# Python Cheat Sheet-8. 类(OOP)\_Classes\_定义，继承，init()构造方法，私有变量/方法

?> PCS-8（    ）：8.1 OOP(Object-oriented programming)\_面向对象编程，及Classes定义和\_\_init\_\_()构造方法；8.2 Inheritance（继承）——superclasses（父类）和subclasses（子类）；8.3 private variables(私有变量/成员)和private methods(私有方法)；8.4 Operator Overloading（运算符/操作符重载） 与类属性格式"字典"

知识点

描述

代码段

运算结果

备注

**8.1** OOP(Object-oriented programming)\_面向对象编程，及Classes定义和\_\_init\_\_()构造方法

* OOP与定义类

面向对象编程（Object-oriented programming，OOP）是一种计算机编程模型，是围绕数据或对象而不是功能和逻辑来组织软件设计，更专注于对象与对象之间的交互，对象涉及的方法（methods）和属性(attributes)都在对象内部。其中类（Classes）是相同种类对象的抽象，是该类对象的公共属性；将该类实例化一个或多个对象（instance objects），即为该对象的实例。例如，定义一个类为鸟类，所有的鸟类都具有觅食的行为，可以将该行为定义为类的函数（成员函数，member functions），称为类的方法；可以将鸟类的飞翔动作定义为类的变量（成员变量），称为类的属性。鸟类通常会根据鸟类特征划分很多子类，例如雨燕目、 鸽形目、 雁形目等，不同子类的羽毛颜色、鸟鸣声都会不同，因此可以基于已有鸟类（父类，superclass），定义继承父类属性方法的子类（subclass）。

python类提供了面向对象编程的所有标准特性：类的继承机制（inheritance mechanism）允许多个基类（basse classes/superclasses）；一个派生类（derived class/subclass）可以覆盖基类或任何类的方法；一个方法可以调用（call/invoke）其基类的方法；对象可以包含任意数量和种类的数据；类具有动态特性（dynamic nature），可以在运行时添加或者删除属性方法等。

类定义的语法，可简单表示为：

class ClassName:  
 <statement-1>  
 .  
 .  
 .  
 <statement-N>

通常类的定义名首字母大写（共识非标准），其中语句一般为函数定义（允许其它类型的语句，例如定义变量赋值语句等）。下面引入[Python官网的几个小案例](https://docs.python.org/3/tutorial/classes.html)①，稍许调整,辅助理解类定义和调用的方式。

下述类定义的类名称为SayHello，该类之下的开始行，函数定义之前，定义了一行赋值语句name\_default='Who'；定义的函数helloWorld(self,name)，其中self参数为类方法定义的第一个必要参数（本身的意思），在方法调用时指向类的实例。同时该函数也传入了另一个参数name，指定默认值为变量name\_default。同时需要注意，通过该函数内print(SayHello.name\_default)语句可以发现，通过Class.attribute的方式可以调用函数外类内的变量。x=SayHello()即实例化类对象语句，将类实例化为变量x，则该实例x具有了该类的所有属性和方法。x.name\_default为调用类属性的方式；x.helloWorld()和x.helloWorld('Swift')为调用类方法（函数）的方式。

类可以使用内置方法\_\_doc\_\_读取类下的注释语句。

class SayHello:  
 """A simple example class"""  
 name\_default='Who'  
  
 def helloWorld(self,name=name\_default):   
 print(SayHello.name\_default) #可以通过Class.attribute的方式获取函数外的变量值  
 return '%s says hello, world!'%name  
   
x=SayHello()  
print(x.name\_default)  
print(x.helloWorld())  
print(x.helloWorld('Swift'))  
  
print('-'\*50)  
print(SayHello.\_\_doc\_\_)

Who  
Who  
Who says hello, world!  
Who  
Swift says hello, world!  
--------------------------------------------------  
A simple example class

通过help()可以查看定义类的内容，这包括开始行的注释语句，方法（Methods）和属性（attributes），及定义的数据描述（data description defined）等。其中\_\_dict\_\_是存储对象属性的一个字典，键为属性名，而值为属性值。通过\_\_dict\_\_可以查看所有类的属性和方法。

print(help(SayHello))  
print(SayHello.\_\_dict\_\_)  
print('-'\*50)  
print(SayHello.\_\_dict\_\_['name\_default'])  
print(x.\_\_dict\_\_)

Help on class SayHello in module \_\_main\_\_:  
  
class SayHello(builtins.object)  
 | A simple example class  
 |   
 | Methods defined here:  
 |   
 | helloWorld(self, name='Who')  
 |   
 | ----------------------------------------------------------------------  
 | Data descriptors defined here:  
 |   
 | \_\_dict\_\_  
 | dictionary for instance variables (if defined)  
 |   
 | \_\_weakref\_\_  
 | list of weak references to the object (if defined)  
 |   
 | ----------------------------------------------------------------------  
 | Data and other attributes defined here:  
 |   
 | name\_default = 'Who'  
  
None  
{'\_\_module\_\_': '\_\_main\_\_', '\_\_doc\_\_': 'A simple example class', 'name\_default': 'Who', 'helloWorld': <function SayHello.helloWorld at 0x00000152FBEC9CA0>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'SayHello' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'SayHello' objects>}  
--------------------------------------------------  
Who  
{}

