, middle + 1, upper)

        else:
            return search(sequence, number, lower, middle)   #如果要查找的数字更大，肯定在右边；如果更小，
```

图表 34 递归在二分查找中的应用

二、类与对象

2.1 基本概念

2.1.1 什么是类？什么是对象？

对象是对客观事物的抽象，类是对对象的抽象。两者的关系：对象是类的实例，类是对象的模板。

2.1.2 什么是子类？什么是超类？

被继承的类一般称为“超类”或“父类”，继承的类称为“子类”。鸟类是一个非常通用（抽象）的类，它有多个子类：你看到的那只鸟可能属于子类“云雀”。你可将“鸟类”视为由所有鸟组成的集合，而“云雀”是其一个子集。一个类的对象为另一个类的对象的子集时，前者就

是后者的子类。因此“云雀”为“鸟类”的子类，而“鸟类”为“云雀”的超类。

2.2 定义类

2.2.1 定义长方形和长方体类

长方形类，数据成员为长、宽。（如图 35）

```
In [1]: class Rectangle:
            def __init__(self, length, width):
                self.length = length
                self.width = width

            def area(self,):
                return self.length * self.width
```

```
In [2]: rectangle_1 = Rectangle(1,2)
```

```
In [3]: print("长:", rectangle_1.length, "宽:", rectangle_1.width)
```

```
长: 1 宽: 2
```

图表 35 定义长方形

长方形木板类（继承长方形），数据成员为长、宽、厚度，设定厚度的默认值为 1。在定义长方形木板时，由于其本身含有长方形的一些属性，所以我们选择继承长方形类，并在此基础上添加只属于长方形木板类的特有属性和成员函数。（如图 37）

```
In [4]: class Cube(Rectangle):
            def __init__(self,length, width, height=1):
                super().__init__(length, width)
                self.height = height

            def area(self,):
                return super().area()

            def volume(self,):
                return self.length * self.width * self.height
```

```
In [5]: cube_1 = Cube(1,2,3)
```

```
In [6]: print("长:", cube_1.length, "宽:", cube_1.width, "高:", cube_1.height)
```

```
长: 1 宽: 2 高: 3
```

图表 36 定义长方形模板类

长方形类的成员函数为计算面积。（如图 38）

```
In [7]: rectangle_2 = Rectangle(2,5)
```

```
In [8]: print("面积:", rectangle_2.area())
```

```
面积: 10
```

图表 37 长方形成员函数

长方形木板类的成员函数为计算底面积和计算体积。（如图 39）

```
In [9]: cube_2 = Cube(5,4,3)
```

```
In [10]: print("底面积：", cube_2.area(), "体积：", cube_2.volume())
```

```
底面积： 20 体积： 60
```

图表 38 长方形模木板类成员函数

2.2.2 类的相关介绍

对象的生成：实例化对象 = 类名(参数)。如图（40-42）

```
In [11]: cube_3 = Cube(2, 3, 4)
```

图表 39 对象的生成

属性：数据成员。（如图 6）

```
In [14]: cube_3.length
Out[14]: 2
```

图表 40 属性：数据成员

普通成员函数是实现类或者对象中的功能的函数，只有在调用时才会作用。（如图 8）

```
In [15]: cube_3.area()
Out[15]: 6
```

图表 41 普通成员函数

构造函数每次实例化类时都会执行一次。super().__ init __：继承父类的 init 方法。self: 指向对象本身。（如图 43）

```
In [16]: class Hello_cube(Cube):
             def __init__(self, length, width, height=1):
                 print("hello")
                 print(isinstance(self, Hello_cube))
                 super().__init__(length, width, height)

In [17]: hello_cube_1 = Hello_cube(2, 3, 4)
         hello
         True
```

图表 42 构造函数

对象的方法(成员函数的调用)：对象名.成员函数。（如图 44）

