武漢大學
WUHAN UNIVERSITY

数据结构与算法

Python 面向对象编程

张晓平

武汉大学数学与统计学院

Table of contents

1. 类和实例

2. 访问限制

3. 继承与多态

面向对象编程——Object Oriented Programming，简称 OOP，是一种程序设计思想。OOP 把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

在 Python 中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

以下举例来说明面向过程与面向对象在程序流程上的不同之处。

假设要处理学生的成绩，为表示一个学生的成绩，面向过程的程序可用一个 dict 来表示：

```
std1 = {'name': 'Michael', 'score': 98}
std2 = {'name': 'Bob', 'score': 81}
```

假设要处理学生的成绩，为表示一个学生的成绩，面向过程的程序可用一个 dict 来表示：

```
std1 = {'name': 'Michael', 'score': 98}
std2 = {'name': 'Bob', 'score': 81}
```

而处理学生成绩可通过函数来实现，比如打印学生的成绩：

```
def print_score(std):
    print(f"{std['name']}: {std['score']}")
print_score(std1)
print_score(std2)
```

```
Michael: 98
Bob: 81
```

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是 Student 这种数据类型应该被视为一个对象，这个对象拥有name和score这两个属性（Property）。

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是 Student 这种数据类型应该被视为一个对象，这个对象拥有name和score这两个属性（Property）。

如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后给对象发一个print_score消息，让对象自己把自己的数据打印出来。

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是 Student 这种数据类型应该被视为一个对象，这个对象拥有name和score这两个属性（Property）。

如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后给对象发一个print_score消息，让对象自己把自己的数据打印出来。

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def print_score(self):
        print(f'{self.name}: {self.score}')
```

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是 Student 这种数据类型应该被视为一个对象，这个对象拥有name和score这两个属性（Property）。

如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后给对象发一个print_score消息，让对象自己把自己的数据打印出来。

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def print_score(self):
        print(f'{self.name}: {self.score}')
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。

类的使用

```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

```
Bart Simpson: 59
Lisa Simpson: 87
```

面向对象的设计思想来源于自然界，因为在自然界中，类 (class) 和实例 (instance) 的概念非常自然。class 是一种抽象概念，比如我们定义的`Student`类，是指学生这个概念，而实例 (instance) 则是一个个具体的`Student`，比如，`bart`和`lisa`是两个具体的`Student`。所以，面向对象的设计思想是抽象出 class，根据 class 创建 instance。

面向对象的设计思想来源于自然界，因为在自然界中，类 (class) 和实例 (instance) 的概念非常自然。class 是一种抽象概念，比如我们定义的`Student`类，是指学生这个概念，而实例 (instance) 则是一个个具体的`Student`，比如，`bart`和`lisa`是两个具体的`Student`。所以，面向对象的设计思想是抽象出 class，根据 class 创建 instance。

面向对象的抽象程度高于函数，因为一个 Class 既包含数据，又包含操作数据的方法。

类和实例

类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，比如 Student 类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记**类是抽象的模板**，比如 Student 类，而**实例是根据类创建出来的一个个具体的“对象”**，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以 Student 类为例，在 Python 中，使用关键字`class`定义类：

```
class Student(object):
    pass
```

`class`后紧跟类名，即`Student`，再紧跟`(object)`，表示该类是从哪个类继承下来的。

类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记**类是抽象的模板**，比如 Student 类，而**实例是根据类创建出来的一个个具体的“对象”**，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以 Student 类为例，在 Python 中，使用关键字`class`定义类：

```
class Student(object):
    pass
```

`class`后紧跟类名，即`Student`，再紧跟`(object)`，表示该类是从哪个类继承下来的。

注

- 类名通常以大写字母开头
- 如果没有合适的继承类，就使用`object`类，它是所有类最终都会继承的类。

类和实例

定义好了Student类，就可以根据它来创建出其实例.

