CSCE 110: Programming I

David Kebo Hougninou

Introduction to Object Oriented Programming
You have learned to use functions to organize code, and built-in types to organize data.

The next step is to learn **object-oriented programming**.

**Object-oriented programming** is a programming concept based on objects.

An object can contain:

• data in the form of **fields/attributes**
• code in the form of **procedures/methods**

e.g. A race car has

**Attributes**: horse power, brand, wheel size, weight, color
**Methods**: accelerate(), navigate(), turn()
Object-Oriented Programming: class

A class is a blueprint or description for an object.
e.g. If manufacturing a race car, a class would contain all the details about the brand, weight, engine size, horse-power etc.

class RaceCar:

The class keyword defines an new class called RaceCar.
Using the class RaceCar, we can build multiple race car objects.

Remember to add a docstring to describe the class!

Object-Oriented Programming: constructor

The constructor is a function that we call whenever a new object is instantiated.
The constructor initializes the attributes of the class.
The constructor is a special class function.
Special class functions begin with double underscore ‘__’.

The name of the constructor is the __init__() function.
__init__ is a reserved method in python classes.
Object-Oriented Programming: constructor

In the example below, the constructor for the class RaceCar initializes the instance attributes brand, gearbox, and horse_power.

```python
class RaceCar:
    """This is a class for a racing car""
    # class attribute
    vehicle = "Racing car"

    # instance attribute
    def __init__(self, brand, gearbox, horse_power):
        self.brand = brand
        self.gearbox = gearbox
        self.horse_power = horse_power
```

Object-Oriented Programming: object

An object or instance is an instantiation of a class.

car1 = RaceCar()
car1 is an object of class RaceCar.

We can create objects with different attributes using one class.

```python
# instantiate the RaceCar class to create an object
car1 = RaceCar("Ferrari", "Automatic", 700)
car2 = RaceCar("McLaren", "Manual", 640)
car3 = RaceCar("Aston Martin", "Manual", 592)
```
Object-Oriented Programming: self parameter

The `self` parameter allows referring to the current object in a method.

When defining an instance method, the first parameter of the method should always be the object itself: `self`. `self` represents the instance of the class. The `self` keyword can access the attributes and methods of the class.

```python
# instance attribute
def __init__(self, brand, gearbox, horse_power):
    self.brand = brand
    self.gearbox = gearbox
    self.horse_power = horse_power
```

Object-Oriented Programming: object

```python
class RaceCar:
    """This is a class for a racing car"""

    # class attribute
    vehicle = "Racing car"

    # instance attribute
    def __init__(self, brand, gearbox, horse_power):
        self.brand = brand
        self.gearbox = gearbox
        self.horse_power = horse_power

    # instantiate the RaceCar class to create an object
    car1 = RaceCar("Ferrari", "Automatic", 700)
    car2 = RaceCar("McLaren", "Manual", 640)
    car3 = RaceCar("Aston Martin", "Manual", 592)

    # access the class attributes
    print("The first vehicle is a {car1.__class__.__vehicle}")
    print("The second vehicle is a {car2.__class__.__vehicle}\n")

    # access the instance attributes
    print("Car1 specs:\nBrand: {car1.brand}\nTrans.: {car1.gearbox}\nHP: {car1.horse_power}\n")
    print("Car2 specs:\nBrand: {car2.brand}\nTrans.: {car2.gearbox}\nHP: {car2.horse_power}\n")
```
Object-Oriented Programming: principles

In Python, Object-oriented programming follows some principles:

<table>
<thead>
<tr>
<th>Inheritance</th>
<th>Enables new objects to take on the properties of existing objects.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation</td>
<td>Hides the private properties of a class from other objects.</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>Enables an object to take on many forms.</td>
</tr>
</tbody>
</table>

Object-Oriented Programming: inheritance

**Inheritance** enables a new class to use using details of an existing class without modifying it.

- The new class is a child class.
- The existing class is a parent class.

The child class **inherits** features from the parent class, adding new features to it.

Note: Python provides a function `issubclass()` that directly tells us if a class is subclass of another class.
Object-Oriented Programming: inheritance

```python
class vehicle:
    """This is a class for a vehicle""
    def __init__(self):
        print("This is a new vehicle")
    def start(self):
        print("Engine ON")

class RaceCar(vehicle):
    """This is a class for a racing car""
    # class attribute
    vehicle = "Racing car"

    # instance attribute
    def __init__(self, brand, gearbox, horse_power):
        self.brand = brand
        self.gearbox = gearbox
        self.horse_power = horse_power

>>> car1.start()
Engine ON
>>> car1.stop()
Engine Off
>>> isinstance(RaceCar, vehicle)
True
```

The child class `RaceCar()` inherits the functions of the parent class `vehicle()`.

Object-Oriented Programming: encapsulation

We can restrict access to methods and variables.

Encapsulation hides the private properties of a class from other objects.

In Python, we denote private attribute using underscore as prefix ‘_’ or double ‘__’.
Object-Oriented Programming: encapsulation

It is not possible to change to private attribute using the assignment on line 17. We use a setter function instead.

```python
class RaceCar:
    """This is a class for a racing car""
    def __init__(self):
        self.__gearbox = "manual"
    def check_gearbox(self):
        print(f"Transmission type: {self.__gearbox}\n")
    def set_gearbox(self, gear):
        self.__gearbox = gear

>>> car = RaceCar()
>>> car.check_gearbox()
# Attempt to change the private attribute gearbox
>>> car.__gearbox = "automatic"
>>> car.check_gearbox()
# Use a setter function to change the private attribute gearbox
>>> car.set_gearbox("automatic")
>>> car.check_gearbox()
```

Object-Oriented Programming: polymorphism

- Polymorphism is the ability of an object to take on many forms.
- Polymorphism allows to define methods in the child class that have the same name as the methods in the parent class.
- When several classes have the same method names, but different implementations for these methods, the classes are polymorphic.
- A function can evaluate these polymorphic methods without knowing which classes are invoked.
Object-Oriented Programming: polymorphism

We defined two classes Racecar and Truck.

Both classes have a common method `accelerate()`

The function of each method is different. The name of the common interface is `accelerate()`

```python
class RaceCar:
    def accelerate(self):
        print("The top speed is 210 MPH")

class Truck:
    def accelerate(self):
        print("The top speed is 80 MPH")

# define a common interface
def accelerate(vehicle):
    vehicle.accelerate()

# instantiate two different objects
vehicle1 = RaceCar()
vehicle2 = Truck()

# use the common interface
accelerate(vehicle1)
accelerate(vehicle2)
```

```
$ python racecar_polymorphism.py
The top speed is 210 MPH
The top speed is 80 MPH
```

Hougninou    CSCE 110: Programming I