#### 1.面向对象编程

目前，我们距离大功告成只差一步之遥了，我们每天都在接触对象，你的车是一个对象，你可以用属性来描述它，它会跑，它就有了一些方法，比如跑的这个方法。今天我们学一下面向对象编程。

**重点**：掌握类的编写，类的继承，类的实例化

#### 2.类的定义

何为类？类是用来描述一类事物的工具，比如鸟这个类，鸟下面有很多品种，我们可以使用下面的代码创建鸟这个类。

class bird:

鸟有一些特征，比如身高，体重，颜色，我们需要在类中为他们创建这些属性，使用下面的初始化方法来为它们创建属性。

class bird:  
 def \_\_init\_\_(self, color,height,weight):  
 self.color = color  
 self.height=height  
 self.weight=weight  
#\_\_init\_\_方法是一个特殊的方法，也被称为构造方法。当创建一个类的实例时，\_\_init\_\_方法会自动调用，用于初始化对象的属性或执行其他必要的设置操作。

这样鸟这个类就有了身高，体重和颜色。

除此之外，鸟还有一些行为特征，比如会飞，会吃饭，我们如何具体形象化的描述呢？我们使用方法来描述类对象的行为。我们可以给出下面代码。

何为方法？就是定义在类内部的函数。

class bird:  
 def \_\_init\_\_(self, color,height,weight):  
 self.color = color  
 self.height=height  
 self.weight=weight  
 def sayHello(self):  
 print(f"我的身高是{self.height}，我的体重是{self.weight}，我的颜色是{self.color}")  
 def fly(self):  
 print("I can fly")

在类内部定义的方法可以调用类的属性。这样我们就创建出来了一个合格的类。

#### 3.类的实例化

类的实例化也叫做对象的创建，比如鸟的这个类里面有很多不同的鸟，比如1号鸟，2号鸟，3号鸟，他们的颜色，身高和体重各不相同，我们可以创建对象时指定不同的属性来为它们具体化。如下，我们创建了三只鸟，颜色，高度和体重各不相同。

bird1=bird("yellow",100,10)  
bird2=bird("black",200,20)  
bird3=bird("white",300,30)

不难看出，它们的格式是变量名=类的实例化。也叫做引用数据类型，在前面讲基本数据类型的时候，我们讲过变量名=基本数据类型，但是也不一定全是基本数据类型，这里我们学到了引用数据类型，变量名=类的实例化同样适用。

现在我有了三只鸟，但是我想让三只鸟sayHello和fly，当然也是可以的，因为我的鸟属于bird类，我的bird类里面有sayHello和fly的行为，我的鸟对象肯定也会这两个行为。写出下面代码。

class bird:  
 def \_\_init\_\_(self, color,height,weight):  
 self.color = color  
 self.height=height  
 self.weight=weight  
 def sayHello(self):  
 print(f"我的身高是{self.height}，我的体重是{self.weight}，我的颜色是{self.color}")  
 def fly(self):  
 print("I can fly")  
  
bird1=bird("yellow",100,10)  
bird2=bird("black",200,20)  
bird3=bird("white",300,30)  
bird1.sayHello()  
bird2.sayHello()  
bird3.fly()  
  
#输出结果  
我的身高是100，我的体重是10，我的颜色是yellow  
我的身高是200，我的体重是20，我的颜色是black  
I can fly

##### 3.1实例化的细节

我们的bird4能不能创建成功呢，因为我们没有指定它的weight。

bird4=bird("yellow",100)  
#输出  
Traceback (most recent call last):  
 File "D:\project\tutorial\python基础\code\3.py", line 20, in <module>  
 bird1=bird("yellow",100)  
 ^^^^^^^^^^^^^^^^^^  
TypeError: bird.\_\_init\_\_() missing 1 required positional argument: 'weight'

可以看到缺少一个参数，那么我们能不能设置一个默认参数呢？也是可以的，需要修改类的/**init**/函数。如下代码所示

class bird:  
 def \_\_init\_\_(self, color="yellow",height=100,weight=100):  
 self.color = color  
 self.height=height  
 self.weight=weight  
 def sayHello(self):  
 print(f"我的身高是{self.height}，我的体重是{self.weight}，我的颜色是{self.color}")  
 def fly(self):  
 print("I can fly")  
  
bird4=bird() #使用默认参数创建一个对象。  
bird4.sayHello()  
  
#输出结果  
我的身高是100，我的体重是100，我的颜色是yellow

可以看到我们在创建bird1,bird2,bird3的时候是用来位置参数来创建的对象，我们也可以打乱它们的位置，使用关键字来创建对象，如下代码。

class bird:  
 def \_\_init\_\_(self, color="yellow",height=100,weight=100):  
 self.color = color  
 self.height=height  
 self.weight=weight  
 def sayHello(self):  
 print(f"我的身高是{self.height}，我的体重是{self.weight}，我的颜色是{self.color}")  
 def fly(self):  
 print("I can fly")  
  
bird5=bird(weight=100,color="bule",height=100)  
  
bird5.sayHello()  
#输出结果  
我的身高是100，我的体重是100，我的颜色是bule

##### 3.2类和对象

类方法：类方法是定义在类中的方法，使用装饰器@classmethod来标识。类方法可以访问类属性，但不能访问实例属性，因为类方法是在类级别上执行，而不是在实例级别上执行。