例: 通过类名 +()来创建实例

```
bart = Student()
lisa = Student()
print(Student)
print(bart)
print(lisa)
```

```
<class '__main__.Student'>
<__main__.Student object at 0x7fb24f350630>
<__main__.Student object at 0x7fb24f3505f8>
```

类和实例

定义好了Student类，就可以根据它来创建出其实例.

例: 通过类名 +()来创建实例

```
bart = Student()
lisa = Student()
print(Student)
print(bart)
print(lisa)
```

```
<class '__main__.Student'>
<__main__.Student object at 0x7fb24f350630>
<__main__.Student object at 0x7fb24f3505f8>
```

由此可看出，Student本身是一个类，而bart指向的是一个Student的实例，后面的0x7fb24f350630是其内存地址，每个 object 的地址都不一样。

类和实例

可以自由地给一个实例变量绑定属性.

例: 给实例bart绑定一个name属性

```
bart.name = 'Bart Simpson'
print(bart.name)
```

```
Bart Simpson
```

类和实例

类起到模板的作用，可在创建实例时把我们认为一些必须绑定的属性强制填写进去。通过定义一个特殊的__init__方法，在创建实例时，把name, score等属性绑上去：

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
```

类和实例

类起到模板的作用，可在创建实例时把我们认为一些必须绑定的属性强制填写进去。通过定义一个特殊的__init__方法，在创建实例时，把name, score等属性绑上去：

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
```

注

- 特殊方法__init__前后分别有两根下划线。
- __init__()的第一个参数永远是 self，表示创建的实例本身。因此，在__init__()内部，就可把各种属性绑定到 self，因为 self 就指向创建的实例本身。

类和实例

有了__init__()，在创建实例时，就不能传入空的参数了，必须传入与__init__()相匹配的参数，但self不需要传，Python 解释器自己会把实例变量传进去：

```
bart = Student('Bart Simpson', 59)
print(bart.name)
print(bart.score)
```

```
Bart Simpson
59
```

类和实例

有了__init__()，在创建实例时，就不能传入空的参数了，必须传入与__init__()相匹配的参数，但self不需要传，Python 解释器自己会把实例变量传进去：

```
bart = Student('Bart Simpson', 59)
print(bart.name)
print(bart.score)
```

```
Bart Simpson
59
```

注

和普通函数相比，在类中定义的函数 (即方法) 只有一点不同，就是第一个参数永远是实例变量self，并且调用时不用传递该参数。除此之外，类的方法和普通函数没有什么区别，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

类和实例

数据封装

数据封装

面向对象编程的一个重要特点就是数据封装。

数据封装

面向对象编程的一个重要特点就是**数据封装**。

对于Student类，每个实例拥有各自的数据，如name和score.

例：可以通过函数来访问这些数据，如打印一个学生的成绩

```
def print_score(std):
    print(f"{std.name}: {std.score}")
print_score(bart)
```

```
Bart Simpson: 59
```

数据封装

既然Student的实例本身就拥有这些数据，要访问他们，就没有必要从外面的函数去访问，可以直接在Student类的内部定义访问数据的函数，这样就把“数据”给封装了。封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法.

例：通过类的方法打印学生的成绩

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def print_score(self):
        print(f"{self.name}: {self.score}")
bart = Student('Bart Simpson', 59)
bart.print_score()
```

```
Bart Simpson: 59
```

数据封装

从外部看Student类，只需在创建实例时给出name和score，而至于如何打印，会在Student类的内部来定义，这些数据和逻辑被“封装”起来了，调用很容易，但不用知道内部实现的细节。

数据封装

封装的另一个好处是可以给Student类增加新的方法，比如get_grade：

```
class Student(object):
    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
bart = Student('Bart Simpson', 59)
print(f'{bart.name}: {bart.get_grade()}')
```

```
Bart Simpson: C
```

访问限制

访问限制

在 Class 内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样就隐藏了内部的复杂逻辑。

但是，从Student类的定义来看，外部代码还是可以自由地修改一个实例的name，score属性：

```
bart = Student('Bart Simpson', 59)
print(bart.score)
bart.score = 99
print(bart.score)
```

