Python 面向对象

一、面向过程 vs 面向对象

（一）面向过程

面向过程的程序设计的核心是**过程**（流水线式思维），过程即解决问题的步骤，面向过程的设计就好比精心设计好一条流水线，考虑周全什么时候处理什么东西。

优点是：极大的降低了程序的复杂度。

缺点是：一套流水线或者流程就是用来解决一个问题，生产汽水的流水线无法生产汽车，即便是能，也得是大改，改一个组件，牵一发而动全身。

应用场景：一旦完成基本很少改变的场景，著名的例子有 Linux 内核，git，以及 Apache HTTP Server 等。

（二）面向对象

面向对象编程(Object Oriented Programming , OOP) 是一种解决软件复用的设计和编程方法。这种方法把软件系统中相近相似的操作逻辑和操作应用数据、状态，以类的型式描述出来，以对象实例的形式在软件系统中复用，以达到提高软件开发效率的作用。

优点是：解决了程序的扩展性。对某一个对象单独修改，会立刻反映到整个体系中，如对游戏中一个人物参数的特征和技能修改都很容易。

缺点是：可控性差，无法像面向过程的程序设计流水线式的可以很精准的预测问题的处理流程与结果，面向对象的程序一旦开始就由对象之间的交互解决问题，即便是上帝也无法预测最终结果。于是我们经常看到一个游戏人某一参数的修改极有可能导致阴霸的技能出现，一刀砍死 3 个人，这个游戏就失去平衡。

应用场景：需求经常变化的软件，一般需求的变化都集中在用户层，互联网应用，企业内部软件，游戏等都是面向对象的程序设计大显身手的好地方。

二、类和对象

类(class)是用来描述具有**相同属性(attribute)和方法(method)**的对象的集合，**对象(object)是类(class)的具体实例**。比如学生都有名字和分数，他们有着共同的属性。这时我们就可以设计一个学生类, 用于记录学生的名字和分数，并自定义方法打印出他们的名字和方法。

属性：类里面用于描述所有对象共同特征的变量或数据。比如学生的名字和分数。

方法：类里面的函数，用来区别类外面的函数, 用来实现某些功能。比如打印出学生的名字和分数。

以建立一个 Student 类为例：

```
class Student(object):
    def __init__(self,name,age,score):
        self._name = name
        self._age = age
        self._score = score

    def show(self):
        print("{}的年龄是{}岁，分数是{}分".format(self._name,self._age,self._score))

aStudent = Student("瑞雯",18,99)
```

属性：_name，_age，_score

方法：__init__()，show()

对象（实例）：aStudent

三、属性

（一）类属性

定义在类中函数体外，属于类

```
class A(object):
    count = 0
    def __init__(self):
        # 类中只有通过类名.类属性访问，self.__class__返回的就是类名
        self.__class__.count += 1     # 或 A.count += 1
a = A()
b = A()
# 类外既可以通过类来访问，也可通过类的实例来访问
print(a.count)                  # 2
print(A.count)                  # 2
```

【注】

要修改类属性必须通过**类名.属性**的方式来修改，而不能通过实例，因为**实例.属性**这种方式其实是创建了同名的实例属性，屏蔽了类的属性，通过 **del 实例.属性**操作后，会发现**实例.属性**还是之前的。

（二）实例属性

定义在__init()__方法中，属于对象

```
class B(object):
    def __init__(self,name,age):
        self.name = name
        self.age = age

a = B("易",18)
b = B("金克斯",8)
```

【注】

一般来说，对象的属性应该设为 private，不能被外界直接访问，这里暂且不管访问权限问题。当能从外界直接访问时，必须通过**对象.属性**来访问，这时候属性是属于对象实例的。

四、访问控制

在 Python 中并没有 public，protected，private 这样的关键字，所以无法实现数据封装，只能从语法上来定义可见性，依靠程序员自觉遵守规约。

在 Python 中，不存在函数的重载，因为函数名和普通变量名一样，都是引用，指向一个对象，是一对一的关系，都是标识符。Python 中就是通过标识符的命名来区分访问的可见性。

