第9章 类

类是面向对象语言的程序设计中的概念，是面向对象编程的基础。

类是对现实生活中一类具有**共同特征**的事物的抽象, 例如：车辆、学生、省份。

面向对象编程(Object Oriented Programming，OOP)是最有效的软件编写方法之一。

在面向对象编程中，你编写表示现实世界中的事物和情景的**类(class)**，并基于这些类来创建**对象(object)/实例(instance)**。 编写类时，定义一大类对象都有的通用行为。

基于类创建对象时，每个对象都自动具备这种通用行为，然后可根据需要赋予每个对象独特的个性（数据）。 使用面向对象编程可模拟现实情景，其逼真程度达到了令人惊讶的地步。

根据类来创建对象称为**实例化(instantiation)** ，这让你能够使用类的实例。

在本章中，将编写一些类并创建其实例。 将指定可在实例中存储什么信息，定义可对这些实例执行哪些操作。 还将编写一些类来扩展既有类的功能，让相似的类能够高效地共享代码。

在本章中，将编写的类存储在模块中，并在自己的程序文件中导入其他程序员编写的类。 理解面向对象编程有助于你像程序员那样看世界，还可以帮助真正明白自己编写的代码：

- 不仅是各行代码的作用，还有代码背后更宏大的概念。 了解类背后的概念可培养逻辑思维，能够通过编写程序来解决遇到的几乎任何问题。
- 随着面临的挑战日益严峻，类还能让你以及与你合作的其他程序员的生活更轻松。
- 如果你与其他程序员基于同样的逻辑来编写代码，你们就能明白对方所做的工作。
- 你编写的程序将能被众多合作者所理解，每个人都能事半功倍。

9.1 创建和使用类

使用类几乎可以模拟任何东西。 下面来编写一个表示狗的简单类Dog，它表示的不是特定的狗，而是任何狗（抽象）。

对于大多数狗，我们都知道些什么呢？ 它们都有名字和年龄。 我们还知道，大多数狗还会蹲下和打滚。 由于大多数狗都具备上述两项信息（名字和年龄）和两种行为（蹲下和打滚），我们的Dog类将包含它们。 这个类让Python知道如何创建表示狗的对象。 编写这个类后，我们将使用它来创建表示特定小狗的实例。

9.1.1 创建Dog类

根据Dog类创建的每个实例都将存储名字和年龄，我们赋予了每条狗蹲下（sit()）和打滚（roll_over()）的能力：

In [10]:
```
class Dog:
    """一次模拟狗的简单尝试，  编写一个类。"""
```

```
    def __init__(self, name, age):
        """初始化属性name和age。"""
        self.name = name
        self.age = age

    def sit(self):
        """模拟小狗收到命令时蹲下。"""
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        """模拟小狗收到命令时打滚。"""
        print(f"{self.name} rolled over!")
```

首先定义了一个名为Dog的类。 根据约定，在Python中，首字母大写的名称指的是类。 这个类定义中没有圆括号，因为要从空白创建这个类。

编写了一个文档字符串，对这个类的功能做了描述。

方法__init__()

```
def \_\_init__(self, name, age):
```

类中的函数称为方法。 前面学到的有关函数的一切都适用于方法，就目前而言，唯一重要的差别是调用方法的方式。

方法__init()是一个特殊方法(构造函数），每当根据**Dog**类创建新实例时，**Python**都会自动运行它。 在这个方法的名称中，开头和末尾各有两个下划线，这是一种约定，旨在避免**Python**默认方法与普通方法发生名称冲突。 务必确保**__init**()的两边都有两个下划线，否则当使用类来创建实例时，将不会自动调用这个方法，进而引发难以发现的错误。

将方法__init__()定义成包含三个形参：self、name和age。

在这个方法的定义中，形参self必不可少，而且必须位于其他形参的前面。 为何必须在方法定义中包含形参self呢？因为Python调用这个方法来创建Dog实例时，将自动传入实参self。 每个与实例相关联的方法调用都自动传递实参self，它是一个指向实例本身的引用，让实例能够访问类中的属性和方法。 创建Dog实例时，Python将调用Dog类的方法__init__()。 通过实参向Dog()传递名字和年龄，**self会自动传递，**因此不需要传递它。 每当根据Dog类创建实例时，都只需给最后两个形参（name和age）提供值。

```
def sit(self):
    """模拟小狗收到命令时蹲下。"""
    print(f"{self.name} is now sitting.")

def roll_over(self):
    """模拟小狗收到命令时打滚。"""
    print(f"{self.name} rolled over!")
```