```
59
99
```

访问限制

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线__。

注

在 Python 中，实例的变量名如果以__开头，就变成了一个私有变量 (private)，只有内部可以访问，外部不能访问。

访问限制

```
class Student(object):
    def __init__(self, name, score):
        self.__name = name
        self.__score = score
    def print_score(self):
        print(f"{self.__name}: {self.__score}")
bart = Student('Bart Simpson', 59)
print(bart.__name)
```

访问限制

```
class Student(object):
    def __init__(self, name, score):
        self.__name = name
        self.__score = score
    def print_score(self):
        print(f"{self.__name}: {self.__score}")
bart = Student('Bart Simpson', 59)
print(bart.__name)
```

```
Traceback (most recent call last):
  File "src/slide02/code/oop3.py", line 19, in <module>
    print(bart.__name)
AttributeError: 'Student' object has no attribute '__name'
```

访问限制

```
class Student(object):
    def __init__(self, name, score):
        self.__name = name
        self.__score = score
    def print_score(self):
        print(f"{self.__name}: {self.__score}")
bart = Student('Bart Simpson', 59)
print(bart.__name)
```

```
Traceback (most recent call last):
  File "src/slide02/code/oop3.py", line 19, in <module>
    print(bart.__name)
AttributeError: 'Student' object has no attribute '__name'
```

对于外部代码来说，没什么变动，但是已经无法从外部访问bart.__name和bart.__score。通过访问限制的保护，就确保了外部代码不能随意修改对象内部的状态，使得代码更加健壮。

访问限制

问

如何外部代码想获取name和score怎么办？

访问限制

问

如何外部代码想获取name和score怎么办？

可以给Student类增加get_name和get_score这样的方法：

```
class Student(object):
    def get_name(self):
        return self.__name
    def get_score(self):
        return self.__score
```

访问限制

问

如果外部代码想修改score怎么办？

访问限制

问

如果外部代码想修改score怎么办？

可以再给Student类增加set_score方法：

```
class Student(object):
    def set_name(self, score):
        self.__score = score
```

访问限制

问

直接通过bart.score=99不是也可以修改score吗？为什么非要大费周折取定义一个方法呢？

访问限制

问

直接通过bart.score=99不是也可以修改score吗？为什么非要大费周折取定义一个方法呢？

因为在方法中，可做参数检查，避免传入无效的参数：

```
class Student(object):
    def set_name(self, score):
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError('bad score')
```

访问限制

请注意在 Python 中`__xxx`, `_xxx`, `__xxx__`等变量的差别:

- 类似于`__xxx`的变量是 private 变量，外部代码不能直接访问。
- 类似于`_xxx`的变量允许外部代码访问，但按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但请把我视为 private 变量，不要随意访问”。
- 类似于`__xxx__`的变量是特殊变量，不是 private 变量，外部代码可直接访问。

访问限制

实例变量__xxx是不是一定不能从外部访问呢？其实也不是。不能直接访问__name是因为 Python 解释器对外把__name变量改成了_Student__name，所以，仍然可以通过_Student__name来访问__name：

```
bart = Student('Bart Simpson', 59)
print(bart._Student__name)
```

```
Bart Simpson
```

但是强烈建议你不要这么干，因为不同版本的 Python 解释器可能会把__name改成不同的变量名。

总而言之，Python 本身没有任何机制阻止你干坏事，一切全靠自觉。

继承与多态

继承与多态

在 OOP 程序设计中，当我们定义一个 class 的时候，可以从某个现有的 class 继承，新的 class 称为子类 (subclass)，而被继承的 class 称为基类、父类或超类 (base class、super class)。

继承与多态

例:

假设已经定义一个名为Animal的 class，其中有一个run()方法直接打印:

```
class Animal(object):
    def run(self):
        print('Animal is running ...')
```

继承与多态

例:

假设已经定义一个名为Animal的 class，其中有一个run()方法直接打印:

```
class Animal(object):
    def run(self):
        print('Animal is running ...')
```

当需要定义Dog和Cat类时，就可以直接从Animal类继承:

```
class Dog(Animal):
    pass
```

```
class Cat(Animal):
    pass
```

继承与多态

例:

假设已经定义一个名为Animal的 class，其中有一个run()方法直接打印:

```
class Animal(object):
    def run(self):
        print('Animal is running ...')
```

当需要定义Dog和Cat类时，就可以直接从Animal类继承:

```
class Dog(Animal):
    pass
```

```
class Cat(Animal):
    pass
```

对于Dog来说，Animal就是它的父类；对于Animal来说，Dog，Cat就是它的子类。

继承与多态

继承有什么好处呢？最大的好处就是子类获得了父类的全部功能。

由于Animial实现了run()，因此，Dog和Cat作为它的子类，什么事也没干，就自动拥有了run()：

```
dog = Dog()
dog.run()
cat = Cat()
cat.run()
```

```
Animal is running ...
Animal is running ...
```

继承与多态

也可以为子类增加一些方法

例:

```
class Dog(Animal):
    def eat(self):
        print('Eating meat ...')
dog = Dog()
dog.run()
dog.eat()
```

```
Animal is running ...
Eating meat ...
```

继承与多态

第二个好处是改进代码。

前面可以看到，无论是Dog还是Cat，调用run()时，显示的都是Animal is running...，符合逻辑的做法是分别显示Dog is running...和Cat is running....

继承与多态

```
class Dog(Animal):
    def run(self):
        print('Dog is running ...')

class Cat(Animal):
    def run(self):
        print('Cat is running ...')
```

```
Dog is running ...
Cat is running ...
```

继承与多态

```
class Dog(Animal):
    def run(self):
        print('Dog is running ...')

class Cat(Animal):
    def run(self):
        print('Cat is running ...')
```

```
Dog is running ...
Cat is running ...
```

当子类和父类都存在相同的run()时，子类的run()覆盖了父类的run()，运行代码时总是会调用子类的 run()。这样，就获得了继承的另一个好处：**多态**。

继承与多态

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个 class 的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和 Python 自带的数据类型，比如 str、list、dict 没什么两样：

```
a = list()
b = Animal()
c = Dog()
```

继承与多态

判断一个变量是否是某个类型可以用 isinstance()：

```
print(isinstance(a, list))
print(isinstance(b, Animal))
print(isinstance(c, Dog))
print(isinstance(c, Animal))
```

```
True
True
True
True
```

a，b，c分别是list，Animal，Dog类型，但c还是Animal类型.

继承与多态

在继承关系中，如果一个实例的数据类型是某个子类，则它的数据类型也可看做是其父类，但反过来不行：

```
print(isinstance(b, Dog))
```

```
False
```

继承与多态

要理解多态的好处，我们再看一个例子.

例: 编写一个函数接受一个Animal类型的变量

```
def run_twice(animal):
    animal.run()
    animal.run()
run_twice(Animal())
run_twice(Dog())
run_twice(Cat())
```

继承与多态

要理解多态的好处，我们再看一个例子.

例: 编写一个函数接受一个Animal类型的变量

```
def run_twice(animal):
    animal.run()
    animal.run()
run_twice(Animal())
run_twice(Dog())
run_twice(Cat())
```

```
Animal is running ...
Animal is running ...
Dog is running ...
Dog is running ...
Cat is running ...
Cat is running ...
```

多态的含义

对于一个变量，只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal，Dog还是Cat对象上，由运行时该对象的确切类型决定，这就是多态真正的威力.

多态的含义

对于一个变量，只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal，Dog还是Cat对象上，由运行时该对象的确切类型决定，这就是多态真正的威力.

调用方只管调用，不管细节. 当新增一种 Animal 的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。 这就是著名的“开闭”原则.

继承与多态

“开闭”原则

- 对扩展开放：允许新增`Animal`子类；
- 对修改封闭：不需要修改依赖`Animal`类型的`run_twice()`等函数。

继承与多态

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类 object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：