①字符开头的标识符，如：age

这种标识符相当于 public，可以通过**对象.属性**或者**对象.方法**来执行。

②双下划线开头结尾的标识符，如：__init__

用户最好不要自定义这种类型的标识符，因为这通常是系统调用。

③单下划线开头的标识符，如：_age

这种标识符相当于 protected，不过 Python 中没有 protected 的概念，所以被视为 private，但是，你可以按照 public 用，属于推荐非强制这种类型。

④双下划线开头的标识符，如：__name

相当于 private，Python 通过更改标识符名（_类名__标识符）来实现无法访问的机制。

五、方法

（一）实例方法

```
class Test(object):
    def __init__(self,name,age):
        self.__name = name
        self.__age = age

    def grow(self,growth):
        self.__age += growth
```

【注】

第一个参数必须是 self，它指向调用这个方法的实例。

（二）类方法

```
class Test(object):
    def __init__(self,name,age):
        self.__name = name
        self.__age = age

    @classmethod
    def getInstance(cls):
        return cls("薇恩",16)
```

【注】

绑定到类的方法，必须在调用函数的上方使用@classmethod 装饰器，同时，第一个参数必须是 cls，这个 cls 代表这个类的类型，如上面例子中的 Test，所以可以用 cls(参数)创建对象，也可以用它调用属于类的属性、其他可使用的方法。这里类似 Java 的 this 关键字。

简单地说，就是 cls 就是类的别名，这里就是 Test 的别名，Test 能干嘛，它能干嘛。特别地，也可以通过实例对象来调用类方法。

（三）静态方法

```
class Test(object):
    def __init__(self,name,age):
        self.__name = name
        self.__age = age

    @staticmethod
    def static_func():
        print("正在调用静态方法")

a = Test("VN",16)
a.static_func()
```

【注】

静态方法不像前两个有特定参数，比较自由，即使你强行把第一个参数名写出 self 和 cls，也不会有相应的作用。静态方法可以通过类或者实例对象调用。

（四）property 方法

内置函数 property()是用来使类的属性能像 Java Bean 那样操作的函数，它的函数定义为：**property(fget=None, fset=None, fdel=None, doc=None)**，其中：

fget	function to be used for getting an attribute value
fset	function to be used for setting an attribute value
fdel	function to be used for deleting an attribute
doc	docstring

举个栗子：

```
class C(object):
    def getx(self): return self._x
    def setx(self, value): self._x = value + 1
    def delx(self): del self._x
    x = property(getx, setx, delx)

c = C()
c.x = 5
print(c.x) # 6
del c.x
# print(c.x) AttributeError: 'C' object has no attribute '_x'
```

这样，既能使对象能简单调用属性，也能保证数据的封装，可是，这么写太麻烦，普遍用的形式是使用装饰器来实现 Bean

```
class Hero(object):
    def __init__(self,name):
        self.__name = name

    @property
    def name(self):
        print("getter 方法...")
        return self.__name

    @name.setter
    def name(self,value):
        print("setter 方法...")
        self.__name = value

    @name.deleter
    def name(self):
        print("deleter 方法...")
        del self.__name

p = Hero("蝙蝠侠")
p.name                                  # getter 方法...
p.name = "游城十代"                      # setter 方法...
del p.name                              # deleter 方法...
```

总结一下，一般@property 只用于私有属性“公有化”，并且 getter 方法和 deleter 方法只能有 self 参数。

六、类的特殊成员

（一）__doc__

表示类的描述信息

```
class Foo(object):
    '''Foo 的描述'''

print(Foo.__doc__)                    # Foo 的描述
```

（二）__module__和__class__

__module__表示当前操作的对象在那个模块；__class__表示当前操作的对象的类是什么，也就是谁创建了这个类，metaclass 还是 type

```
class Foo(object):
    pass

print(Foo.__module__)                 # __main__
print(Foo.__class__)                  # <class 'type'>
```

（三）__init__

构造方法，创建对象时自动执行

```
class Foo(object):
    def __init__(self):
        print("init 方法...")
        pass

f = Foo()                             # init 方法...
```