类属性：类属性是定义在类中的变量，它属于类本身，而不是类的实例。类属性在所有类的实例之间是共享的，可以通过类名或实例来访问。

对象方法：对象方法是定义在类中的方法，用于操作类的实例。对象方法可以访问和修改实例属性，并且可以通过self参数来访问其他对象方法。

对象属性：对象属性是定义在类的实例中的变量，它属于类的每个实例单独拥有。每个实例的对象属性可以具有不同的值。

class Student:  
 school = "ABC School" # 类属性  
  
 def \_\_init\_\_(self, name, age):  
 self.name = name # 对象属性  
 self.age = age  
  
 @classmethod  
 def get\_school(cls):  
 return cls.school # 访问类属性  
  
 def get\_name(self):  
 return self.name # 访问对象属性  
  
 def set\_name(self, new\_name):  
 self.name = new\_name # 修改对象属性  
  
 def get\_age(self):  
 return self.age  
  
 def set\_age(self, new\_age):  
 self.age = new\_age  
  
 def display\_info(self):  
 print(f"Name: {self.name}, Age: {self.age}, School: {self.get\_school()}")  
  
# 创建学生对象  
student1 = Student("Alice", 18)  
student2 = Student("Bob", 17)  
  
# 调用对象方法  
student1.display\_info() # 输出：Name: Alice, Age: 18, School: ABC School  
student2.display\_info() # 输出：Name: Bob, Age: 17, School: ABC School  
  
# 调用类方法  
print(Student.get\_school()) # 输出：ABC School  
  
# 修改对象属性  
student1.set\_name("Alex")  
student1.set\_age(19)  
student1.display\_info() # 输出：Name: Alex, Age: 19, School: ABC School

#### 4.类的继承

继承允许我们创建新的类（称为子类）从现有的类（称为父类）继承属性和方法。通过继承，子类可以重用父类的代码，并在此基础上添加新的功能或修改现有功能。

##### 4.1单继承

class Animal:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def eat(self):  
 print(f"{self.name} is eating.")  
  
 def sleep(self):  
 print(f"{self.name} is sleeping.")  
  
class Dog(Animal): #子类如果没有构造方法，会自动调用父类的构造方法  
 def bark(self):  
 print(f"{self.name} is barking.")  
 def eat(self): #方法重写  
 print(f"{self.name} likes eating meat")  
  
# 创建父类实例  
animal = Animal("Animal")  
animal.eat() # 输出：Animal is eating.  
  
# 创建子类实例  
dog = Dog("huahua")  
dog.eat() # 输出：huahua likes eating meat.  
dog.bark() # 输出：huahua is barking.

在上面的例子中，我们定义了一个父类Animal，它具有eat和sleep两个方法。然后我们定义了一个子类Dog，它继承自父类Animal。

通过继承，子类Dog自动获得了父类Animal的属性和方法，包括name属性和eat方法。子类可以直接使用继承得到的属性和方法，无需重新编写。

此外，子类还可以添加自己特定的属性和方法。在上面的例子中，子类Dog添加了一个bark方法，用于输出狗叫声。

通过创建父类实例animal和子类实例dog，我们可以看到子类继承了父类的方法，并可以调用这些方法。此外，子类还可以调用自己特定的方法。

##### 4.2多层继承

多层继承时，一个子类可以继承一个父类，并且可以被另一个类继承，形成继承链。

class Vehicle:  
 def \_\_init\_\_(self, brand):  
 self.brand = brand  
  
 def drive(self):  
 print(f"{self.brand} is driving.")  
  
class Car(Vehicle):  
 def honk(self):  
 print(f"{self.brand} is honking.")  
  
class ElectricCar(Car):  
 def charge(self):  
 print(f"{self.brand} is charging.")  
  
# 创建实例  
vehicle = Vehicle("Vehicle")  
vehicle.drive() # 输出：Vehicle is driving.  
  
car = Car("Toyota")  
car.drive() # 输出：Toyota is driving.  
car.honk() # 输出：Toyota is honking.  
  
electric\_car = ElectricCar("Tesla")  
electric\_car.drive() # 输出：Tesla is driving.  
electric\_car.honk() # 输出：Tesla is honking.  
electric\_car.charge() # 输出：Tesla is charging.

我们定义了一个父类Vehicle，它具有drive方法。子类Car继承自父类Vehicle，并添加了自己的方法honk。另一个子类ElectricCar继承自子类Car，并添加了自己的方法charge。

通过多层继承，子类ElectricCar继承了父类Vehicle和子类Car的属性和方法。因此，electric\_car实例可以调用继承的方法drive和honk，以及自己特定的方法charge。

**继承链的顺序很重要，它决定了属性和方法的优先级**。在上面的例子中，ElectricCar继承了父类Vehicle和子类Car的属性和方法，如果某个属性或方法在多个层次上都存在，则优先使用最靠近子类的定义。

需要注意的是，虽然多层继承可以带来灵活性和代码重用，但过度的层次嵌套可能会导致代码复杂性增加。因此，需要谨慎设计继承关系，避免继承链过长或过于复杂。可以使用UML类图画出来类之间的关系，然后再编程。

##### 4.3多重继承

多重继承是指一个子类可以同时继承多个父类的属性和方法。在Python中，可以通过在类定义时列出多个父类来实现多重继承。

class Flyable:  
 def fly(self):  
 print("Flying...")  
  
class Swimmable:  
 def swim(self):  
 print("Swimming...")  
  
class Bird(Flyable):  
 def chirp(self):  
 print("Chirping...")  
  
class Fish(Swimmable):  
 def breatheUnderWater(self):  
 print("Breathing underwater...")  
  
class Duck(Bird, Swimmable):  
 def quack(self):  
 print("Quacking...")  
  
# 创建实例  
duck = Duck()  
duck.fly() # 输出：Flying...  
duck.swim() # 输出：Swimming...  
duck.chirp() # 输出：Chirping...  
duck.quack() # 输出：Quacking...  
duck.breatheUnderWater() # 输出：Breathing underwater...

在上面的例子中，我们定义了两个父类Flyable和Swimmable，它们分别具有fly和swim方法。子类Bird继承自父类Flyable，并添加了自己的方法chirp。子类Fish继承自父类Swimmable，并添加了自己的方法breatheUnderWater。最后，子类Duck继承自子类Bird和Fish，并添加了自己的方法quack。

通过多重继承，子类Duck继承了父类Flyable和Swimmable的属性和方法，以及子类Bird和Fish的属性和方法。因此，duck实例可以调用继承的方法fly、swim、chirp、quack和breathe\_underwater。

##### 4.4super关键字

在Python中，super()函数用于调用父类的方法，通过super()函数，我们可以在子类中调用父类的方法，而无需显式地指定父类的名称。

class Animal:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def eat(self):  
 print(f"{self.name} is eating.")  
  
class Dog(Animal):  
 def \_\_init\_\_(self, name, breed):  
 super().\_\_init\_\_(name) # 调用父类的\_\_init\_\_方法  
 self.breed = breed  
  
 def bark(self):  
 print(f"{self.name} is barking.")  
  
 def eat(self):  
 super().eat() # 调用父类的eat方法  
 print(f"{self.name} is eating loudly.")  
  
# 创建实例  
dog = Dog("Bobby", "Labrador")  
dog.bark() # 输出：Bobby is barking.  
dog.eat() # 输出：Bobby is eating. Bobby is eating loudly.

在上面的例子中，我们定义了一个父类Animal，它具有eat方法。子类Dog继承自父类Animal，并添加了自己的方法bark。子类Dog的\_\_init\_\_方法通过使用super()函数调用了父类Animal的\_\_init\_\_方法，以获得父类的属性name。同样地，子类Dog的eat方法通过使用super()函数调用了父类Animal的eat方法，以获得父类的行为，并在此基础上添加了自己的行为。

##### 4.5私有属性和私有方法

在Python中，私有属性和方法是以双下划线（\_\_）开头的命名约定，它们用于限制对类的某些成员的访问权限。继承关系中，子类通常不能直接访问父类的私有属性和方法。然而，子类可以通过一些机制间接地访问父类的私有成员，例如使用公有方法或受保护方法。

下面是一个示例，演示了继承关系中私有属性和方法的访问：

class Person:  
 def \_\_init\_\_(self, name):  
 self.\_\_name = name  
  
 def get\_name(self):  
 return self.\_\_name  
  
 def set\_name(self, new\_name):  
 self.\_\_name = new\_name  
 def \_\_privateMethod(self):  
 print("private method")  
 def getPrivateMethod(self):  
 return self.\_\_privateMethod()  
# 创建实例  
person = Person("huahai2022")  
  
# 使用 getter 方法获取私有属性值  
print(person.get\_name())  
  
# 使用 setter 方法修改私有属性值  
person.set\_name("huahua Gou")  
print(person.get\_name())  
person.getPrivateMethod()

#### 5.面向对象总结

面向对象的灵魂：封装，继承，多态

继承：继承是面向对象编程中的一个重要特性，它允许一个类（称为子类或派生类）继承另一个类（称为父类或基类）的属性和方法。子类可以继承父类的特征，同时可以根据需要添加自己的特征或修改继承的特征。

封装：封装是面向对象编程中的一种机制，它将数据和操作数据的方法封装在一个单独的实体中，以实现对数据的隔离和保护。通过封装，我们可以隐藏实现细节，只暴露必要的接口供其他对象使用。

多态：允许使用相同的接口来处理不同类型的对象，从而实现代码的通用性和灵活性。多态性使得不同的对象可以对相同的消息做出不同的响应。