Dog类还定义了另外两个方法：sit()和roll_over()。 这些方法执行时不需要额外的信息，因此它们只有一个形参self。 我们随后将创建的实例能够访问这些方法，换句话说，它们都会蹲下和打滚。 当前，sit()和roll_over()所做的有限，只是打印一条消息，指出小狗正在蹲下或打滚。 但可以扩展这些方法以模拟实际情况：如果这个类包含在一个计算机游戏中，这些方法将包含创建小狗蹲下和打滚动画效果的代码； 如果这个类是用于控制机器狗的，这些方法将让机器狗做出蹲下和打滚的动作。

9.1.2 根据类创建实例

可将类视为有关如何创建实例的说明。Dog类是一系列说明，让Python知道如何创建表示特定小狗的实例。下面来创建一个表示特定小狗的实例：

In [11]:
```
my_dog = Dog('Willie', 6)
print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
```

```
My dog's name is Willie.
My dog is 6 years old.
```

这里使用的是前一个示例中编写的Dog类。 首先，让Python创建一条名字为'Willie'、年龄为6的小狗。 遇到这行代码时，Python使用实参'Willie'、6和调用Dog类的方法__init**()** 。 方法**__init**()创建一个表示特定小狗的实例，并使用提供的值来设置属性name和age。 接下来，Python返回一个表示这条小狗的实例，而我们将这个实例赋给了变量my_dog。

在这里，命名约定很有用：通常可认为首字母大写的名称（如Dog）指的是类，而小写的名称（如my_dog）指的是根据类创建的实例。

访问属性

要访问实例的属性，可使用句点表示法。 访问my_dog 的属性name的值：

In [12]:
```
my_dog.name
```

Out[12]:
```
'Willie'
```

调用方法

In [ ]:
```
my_dog.sit()
my_dog.roll_over()
```

创建多个实例

可按需求根据类创建任意数量的实例。 下面再创建一个名为your_dog 的小狗实例：

In [13]:
```
my_dog = Dog('Willie', 6)
print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

your_dog = Dog('Brad', 3)
print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Brad.
Your dog is 3 years old.
Brad is now sitting.
```

9.2 使用类和实例

In [21]:
```
class Car:
```

```
    """一次模拟汽车的简单尝试。"""

    def __init__(self, make, model, year):
        """初始化描述汽车的属性。"""
        self.__make = make
        self.model = model
        self.year = year

        # 设置默认值
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """返回整洁的描述性信息。"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """打印一条指出汽车里程的消息。"""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """将里程表读数设置为指定的值。"""
        if mileage < self.odometer_reading:
            pass
        else:
            self.odometer_reading = mileage

    def increment_odometer(self, miles):
        """将里程表读数增加指定的量。"""
        self.odometer_reading += miles

    def car_wash(self, is_da_la = False):
        if is_da_la:
            print('washed my car and also da_la')
        else:
            print('I am poor, washed my car without da_la')

    def add_alarm_system(self):
        print("alarm system has been installed!")
```

In [22]:

```
my_new_car = Car('Audi', 'A6', 2022)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
my_new_car.car_wash()
my_new_car.add_alarm_system()

print()

li_old_car = Car('Toyota', 'camery', 1990)
print(li_old_car.get_descriptive_name())
li_old_car.update_odometer(300000)
li_old_car.read_odometer()
li_old_car.car_wash(is_da_la = True)
my_new_car.add_alarm_system()
```

```
2022 Audi A6
This car has 0 miles on it.
I am poor, washed my car without da_la
alarm system has been installed!

1990 Toyota Camery
This car has 300000 miles on it.
washed my car and also da_la
alarm system has been installed!
```