（四）__del__

析构方法，当对象在内存中被释放时，自动触发执行。

此方法一般无须定义，因为 Python 是一门高级语言，程序员在使用时无需关心内存的分配和释放，因为此工作都是交给 Python 解释器来执行，所以，析构函数的调用是由解释器在进行垃圾回收时自动触发执行的。

（五）__call__

对象后面加括号，触发执行，也就是当做函数执行时的调用代码

```
class Demo(object):
    def __init__(self):
        print("执行 init...")

    def __call__(self, *args, **kwargs):
        print("执行 call...")

d = Demo()                            #  执行 init...
d()                                   #  执行 call...
```

【注】

构造方法的执行是由创建对象触发的，即：**对象 = 类名()**，而对于__call__方法的执行是由对象后加括号触发的，即：**对象()**或者**类()()**

（六）__dict__

返回类或对象中的所有成员

```
class Province(object):

    country = 'China'

    def __init__(self, name, count):
        self.name = name
        self.count = count

    def func(self, *args, **kwargs):
        print('func')

print(Province.__dict__)                    # 返回类的所有成员
'''
{'__module__': '__main__', 'country': 'China',
'__init__': <function Province.__init__ at 0x0000022A0C5278C8>,
'func': <function Province.func at 0x0000022A0C527950>,
'__dict__': <attribute '__dict__' of 'Province' objects>,
'__weakref__': <attribute '__weakref__' of 'Province' objects>,
 '__doc__': None}
'''

print(Province("卡兹克",1).__dict__)        # 返回对象的成员
'''
{'name': '卡兹克', 'count': 1}
'''
```

【注】

由此可以看出属于类和属于对象是两码事，而__dict__就是返回相应范围的所有成员。

（七）__str__、__repr__

如果一个类中定义了__str__方法，那么在打印对象时，默认输出该方法的返回值。

```
class Text(object):
    def __init__(self,text):
        self.__text = text

    def __str__(self):
        return self.__text
    __repr__ = __str__                      # 一般都这么写偷懒

t = Text("无极剑圣")
print(t)                                    # 无极剑圣
```

【注】

类似 Java 中的 toString()方法只是打印对象有用，如果是类或类型，打印出来就是<class XXX>

str 返回的是用户看到的字符串，repr 返回的是开发者看到的字符串，为调试服务。

（八）__getitem__、__setitem__、__delitem__

用于索引操作，如字典。以上分别表示获取、设置、删除数据

```
class Subject(object):
    def __getitem__(self, item):                # 根据 item 返回一个值
        print("调用 getitem...")

    def __setitem__(self, key, value):          # 设置 key 对应的值为 value
        print("调用 setitem...")

    def __delitem__(self, key):                 # 删除这组键值对
        print("调用 delitem...")

m = Subject()
n = m['math']                                   # 调用 getitem...
m['math'] = 100                                 # 调用 setitem...
del m['math']                                   # 调用 delitem...
```

就我理解而言，就是把类改造成类似键值对的形式，可以像键值对那样根据 key 取 value，seq 中的下标本质上也是键值对，也可以通过这种方法建立。特别地，__getitem__方法的参数可以是 slice，进而实现切片操作。

首先介绍 slice()内置函数：

slice(stop)

slice(start,stop,step=1)

```
>>> slice(2,5)
slice(2, 5, None)
>>> a = slice(2,5)
>>> b = [1,2,3,4,5,6,7,8,9,10]
>>> b[a]
[3, 4, 5]
```

下面看看传入 slice 的情况

```
class MyList(object):
    def __init__(self,seq):
        self.__seq = [x*x for x in seq]

    def __getitem__(self, item):
        if isinstance(item,int):
            return self.__seq[item]
        elif isinstance(item,slice):
            return self.__seq[item]
        else:
            raise ValueError("参数错误")

l = MyList([1,2,3,4,5,6])
print(l[2])                                     # 9
print(l[2:5])                                   # [9, 16, 25]
print(l[-1])                                    # 36
print(l['a'])                                   # ValueError: 参数错误
```

（九）__iter__

用于迭代器，之所以列表、字典、元组可以进行 for 循环，是因为类型内部定义了__iter__。

```
class MyList(object):
    def __init__(self,seq):
        self.__seq = [x*x for x in seq]

    def __iter__(self):
        return iter(self.__seq)

l = MyList([1,2,3,4,5,6])
for x in l:
    print(x,end=" ")
"""输出：1 4 9 16 25 36 """
```

可以看出，实际上是利用 iter()函数把要迭代的数据转成 Iterator（迭代器）。

（十）__new__

1. 传统创建类

```
class Demo(object):
    def __init__(self,name):
        self.__name = name

d = Demo("乐芙兰")
```

d 是通过 Demo 类实例化的对象，其实不仅 d 是一个对象，Demo 本身也是一个对象，因为在 Python 中，一切皆对象，d 是通过执行 Demo 类的构造方法创建，那么 Demo 也应该是通过某个类的构造方法创建。

```
print(type(d))# <class '__main__.Demo'>        # 表示 d 由 Demo 类创建
print(type(Demo))# <class 'type'>              # 表示 Demo 由 type 类创建
```

所以，d 对象是 Demo 类的一个实例，Demo 类对象是 type 类的一个实例，即：Demo 类对象是通过 type 类的构造方法创建。

2. type 创建类

语法：**type('类名',父类元组,成员字典)**

```
def init(self,name):
    self.__name = name

def show(self):
    print(self.__name)

Demo = type('Demo',(object,),{'__init__':init,'output':show,'a':3})
d = Demo("诡术妖姬")
print(d)                                # <__main__.Demo object at 0x0000017F3C0F32E8>
d.output()                              # 诡术妖姬
```

【注】

一般用类名同名的变量来接受创建的类；父类只有 object 时，注意元组单元素时的逗号，成员字典中，成员名是字符串，对应的值可以是方法地址，可以是属性值；**另外，init，show 这些方法可以在前面加@classmethod 或者@staticmethod** 等，创建完一样有用。

3. __new__方法

new 方法是类自带的一个方法，可以重构，__new__方法在实例化的时候也会执行，并且先于__init__方法之前执行，简单理解，创建对象和初始化对象。

```
class Foo(object):
    def __init__(self, name):
        self.name = name
        print("Foo __init__")

    def __new__(cls, *args, **kwargs):
        print("Foo __new__", cls, *args, **kwargs)
        return object.__new__(cls)

f = Foo("凯特琳")
"""
Foo __new__ <class '__main__.Foo'> 凯特琳
Foo __init__
"""
```

4. 重构__new__方法

重构时，必须要调用父类的 new 方法，不然会覆盖父类的 new 方法，实例创建不了

```
class Foo(object):
    def __init__(self, name):
        self.name = name
        print("Foo __init__")

    def __new__(cls, *args, **kwargs):                    # cls 相当于传入类 Foo
        print("Foo __new__", cls, *args, **kwargs)
        return object.__new__(cls) #  继承父类的__new__方法，必须以返回值的形式继承

f = Foo("凯特琳")
"""
Foo __new__ <class '__main__.Foo'> 凯特琳
Foo __init__
"""
```

__new__方法可以创建类的变量，__init__方法创建的是对象的变量，都是个性化定制。

__new__()方法接收到的参数依次是：

①当前准备创建的类的对象；
②类的名字；
③类继承的父类集合；
④类的方法集合。

看到这里应该比较熟悉了，和 type()方法创建类的参数基本相同，实际上__new__()方法就是通过 type()函数动态创建类的。

（十一）metaclass

metaclass，直译为元类，简单的解释就是，当我们定义了类以后，就可以根据这个类创建出实例，所以，先定义类，然后创建实例。但是如果我们想创建出类呢？那就必须根据 metaclass 创建出类，所以，先定义 metaclass，然后创建类。

连接起来就是：先定义 metaclass，就可以创建类，最后创建实例。

所以，metaclass 允许你创建类或者修改类。换句话说，你可以把类看成是 metaclass 创建出来的“实例”。

metaclass 是 Python 面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会碰到需要使用 metaclass 的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。