```
In [18]: cube_3.volume()
Out[18]: 24
```

图表 43 函数成员的调用

2.2.3 什么是封装

指的是向外部隐藏不必要的细节。

2.2.4 什么是多态

即便不知道变量指向的是哪种对象，也能够对其执行操作，且操作的行为将随对象所属的类型（类）而异。（如图 45）

```
In [19]: x = Cube(1, 2, 3)
         x_1 = Rectangle(1, 2)

In [20]: x.area()
Out[20]: 2

In [22]: x_1.area()
Out[22]: 2
```

图表 44 多态

2.2.5 什么是继承

它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

2.3 类的命名空间

在 class 语句中定义的代码都是在一个特殊的命名空间（类的命名空间）内执行的，而类的所有成员都可访问这个命名空间。

2.3.1 类体内函数体外的变量

类体内函数外的变量，对于每个该类初始化的对象来说都是公共的。（如图 46）

```
In [36]: class MemberCounter:
             members = 0
             def __init__(self, step=1):
                 self.step = step
                 MemberCounter.members += 1 # 静态成员变量

In [37]: m1 = MemberCounter()

In [38]: m1.members
Out[38]: 1

In [39]: m1.step
Out[39]: 1

In [40]: MemberCounter.members
Out[40]: 1

In [41]: MemberCounter.step
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-41-70446328274d> in <module>
----> 1 MemberCounter.step

AttributeError: type object 'MemberCounter' has no attribute 'step'
```

图表 45 类体内函数体外的变量

2.3.2 该类变量的应用

该类变量的主要应用适用于对象计数。（如图 47）

```
In [42]: class MemberCounter:
             members = 0
             def __init__(self, step=1):
                 self.step = step
                 MemberCounter.members += 1

In [43]: m1 = MemberCounter()
         m2 = MemberCounter()

In [44]: m1.members
Out[44]: 2

In [45]: m2.members
Out[45]: 2
```

图表 46 该类变量的应用

2.4 如何修改属性（数据成员）的值

2.4.1 直接修改

```
In [46]: class Dog:
             def __init__(self, name):
                 self.name = name

             def change_name(self, name):
                 self.name = name

In [47]: dog_1 = Dog("jim")

In [48]: dog_1.name = "jack"

In [49]: dog_1.name
Out[49]: 'jack'
```

图表 47 直接修改

2.4.2 通过成员函数修改

```
In [51]: class DOG():
             def __init__(self, name):
                 self.__name = name

             def change_name(self, name):
                 self.__name = name
                 return self.__name

In [53]: dog_2 = DOG("jim")

In [56]: dog_2.__name
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-56-030884015482> in <module>
----> 1 dog_2.__name

AttributeError: 'DOG' object has no attribute '__name'

In [57]: dog_2.change_name("jack")
Out[57]: 'jack'
```

图表 48 通过成员函数修改

一旦我们将对象的属性设为私有的，那么我们之恩那个通过成员函数来进行修改。这样的好处在于防止误操作。（如图 49）

2.5 多重继承

哺乳类：能跑的哺乳类，能飞的哺乳类；鸟类：能跑的鸟类，能飞的鸟类。（如图 50）

```
In [45]: class Animal(object):
             pass

In [46]: class Mammal(Animal):
             pass

         class Bird(Animal):
             pass

In [47]: class Runnable(object):
             def run(self):
                 print('Running...')

         class Flyable(object):
             def fly(self):
                 print('Flying...')

In [48]: class Dog(Mammal, Runnable):
             pass

In [49]: class Bat(Mammal, Flyable):
             pass
```

图表 49 多重继承

2.6 抽象基类

"抽象基类"这个词可能听着比较"深奥",其实"基类"就是"父类","抽象"就是"假"的意思, "抽象基类"就是"假父类."

```
In [50]: from abc import ABC, abstractmethod

In [51]: class Talker(ABC):
             @abstractmethod
             def talk(self):
                 pass

In [53]: class Knigget(Talker):
             pass

In [54]: Knigget()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-54-804f3a16fc5d> in <module>
----> 1 Knigget()

TypeError: Can't instantiate abstract class Knigget with abstract methods talk

In [55]: class Knigget(Talker):
             def talk(self):
                 print("Ni!")

In [56]: k = Knigget()

In [57]: k.talk()
         Ni!
```

图表 50 抽象基类

抽象基类的好处在与我们定义一个类的时候，并不要清楚是如何实现的。在具体操作的时候可以进行具体化，从而实现多态性。（如图 51）

2.7 hasattr() setattr() getattr()

hasattr() 函数用于判断对象是否包含对应的属性。（如图 52）

```
In [62]: hasattr(rectangle_1, 'length')
Out[62]: True

In [63]: hasattr(rectangle_1, 'height')
Out[63]: False
```

图表 51 hasattr() 函数

setattr() 函数的功能相对比较复杂，它最基础的功能是修改类实例对象中的属性值。（如图 53）

```
In [64]: cube_1.height
Out[64]: 4

In [66]: setattr(cube_1,"height",56)

In [67]: cube_1.height
Out[67]: 56
```

图表 52 setattr() 函数

getattr() 函数获取某个类实例对象中指定属性的值。（如图 54）

```
In [69]: cube_1.length
Out[69]: 2

In [70]: getattr(cube_1, "length")
Out[70]: 2

In [71]: getattr(Cube, "area")
Out[71]: <function __main__.Cube.area(self)>
```

图表 53 getattr() 函数

三、魔法方法

3.1 概念介绍："方法"VS"函数"VS"属性"

1.函数（FunctionType）：函数是封装了一些独立的功能，可以直接调用，能将一些数据（参数）传递进去进行处理，然后返回一些数据（返回值），也可以没有返回值。可以直接在模块中进行定义使用。 所有传递给函数的数据都是显式传递的。

2.方法（MethodType）：方法和函数类似，同样封装了独立的功能，但是方法是只能依靠类或者对象来调用的，表示针对性的操作。方法中的数据 self 和 cls 是隐式传递的，即方法的调用者；方法可以操作类内部的数据。

简单的说，函数在 python 中独立存在，可直接使用的，而方法是必须被别人调用才能实现的。

3.属性(Attribute)：类和对象的属性，储存某个值（int,str,bool...），代表对象的某种性质或特征。

然而，对于本模块需要讲解的"魔法方法"，我们可以理解为与普通方法的调用和使用有所不同的方法，具有特殊的固定名称，便于人们辨别。

我们可以使用函数 dir()来查看一划线 "__xxx__" 的为隐函数（图 X），我们今天所介绍的魔法方法就包含其中。（如图 55）

```
dir(list)
```

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
```

图表 54 查看类的方法和属性

列表类中的 add 函数，作为一种“魔法方法”，我们可以用“+”对其调用（如图 56）。

```
a=[1,2,3,4,5]
print(a+[6])
print(a.__add__([6]))
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

图表 55 魔法方法——加法

3.2 基本的序列（字符串、列表、元祖）和映射（字典）的协议（方法）

序列和映射基本上是元素（item）的集合，要实现它们的基本行为（协议），不可变对象需要实现 2 个方法，而可变对象需要实现 4 个。

1.__len__(self)：这个方法应返回集合包含的项数，对序列来说为元素个数，对映射来说为键值对数。如果__len__返回零（且没有实现覆盖这种行为的__nonzero__），对象在布尔上下文中将被视为假（就像空的列表、元组、字符串和字典一样）。

2.__getitem__(self, key)：这个方法应返回与指定键相关联的值。对序列来说，键应该是 0~n-1 的整数，其中 n 为序列的长度。对映射来说，键可以是任何类型。

3.__setitem__(self, key, value)：这个方法应以与键相关联的方式存储值，以便以后能够使用__getitem__来获取。当然，仅当对象可变时才需要实现这个方法。(

4.__delitem__(self, key)：这个方法在对对象的组成部分使用__del__语句时被调用，应删除与 key 相关联的值。同样，仅当对象可变（且允许其项被删除）时，才需要实现这个方法。

例如：列表作为可变对象，可以使用以上四种方法（如图 57）