9.2.3 修改属性的值

能以三种方式修改属性的值：

- 直接通过实例进行修改
- 通过方法进行设置

In [35]:
```
my_new_car = Car('audi', 'A4', 2019)
print(my_new_car.get_descriptive_name())

#your_new_car = Car('BMW', '3', 2022)
#print(my_new_car.get_descriptive_name())

#直接通过实例进行修改
my_new_car.read_odometer()
my_new_car.odometer_reading = -100
my_new_car.read_odometer()

#通过方法修改属性的值
#my_new_car.update_odometer(200)
my_new_car.read_odometer()

#通过方法对属性的值进行递增
#your_new_car.update_odometer(40)
#your_new_car.increment_odometer(10)
#your_new_car.read_odometer()
```

```
2019 Audi A4
This car has 0 miles on it.
This car has -100 miles on it.
This car has -100 miles on it.
```

封装

封装（encapsulation），即隐藏对象的属性和实现细节，仅对外公开接口，控制在程序中属性的读和修改的访问级别；

将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体，也就是将数据与操作数据的源代码进行有机的结合，形成"类"，其中数据和函数都是类的成员。

在面向对象编程中，封装是将对象运行所需的资源封装在程序对象中——基本上，是方法和数据。对象是"公布其接口"。其他附加到这些接口上的对象不需要关心对象实现的方法即可使用这个对象。这个概念就是**"不要告诉我你是怎么做的，只要做就可以了。"**对象可以看作是一个自我包含的原子。对象接口包括了公共的方法和初始化数据。

私有变量的概念要从面向对象中的封装谈起。我们定义一个类/对象，本质上是在描述这个类/对象的状态和行为，其中状态由成员变量表示，行为由成员方法表示。在这些状态和行为中，有些我们需要公开作为外界调用的接口，有些我们需要隐藏作为内部具体的实现。

私有变量 private variable

不存在的

Python中没有禁止访问类中某一成员的保护机制。Java是一门非常工程化的语言，其哲学就是为工程服务，通过各种限制，尽可能减少程序员编写错误代码的机会。而Python则相反，其哲学为信任编码者，给程序员最少的限制，但**程序员必须为自己编写的代码负责**。

那是否意味着Python认为面向对象编程不需要封装呢？答案也是否定的。Python通过编码规范而不是语言机制来完成封装，具体而言，Python规定了对变量命名的公约，约定什么样的变量名表示变量是私有的，不应该被访问(而不是不能被访问)。

根据PEP 8(Python Enhancement Proposal #8)的建议：

https://www.python.org/dev/peps/pep-0008/#designing-for-inheritance

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public. ... We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work).

参考于： https://zhuanlan.zhihu.com/p/79280319

In [49]:
```
class Foo(object):
    def __init__(self):
        self.__baz = 42
    def foo(self):
        print(self.__baz)

class Bar(Foo):
    def __init__(self):
        super(Bar, self).__init__()
        self.__baz = 21
    def bar(self):
        print(self.__baz)

x = Bar()
x.foo()
x.bar()
print(x.__dict__)
{'_Bar__baz': 21, '_Foo__baz': 42}
```

Out[49]:
```
42
21
{'_Foo__baz': 42, '_Bar__baz': 21}
{'_Bar__baz': 21, '_Foo__baz': 42}
```

9.3 继承

编写类时，并非总是要从空白开始。 如果要编写的类是另一个现成类的特殊版本，可使用继承**(inheritance**）。

一个类继承另一个类时，将自动获得另一个类的所有属性和方法。 原有的类称为父类（parent class)，而新类称为子类(child class)。 子类继承了父类的所有属性和方法，同时还可以定义自己的属性和方法。

9.3.1 子类的方法__init__()

在既有类的基础上编写新类时，通常要调用父类的方法__init**()** 。 **这将初始化在父类__init**()方法中定义的所有属性，从而让子类包含这些属性。

例如，下面来模拟电动汽车。电动汽车是一种特殊的汽车，因此可在前面创建的Car 类的基础上创建新类ElectricCar。 这样就只需为电动汽车特有的属性和行为编写代码。

下面来创建ElectricCar 类的一个简单版本，它具备Car 类的所有 功能：

In [2]:
```
class Car:
    """一次模拟汽车的简单尝试。"""
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0
        self.gas = 10

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

    def fill_gas_tank(self, amount):
        self.gas += amount
        print(f'fill {amount}L gas, now has {self.gas}L gas ')
```

In [37]:
```
class ElectricCar(Car):
    """电动汽车的独特之处。"""

    def __init__(self, make, model, year):
        """初始化父类的属性。"""
        super().__init__(make, model, year)

my_byd = ElectricCar('BYD', 'Song', 2022)
print(my_byd.get_descriptive_name())


print(my_byd.make)
```

```
2022 Byd Song
BYD
```

