Introduction to Programming in Python Part 2

By William Chan

Objectives

- Learn to write classes to enable object oriented programming
- Learn how to incorporate classes and methods in classes to store data and behavior
- Learning inheritance and how classes inherit from other classes to obtain similar behavior and properties

Creating Objects in Python

Objects are created with classes in Python. It can be thought of as writing a “class” to create objects from and the objects created would be of that “class”.

For instance a `Vehicle` class can create `vehicle` objects. Think of it as a `Vehicle` class of objects.

A simple class

A simple class can be written using the following template. The object in the parentheses represents the class that is being inherited in which case all classes will implicitly inherit from the basic `object` class. If it doesn't inherit anything it can be left out. The `pass` keyword allows the defined class to have an empty body.

```
class NameOfClass(object):
    pass


// or


class NameOfClass:
    pass
```

Creating an object of the simple class

Creating an object of the simple class is like calling a function from the name of the function like the following of a previously defined class.

```
class SomeClass:          # defines a class with the name of `SomeClass`
    pass

some_obj = SomeClass()    # creates an object of SomeClass
                          # and assigns it to some_obj
```

Writing a simple class with a constructor

The constructor is the `__init__` method inside of SimpleClass. The `__init__` method can be thought of as a magic method that gets called when attempting to create an object using that class. The term of such a method is “constructor”.

The self in parentheses in the `__init__` method represents the receiver which is also explicitly the object that the SimpleClass will create at that point of time. This will become more apparent as the class creates an object with certain properties.

```
class SimpleClass:
    def __init__(self):
        print('in the constructor')


some_obj = SimpleClass()  # prints 'in the constructor' and
                          # creates the object from SimpleClass
```

What are instance variables?

Instance variables are variables that belong instances of a defined class. Instance variables are defined inside the constructor using the first parameter as the “receiver” and the instance it is trying to create.

```
class SimpleClass:
    def __init__(self):
        print('calling the constructor')
        self.instance_variable = 'i am an instance variable'

some_obj = SimpleClass()                # prints calling the 'constructor'
print(some_obj.instance_variable)       # prints 'i am an instance variable'
```

What are class variables?

Class variables are variables that belong and can be accessed from a class. A simple example shows how a class variable may be defined.

```
class SimpleClass:
    class_variable = "i am a class variable"

# accessing the class variable using “dot” notation
print(SimpleClass.class_variable)      # prints “i am a class variable”
```

What is a method?

A method is a function that belongs to a class or an object. There are multiple different types of methods. They are the following:

- instance methods - methods that belong to objects and can access instance variables (in Python they can also access class variables)
- class methods - methods that belong to classes and can access class variables
- static methods - methods that are defined in classes but do not necessarily belong to the class and can not access neither class or instance variables

Defining instance methods

Instance methods are defined inside of the class. The first parameter of the method is self again which is represents the instance that it is called on. In `some_obj.instance_method()`, the `self` represents `some_obj` while the self in `another_obj.instance_method()` represents `another_obj`.

```
class SimpleClass:
     def instance_method(self):
          print('calling the instance method')

some_obj = SimpleClass()
some_obj.instance_method()    # prints ‘calling the instance method’

another_obj = SimpleClass()
another_obj.instance_method() # prints ‘calling the instance method’
```

Defining class methods

In order to differentiate a class method from an instance method, it uses `cls` as the first parameter of the method to be the receiver. This is by convention. It can access class variables via the receiver variable, `cls` and is called via the class and not the instance. `@classmethod` is the decorator that makes the method a class method instance of an instance method.

```
class SimpleClass:
     class_variable = 'i am a class variable'
     @classmethod
     def class_method(cls):
          print('calling the class method')
          print(cls.class_variable)


SimpleClass.class_method()     # prints ‘calling the class method’
                               # prints ‘i am a class variable’
```