* '\_\_init\_\_()'构造方法与可变对象

如果类实例化时要传入参数（初始化参数），则可以定义类时，定义一个名为\_\_init\_\_()的特殊方法（constructor method，构造方法），下述案例为构造方法传入了一个name参数。在实例化时，如果该参数未指定默认值，则需要传入该参数值，如d = Dog('Fido')。类的实例化会自动掉调用\_\_init\_\_()来处理新创建的类的实例，通过self.name = name，初始化对应的参数，则self.name变量的作用域为整个类，内部的所有函数均可以调用self.name变量，外部实例化对象也可以通过d.name访问。也可以在该构造函数内书写其它代码完成相关的任务，例如定义新的变量，或相关计算。

下述引用官网的案例在说明\_\_init\_\_()构造方法初始化参数值和相关运算，也阐述在类下函数外定义的变量，如果为可变对象（mutable objects）例如tricks = []，的变量tricks为一个列表，是可变对象，则实例化多个对象，执行类的add\_trick()方法时，新的值都会被追加到tricks列表中，即各个实例化对象的tricks属性值相同，为共同追加的值列表。如果要将其分开，则需要在初始化时，通过self.tricks = []方法定义可变变量，self指向各个实例化对象自身，而不是类本身，因此在各各个实例化对象（d和e）执行add\_trick()方法时，仅会将值追加在各自的self.tricks列表中。

在SayHello案例中，因为没有以self引入的变量，因此print(x.\_\_dict\_\_)时为空。在Dog案例中，因为存在变量self.name，因此print(d.\_\_dict\_\_)时，会返回{'name': 'Fido'}。

class Dog:  
  
 tricks = [] # mistaken use of a class variable  
  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def add\_trick(self, trick):  
 self.tricks.append(trick)  
   
d = Dog('Fido')  
print(d.name)  
e = Dog('Buddy')  
d.add\_trick('roll over')  
e.add\_trick('play dead')  
print(d.tricks)  
print(e.tricks)  
  
print("-"\*50)  
print(d.\_\_dict\_\_)

Fido  
['roll over', 'play dead']  
['roll over', 'play dead']  
--------------------------------------------------  
{'name': 'Fido'}

class Dog:  
  
 def \_\_init\_\_(self, name):  
 self.name = name  
 self.tricks = [] # creates a new empty list for each dog  
  
 def add\_trick(self, trick):  
 self.tricks.append(trick)  
  
d = Dog('Fido')  
e = Dog('Buddy')  
d.add\_trick('roll over')  
e.add\_trick('play dead')  
print(d.tricks)  
print(e.tricks)

['roll over']  
['play dead']

**8.2** Inheritance（继承）——superclasses（父类）和subclasses（子类）

继承主要作用是实现代码的复用，使子类拥有父类的方法和属性，在已有类的继承上扩展子类，可以大幅度降低代码书写量，并避免重复；同时，可以更加方便的理清类之间、子类父类之间的关系，这使得代码逻辑更加清晰；配置类的组合关系，则使得代码书写具有更好的弹性。

继承的语法为：

class DerivedClassName(BaseClassName):  
 <statement-1>  
 .  
 .  
 .  
 <statement-N>

如果父类和子类不在同一个模块（module）中（即不在同一个文件），则可以用下述方法：

class DerivedClassName(modname.BaseClassName):

python的类支持多重继承（multiple inheritance），即可以同时继承多个父类，语法如下：

class DerivedClassName(Base1, Base2, Base3):  
 <statement-1>  
 .  
 .  
 .  
 <statement-N>

* 继承搜索

具有继承关系类的属性和方法搜索，基本路径是自下而上，自左而右。例如上述多重继承，首先在子类自身DerivedClassName中搜索，如果搜索不到，则向上在父类中搜索。首先在Base1中搜索，如果仍搜索不到，则继续搜索Base2和Base3，直至锁定，或者找不到属性和方法，则引发异常。

* 继承的第1个案例——1个父类+1个子类

下述案例定义了父类Bird，拥有的属性有：fly和hungry；拥有的方法有：eat()。定义的子类Apodidae(Bird)，继承父类Bird的所有属性和方法，同时增加了自身的属性有：sound；方法有：sing()。通过super(Apodidae,self).\_\_init\_\_(hungry)语句，为调用父类的初始化方法来初始化子类，也可以在子类中重新定义初始化方法来覆盖父类初始化的属性结果。

class Bird:  
 fly='Whirring' #美 /wɜːr/   
 def \_\_init\_\_(self,hungry=True):  
 self.hungry=hungry  
 def eat(self):  
 if self.hungry:  
 print('Aaaah...')  
 self.hungry=False  
 else:  
 print('No.Thanks!')  
  
class Apodidae(Bird): #/'æpədədi:/  
 def \_\_init\_\_(self,sound='Squawk!',hungry=True):  
 super(Apodidae,self).\_\_init\_\_(hungry)  
 self.sound=sound #美 /skwɔːk/   
 def sing(self):  
 print(self.sound)

将子类实例化为两个实例对象，为swift和blackswift，实例化时，传入的初始化参数不同，对于swift使用了默认参数值，即(Squawk,True)，而blackswift传入的参数为('hooting',False)。子类具有自身定义的属性（sound）和方法(sing())，也继承了父类的属性（hungry,fly）和方法(eat())。

swift=Apodidae()  
print("swift.fly->{}".format(swift.fly))  
print("swift.hungry->{}".format(swift.hungry))  
swift.eat()  
print("swift.hungry->{}".format(swift.hungry))  
swift.eat()  
print("swift.sing->{}".format(swift.sound))  
swift.sing()  
  
print("-"\*50)  
blackswift=Apodidae('hooting',False)  
print("blackswift.fly->{}".format(blackswift.fly))  
print("blackswift.hungry->{}".format(blackswift.hungry))  
blackswift.eat()  
print("blackswift.hungry->{}".format(blackswift.hungry))  
print("blackswift.sing->{}".format(blackswift.sound))  
blackswift.sing()

swift.fly->Whirring  
swift.hungry->True  
Aaaah...  
swift.hungry->False  
No.Thanks!  
swift.sing->Squawk!  
Squawk!  
--------------------------------------------------  
blackswift.fly->Whirring  
blackswift.hungry->False  
No.Thanks!  
blackswift.hungry->False  
blackswift.sing->hooting  
hooting

如果将父类Bird实例化为对象crow，则crow仅具有Bird类的属性和方法，而不具有子类的属性和方法。

crow=Bird()  
print("crow.fly->{}".format(crow.fly))  
print("crow.hungry->{}".format(crow.hungry))  
crow.eat()  
print("crow.hungry->{}".format(crow.hungry))  
print("-"\*50)  
crow.sing()

crow.fly->Whirring  
crow.hungry->True  
Aaaah...  
crow.hungry->False  
--------------------------------------------------  
  
  
  
---------------------------------------------------------------------------  
  
AttributeError Traceback (most recent call last)  
  
Input In [20], in <cell line: 7>()  
 5 print("crow.hungry->{}".format(crow.hungry))  
 6 print("-"\*50)  
----> 7 crow.sing()  
  
  
AttributeError: 'Bird' object has no attribute 'sing'

* 继承的第2个案例——连续继承与\_\_repr\_\_()重写（overload）

该案例引自Mark Lutz. (2013). Learning python(5th Edition).O'Reilly Media, Inc, USA中关于类阐述的章节。其中AttriDisplay是TopTest的父类，TopTest是SubTest的父类。因为\_\_dict\_\_可以获取类的所有属性和方法的键值对，而getattr()内置函数用于一个对象的属性值，因此AttrDisplay方法gatherAttrs()提取了self（实例化对象）具有的属性，并存储在attrs列表中。

repr()通过\_\_repr\_\_()这个特殊方法来得到一个对象的字符串表示形式，方便查看。如果对象没有实现\_\_repr\_\_()，返回的字符串则如下述的<\_\_main\_\_.Example object at 0x00000152FBE9E190>。在AttriDisplay类定义中，则定义了\_\_repr\_\_()方法，该方法返回值必须是字符串对象。因此，当打印实例化对象时，则会执行\_\_repr\_\_()方法，返回对应格式化后的字符串。

string='example'  
print(repr(string))  
  
class Example:pass  
print(repr(Example()))

<\_\_main\_\_.Example object at 0x00000152FBE9E190>  
'example'

class AttrDisplay:  
 """  
 Provides an inheritable display overload method that shows  
 instances with their class names and a name=value pair for  
 each attribute stored on the instance itself (but not attrs  
 inherited from its classes). Can be mixed into any class,  
 and will work on any instance.  
 """  
 def gatherAttrs(self):  
 attrs = []  
 for key in sorted(self.\_\_dict\_\_):  
 attrs.append('%s=%s' % (key, getattr(self, key)))  
 return ', '.join(attrs)  
 def \_\_repr\_\_(self):  
 return '[%s: %s]' % (self.\_\_class\_\_.\_\_name\_\_, self.gatherAttrs())  
   
class TopTest(AttrDisplay):  
 count = 0  
 def \_\_init\_\_(self):  
 self.attr1 = TopTest.count  
 self.attr2 = TopTest.count+1  
 TopTest.count += 2  
class SubTest(TopTest):  
 pass   
  
print(AttrDisplay()) #父类AttriDisplay没有self引导的属性  
X, Y = TopTest(), SubTest()   
print(X)   
print(Y)   
  
  
print("-"\*50)  
print(getattr(X,'count'))

[AttrDisplay: ]  
[TopTest: attr1=0, attr2=1]  
[SubTest: attr1=2, attr2=3]  
--------------------------------------------------  
4

* 继承的第3个案例——[shelve](https://docs.python.org/3/library/shelve.html)②数据录入、更新、删除、与读取

shelve，是python数据储存的一种方式，类似key-value数据库存储形式，便于保存python对象。shelve只有一个open（）函数，用来打开指定的文件（字典），返回一个类似字典的对象。为了方便shelve存储方式的操作，编写下述DB\_Shelve类，实现读、写、更新和删除4类基本操作。同时继承了AttrDisplay类，方便查看属性对象，当调试完后，可以移除该父类。

写入(write())与更新(update)数据的方法使用了关键字(\*\*kwargs)的形式。同时为了方便查看操作结果，增加了feedback()方法，可以在读写更新等方法中调用，反馈操作结果等信息。

import shelve  
  
class DB\_Shelve(AttrDisplay):   
 '''  
 为方便shelve库方式的数据存储，编写数据写入、读取、更新和删除等操作，方便调用。  
 '''  
   
 def \_\_init\_\_(self,db\_fn,flag='c'):  
 '''  
 初始化读写shelve数据库（存储文件）的基本参数  
  
 Parameters  
 ----------  
 db\_fn : string  
 存储文件路径名.  
 flag : string, optional  
 读写方式。包括r-只读；  
 w-可读写;  
 n-每次调用open()都重新创建一个空的文件，可读写.   
 The default is 'c'-如果数据文件不存在，就创建，允许读写.  
  
 Returns  
 -------  
 None.  
  
 '''  
 self.db\_fn=db\_fn  
 self.flag=flag  
   
 def feedback(self,process=None,msg='OK.',data=None):  
 '''  
 用于反馈读写信息  
  
 Parameters  
 ----------  
 process : string, optional  
 标识操作，例如write, read,update或trash等. The default is None.  
 msg : string, optional  
 返回信息，表述运行是否成功等. The default is 'OK.'.  
 data : any python built-in types, optional  
 打印数据. The default is None.  
  
 Returns  
 -------  
 dict  
 反馈操作结果信息.  
  
 '''  
 if data:  
 return {"process":process,'msg':msg,'data':data}  
 else:  
 return {"process":process,'msg':msg}  
   
 def write(self,\*\*kwargs):  
 '''  
 向存储文件中写入数据  
  
 Parameters  
 ----------  
 \*\*kwargs : dict/kwargs  
 以字典形式（加\*\*dict）或关键字参数方式输入数据.  
  
 Returns  
 -------  
 dict  
 反馈操作结果.  
  
 '''  
 with shelve.open(filename=self.db\_fn,flag=self.flag) as db:  
 for k,v in kwargs.items():  
 db[k]=v  
 return self.feedback('write')  
   
 def read(self,keys\_selection=[]):  
 '''  
 从存储文件中读取数据  
  
 Parameters  
 ----------  
 keys\_selection : list, optional  
 待读取存储文件数据的键. The default is [].  
  
 Returns  
 -------  
 dict  
 反馈操作结果.  
  
 '''  
 with shelve.open(filename=self.db\_fn,flag=self.flag) as db:  
 keys=list(db.keys())  
 data={}  
 if keys\_selection:  
 for k in keys\_selection:  
 data[k]=db[k]  
 else:  
 for k in keys:  
 data[k]=db[k]   
 return self.feedback('read',data=data)  
   
 def trash(self,trash\_keys):  
 '''  
 根据指定的键，删除存储文件对应键值数据  
  
 Parameters  
 ----------  
 trash\_keys : list  
 待删除的存储文件键列表.  
  
 Returns  
 -------  
 dict  
 反馈操作结果.  
  
 '''  
 with shelve.open(filename=self.db\_fn,flag=self.flag) as db:  
 keys=set(db.keys())  
 if set(trash\_keys).issubset(keys):  
 for t\_k in trash\_keys:  
 del db[t\_k]  
 return self.feedback('trash')  
 else:  
 return self.feedback('trash','failed, %s not exist.'%trash\_keys)  
   
 def update(self,clear\_all=False,\*\*kwargs):  
 '''  
 给定键，更新存储文件对应键的值  
  
 Parameters  
 ----------  
 clear\_all : bool, optional  
 是否清空存储文件所有数据. The default is False.  
 \*\*kwargs : dict/kwargs  
 以字典形式（加\*\*dict）或关键字参数方式输入更新数据.  
  
 Returns  
 -------  
 dict  
 反馈操作结果.  
  
 '''  
 with shelve.open(filename=self.db\_fn,flag=self.flag) as db:  
 if clear\_all:  
 keys=list(db.keys())  
 for k in keys:  
 del db[k]  
 if len(kwargs)>0:  
 for k,v in kwargs.items():  
 db[k]=v  
 return self.feedback('update')  
   
db\_fn='./database/shelve\_db'   
db=DB\_Shelve(db\_fn)   
print(db)  
  
print("-"\*50)  
fb\_w=db.write(\*\*{'commodity\_name':'muskmelon','sales':1100})  
print(fb\_w)  
fb\_r\_all=db.read()  
print(fb\_r\_all)  
fb\_r\_selection=db.read(['sales'])  
print(fb\_r\_selection)  
db.write(\*\*{'exporting\_country\_name':'peru'})  
  
print("-"\*50)  
print(db.read())  
fb\_t=db.trash(['sales'])  
print(fb\_t)  
print(db.read())  
  
print("-"\*50)  
db.update(sales=1100)  
db.update(commodity\_code=101,idx=1101)

[DB\_Shelve: db\_fn=./database/shelve\_db, flag=c]  
--------------------------------------------------  
{'process': 'write', 'msg': 'OK.'}  
{'process': 'read', 'msg': 'OK.', 'data': {'commodity\_name': 'muskmelon', 'exporting\_country\_name': 'peru', 'commodity\_code': 101, 'idx': 1101, 'commodity\_dic': {'commodity\_code': 103, 'commodity\_name': 'muskmelon'}, 'sales': 1100}}  
{'process': 'read', 'msg': 'OK.', 'data': {'sales': 1100}}  
--------------------------------------------------  
{'process': 'read', 'msg': 'OK.', 'data': {'commodity\_name': 'muskmelon', 'exporting\_country\_name': 'peru', 'commodity\_code': 101, 'idx': 1101, 'commodity\_dic': {'commodity\_code': 103, 'commodity\_name': 'muskmelon'}, 'sales': 1100}}  
{'process': 'trash', 'msg': 'OK.'}  
{'process': 'read', 'msg': 'OK.', 'data': {'commodity\_name': 'muskmelon', 'exporting\_country\_name': 'peru', 'commodity\_code': 101, 'idx': 1101, 'commodity\_dic': {'commodity\_code': 103, 'commodity\_name': 'muskmelon'}}}  
--------------------------------------------------  
  