首先是Car类的代码。 随后定义了子类ElectricCar。

定义子类时，必须在圆括号内指定父类的名称。 方法__init()接受创建Car实例所需的信息。 super()是一个特殊函数，让你能够调用父类的方法。 这行代码让Python调用Car类的方法__init()，让ElectricCar实例包含这个方法中定义的所有属性。 父类也称为超类（superclass），名称super由此而来。 为测试继承能够正确地发挥作用，我们尝试创建一辆电动汽车，但提供的信息与创建普通汽车时相同。 创建ElectricCar类的一个实例，并将其赋给变量my*byd*。*这行代码调用ElectricCar 类中定义的方法*_init() ， 后者让Python调用父类Car中定义的方法__init() 。我们提供了实参'byd'、'Song' 和2022。

除方法__init___() 外，电动汽车没有其他特有的属性和方法。 当前，我们只想确认电动汽车具备普通汽车的行为： 2022_BYD SONG_ElectricCar实例的行为与Car实例一样，现在可以开始定义电动 汽车特有的属性和方法了。

9.3.2 给子类定义属性和方法

让一个类继承另一个类后，就可以添加区分子类和父类所需的新属性和新方法了。

下面来添加一个电动汽车特有的属性（电瓶），以及一个描述该属性的方法。 我们将存储电瓶容量，并编写一个打印电瓶描述的方法：

In [5]:
```
class ElectricCar(Car):
    """电动汽车的独特之处。"""

    def __init__(self, make, model, year):
        """
        初始化父类的属性。
        再初始化电动汽车特有的属性。
        """
        super().__init__(make, model, year)
        self.battery_size = 75

    def describe_battery(self):
        """打印一条描述电瓶容量的消息。"""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_byd = ElectricCar('BYD', 'Song', 2022)
print(my_byd.make)
print(my_byd.gas)
my_byd.describe_battery()
```

```
BYD
10
This car has a 75-kWh battery.
```

有问题的父类的方法

In [49]:
```
my_byd = ElectricCar('BYD', 'Song', 2022)
my_byd.describe_battery()
my_byd.fill_gas_tank(30)
```

```
This car has a 75-kWh battery.
fill 30L gas, now has 40L gas
```

9.3.3 重写父类的方法 (Override)

override可以翻译为覆盖，从字面就可以知道，它是覆盖了一个方法并且对其重写，以求达到不同的作用。

对我们来说最熟悉的覆盖就是对接口方法的实现，在接口中一般只是对方法进行了声明，而我们在实现时，就需要实现接口声明的所有方法。

除了这个典型的用法以外，我们在继承中也可能会在子类覆盖父类中的方法。

In [6]:
```
class ElectricCar(Car):
    """电动汽车的独特之处。"""

    def __init__(self, make, model, year):
        """
        初始化父类的属性。
        再初始化电动汽车特有的属性。
        """
        super().__init__(make, model, year)
        self.battery_size = 75

    def describe_battery(self):
        """打印一条描述电瓶容量的消息。"""
        print(f"This car has a {self.battery_size}-kWh battery.")
```

```
    def fill_gas_tank(self, amount):
        """电动汽车没有油箱。"""
        print("I am an Electric Car, doesn't have a gas tank!")

my_byd = ElectricCar('BYD', 'Song', 2022)
print(my_byd.get_descriptive_name())
my_byd.describe_battery()
my_byd.fill_gas_tank(30)
```

```
2022 Byd Song
This car has a 75-kWh battery.
I am an Electric Car, doesn't have a gas tank!
```

现在，如果有人对电动汽车调用方法fill_gas_tank()，Python将忽略Car类中的方法fill_gas_tank()，转而运行上述代码。 使用继承时，可让子类保留从父类那里继承而来的精华，并剔除不需要的糟粕。

9.3.4 将实例用作属性

使用代码模拟实物时，你可能会发现自己给类添加的细节越来越多：
属性和方法清单以及文件都越来越长。 在这种情况下，可能需要将类的一部分提取出来，作为一个独立的类。 可以将大型类拆分成多个协同工作的小类。