Defining static methods

Static methods are defined as a method without the self keyword since it doesn’t belong to an instance or a class. It is called the same way as a class method as the method seemingly attaches to the class but has no access to anything within the class.

```
class SimpleClass:
    @staticmethod
    def static_method():
        print('calling the static method')


SimpleClass.static_method() # prints ‘calling the static method’
```

Inheritance

Python allows classes to have inheritance where another class can inherit behavior and properties of another class.

```
# defining the classes and inheritance
class Vehicle:
      def __init__(self):
            self.speed = 0
      def accelerate(self, miles):
            print('speeding up ' + miles + ' miles')
            self.speed += miles
      def decelerate(self, miles):
            print('slowing down ' + miles + ' miles')
            self.speed -= miles
# "Car" inherits from "Vehicle"
def Car(Vehicle):
      pass
```

```
# the calling code
toyota = Car()
toyota.accelerate(10)
print(toyota.speed)
```

Overriding methods and properties in parent classes

A child class overrides the method of the parent class if they have the same name. If the parent and child class have different method signatures (ie. different number of parameters), the method signature in the child class will take precedence.

```
class Vehicle:
     def accelerate(self):
          print('speeding up')

class Boat(Vehicle):
     # overrides Vehicle's speed_up method because it has the same definition
     def accelerate(self):
          print('the boat is accelerating')

# the calling code
boat = Boat()
boat.accelerate()
```

Multiple Inheritance

A class can inherit from multiple classes besides itself. If a class were to inherit multiple classes with similar properties and methods, it would use the methods and properties of the leftmost class.

```
class Vehicle:
      def accelerate(self):
            print('accelerating')
      def turn(self, direction):
            print('turning ' + direction)
class Ship:
      def turn(self, direction):
            print('turning ' + direction + ' swiftly')

class BattleShip(Ship, Vehicle):
      def launch_missile(self):
            print('launching missile')
```

```
battleship = BattleShip()

# prints 'launching missile'
battleship.launch_missile()

# prints 'accelerating'
battleship.speed_up()

# prints 'turning left swiftly'
battleship.turn('left')
```

An object oriented exercise

Write a simple rock-paper-scissors game between the computer and a player. This will require taking an input from the user and comparing it to a randomly generated value for the computer. It will print out ‘you win’, ‘draw’, or ‘you lost’ under the appropriate scenario.

Hint: Think about the objects in the program. Typically the nouns are the objects and the verbs are the behavior of the object. What are the nouns and the verbs and how do they map to each other?

Another Hint: `raw_input('Input your value here: ')` allows the user to enter in the value in the terminal. The following gets the value that is inputted by the user. It will stop the program until a value is actually entered into the terminal and return is hit.

```
entered_value = raw_input('input your value here: ')
```

Summary

- Objects are created with classes
- Objects are instantiated via the constructor, `def __init__(self):` method of the class
- Methods that belong to objects are defined in classes and are called “instance methods”
- Properties that belong to objects and defined in the constructor of classes are called “instance variables”
- Class variables are defined inside the class but outside of any method
- Class methods are defined with the `@classmethod` decorator placed immediately above the method that should be a class method
- Classes can inherit properties and behavior from one or many classes
- In the case where a class inherits multiple classes, the classes on the left takes precedence over the classes to the right if it shall have the same properties and methods

More Advanced Python Concepts

- list comprehensions
- public methods
- private methods
- encapsulation and its definition
- imports
- continue statements inside flow control
- break statements inside flow control
- more depth on object oriented programming theory
- args as arbitrary number of arguments
- kwargs as keyword arguments

Some common frameworks and libraries to continue your learning

- Django - a web development framework that includes the batteries to roll a quick python web application with a database
- Flask - a microframework for web development that is primarily a routing layer differing from Django as it requires more manual setup
- NumPy and SciPy - libraries for numerical and scientific computing for more advanced mathematics like linear algebra, etc
- Matplotlib - a library for building data visualizations
- pandas - a data analysis library that allows data visualizations and relationships of data to be analyzed