  
  
  
  
{'process': 'update', 'msg': 'OK.'}

DB\_Shelve类是对shelve数据存储的基本操作，如果要拓展列表追加或字典更新等值的存储类型，可以以子类的方式增加功能。同时，也可以对原有父类的某些方法重新。这里仅增加了update\_lstExtend(()和update\_dict两种方法，未重写父类方法。注意，要列表追加或更新字典，需要配置shelve.open(file\_name, flag='', writeback=True\False)中的writeback为True。

class DB\_Shelve\_Extension(DB\_Shelve):  
 '''  
 拓展DB\_Shelve操作shelve数据存储的方式，包括值类型的，列表形式追加和字典更新  
 '''  
 def \_\_init\_\_(self,db\_fn,flag='c',writeback=True):  
 '''  
 初始化读写shelve数据库（存储文件）的基本参数，较之DB\_Shelve类，增加了writeback参数  
  
 Parameters  
 ----------  
 db\_fn : string  
 存储文件路径名.  
 flag : string, optional  
 读写方式。包括r-只读；  
 w-可读写;  
 n-每次调用open()都重新创建一个空的文件，可读写.   
 The default is 'c'-如果数据文件不存在，就创建，允许读写.  
 writeback : bool, optional  
 当设置为True以后，shelve对象为所有访问过的条目保留缓存并在close()或sync()时将它们写回到DB. The default is True.  
  
 Returns  
 -------  
 None.  
  
 '''  
 super(DB\_Shelve\_Extension,self).\_\_init\_\_(db\_fn,flag)  
 self.writeback=writeback  
   
 def update\_lstExtend(self,\*\*kwargs):  
 '''  
 对值为列表形式的数据，追加新的数据  
  
 Parameters  
 ----------  
 \*\*kwargs : dict/kwargs  
 以字典形式（加\*\*dict）或关键字参数方式输入更新数据，值为列表.  
  
 Returns  
 -------  
 dict  
 反馈操作结果.  
  
 '''  
 with shelve.open(filename=self.db\_fn,flag=self.flag,writeback=self.writeback) as db:  
 if len(kwargs)>0:  
 for k,v in kwargs.items():  
 #print('-'\*30,k,v)  
 #print(db[k])  
 db[k].extend(v)   
 #print(db[k])  
 return self.feedback('update\_lstExtend')  
   
 def update\_dict(self,\*\*kwargs):  
 '''  
 对值为字典形式的数据，根据新的字典数据更新值  
  
 Parameters  
 ----------  
 \*\*kwargs : dict/kwargs  
 以字典形式（加\*\*dict）或关键字参数方式输入更新数据，值为字典.  
  
 Returns  
 -------  
 dict  
 反馈操作结果.  
  
 '''  
 with shelve.open(filename=self.db\_fn,flag=self.flag,writeback=self.writeback) as db:  
 if len(kwargs)>0:  
 for k,v in kwargs.items():  
 #print(db[k])  
 db[k].update(v)  
 #print(db[k])  
 return self.feedback('update\_dict')   
   
db\_fn='./database/shelve\_db'   
db\_ex=DB\_Shelve\_Extension(db\_fn)   
print(db\_ex)  
  
print("-"\*50)  
print(db\_ex.read())  
  
print("-"\*50)  
db\_ex.update(commodity\_name=['muskmelon','strawberry','apple'])  
print(db\_ex.read())  
db\_ex.update\_lstExtend(commodity\_name=['lemon'])  
print(db\_ex.read(keys\_selection=['commodity\_name']))  
  
print("-"\*50)  
db\_ex.write(commodity\_dic={'commodity\_code':101,'commodity\_name':'muskmelon'})  
print(db\_ex.read())  
  
print("-"\*50)  
db\_ex.update\_dict(commodity\_dic={'commodity\_code':103})  
print(db\_ex.read())

[DB\_Shelve\_Extension: db\_fn=./database/shelve\_db, flag=c, writeback=True]  
--------------------------------------------------  
{'process': 'read', 'msg': 'OK.', 'data': {'commodity\_name': ['muskmelon', 'strawberry', 'apple', 'lemon'], 'exporting\_country\_name': 'peru', 'commodity\_code': 101, 'idx': 1101, 'commodity\_dic': {'commodity\_code': 103, 'commodity\_name': 'muskmelon'}, 'sales': 1100}}  
--------------------------------------------------  
{'process': 'read', 'msg': 'OK.', 'data': {'commodity\_name': ['muskmelon', 'strawberry', 'apple'], 'exporting\_country\_name': 'peru', 'commodity\_code': 101, 'idx': 1101, 'commodity\_dic': {'commodity\_code': 103, 'commodity\_name': 'muskmelon'}, 'sales': 1100}}  
{'process': 'read', 'msg': 'OK.', 'data': {'commodity\_name': ['muskmelon', 'strawberry', 'apple', 'lemon']}}  
--------------------------------------------------  
{'process': 'read', 'msg': 'OK.', 'data': {'commodity\_name': ['muskmelon', 'strawberry', 'apple', 'lemon'], 'exporting\_country\_name': 'peru', 'commodity\_code': 101, 'idx': 1101, 'commodity\_dic': {'commodity\_code': 101, 'commodity\_name': 'muskmelon'}, 'sales': 1100}}  
--------------------------------------------------  
{'process': 'read', 'msg': 'OK.', 'data': {'commodity\_name': ['muskmelon', 'strawberry', 'apple', 'lemon'], 'exporting\_country\_name': 'peru', 'commodity\_code': 101, 'idx': 1101, 'commodity\_dic': {'commodity\_code': 103, 'commodity\_name': 'muskmelon'}, 'sales': 1100}}