我们先看一个简单的例子，这个 metaclass 可以给我们自定义的 MyList 增加一个 add 方法。定义 ListMetaclass，按照默认习惯，metaclass 的类名总是以 Metaclass 结尾，以便清楚地表示这是一个 metaclass。

```
# metaclass 是创建类，所以必须从 type 类型派生：
class ListMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value: self.append(value)
        return type.__new__(cls, name, bases, attrs)

class MyList(list,metaclass=ListMetaclass):
    pass
```

当我们写下 metaclass = ListMetaclass 语句时，魔术就生效了，它指示 Python 解释器在创建 MyList 时，要通过 ListMetaclass.__new__()来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

测试一下 MyList 是否可以调用 add()方法：

```
l = MyList()
l.add(1)
print(l)                                    # [1]
```

元类中，最好只定义__new__()方法，因为其他类如果使用了“metaclass=XXXMetaclass”，只会调用这个 metaclass 的 new 方法来创建类，也就是避免了直接创建类，在创建类之前还封装了一组操作，但是，metaclass 中其他的方法、属性等，是不会让使用 metaclass 创建的类继承的，所以一般在元类中，只写__new__()方法。**metaclass 可以隐式继承到子类**。

（十二）__slots__

一般情况下，可以任意地给对象添加属性，这会造成很大的漏洞，影响程序安全，为了达到限制的目的，Python 允许在定义 class 的时候，定义一个特殊的__slots__变量，来限制该 class 实例能添加的属性。

```
class Student(object):
    __slots__ = ('name', 'age') #  用 tuple 定义允许绑定的属性名称

s = Student()
s.name = 'Michael'
s.age = 25
s.score = 99                            # AttributeError: 'Student' object has no attribute 'score'
```

显然，这时候不能添加限制以外的属性，但要注意，**__slots__定义的属性仅对当前类实例起作用，对继承的子类的属性不做限制，除非在子类中也定义__slots__，这样，子类实例允许定义的属性就是自身的__slots__加上父类的__slots__**。

七、继承

（一）继承语法

```
class Animal(object):
    def __init__(self,name):
        self.__name = name

class Dog(Animal):
    kind = "Dog"
    def __init__(self,name,age):
        super.__init__(self,name)
        self.__age = age
```

可以看出，定义类时后面圆括号里的就是父类（基类），如果有多个父类或者 metaclass 时，用逗号隔开。只要子类成员有与父类成员同名，默认覆盖父类成员的，类似 Java 中的方法重写(@override。)

（二）多继承与 MRO 顺序

1. 多继承

多继承指的是子类继承多个父类，可以通过三种方式访问父类的方法：

- **父类名.父类方法(self)**
  这种方式容易造成父类方法被调用多次的问题（菱形继承问题），而且一旦父类名称发生变化，子类调用的地方都需要修改。
- **super(指定某个类名, self).父类方法()**
  从指定类名的 MRO 下一级开始调用
- **super().父类方法()**
  按照 MRO 顺序查找上级父类的方法

2. MRO(Method Resolution Order)——方法解析顺序

可以通过**类名.__MRO__**属性查找出来当前类的调用顺序，其顺序由 C3 算法来决定，保证每一个类只调用一次。

例子（O 表示 object）：

```
class A(O):pass
class B(O):pass
class C(O):pass
class E(A,B):pass
class F(B,C):pass
class G(E,F):pass
```

【MRO 的 C3 计算过程】

①A、B、C 都继承至一个基类，所以 mro 序列依次为[A,O]、[B,O]、[C,O]

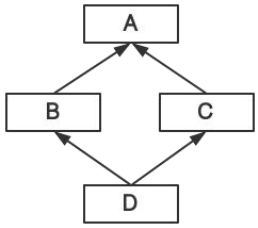
②**mro(E) = [E] + merge(mro(A),mro(B),[A,B])**
= [E] + merge([A,O] ,[B,O], [A,B])

③A 是序列[A,O]中的第一个元素，在序列[B,O]中不出现，在序列[A,B]中也是第一个元素，所以从执行 merge 操作的序列([A,O]、[B,O]、[A,B])中删除 A，合并到当前 mro，[E]中：**mro(E) = [E,A] + merge([O], [B,O], [B])**

