**1、什么是“鸭子类型”**

Python崇尚“鸭子类型”

对于鸭子模型常见的说法是：“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

**鸭子类型**（英语：**duck typing**）在程序设计中是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由“当前方法和属性的集合”决定

在鸭子类型中，关注点在于对象的行为能做什么；而不是关注对象所属的类型。例如，在不使用鸭子类型的语言中，我们可以编写一个函数，它接受一个类型为"鸭子"的对象，并调用它的"走"和"叫"方法。在使用鸭子类型的语言中，这样的一个函数可以接受一个任意类型的对象，并调用它的"走"和"叫"方法。如果这些需要被调用的方法不存在，那么将引发一个运行时错误。任何拥有这样的正确的"走"和"叫"方法的对象都可被函数接受的这种行为引出了以上表述，这种决定类型的方式因此得名

鸭子类型通常得益于"不"测试方法和函数中参数的类型，而是依赖文档、清晰的代码和测试来确保正确使用

在常规类型中，我们能否在一个特定场景中使用某个对象取决于这个对象的类型，而在鸭子类型中，则取决于这个对象是否具有某种属性或者方法——即只要具备特定的属性或方法，能通过鸭子测试，就可以使用。

内容参考自维基百科：[https://zh.wikipedia.org/wiki/鸭子类型](https://zh.wikipedia.org/wiki/%E9%B8%AD%E5%AD%90%E7%B1%BB%E5%9E%8B)

**2、“鸭子类型”从何而来**

**2.1 多态**

先来看看“多态”

大学时学习过C、Java基础这一类强类型语言，面向对象编程的三大特性之一有个概念叫做“多态”

简单来说，定义时的类型和运行时的类型不一样就是多态

更通俗的来说，多态是指一类事物有多种形态。比如动物有多种形态，人、狗、猫等等

放到二进制的世界，例如文件，文件有多种形态：文本文件、可执行文件

总而言之，多态即某类的再分类，再分的每一类就是父类的多种形态的一种

例如下面代码示例：

* 程序在运行到list这一步是只把Cat，Dog，Duck当成变量
* 只有在最后实例化调用类方法的时候才会明白是一个类，这就表示多态

class Cat(object):

def info(self):

print("我是一只猫")

class Dog(object):

def info(self):

print("我是一只狗")

class Duck(object):

def info(self):

print("我是一只鸭")

list=[Cat,Dog,Duck]

for animal in list:

animal().info()

**2.2 多态的使用**

再来一个偏实际的应用示例代码

from abc import ABCMeta,abstractmethod #(抽象方法)

class Payment(metaclass=ABCMeta): # metaclass 元类 metaclass = ABCMeta表示Payment类是一个规范类

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

self.money=money

self.name=name

@abstractmethod # @abstractmethod表示下面一行中的pay方法是一个必须在子类中实现的方法

def pay(self,\*args,\*\*kwargs):

pass

class AliPay(Payment):

def pay(self):

# 支付宝提供了一个网络上的联系渠道

print('%s通过支付宝消费了%s元'%(self.name,self.money))

class WeChatPay(Payment):

def pay(self):

# 微信提供了一个网络上的联系渠道

print('%s通过微信消费了%s元'%(self.name,self.money))

class Order(object):

def account(self,pay\_obj):

pay\_obj.pay()

pay\_obj = WeChatPay("wang",100)

pay\_obj2 = AliPay("zhang",200)

order = Order()

order.account(pay\_obj)

order.account(pay\_obj2)

从上述代码可见，在调用order对象的account()方法时，程序并不关心为该方法传入的参数对象pay\_obj是谁，只要求此参数对象pay\_obj包含pay()方法即可，而调用者传入的参数对象类型pay\_obj是子类对象还是其他类对象，对Python来说无所谓