**8.3** private variables(私有变量/成员)和private methods(私有方法)

python实际上没有类似public，private等关键字修饰的成员变量和方法，通常在变量名或方法名前增加\_\_两个下划线，将公有变量和方法转换为私有变量和方法，实际上私有变量和方法是可以在外部访问的，只需要将例如\_\_data的私有变量，\_\_say()或\_\_subPrintData()的私有方法，替换为\_PA\_\_data，\_PA\_\_say()和\_PB\_\_subPrintData()（类名之前一个下划线，私有变量和方法前两个下划线），因此python中并没有真正的私有变量和方法。私有变量和方法仅在类内访问，其子类和父类无法直接访问。

class PA:  
 def \_\_init\_\_(self):  
 self.\_\_data=[] #私有变量，实际被替换为：self.\_PA\_\_data=[]  
 self.\_\_name='Guido van Rossum' #私有变量，实际被替换为：self.\_PA\_\_name='Guido van Rossum'  
 def add(self,item):  
 self.\_\_data.append(item) #实际被替换为：self.\_PA\_\_data.append(item)  
 def printData(self):  
 print(self.\_\_data) #实际被替换为：self.\_PA\_\_data  
 def \_\_say(self): #私有方法，实际被替换为：def \_PA\_\_say(self)  
 print("%s created python."%self.\_\_name) #实际被替换为：self.\_PA\_\_name  
   
class PB(PA):  
 def printSuperclassAttri(self):  
 print(self.\_PA\_\_data) #self.\_\_data无法访问，属于父类的私有变量，需按替换语句操作   
 def \_\_subPrintData(self):  
 print('This is a private method of the subclass.')  
   
instance=PB()  
for i in ['c','c#','C++','python']:instance.add(i)  
instance.printData()  
print(instance.\_PA\_\_data) #访问私有变量，需按替换语句操作  
print(instance.\_PA\_\_say()) #调用私有方法，需按替换语句操作  
  
print("-"\*50)  
instance.printSuperclassAttri()  
instance.\_PB\_\_subPrintData()

['c', 'c#', 'C++', 'python']  
['c', 'c#', 'C++', 'python']  
Guido van Rossum created python.  
None  
--------------------------------------------------  
['c', 'c#', 'C++', 'python']  
This is a private method of the subclass.

**8.4** Operator Overloading（运算符/操作符重载） 与类属性格式”字典”

* Operator Overloading（操作符重载）

操作符重载理解为，如果在类的内部重写了python内置操作（build-in operations），类的实例化时会自动调用重写的方法，由重写的方法计算的结果作为返回值。对于操作符重载： 即重写的方法拦截了python正常的内置操作方法；类可以重载所有的python表达式操作；类也可以重载python内置操作，例如打印（printing），函数调用（function calls）和属性访问（attribute access）等； 重载使类的实例表现得更像内置类型；重载是通过在一个类中提供特别命名的方法来实现的。

例如下述案例在类里重写了python的运算符加号+，即\_\_add\_\_方法。为了区别正常的加法运算，这里在两个数相加后，再加10。下述类重写加法运算符直接调用了重写的方法运算方式，同时调用类进行计算时类似于常规的内置类型运算方式，直接使用+进行计算，而不必按照类调用方法的途径实现。

class OO\_Add:  
 def \_\_init\_\_(self,data):  
 self.data=data  
 def \_\_add\_\_(self,y):  
 return OO\_Add(self.data+y+10)  
  
x=OO\_Add(3)  
y=x+9  
print(y.data)  
  
print("-"\*50)  
class OO\_Add\_:  
 def \_\_init\_\_(self,data):  
 self.data=data  
 def \_\_add\_\_(self,y):  
 return self.data+y+10  
print(OO\_Add\_(3)+9)

22  
--------------------------------------------------  
22

常见可用于操作符重载方法的内置运算符：

| Method | Implements | Called for |
| --- | --- | --- |
| \_\_init\_\_ | Constructor 构造函数 | Object creation: X=Class(args) |
| \_\_del\_\_ | Destructor 析构函数 | Object deletion of X |
| \_\_add\_\_ | Operator + 加法 | X+Y, X+=Y if no \_\_iadd\_\_ |
| \_\_or\_\_ | Operator \|(bitwise OR) 运算符\| | X|Y, X|=Y if no \_\_ior\_\_ |
| \_\_repr\_\_, \_\_str\_\_ | Printing, conversions 打印／转换 | print(X), repr(X),str(X) |
| \_\_call\_\_ | Function calls 函数调用 | X(\*args, \*\*kwargs) |
| \_\_getattr\_\_ | Attribute fetch 属性引用 | X.undefined |
| \_\_setattr\_\_ | Attribute assignment 属性赋值 | X.attribute=value |
| \_\_delattr\_\_ | Attribute deletion 属性删除 | del X.attribute |
| \_\_getattribute\_\_ | Attribute fetch 属性获取 | X.any |
| \_\_getitem\_\_ | Indexing, slicing, iteration 索引运算 | X[Key], X[i:j], for loops and other iterations if no \_\_iter\_\_ |
| \_\_setitem\_\_ | Index and slice assignment 索引赋值 | X[Key]=value, X[i:j]=iterable |
| \_\_delitem\_\_ | Index and slice deletion 索引和分片删除 | del X[Key],del X[i:j] |
| \_\_len\_\_ | Length 长度 | len(X), truth tests if no \_\_bool\_\_ |
| \_\_bool\_\_ | Boolean tests 布尔测试 | bool(X), truth tests(named \_\_nonzero\_\_ in 2.X) |
| \_\_lt\_\_, \_\_gt\_\_ , \_\_le\_\_,\_\_ge\_\_,\_\_eq\_\_,\_\_ne\_\_ | Comparisons 特定的比较 | X<Y, X>Y, X<=Y, X>=Y, X==Y, X!=Y |
| \_\_radd\_\_ | Right-side operators 右侧加法 | Other+X |
| \_\_iadd\_\_ | In-place augmented operators 实地（增强的）加法 | X+=Y (or else \_\_add\_\_) |
| \_\_iter\_\_, \_\_next\_\_ | Iteration contexts 迭代 | I=iter(X), next(I); for loops, in if no \_\_contains\_\_, all comprehensions, map(F,X), others(\_\_next\_\_ is named next in 2.X) |
| \_\_contains\_\_ | Membership test 成员关系测试 | item in X(any iterable) |
| \_\_index\_\_ | Integer value 整数值 | hex(X), bin(X), oct(X), O[X],O[X:] |
| \_\_enter\_\_, \_\_exit\_\_ | Context manager 环境管理器 | with obj as var: |
| \_\_get\_\_,\_\_set\_\_,\_\_delete\_\_ | Descriptor attributes 描述符属性 | X.attr, X.attr=value, del X.attr |
| \_new\_\_ | Creation 创建 | Object creation, before \_\_init\_\_ |

* 类属性格式”字典”

如果以dict.key获取类属性的形式替代字典dict[key]的形式，在代码书写上相对简单。要达到这样的目的，可以建立一个空的类，通过赋予属性的方式实现，例如下述案例中的class Sales:pass；Sales类方法在赋值时需要逐一赋值，也不能够将已有的字典转换为属性字典的形式，因此定义类class AttrDict\_simple(dict)，该种方法继承了父类dict内置字典类，并以父类的方式super(AttrDict\_simple,self).\_\_init\_\_(\*args,\*\*kwargs)初始化参数值，并通过self.\_\_dict\_\_=self将实例化对象本身赋予给\_\_dict\_\_对象（可以获取类的所有属性和方法的键值对），因此可以通过属性方式访问字典成员；AttrDict\_simple类方式虽然可以直接将字典转换为属性字典的格式，但是缺少编辑功能，例如增减成员，因此定义class AttriDict(dict)类，重写（overload）了部分父类dict的方法，实现增加键值对（属性值）的方法。

类似 \_\_xx\_\_ 或 \_\_xx\_\_() 前后都有2个下划线的变量或方法，通常是python中内置的特殊变量属性或方法的标识，在书写代码时应尽量避免用该类方式自定义变量或方法。

help(dict)

Help on class dict in module builtins:  
  
class dict(object)  
 | dict() -> new empty dictionary  
 | dict(mapping) -> new dictionary initialized from a mapping object's  
 | (key, value) pairs  
 | dict(iterable) -> new dictionary initialized as if via:  
 | d = {}  
 | for k, v in iterable:  
 | d[k] = v  
 | dict(\*\*kwargs) -> new dictionary initialized with the name=value pairs  
 | in the keyword argument list. For example: dict(one=1, two=2)  
 |   
 | Built-in subclasses:  
 | StgDict  
 |   
 | Methods defined here:  
 |   
 | \_\_contains\_\_(self, key, /)  
 | True if the dictionary has the specified key, else False.  
 |   
 | \_\_delitem\_\_(self, key, /)  
 | Delete self[key].  
 |   
 | \_\_eq\_\_(self, value, /)  
 | Return self==value.  
 |   
 | \_\_ge\_\_(self, value, /)  
 | Return self>=value.  
 |   
 | \_\_getattribute\_\_(self, name, /)  
 | Return getattr(self, name).  
 |   
 | \_\_getitem\_\_(...)  
 | x.\_\_getitem\_\_(y) <==> x[y]  
 |   
 | \_\_gt\_\_(self, value, /)  
 | Return self>value.  
 |   
 | \_\_init\_\_(self, /, \*args, \*\*kwargs)  
 | Initialize self. See help(type(self)) for accurate signature.  
 |   
 | \_\_iter\_\_(self, /)  
 | Implement iter(self).  
 |   
 | \_\_le\_\_(self, value, /)  
 | Return self<=value.  
 |   
 | \_\_len\_\_(self, /)  
 | Return len(self).  
 |   
 | \_\_lt\_\_(self, value, /)  
 | Return self<value.  
 |   
 | \_\_ne\_\_(self, value, /)  
 | Return self!=value.  
 |   
 | \_\_repr\_\_(self, /)  
 | Return repr(self).  
 |   
 | \_\_reversed\_\_(self, /)  
 | Return a reverse iterator over the dict keys.  
 |   
 | \_\_setitem\_\_(self, key, value, /)  
 | Set self[key] to value.  
 |   
 | \_\_sizeof\_\_(...)  
 | D.\_\_sizeof\_\_() -> size of D in memory, in bytes  
 |   
 | clear(...)  
 | D.clear() -> None. Remove all items from D.  
 |   
 | copy(...)  
 | D.copy() -> a shallow copy of D  
 |   
 | get(self, key, default=None, /)  
 | Return the value for key if key is in the dictionary, else default.  
 |   
 | items(...)  
 | D.items() -> a set-like object providing a view on D's items  
 |   
 | keys(...)  
 | D.keys() -> a set-like object providing a view on D's keys  
 |   
 | pop(...)  
 | D.pop(k[,d]) -> v, remove specified key and return the corresponding value.  
 | If key is not found, d is returned if given, otherwise KeyError is raised  
 |   
 | popitem(self, /)  
 | Remove and return a (key, value) pair as a 2-tuple.  
 |   
 | Pairs are returned in LIFO (last-in, first-out) order.  
 | Raises KeyError if the dict is empty.  
 |   
 | setdefault(self, key, default=None, /)  
 | Insert key with a value of default if key is not in the dictionary.  
 |   
 | Return the value for key if key is in the dictionary, else default.  
 |   
 | update(...)  
 | D.update([E, ]\*\*F) -> None. Update D from dict/iterable E and F.  
 | If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]  
 | If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v  
 | In either case, this is followed by: for k in F: D[k] = F[k]  
 |   
 | values(...)  
 | D.values() -> an object providing a view on D's values  
 |   
 | ----------------------------------------------------------------------  
 | Class methods defined here:  
 |   
 | fromkeys(iterable, value=None, /) from builtins.type  
 | Create a new dictionary with keys from iterable and values set to value.  
 |   
 | ----------------------------------------------------------------------  
 | Static methods defined here:  
 |   
 | \_\_new\_\_(\*args, \*\*kwargs) from builtins.type  
 | Create and return a new object. See help(type) for accurate signature.  
 |   
 | ----------------------------------------------------------------------  
 | Data and other attributes defined here:  
 |   
 | \_\_hash\_\_ = None