```
a=[1,4,5,7,8]
print(len(a))
print(a[3])
a[4]=2
print(a[4])
del(a[2])
print(a)
```

```
5
7
2
[1, 4, 7, 2]
```

图表 56 可变对象适用四种方法

但是，元组作为不可变对象，只能使用前两种方法，使用”__setitem__”和“__delitem__”时会报错（如图 58）

```
b=(1,2,3,4,5,6)
print(len(b))
print(b[3])
#b[2]=4
#del(b[1])
```

```
6
4
```

```
b=(1,2,3,4,5,6)
#print(len(b))
#print(b[3])
b[2]=4
#del(b[1])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-ea2166f926a8> in <module>
      2 #print(len(b))
      3 #print(b[3])
----> 4 b[2]=4
      5 #del(b[1])

TypeError: 'tuple' object does not support item assignment
```

```
b=(1,2,3,4,5,6)
#print(len(b))
#print(b[3])
#b[2]=4
del(b[1])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-6b8593684fa7> in <module>
      3 #print(b[3])
      4 #b[2]=4
----> 5 del(b[1])

TypeError: 'tuple' object doesn't support item deletion
```

图表 57 不可变对象只适用两种方法

我们知道 range 函数既可以生成一系列连续的整数，也可以生成指定步长的序列。下面我们将改写”__len__”和“__getitem__”，实现这一功能。（如图 59）

```
class Range:

    def __init__(self, start, stop = None, step = 1) :
    ###Initialize a Range instance .
        if step == 0:
            raise ValueError('step cannot be 0')
        if stop is None: # special case of range (n)
            start, stop = 0, start # should be treated as if range (0 ,n)
         # calculate the effective length once
        self._length = max(0, (stop - start + step - 1) // step)
        # need knowledge of start and step ( but not stop ) to support __getitem__
        self._start = start
        self._step = step

    def __len__(self):
    #Return number of entries in the range .
        return self._length

    def __getitem__(self, k):
    ##Return entry at index k ( using standard interpretation if negative ).
        if k < 0:
            k += len(self) # attempt to convert negative index
        if not 0 <= k < self._length :
            raise IndexError('index out of range')
        return (self._start + k * self._step)
```

```
a=Range(1,10,2)
print(len(a))
print(a[2])
```

```
5
5
```

图表 58 改写"__len__"和"__getitem__"

3.3 静态方法和类方法

3.3.1 概念介绍

1.实例方法 ：第一个参数必须是实例对象，该参数名一般约定为“self”，通过它来传递实例的属性和方法（也可以传类的属性和方法）；只能由实例对象调用。

2.类方法：使用装饰器@classmethod。第一个参数必须是当前类对象，该参数名一般约定为“cls”，通过它来传递类的属性和方法（不能传实例的属性和方法）；实例对象和类对象都可以调用。

3.静态方法：使用装饰器@staticmethod。参数随意，没有“self”和“cls”参数，但是方法体中不能使用类或实例的任何属性和方法；实例对象和类对象都可以调用。

参考于 https://www.cnblogs.com/geogre123/p/10142510.html

3.3.2 类方法:

使用装饰器@classmethod。原则上，类方法是将类本身作为对象进行操作的方法。假设有个方法，且这个方法在逻辑上采用类本身作为对象来调用更合理，那么这个方法就可以定义为类方法。另外，如果需要继承，也可以定义为类方法。

例题：假设我有一个学生类和一个班级类，想要实现的功能为：

1.执行班级人数增加的操作、获得班级的总人数；

2.学生类继承自班级类，每实例化一个学生，班级人数都能增加；

3.最后，我想定义一些学生，获得班级中的总人数。（图 X）

那么这个问题用类方法做比较合适，为什么？

因为我实例化的是学生，但是如果我从学生这一个实例中获得班级总人数，在逻辑上显然是不合理的。同时，如果想要获得班级总人数，如果生成一个班级的实例也是没有必要的。（如图 60）