④再执行 merge 操作，O 是序列[O]中的第一个元素，但 O 在序列[B,O]中出现并且不是其中第一个元素。继续查看[B,O]的第一个元素 B，B 满足条件，所以从执行 merge 操作的序列中删除 B，合并到[E, A]中：**mro(E) = [E,A,B] + merge([O], [O])= [E,A,B,O]**

3. super(Class,self)指定解析顺序

如果不想按照这样的顺序进行调用,可以使用 **super(指定某个类名, self).父类方法()**从指定类名的 MRO 下一级开始调用。



例如，四个类都有 show()方法，功能是打印自己的类名（A,B,C,D）

```
class D(B,C):
    def f(self):
        print("D")
        super().f()
d = D()
d.f()                                   # D B
```

但是如果人为指定：

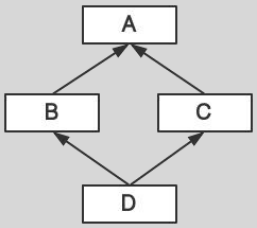
```
class D(B,C):
    def f(self):
        print("D")
        super(B,self).f()
d = D()
d.f()                                   # D C
```

从这可以看出，什么叫从指定类名的 MRO 下一级开始调用：D 的 mro 列表是[D,B,C,A]，我们参数指定了 B，方法搜索顺序就成了[C,A]。

（三）菱形继承问题

```
class A(object):
    def __init__(self,a,c,*args,**kwargs):
        pass
class B(A):
    def __init__(self,a,*args,**kwargs):
        super().__init__(a,*args,**kwargs)
class C(A):
    def __init__(self,a,b,*args,**kwargs):
        super().__init__(a,*args, **kwargs)
class D(B,C):
    def __init__(self,a,b,c,d):
        super().__init__(a,b,c,d)

d = D(1,2,3,4)
```



其实不止菱形继承中，多继承都有调用父类方法的参数传递问题，因为 super()方法可以自由确定你想用哪个类的函数，万一这些方法参数个数都不一致，你要是修改或重构代码，要改的就太多了，所以一般最好是把需要的参数写成位置参数，同时在方法的参数列表末尾加上不定长参数和关键字参数。简而言之，**子类参数全传递，父类和超类按需取值**。

【注】如果定义 func(a,b,*args)，可以这么调用：func(*[1,2,3])，也可以 func(2,*[5,8,9])。

（四）C3 线性算法△

下面是我自己实现的一个求 mro 列表的算法，merge()核心操作类似于拓扑排序。

```
import abc
def mro(cls)->list:
    if type(cls) == type(object) or type(cls) == type(abc.ABC):
        # object 是最顶层类，如果传入的是 object，那么就是[object]
        if issubclass(object,cls):
            return [cls]
        else:
            bases = cls.__bases__
            return __merge(cls, *[mro(x) for x in bases], list(bases))
    else:
        raise TypeError("mro()方法需要一个位置参数 cls，类型为 class")

def __merge(*args)->list:
    result = [args[0]]
    operation_list = list(args[1:])
    """
    any()函数：
        Return True if bool(x) is True for any x in the iterable.
        If the iterable is empty, return False.
    """
    while any(operation_list):
        # 将空的列表删除，因为有可能传进来的是[[],[],['object']]
        while [] in operation_list:
            operation_list.remove([])

        # 拓扑序列中的每个元素（类名列表）
        for y in operation_list:
            temp = y[0]
            need = True
            for t in operation_list:
                if temp in t and t.index(temp) > 0:
                    need = False
                    break
            if need:
                break;

        # 将这个元素添加到结果列表
        result.append(temp)
        # 拓扑排序的列表中删除这个元素节点
        for p in operation_list:
            while temp in p:
                p.remove(temp)
    else:
        return result
```

八、抽象类

与 java 一样，python 也有抽象类的概念但是同样需要借助模块实现，抽象类是一个特殊的类，它的特殊之处在于只能被继承，不能被实例化。如果一个抽象基类要求实现指定的方法，而子类没有实现的话，当试图创建子类或者执行子类代码时会抛出异常。

Python 中，定义抽象类需要借助 abc 模块，abc 模块提供了一个使用某个抽象基类声明协议的机制，并且子类一定要提供了一个符合该协议的实现。

```
from abc import ABC
import abc
# class Animal(metaclass=abc.ABCMeta):
class Animal(ABC):
    @abc.abstractmethod
    def test(cls):
        pass

    @abc.abstractmethod
    def bark(self):
        pass

class Dog(Animal):
    # 子类需要实现抽象方法，否则报错
    def bark(self):
        print("汪汪汪...")

    @classmethod
    def test(cls):
        print("Dog 的 test")

class Cat(Animal):
    def bark(self):
        print("喵喵喵...")

    def test(self):
        print("Cat 的 test")

# a = Animal()                                  # 抽象类不能实例化
d = Dog()
d.bark()
Dog.test()

c = Cat()
c.bark()
c.test()
```

【注】

①必须导入 abc 模块

②抽象方法用装饰器@abc.abstractmethod，子类实现时确认是实例、类还是静态方法

③抽象类，要么继承 ABC，要么不写父类，写 metaclass=ABCMeta

九、枚举类

例如星期、月份这些类型，它们的值是公认的，不会随意更改，所以可以事先将这些值都定义出来，用的时候直接拿过来用，这就是枚举类和枚举类型，最主要的一点是穷尽，枚举类型必须是一个枚举类的所有可能结果。这些枚举类型都是只创建一次对象的，每个人要用都是用一样的对象，是不可变的。

```
from enum import Enum, unique

# 创建月份的枚举类
# 枚举类中的每个属性值，默认是从 1 开始递增的整数
# 如果指定了一个属性的值，它后面的属性则会从该值开始递增
# PS：下面这个就是个构造方法，Enum 就是个类
Month = Enum('Month',('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                      'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))

# 如果想自定义属性的值可以这么写，继承 Enum 类
# unique 装饰器可以帮助我们检查，保证没有重复值
@unique
class WeekDay(Enum):
    Sun = 0                         # Sun 的 value 被设定为 0
    Mon = 1
    Tue = 2
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6

# 访问枚举类型的值
print(WeekDay.Mon)                  # WeekDay.Mon
print(WeekDay['Tue'])               # WeekDay.Tue
print(WeekDay.Wed.name)             # Wed
print(WeekDay.Wed.value)            # 3
print(WeekDay(4))                   # WeekDay.Thu
```

十、反射机制

在程序开发中，常常会遇到这样的需求：在执行对象中的某个方法，或者在调用对象的某个变量，但是由于一些原因，我们无法确定或者并不知道该方法或者变量是否存在，这时我们需要一个特殊的方法或者机制来访问或操作该未知的方法或变量，这种机制就被称之为反射。

反射机制：反射就是通过字符串的形式，导入模块；通过字符串的形式，去模块中寻找指定函数，对其进行操作。也就是利用字符串的形式去对象(模块)中操作(查找 or 获取 or 删除 or 添加)成员，一种基于字符串的事件驱动。

下面介绍反射机制的四个方法：

1. hasattr()函数

语法：**hasattr(object, name)**

功能：判断 object 中是否有属性或这方法 name，其中 object 可以是对象、可以是类、可以是模块名，存在返回 True，不存在返回 False。

2. getattr()函数

语法：**getattr(object, name, default=None)**

功能：返回 object 的 name 属性（或方法），不存在则看 default 有没有传，没有就会报错，传了的话返回 default 值作为属性（或方法）不存在的返回值。