多态性就是相同的消息（函数方法触发）使得不同的类做出不同的响应，这就是典型的类编程中多态性的应用实例

**2.3 鸭子类型产生**

在上面的例子中order.account(pay\_obj)中pay\_obj不需要类型声明，而java在使用时要定义好类型

（order.account(Payment pay\_obj)），所以你传入别的类型对象一定报错

但是python因为是动态语言所以传入的对象只要拥有调用的方法即可视为Payment类型对象，这就是所谓的鸭子类型

**class Duck():**

**def walk(self):**

**print('I walk like a duck')**

**def swim(self):**

**print('i swim like a duck')**

**class Person():**

**def walk(self):**

**print('this one walk like a duck')**

**def swim(self):**

**print('this man swim like a duck')**

可以很明显的看出，Person类拥有跟Duck类一样的方法，当有一个函数调用Duck类，并利用到了两个方法walk()和swim()。我们传入Person类也一样可以运行，函数并不会检查对象的类型是不是Duck，只要他拥有walk()和swim()方法，就可以正确的被调用

**3、小结**

* 在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的

例如，在不使用鸭子类型的语言中，我们可以编写一个函数，它接受一个类型为"鸭子"的对象，并调用它的"走"和"叫"方法

* 在使用鸭子类型的语言中，这样的一个函数可以接受一个任意类型的对象，并调用它的"走"和"叫"方法

如果这些需要被调用的方法不存在，那么将引发一个运行时错误

任何拥有这样的正确的"走"和"叫"方法的对象都可被函数接受的这种行为引出了以上表述，这种决定类型的方式因此得名

* 鹅不是鸭子，但是鹅一旦拥有鸭子的一些方法，就可以被当成鸭子类型了
* 鸭子类型的好处就在于能够避免一些类的重写，无需大量复制相同的代码
* 鸭子类型使用的前提是需要良好的文档支持，不然会让代码变得很混乱，如果没有良好的文档及说明，有可能会导致你的“鸭子”不是我的“鹅”了

## Python：多态、鸭子模型和抽象基类

# 1. 多态

* 什么是多态  
  -- 多态，指的是一种事务具有多种形态；  
  -- python是一种动态语言，默认支持多态，**同一个方法** 调用 **不同的类对象** ，执行的 **结果各不相同**；
* 多态实现  
  -- 继承：**不同子类** 继承 **同一父类**；  
  -- 重写：子类重写 **同一个方法**，保证执行结果各不相同；
* 示例  
  -- 有如下代码：

>>> class Animals():

... def talk(self):

... print("Animal talk")

...

>>>

>>> class People(Animals): # 继承 Animals 类

... def talk(self):

... print('People speak language')

...

>>>

>>> class Cat(Animals): # 继承 Animals 类

... def talk(self):

... print('Cat say miaomiao')

...

>>>

>>> cat = Cat()

>>> peo = People()

>>>

>>> cat.talk() # 调用 talk 方法

Cat say miaomiao

>>> peo.talk() # 调用 talk 方法

People speak language

* 如上所示：  
  -- cat 和 peo 两个对象调用同一个 talk() 方法；  
  -- 最后得到两种不同的结果；
* 多态的优点：  
  -- 多态可以增加代码的灵活度；  
  -- 是调用方法的技巧，不会影响到类的内部设计；  
  -- 多态可以看做 **接口函数的重用**，**同一种接口方法** 通过 **接收不同的类** 对象，从而实现不同的功能；
* 多态使用场景：  
  -- 方法参数接收同一父类的不同子类对象。

# 2. 鸭子模型

* 什么是鸭子模型  
  -- 当看到一只鸟走起来像鸭子，游泳起来也像鸭子，叫起来也像鸭子，那么这只鸟就可以被称为鸭子；  
  -- 鸭子模型和多态一样，都是接受不同的类对象，并调用相同的方法（即：鸭子的 游泳 和 叫 方法）；  
  -- 对于一个鸭子模型来说，我们并 **不关心接收的类对象是否真的是鸭子类**，只关心这个类是如何被使用的；  
  -- 注意：如果这些需要被调用的方法不存在，那么将引发一个运行时错误。
* 示例  
  -- 有如下代码：

>>> class Duck:

... def quack(self):

... print("duck quack")

...

>>>

>>> class Bird: # Bird 类与 Duck 类无继承关系

... def quack(self):

... print("bird quack")

...

>>>

>>> class Dog: # Dog类与 Duck 类无继承关系

... def quack(self):

... print("dog quack")

...

>>>

>>> def animal\_quack(animal): # animal\_quack 方法可以调用任何对象的 quack() 方法，不关心对象是谁

... animal.quack()

...

