Python3 面向对象

类(Class)

- 类(Class): 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- 类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- 实例变量：在类的声明中，属性是用变量来表示的，这种变量就称为实例变量，实例变量就是一个用 self 修饰的变量。

类名的首字母一般要大写

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

- **类的方法**:在类的内部，使用 `def` 关键字来定义一个方法，与一般函数定义不同，类方法必须包含参数 self, 且为第一个参数，self 代表的是类的实例。
  实例：

```
    #类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
        self.__weight = w
    def speak(self):
        print("%s 说: 我 %d 岁。" %(self.name,self.age))

# 实例化类
p = people('agul',10,30)
p.speak()
```

运行结果：

```
agul 说: 我 10 岁。
```

- 在Python中，可以使用内置方法 `isinstance()` 来测试一个对象是否为某个类的实例，下面的代码演示了 `isinstance()` 的用法。
- 类的所有实例方法都必须至少有一个名为“self”的参数，并且必须是方法的第一个形参（如果有多个形参的话），“self”参数代表将来要创建的对象本身。在类的实例方法中访问实例属性时需要以“self”为前缀，但在外部通过对象名调用对象方法时并不需要传递这个参数，如果在外部通过类名调用对象方法则需要显式为self参数传值。

构造函数 `__init__`

- `__init__` 允许我们在执行实例化过程时传入一些参数
- 新建的实例本身，连带其中的参数，会一并传给 `__init__` 函数自动并执行它。所以 `__init__` 函数的参数列表会在开头多出一项，它永远指代新建的那个实例对象，Python语法要求这个参数必须要有，而名称随意，习惯上就命为 `self` 。
- **独立的命名空间**，也就是说函数内新引入的变量均为局部变量，新建的实例对象对这个函数来说也只是通过第一参数 `self` 从外部传入的，故无论设置还是使用它的属性都得利用 `self.<属性名>` 。
- `__init__` 还是有个特殊之处，那就是它不允许有返回值。如果你的 `__init__` 过于复杂有可能要提前结束的话，使用单独的return就好，不要带返回值。

析构函数 `__del__`

- Python中类的析构函数是**del**，用来释放对象占用的资源，在Python收回对象空间之前自动执行。如果用户未涉及析构函数，Python将提供一个默认的析构函数进行必要的清理工作。

实例属性和类属性

- 实例属性属于实例(对象)只能通过对象名访问；类属性属于类可通过类名访问，也可以通过对象名访问，为类的所有实例共享。

实例：

```
#定义含有实例属性（姓名name， 年龄age）和类属性（人数num）的Person人员类
class Person:
    num=1           #类属性
    def __init__(self, str,n):  #构造函数
        self.name = str #实例属性
        self.age=n
    def SayHello(self):         #成员函数
        print("Hello!")
    def PrintName(self):        #成员函数
        print("姓名：", self.name,  "年龄：", self.age)
    def PrintNum(self):         #成员函数
        print(Person.num)   #由于是类属性，所以不写self .num
#主程序
P1= Person("夏敏捷",42)
P2= Person("王琳",36)
P1.PrintName()
P2.PrintName()
```

运行结果：

```
姓名： 夏敏捷 年龄： 42
姓名： 王琳 年龄： 36
```

- 类是模板，而实例则是根据类创建的对象。
- 绑定在一个实例上的属性不会影响其他实例，但是，类本身也是一个对象，如果在类上绑定一个属性，则所有实例都可以访问类的属性，并且，所有实例访问的类属性都是同一个！也就是说，实例属性每个实例各自拥有，互相独立，而类属性有且只有一份。
- 定义类属性可以直接在 **class** 中定义

私有成员与公有成员

- Python并没有对私有成员提供严格的访问保护机制。在定义类的属性时，如果属性名以两个下划线“__”开头则表示是私有属性，否则是公有属性。**私有属性在类的外部不能直接访问，需通过调用对象的公有成员方法来访问，或者通过Python支持的特殊方式来访问。**Python提供了访问私有属性的特殊方式，可用于程序的测试和调试，对于成员方法也有同样性质

实例：

```
class Car:
    price = 100000                #定义类属性
    def __init__(self, c, w):
        self.color = c                #定义公有属性color
        self. __weight= w             #定义私有属性__weight
#主程序
car1 = Car("Red",10.5)
car2 = Car("Blue",11.8)
print(car1.color)
print(car1. _Car__weight)
print(car1. __weight)             # AttributeError
```

