Python Object Oriented Programming

Introduction to OOPs in Python

Python is a multi-paradigm programming language. Meaning, it supports different programming approach.
One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
An object has two characteristics:

- attributes
- behavior

Let's take an example:
Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

Class

A class is a blueprint for the object.

We can think of class as an sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```

Here, we use `class` keyword to define an empty class `Parrot`. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, `obj` is object of class `Parrot`.

Suppose we have details of parrot. Now, we are going to show how to build the class and objects of parrot.

Example 1: Creating Class and Object in Python

```
class Parrot:
    # class attribute
    species = "bird"
    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age
# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)
# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))
# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

When we run the program, the output will be:

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we create a class with name `Parrot`. Then, we define attributes. The attributes are a characteristic of an object.
Then, we create instances of the `Parrot` class. Here, `blu` and `woo` are references (value) to our new objects.
Then, we access the class attribute using `__class __.species`. Class attributes are same for all instances of a class. Similarly, we access the instance attributes using `blu.name` and `blu.age`. However, instance attributes are different for every instance of a class.

Instance Attributes

All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph). Use the __init__() method to initialize (e.g., specify) an object's initial attributes by giving them their default value (or state). This method must have at least one argument as well as the self variable, which refers to the object itself (e.g., Dog).

```
class Dog:

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

In the case of our Dog() class, each dog has a specific name and age, which is obviously important to know for when you start actually creating different dogs. Remember: the class is just for defining the Dog, not actually creating *instances* of individual dogs with specific names and ages; we'll get to that shortly.

Similarly, the self variable is also an instance of the class. Since instances of a class have varying values we could state Dog.name = name rather than self.name = name. But since not all dogs share the same name, we need to be able to assign different values to different instances. Hence the need for the special self variable, which will help to keep track of individual instances of each class.

NOTE: You will never have to call the __init__() method; it gets called automatically when you create a new 'Dog' instance.

## Class Attributes

While instance attributes are specific to each object, class attributes are the same for all instances—which in this case is *all* dogs.

```
class Dog:

    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

So while each dog has a unique name and age, every dog will be a mammal.

D=Dog("Roky","5")

The `self`

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object *itself*, and by convention, it is given the name `self`.

Although, you can give any name for this parameter, it is *strongly recommended* that you use the name `self` - any other name is definitely frowned upon. There are many advantages to using a standard name - any reader of your program will immediately recognize it and even specialized IDEs (Integrated Development Environments) can help you if you use `self`.

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

The `__init__` method

There are many method names which have special significance in Python classes. We will see the significance of the `__init__` method now.

The __init__ method is similar to constructors in C++ and Java. The `__init__` method is run as soon as an object of a class is instantiated (i.e. created). The method is useful to do any *initialization* (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores both at the beginning and at the end of the name.

Example

```python
class Person:
    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Swaroop')
p.say_hi()
# The previous 2 lines can also be written as
# Person('Swaroop').say_hi()
```

Output:

```
$ python oop_init.py
Hello, my name is Swaroop
```

Example 2 : Creating Methods in Python

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)
    def dance(self):
        return "{} is now dancing".format(self.name)
# instantiate the object
blu = Parrot("Blu", 10)
# call our instance methods
print(blu.sing("'Happily'"))
print(blu.dance())
```

When we run program, the output will be:

```
Blu sings 'Happily'
Blu is now dancing
```

In the above program, we define two methods i.e `sing()` and `dance()`. These are called instance method because they are called on an instance object i.e `blu`.

Basic Principles of OOP

In Python, the concept of OOP follows some basic principles:

Inheritance	A process of using details from a new class without modifying existing class.
Encapsulation	Hiding the private details of a class from other objects.
Polymorphism	A concept of using common operation in different ways for different data input.

Inheritance

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example 3: Use of Inheritance in Python

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")
    def whoisThis(self):
        print("Bird")
    def swim(self):
        print("Swim faster")
# child class
class Penguin(Bird):
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")
    def whoisThis(self):
        print("Penguin")
    def run(self):
        print("Run faster")
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

When we run this program, the output will be:

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

In the above program, we created two classes i.e. `Bird` (parent class) and `Penguin` (child class). The child class inherits the functions of parent class. We can see this from `swim()` method. Again, the child class modified the behavior of parent class. We can see this from whoisThis() method. Furthermore, we extend the functions of parent class, by creating a new `run()` method.
Additionally, we use `super()` function before `__init__()` method. This is because we want to pull the content of `__init__()` method from the parent class into the child class.

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single “ _ “ or double “ __“.

Example 4: Data Encapsulation in Python

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price
c = Computer()
c.sell()
# change the price
c.__maxprice = 1000
c.sell()
# using setter function
c.setMaxPrice(1000)
c.sell()
```

When we run this program, the output will be:

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

In the above program, we defined a class `Computer`. We use `__init__()` method to store the maximum selling price of computer. We tried to modify the price. However, we can’t change it because Python treats the `__maxprice` as private attributes. To change the value, we used a setter function i.e `setMaxPrice()` which takes price as parameter.

Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

Example 5: Using Polymorphism in Python

```
class Parrot:
    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")
```

```
class Penguin:
    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")
# common interface
def flying_test(bird):
    bird.fly()
#instantiate objects
blu = Parrot()
peggy = Penguin()
# passing the object
flying_test(blu)
flying_test(peggy)
```

When we run above program, the output will be:

```
Parrot can fly
Penguin can't fly
```

In the above program, we defined two classes `Parrot` and `Penguin`. Each of them have common method `fly()` method. However, their functions are different. To allow polymorphism, we created common interface i.e `flying_test()` function that can take any object. Then, we passed the objects `blu` and `peggy` in the `flying_test()` function, it ran effectively.