>>>

>>> duck = Duck()

>>> bird = Bird() # bird 实例与 duck 实例无任何关系

>>> dog = Dog() # dog 实例与 duck 实例无任何关系

>>>

>>> for animal in [duck, bird, dog]:

... animal\_quack(animal)

...

duck quack

bird quack

dog quack

-- 如上所示：  
-- duck、bird、dog 分别来自三个不同的类，而且类之间是 **没有继承关系** 的；  
-- duck、bird、dog 调用 animal\_quack 方法，得到三种不同的结果，符合多态的特征；

* 鸭子模型的优点：  
  -- 鸭子模型不关关心类对象是什么，不需要类之间具有继承关系；  
  -- 鸭子模型让代码比多态更加灵活度；
* 多态使用场景：  
  -- 鸭子模型中，接收不同的类将会产生不同的行为，而无须明确知道这个类实际上是什么，这是多态的重要应用场景；  
  -- 实际生产环境中，主要用于 **接口开发**，即用同一个函数接收不同的类对象，从而实现不同的功能，而且无需关注对象之间的继承关系；

# 3. 抽象基类

* 什么是抽象基类  
  -- 抽象基类，这个词可能听着比较"深奥"，其实 抽象 就是 假 的意思，基类 就是 父类，抽象基类 就是 假父类；  
  -- 具体来说，由 abc.ABCMeta 这个元类实现的类，就是抽象基类；
* 示例：  
  -- 如下代码中的 AbstractClass 类继承自 abc.ABCMeta，AbstractClass 就是抽象基类；

class AbstractClass(metaclass=abc.ABCMeta):

pass

* 抽象基类的作用  
  -- 判断是否为某个对象的实例

>>> class MyList(object):

... def \_\_init\_\_(self, my\_list):

... self.my\_list= my\_list

... def \_\_len\_\_(self):

... return len(self.my\_list)

...

>>>

>>> class NewList(MyList): # NewList 继承自 MyList

... pass

...

>>> ml = MyList(["a", "b", "c"])

>>>

>>> from collections.abc import Sized, Iterable

>>>

>>> print(isinstance(ml, Sized))

True # 返回 True，因为这里会检查实例对象中有没有\_\_len\_\_方法，有即输出True

>>> nl = NewList([1, 2, 3])

>>> print(isinstance(nl, MyList))

True # 返回 True，因为 nl 实例化的类 NewList 同时也是 MyList 的子类

-- 强制要求父类被子类继承，并在子类实现某个方法，否则子类初始化时就会报错；

>>> from abc import ABCMeta,abstractmethod

>>>

>>>

>>> class Source(metaclass=ABCMeta): # 创建抽象基类 Source

... @abstractmethod # 表示装饰的方法必须被子类所实现，否则会报错

... def get(self,key):

... pass

...

>>>

>>> class Mysource(Source): # 子类 Mysource 继承自 抽象基类 Source

... def get(self,key): # 实现 get 方法，这个方法是 抽象基类 Source 强制要求实现的

... pass

...

>>>

>>> class Mysource1(Source): # 子类 Mysource1 没有实现 抽象基类 Source 强制要求实现的 get 方法

... pass

...

>>> test = Source() # test 直接实例化 Source 父类

Traceback (most recent call last): # 此处报错，因为抽象类无法实现实例化

File "<stdin>", line 1, in <module>

TypeError: Can't instantiate abstract class Source with abstract methods get

>>>

>>> test = Mysource() # 此处实例化 Mysource，未报错

>>>

>>> test = Mysource1()

Traceback (most recent call last): # 报错，继承类必须实现抽象类的方法

File "<stdin>", line 1, in <module>

TypeError: Can't instantiate abstract class Mysource1 with abstract methods get

* 抽象基类使用场景  
  -- 接口强制规定，主要是 **强制子类实现某个方法**，否则就提示报错；
* 抽象基类的有点：  
  -- 处理继承问题方面更加规范、系统；  
  -- 明确调用之间的相互关系，使得继承层次更加清晰；
* 抽象基类的缺点：  
  -- 抽象基类在 python 并非在于用来继承，主要用来理解 python继承 的定义，应该 **尽量使用鸭子模型**；  
  -- 如果一定要继承接口的话，比较 **推荐多继承**，抽象基类容易 **设计过度**；