Object Oriented Programming

Intro –

Object oriented programming is a very effective efficient way of writing code. It involves writing classes that represent real world things and situations, and you create object based on these classes. When you write a class, you write a general behaviour that a whole category of objects can have.

When you create individual objects from the class, each object is automatically equipped with the general behaviour; you can then give each object whatever unique traits you desire. You'll be amazed how well real-world situations can be modelled with object-oriented programming.

Making an object from a class is called ***instantiation***, and you work with ***instances*** of a class. You'll specify the kind of information that can be stored in instances, and you'll define actions that can be taken with these instances. You'll also write classes that extend the functionality of existing classes, so similar classes can share code efficiently. You'll store your classes in modules and import classes written by other programmers into your own program files.

Creating and using a class –

You can model almost anything using classes. Let's start by writing a simple class, Dog, that represents a dog—not one dog in particular, but any dog. What do we know about most pet dogs? Well, they all have a name and age. We also know that most dogs sit and roll over. Those two pieces of information (name and age) and those two behaviours (sit and roll over) will go in our Dog class because they're common to most dogs.

Creating the Dog Class

Each instance created from the `Dog` class will store a `name` and an `age`, and we'll give each dog the ability to `sit()` and `roll_over()`:

dog.py

```
❶ class Dog():
❷     """A simple attempt to model a dog."""

❸     def __init__(self, name, age):
          """Initialize name and age attributes."""
❹         self.name = name
          self.age = age

❺     def sit(self):
          """Simulate a dog sitting in response to a command."""
          print(self.name.title() + " is now sitting.")

      def roll_over(self):
          """Simulate rolling over in response to a command."""
          print(self.name.title() + " rolled over!")
```

The __init__() Method –

A function that's part of a class is a ***method***. The __init__() method at (3) is a special method Python runs automatically whenever we create a new instance based on the Dog class. This method has two leading underscores and two trailing underscores, a convention that helps prevent Python's default method names from conflicting with your method names. We define the __init__() method to have three parameters: self, name, and age.
The ***self*** parameter is required in the method definition, and it must come first before the other parameters. It must be included in the definition because when Python calls this __init__() method later (to create an instance of Dog), the method call will automatically pass the self argument. Every method call associated with a class automatically passes self, which is a reference to the instance itself; it gives the individual instance access to the attributes and methods in the class. When we make an instance of Dog, Python will call the __init__() method from the Dog class. We'll pass Dog() a name and an age as arguments; self is passed automatically, so we don't need to pass it. Whenever we want to make an instance from the Dog class, we'll provide values for only the last two parameters, name and age.

The two variables defined at (4) each have the prefix self. Any variable prefixed with self is available to every method in the class, and we'll also be able to access these variables through any instance created from the class. *self.name* =

name takes the value stored in the parameter name and stores it in the variable name, which is then attached to the instance being created. The same process happens with *self.age* = age. Variables that are accessible through instances like this are called ***attributes***.
The Dog class has two other methods defined: sit() and roll_over() (5). Because these methods don't need additional information like a name or age, we just define them to have one parameter, self. The instances we create later will have access to these methods.

Making an Instance from a Class

Think of a class as a set of instructions for how to make an instance. The class `Dog` is a set of instructions that tells Python how to make individual instances representing specific dogs.

Let's make an instance representing a specific dog:

```
class Dog():
    --snip--

❶ my_dog = Dog('willie', 6)

❷ print("My dog's name is " + my_dog.name.title() + ".")
❸ print("My dog is " + str(my_dog.age) + " years old.")
```

The Dog class we're using here is the one we just wrote in the previous example. At (1) we tell Python to create a dog whose name is 'willie' and whose age is 6. When Python reads this line, it calls the __init__() method in Dog with the arguments 'willie' and 6. The __init__() method creates an instance representing this particular dog and sets the name and age attributes using the values we provided. The __init__() method has no explicit return statement, but Python automatically returns an instance representing this dog. We store that instance in the variable my_dog. The naming convention is helpful here: we can usually assume that a capitalized name like Dog refers to a class, and a lowercase name like my_dog refers to a single instance created from a class.

Accessing Attributes

To access the attributes of an instance, you use dot notation. At ❷ we access the value of `my_dog`'s attribute `name` by writing:

```
my_dog.name
```

Dot notation is used often in Python. This syntax demonstrates how Python finds an attribute's value. Here Python looks at the instance my_dog and then finds the attribute name associated with my_dog. This is the same attribute referred to as self.name in the class Dog.

The output is a summary of what we know about `my_dog`:

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

Calling Methods

After we create an instance from the class `Dog`, we can use dot notation to call any method defined in `Dog`. Let's make our dog sit and roll over:

```
class Dog():
    --snip--

my_dog = Dog('willie', 6)
my_dog.sit()
my_dog.roll_over()
```

To call a method, give the name of the instance (in this case, my_dog) and the method you want to call, separated by a dot. When Python reads my_dog.sit(), it looks for the method sit() in the class Dog and runs that code. Python interprets the line my_dog.roll_over() in the same way. Now Willie does what we tell him to:

```
Willie is now sitting.
Willie rolled over!
```

TRY IT YOURSELF

9-1. Restaurant: Make a class called `Restaurant`. The `__init__()` method for `Restaurant` should store two attributes: a `restaurant_name` and a `cuisine_type`. Make a method called `describe_restaurant()` that prints these two pieces of information, and a method called `open_restaurant()` that prints a message indicating that the restaurant is open.

Make an instance called `restaurant` from your class. Print the two attributes individually, and then call both methods.

9-2. Three Restaurants: Start with your class from Exercise 9-1. Create three different instances from the class, and call `describe_restaurant()` for each instance.

9-3. Users: Make a class called `User`. Create two attributes called `first_name` and `last_name`, and then create several other attributes that are typically stored in a user profile. Make a method called `describe_user()` that prints a summary of the user's information. Make another method called `greet_user()` that prints a personalized greeting to the user.

Create several instances representing different users, and call both methods for each user.