```
class ClassTest(object):
    __num = 0

    @classmethod
    def addNum(cls):
        cls.__num += 1

    @classmethod
    def getNum(cls):
        return cls.__num

    # 这里我用到魔术函数__new__，主要是为了在创建实例的时候调用人数累加的函数。
    def __new__(self):
        ClassTest.addNum()
        return super(ClassTest, self).__new__(self)


class Student(ClassTest):
    def __init__(self):
        self.name = ''

a = Student()
b = Student()
print(ClassTest.getNum())
```

```
2
```

图表 59 类方法示例

3.3.3 静态方法：

使用装饰器@staticmethod。静态方法是类中的函数，不需要实例。静态方法主要是用来存放逻辑性的代码，逻辑上属于类，但是和类本身没有关系，也就是说在静态方法中，不会涉及到类中的属性和方法的操作。可以理解为，静态方法是个独立的、单纯的函数，它仅仅托管于某个类的名称空间中，便于使用和维护。

譬如，无论什么狗都是汪汪叫的.（如图 61）

```
class Dog():
    @staticmethod
    def dark():
        print('wang,wang,wang...')
Dog.dark()

d=Dog()
d.dark()
```

```
wang,wang,wang...
wang,wang,wang...
```

图表 60 静态方法示例

如上，使用了静态方法（函数），然而方法体中并没使用（也不能使用）类或实例的属性（或方法）。若要获得当前时间的字符串时，并不一定需要实例化对象，此时对于静态方法而言，所在类更像是一种名称空间。

其实，我们也可以在类外面写一个同样的函数来做这些事，但是这样做就打乱了逻辑关系，也会导致以后代码维护困难。

3.4 类和对象的魔法方法

既对类和对象的属性的系列操作：

这些魔法方法是拦截对对象属性的所有访问企图，在属性被访问时执行的一段代码。下面的四个魔法方法提供了你需要的所有功能（在旧式类中，只需使用后面三个）。（Python3 默认都是新式类）

1.__getattribute__(self, name)：在属性被访问时自动调用（只适用于新式类）。

2.__getattr__(self, name)：在属性被访问而对象没有这样的属性时自动调用。

3.__setattr__(self, name, value)：试图给属性赋值时自动调用。

4.__delattr__(self, name)：试图删除属性时自动调用。

我们随意构建一个类 A，使用“dir()”函数可以看到类 A 具有以上四种方法。（”__getattr__”方法没有出现，由于其功能被” __getattribute__”所包含了。）（如图 62）

```
class A:
    def __init__(self,height=1,width=2):
        self.height=height
        self.width=width
a=A()
print(dir(a))
print(a.width)
a.height=5
print(a.height)
del(a.height)
a.height
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get
attribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'height', 'width']
```

图表 61 查看类和对象的魔法方法

以下是实验这四种方法的结果（如图 63）：

```
class A:
    def __init__(self,height=1,width=2):
        self.height=height
        self.width=width
a=A()
dir(a)
print(a.width)
a.height=5
print(a.height)
del(a.height)
a.height
```

```
2
5
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-8dbb58c93250> in <module>
      9 print(a.height)
     10 del(a.height)
---> 11 a.height

AttributeError: 'A' object has no attribute 'height'
```

图表 62 这四种魔法方法的使用

我们通过构建“Recrangle”为例，通过改写“__setattr__”和“__getattr__”，实现设置和获取长方形长和宽的功能（如图 64）：

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def __setattr__(self, name, value):
        if name == 'size':
            self.width, self.height = value
        else:
            self.__dict__[name] = value
    def __getattr__(self, name):
        if name == 'size':
            return self.width, self.height
        else:
            raise AttributeError()

a=Rectangle()
a.width=3
a.height=4
a.size
```

```
(3, 4)
```

图表 63 改写"__setattr__"和"__getattr__"