运行结果：

```
Red
10.5
Traceback (most recent call last):
  File "GitHub\Python\oob\私有成员与公有成员.py", line 11, in <module>
    print(car1. __weight)                          # AttributeError
AttributeError: 'Car' object has no attribute '__weight'
```

方法

- 在类中定义的方法可以粗略分为3大类：公有方法、私有方法、静态方法。
- **公有方法、私有方法**都属于对象，私有方法的名字以两个下划线“__”开始，每个对象都有自己的公有方法和私有方法，在这两类方法中可以访问属于类和对象的成员；
- **公有方法**通过对象名直接调用，私有方法不能通过对象名直接调用，只能在属于对象的方法中通过“self”调用或在外部通过Python支持的特殊方式来调用。
- **静态方法可以通过类名和对象名调用**，但不能直接访问属于对象的成员，只能访问属于类的成员

实例：

```
class Person:
    num=0           #类属性
    def __init__(self, str,n,w):    #构造函数
```

```
        self.name = str      #对象实例属性（成员）
        self.age=n
        self.__weight= w         #定义私有属性__weight
        Person.num += 1
    def __outputWeight(self):    #定义私有方法outputWeight
        print("体重：",self.__weight)  #访问私有属性__weight
    def PrintName(self):         #定义公有方法（成员函数）
        print("姓名：", self.name,  "年龄：", self.age, end=" ")
        self.__outputWeight( )  #调用私有方法outputWeight
    def PrintNum(self):          #定义公有方法（成员函数）
        print(Person.num)        #由于是类属性，所以不写self.num
    @ staticmethod
    def getNum():        #定义静态方法getNum
         return Person.num

#主程序
P1= Person("夏敏捷",42,120)
P2= Person("张海",39,80)
#P1.outputWeight()           #错误'Person' object has no attribute 'outputWeight'
P1.PrintName()
P2.PrintName()
Person.PrintName(P2)
print("人数：",Person.getNum())
print("人数：",P1.getNum())
```

运行结果：

```
姓名： 夏敏捷 年龄： 42 体重： 120
姓名： 张海 年龄： 39 体重： 80
姓名： 张海 年龄： 39 体重： 80
人数： 2
人数： 2
```

定义类方法

- 和属性类似，方法也分实例方法和类方法。
- 在**class**中定义的全部是实例方法，实例方法第一个参数 **self** 是实例本身。
- 要在class中定义类方法，需要这么写：

```
class Person(object):
    count = 0
    @classmethod
    def how_many(cls):
        return cls.count
    def __init__(self,name):
        self.name = name
        Person.count = Person.count + 1

print(Person.how_many())
p1 = Person('Bob')
print(Person.how_many())
```

运行结果：

```
0
1
```

类的继承

- **继承**用于指定一个类将从其父类获取其大部分或全部功能。 它是面向对象编程的一个特征。 这是一个非常强大的功能，方便用户对现有类进行几个或多个修改来创建一个新的类。新类称为子类或派生类，从其继承属性的主类称为基类或父类。
- **子类或派生类**继承父类的功能，向其添加新功能。 它有助于代码的可重用性。
- **基类名写在括号里**
- **在Python中继承的一些特点：**
    1. 在继承中基类的构造函数（**init**()方法）不会被自动调用，它需要在其派生类的构造中亲自专门调用。
    2. 如果需要在派生类中调用基类的方法时，通过“基类名.方法名(self)”的方式来实现，需要加上基类的类名前缀。区别于在类中调用普通函数时并不需要带上self参数。也可以使用内置函数super()实现这一目的。
    3. Python总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。（先在本类中查找调用的方法，找不到才去基类中找）。

实例：

```
class Person(object):
   def __init__(self, name, gender):
      self.name = name
      self.gender = gender

class Student(Person):
   def __init__(self, name, gender,score):
       super(Student, self).__init__(name, gender)
       self.score = score

s1=Student('张三','Female',89)
print(s1.name)
```

- 一定要用 `super(Student, self).__init__(name, gender)` 去初始化父类，否则，继承自 `Person` 的 `Student` 将没有 `name` 和 `gender`。
- 函数 `super(Student, self)` 将返回当前类继承的父类，即 `Person` ，然后调用 `__init__()` 方法，注意 `self` 参数已在 `super()` 中传入，在 `__init__()` 中将隐式传递，不需要写出（也不能写）。