3. setattr()函数

语法：**setattr(object,name,value)**

功能：动态给属性赋值，如果属性不存在，则先创建属性再赋值，另外，这是运行时修改，不会影响文件代码的内容。

4. delattr()函数

语法：**delattr(object,name)**

功能：删除 object 的 name 属性，，属性不存在则报错

```
import functools

class Student(object):
    std_no = 1001
    def __init__(self,name):
        self.name =name
    def f(self,string):
        print(string)

s = Student("芜情")
print(hasattr(functools,"reduce"))          # True
print(hasattr(Student,"std_no"))            # True
print(hasattr(Student,"f"))                 # True
print(hasattr(s,"name"))                    # True

print(getattr(Student,"name","None"))       # None
getattr(Student,"f","None")(s,"WCG"))       # WCG

setattr(Student,"sex","male")
print(getattr(s,"sex","None"))              # male
setattr(Student,"f",functools.reduce)
print(getattr(s,"f"))                       # <built-in function reduce>

setattr(Student,"sex","male")
print(getattr(s,"sex","None"))              # male
setattr(Student,"f",functools.reduce)
print(getattr(s,"f"))                       # <built-in function reduce>

delattr(Student,"std_no")
print(getattr(Student,"std_no","None"))     # None
delattr(Student,"f")
print(getattr(Student,"f","None"))          # None
```

十一、dataclasses 数据类

（一）dataclass 函数

dataclass(*, init = True, repr = True, eq = True, order = False, unsafe_hash = False, frozen = False)

【注】全都是命名关键字参数，注意了！！！

1. 参数含义

init：默认 True，则自动生成__init__()方法

repr：默认 True，则自动生成__repr__()方法，格式为类名和各参数书名以及参数值

eq：默认 True，自动生成__eq__()方法，此方法按顺序比较属性的元组

order：默认 False，如果为 True，则自动生成__lt__()、__le__()、__gt__()、__ge__()方法

unsafe_hash：暂且不管，和 hash 值有关，一般遇不到

frozen：默认 False，如果为 True，则禁止更改属性值

2. 示例

首先，有三种使用方法是等价的：

```
@dataclass
@dataclass()
@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)
```

下面给出具体例子：

```
@dataclass(order=True)
class Student(object):
    name:str
    age:int = 18

s = Student("yee",16)
t = Student("sky")
print(s)                                # Student(name='yee', age=16)
print(s==t)                             # False
print(s>t)                              # True
t.name = "疾风剑豪"                      # frozen=True 时，此方法报错
print(t)                                # Student(name='疾风剑豪', age=18)
```

3. 后期初始化处理

比如，有些操作需要在初始化后进行，如分离浮点数的整数部分和小数部分：

```
from dataclasses import dataclass,field
import math
@dataclass
class FloatNumber:
    val: float = 0.0
    def __post_init__(self):            # 方法签名固定，不能改
        self.decimal, self.integer = math.modf(self.val)

a = FloatNumber(2.2)
print(a)                                # FloatNumber(val=2.2)
print(a.val)                            # 2.2
print(a.integer)                        # 2.0
print(a.decimal)                        # 0.20000000000000018
```

（二）field()函数

```
field(*, default=MISSING, default_factory=MISSING,
                  init=True,repr=True,hash=None,
                      compare=True,metadata=None):
```

【注】MISSING 只是定义的缺省类，类体就是 pass 语句

1. 参数含义

default：如果提供，这将是此字段的默认值。这是必需的，因为 field()调用本身取代了默认值的正常位置

default_factory：如果提供，它必须是零参数可调用，当此字段需要默认值时将调用该调用。default_factory 不能和 default 同时出现。

init：如果为 true（默认值），则此字段作为参数包含在生成的__init__()方法中。

repr：如果为 true（默认值），则此字段包含在生成的__repr__()方法返回的字符串中

hash：一般不用

compare：如果为真（默认值），则该字段被包括在所产生的==和比较方法

metadata：这是给第三方的 API，我们用不到

2. 示例：

```
from dataclasses import dataclass,field

@dataclass
class Student(object):
    Chinese:float
    Maths:float
    English:float
    total:float = field(init=False)
    def __post_init__(self):
        self.total = self.Chinese + self.Maths + self.English

s = Student(98,99,95)
print(s)                          # Student(Chinese=98, Maths=99, English=95, total=292)
```

（三）继承问题

这里继承和一般的继承一样，子类会继承父类的属性和方法，按照 mro 列表的顺序，查找所调用的函数和属性，都找不到则报错。