from datetime import datetime  
date=datetime.strptime('2020-3-30','%Y-%m-%d').date()  
  
sales\_dict={'idx':1101,'date':date,'exporting\_country':'peru','sales':2500,'commodity\_name':'lemon'}  
print(sales\_dict['commodity\_name'])  
  
print("-"\*50)  
class Sales:pass  
sales\_rec=Sales()  
sales\_rec.idx=1101  
sales\_rec.date=date  
sales\_rec.exporting\_country='peru'  
sales\_rec.sales=2500  
sales\_rec.commodity\_name='lemon'  
print(sales\_rec.commodity\_name)  
  
print("-"\*50)  
class AttrDict\_simple(dict):  
 def \_\_init\_\_(self,\*args,\*\*kwargs):  
 super(AttrDict\_simple,self).\_\_init\_\_(\*args,\*\*kwargs)  
 self.\_\_dict\_\_=self  
   
sales\_attrDict\_simple=AttrDict\_simple(sales\_dict)  
print(sales\_attrDict\_simple.commodity\_name)

lemon  
--------------------------------------------------  
lemon  
--------------------------------------------------  
lemon

class AttriDict(dict):  
 def \_\_init\_\_(self, \*args, \*\*kwargs):  
 super(AttriDict, self).\_\_init\_\_(\*args, \*\*kwargs)  
 for arg in args:  
 if isinstance(arg, dict):  
 for k, v in arg.items():  
 self[k]=v  
 if kwargs:  
 for k, v in kwargs.items():  
 self[k]=v   
   
 def \_\_getattr\_\_(self, attr):  
 return self.get(attr)   
   
 def \_\_setattr\_\_(self, key, value):  
 self.\_\_setitem\_\_(key, value)  
   
 def \_\_setitem\_\_(self, key, value):  
 super(AttriDict, self).\_\_setitem\_\_(key, value)  
 self.\_\_dict\_\_.update({key: value})  
   
 def \_\_delattr\_\_(self, item):  
 self.\_\_delitem\_\_(item)   
  
 def \_\_delitem\_\_(self, key):  
 super(AttriDict, self).\_\_delitem\_\_(key)  
 del self.\_\_dict\_\_[key]   
  
sales\_attrDict=AttriDict(sales\_dict)   
print(sales\_attrDict)  
print(sales\_attrDict.commodity\_name)  
sales\_attrDict.commodity\_code=102  
print(sales\_attrDict)  
sales\_attrDict['exporting\_country\_ID']=25  
print(sales\_attrDict)  
del sales\_attrDict.idx #或用：del sales\_attrDict['idx']  
print(sales\_attrDict)

{'idx': 1101, 'date': datetime.date(2020, 3, 30), 'exporting\_country': 'peru', 'sales': 2500, 'commodity\_name': 'lemon'}  
lemon  
{'idx': 1101, 'date': datetime.date(2020, 3, 30), 'exporting\_country': 'peru', 'sales': 2500, 'commodity\_name': 'lemon', 'commodity\_code': 102}  
{'idx': 1101, 'date': datetime.date(2020, 3, 30), 'exporting\_country': 'peru', 'sales': 2500, 'commodity\_name': 'lemon', 'commodity\_code': 102, 'exporting\_country\_ID': 25}  
{'date': datetime.date(2020, 3, 30), 'exporting\_country': 'peru', 'sales': 2500, 'commodity\_name': 'lemon', 'commodity\_code': 102, 'exporting\_country\_ID': 25}

注释（Notes）：

① 关于Class类Python官网的几个小案例，（<https://docs.python.org/3/tutorial/classes.html>）。

② shelve — Python object persistence，（<https://docs.python.org/3/library/shelve.html>）。

PC8-ipynb download