运行结果：

```
张三
```

多重继承

- 除了从一个父类继承外，Python允许从多个父类继承，称为多重继承。

实例：

```
class A(object):
    def __init__(self, a):
        print('init A...')
        self.a = a

class B(A):
    def __init__(self, a):
        super(B, self).__init__(a)
        print('init B...')

class C(A):
    def __init__(self, a):
        super(C, self).__init__(a)
        print ('init C...')

class D(B,C):
    def __init__(self, a):
        super(D, self).__init__(a)
        print('init D...')

d = D('d')
```

运行结果：

```
init A...
init C...
init B...
init D...
```

- 像这样，**D** 同时继承自 **B** 和 **C**，也就是 **D** 拥有了 **A**、**B**、**C** 的全部功能。多重继承通过 `super()` 调用 `__init__()` 方法时，**A** 虽然被继承了两次，但 `__init__()` 只调用一次：
- **多重继承的目的**是从两种继承树中分别选择并继承出子类，以便组合功能使用。

方法重写

- 重写必须出现在继承中。它是指当派生类继承了基类的方法之后，如果基类方法的功能不能满足需求，需要对基类中的某些方法进行修改，可以在派生类重写基类的方法，这就是**重写**。

实例：

```
class Animal:               # 定义父类
   def run(self):
      print ('调用父类方法')
class Cat (Animal):             # 定义子类
   def run (self):
      print ('调用子类方法')
class Dog (Animal):         # 定义子类
   def run (self):
      print ('调用子类方法')

c = Dog()                   # 子类实例
c. run ()                   # 子类调用重写方法
```

运行结果：

```
调用子类方法
```

多态

- 类具有继承关系，并且子类类型可以向上转型看做父类类型，如果我们从 **Person** 派生出 **Student** 和**Teacher** ，并都写了一个 `whoAmI()` 方法：

实例：

```
class Person(object):
    def __init__(self, name, gender):
      self.name = name
      self.gender = gender
    def whoAmI(self):
        return 'I am a person, my name is %s' %self.name


class Student(Person):
    def __init__(self, name, gender,score):
       super(Student, self).__init__(name, gender)
       self.score = score
    def whoAmI(self):
        return 'I am a Student, my name is %s' %self.name

class Teacher(Person):
    def __init__(self, name, gender,course):
       super(Teacher, self).__init__(name, gender)
       self.course = course
    def whoAmI(self):
        return 'I am a Teacher, my name is %s' %self.name

def who_am_i(x):
    print(x.whoAmI())

p=Person('Tim','Male')
s=Student('Bob','Female',88)
t=Teacher('Alice', 'Female', 'English')

who_am_i(p)
who_am_i(s)
```

```
who_am_i(t)
```

- 在一个函数中，如果我们接收一个变量 **x**，则无论该 **x** 是 **Person**、**Student**还是 **Teacher**，都可以正确打印出结果：

运行结果：

```
I am a person, my name is Tim
I am a Student, my name is Bob
I am a Teacher, my name is Alice
```

- 这种行为称为多态。也就是说，方法调用将作用在 **x** 的实际类型上。**s** 是**Student**类型，它实际上拥有自己的 `whoAmI()` 方法以及从 Person继承的 `whoAmI` 方法，但调用 `s.whoAmI()` 总是先查找它自身的定义，如果没有定义，则顺着继承链向上查找，直到在某个父类中找到为止。
- 由于Python是动态语言，所以，传递给函数 `who_am_i(x)` 的参数 **x** 不一定是 Person 或 Person 的子类型。任何数据类型的实例都可以，只要它有一个`whoAmI()`的方法即可：
- 这是动态语言和静态语言（例如Java）最大的差别之一。动态语言调用实例方法，不检查类型，只要方法存在，参数正确，就可以调用。
- 多态的好处就是，当我们需要传入Dog、Cat、Tortoise……时，我们只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise……都是Animal类型，然后，按照Animal类型进行操作即可。由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思。

import 导入模块

想使用 Python 源文件，只需在另一个源文件里执行 import 语句，语法如下:

```
import module1[, module2[,... moduleN]
```

- 一个模块只会被导入一次，不管你执行了多少次import。这样可以防止导入模块被一遍又一遍地执行。

Python 的 from 语句让你从模块中导入一个指定的部分到当前命名空间中，语法如下：

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

- 通过这种方式引入的时候，调用函数时只能给出函数名，不能给出模块名

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

```
from modname import *
```

一个模块被另一个程序第一次引入时，其主程序将运行。如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用**name**属性来使该程序块仅在该模块自身运行时执行。

实例：

```
# Filename: using_name.py
if __name__ == '__main__':
   print('程序自身在运行')
else:
   print('我来自另一模块')
```

运行结果：

```
$ python using_name.py
程序自身在运行
$ python
>>> import using_name
我来自另一模块
>>>
```

- 每个模块都有一个**name**属性，当其值是'**main**'时，表明该模块自身在运行，否则是被引入。

文件

`open()` 方法

- `Python open()` 方法用于打开一个文件，并返回文件对象，在对文件进行处理过程都需要使用到这个函数，如果该文件无法被打开，会抛出 `OSError`
- **注意：**使用 `open()` 方法一定要保证关闭文件对象，即调用 `close()` 方法。
- `open()` 函数常用形式是接收两个参数：文件名(file)和模式(mode)。

  ```
  open(file, mode='r')
  ```

  完整的语法格式为

  ```
  open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
  closefd=True, opener=None)
  ```

  参数说明：

参数	作用
file	必需，文件路径（相对或者绝对路径）
mode	可选，文件打开模式
buffering	设置缓冲
encoding	一般使用utf8
newline	区分换行符
closefd	传入的 file 参数类型
opener	设置自定义开启器，开启器的返回值必须是一个打开的文件描述符。

  mode 参数有：

模式	描述
t	文本模式 (默认)
x	写模式，新建一个文件，如果该文件已存在则会报错
b	二进制模式
+	打开一个文件进行更新(可读可写)
U	通用换行模式（**Python 3 不支持**）
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。一般用于非文本文件如图片等
r+	打开一个文件用于读写。文件指针将会放在文件的开头
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。一般用于非文本文件如图片等
w	打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等
w+	打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写

读取文本文件

- `read()` 方法

  不设置参数的read()方法将整个文件的内容读取为一个字符串

  实例:

```
helloFile=open("d:\\hello.txt")
fileContent=helloFile.read()
helloFile.close()
print(fileContent)
```

- `readline()` 方法

  `readline()` 方法从文件中获取一个字符串，每个字符串就是文件中的每一行

```
helloFile=open("d:\\hello.txt")
fileContent=""
while True:
   line=helloFile.readline()
   if line=="":    # 或者 if not line
      break
   fileContent+=line
helloFile.close()
print(fileContent)
```

写文本文件

写文件与读文件相似，都需要先创建文件对象连接。所不同的是，打开文件时是以“写”模式或“添加”模式打开。如果文件不存在，则创建该文件。

与读文件时不能添加或修改数据类似，写文件时也不允许读取数据。“w”写模式打开已有文件时，会覆盖文件原有内容，从头开始，就像我们用一个新值覆写一个变量的值

- `write()` 方法

  write方法将字符串参数写入文件

```
helloFile=open("d:\\hello.txt","a")
helloFile.write("third line. ")
helloFile.close()
```

- `write lines()` 方法

  write lines()方法将字符串列表参数写入文件。

  注意换行需要自己添加

```
b=["First line\n","Second line\n","third line\n"]
helloFile=open("d:\\hello.txt","w")
helloFile.writelines(b)
helloFile.close()
```

文件内移动

无论读或写文件，Python都会跟踪文件中的读写位置。在默认情况下，文件的读/写都从文件的开始位置进行。Python提供了控制文件读写起始位置的方法，使得我们可以改变文件读/写操作发生的位置。

- `tell()` 函数可以计算文件当前位置和开始位置之间的字节偏移量。

文件的关闭

- 应该牢记使用 `close()` 方法关闭文件

```
helloFile=open("d:\\hello.txt","w")
try :
    helloFile.write("Hello,Sunny Day!")
finally:
    helloFile.close()
# 也可以使用with语句自动关闭文件:
with open("d:\\hello.txt") as helloFile:
    s=helloFile.read()
print(s)
```

文件操作

操作	作用
os.path.dirname(path)	返回path参数中的路径名称字符串
os.path.basename(path)	返回path参数中的文件名
os.path.split(path)	返回参数的路径名称和文件名组成的字符串元组
os.path.exists(path)	判断参数path的文件或文件夹是否存在。存在返回true，否则返回false
os.path.isfile(path)	判断参数path存在且是一个文件，则返回true，否则返回false
os.path.isdir(path)	判断参数path存在且是一个文件夹，则返回true，否则返回false
os.path.getsize()	查看文件大小
os.rename()	重命名文件
shutil.move(source,destination)	shutil.move()函数与shutil.copy()函数用法相似，参数destination 既可以是一个包含新文件名的路径，也可以仅包含文件夹
os.remove(path)/os.unlink(path)	删除参数path指定的文件