例如，不断给ElectricCar 类添加细节时，我们可能发现其中包含很多专门针对汽车电瓶的属性和方法。 在这种情况下，可将这些属性和方法提取出来，放到一个名为Battery的类中，并将一个Battery实例作为ElectricCar类的属性：

In [53]:
```
class Battery:
    """一次模拟电动汽车电瓶的简单尝试。"""

    def __init__(self, battery_size=75):
        """初始化电瓶的属性。"""
        self.battery_size = battery_size

    def describe_battery(self):
        """打印一条描述电瓶容量的消息。"""
        print(f"This car has a {self.battery_size}-kWh battery.")

class ElectricCar(Car):
    """电动汽车的独特之处。"""

    def __init__(self, make, model, year):
        """
        初始化父类的属性。
        再初始化电动汽车特有的属性。
        """
        super().__init__(make, model, year)
        self.battery = Battery()

my_byd = ElectricCar('byd', 'Song', 2022)
print(my_byd.get_descriptive_name())
my_byd.battery.describe_battery()
```

```
2022 Byd Song
This car has a 75-kWh battery.
```

In [8]:
```
class Battery:
    """一次模拟电动汽车电瓶的简单尝试。"""

    def __init__(self, battery_size=75):
        """初始化电瓶的属性。"""
        self.battery_size = battery_size
```

```
    def describe_battery(self):
        """打印一条描述电瓶容量的消息。"""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """打印一条消息，指出电瓶的续航里程。"""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315
        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    """电动汽车的独特之处。"""

    def __init__(self, make, model, year):
        """
        初始化父类的属性。
        再初始化电动汽车特有的属性。
        """
        super().__init__(make, model, year)
        self.battery = Battery()

my_byd = ElectricCar('byd', 'Song', 2022)
print(my_byd.get_descriptive_name())
my_byd.battery.describe_battery()
my_byd.battery.get_range()
```

```
2022 Byd Song
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

9.3.5 模拟实物

模拟较复杂的物件（如电动汽车）时，需要解决一些有趣的问题。 续航里程是电瓶的属性还是汽车的属性呢？ 如果只描述一辆汽车，将方法get_range()放在Battery类中也许是合适的，但如果要描述一家汽车制造商的整个产品线，也许应该将方法get_range() 移到ElectricCar类中。

在这种情况下，get_range()依然根据电瓶容量来确定续航里程，但报告的是一款汽车的续航里程。 也可以这样做：
仍将方法get_range()留在Battery类中，但向它传递一个参数，如car_model。 在这种情况下，方法get_range()将根据电瓶容量和汽车型号报告续航里程。 这让你进入了程序员的另一个境界：解决上述问题时，从较高的逻辑层面（而不是语法层面）考虑； 考虑的不是Python，而是如何使用代码来表示实物。 达到这种境界后，你会经常发现，对现实世界的建模方法没有对错之分。 有些方法的效率更高，但要找出效率最高的表示法，需要经过一定的实践。 只要代码像你希望的那样运行，就说明你做得很好！ 即便发现自己不得不多次尝试使用不同的方法来重写类，也不必气馁。 要编写出高效、准确的代码，都得经过这样的过程。

9.4 导入类

见9-3

In [ ]:
```
# The end
```

In [1]:
```
class A:

    def __init__(self):
        self.data = 'A'
```

```
    def show_info(self):
        print(self.data)

class B:
    def __init__(self):
        self.data = 'B'

    def show_info(self):
        print(self.data)
```

In [8]:
```
class C(B, A):

    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        pass
```

In [9]:
```
c_object = C()

c_object.show_info()
```

```
B
```

共有属性 VS 私有属性

In [39]:
```
class A:

    def __init__(self):
        self.__private = 10
        self.pub = 20

    def pub_show(self):
        print(f'A public show, {self.__private},{self.pub}')
        self.__private_show()

    def __private_show(self):
        print(f'A private show, {self.__private},{self.pub}')
```

In [45]:
```
a = A()

a.pub_show()
#a.__private_show()

print(a._A__private)
a._A__private_show()
```

```
A public show, 10,20
A private show, 10,20
10
A private show, 10,20
```

In [42]:
```
dir(a)
```

Out[42]:
```
['_A__private',
 '_A__private_show',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
```

```
'__getattribute__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__le__',
'__lt__',
'__module__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'pub',
'pub_show']
```

In [ ]: