30. Introduction to OOPs in Python

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** and **behavior**. E.g. 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). In Python, the concept of OOP follows some basic principles:
- **Inheritance**: using details from a new class without modifying existing class;
- **Encapsulation**: hiding the private details of a class from other objects;
- **Polymorphism**: using common operation in different ways for different data input.

1 Class

A class is a blueprint for the 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. **Instance is a specific object created from a particular class.**

2 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()
```

Example 1: Creating Class and Object in Python

```
class Parrot:
    species = "bird"  # class attribute

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

# instantiate the Parrot class:
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes:
print(f'Blu is a {blu.__class__.species}')
print(f'Woo is also a {woo.__class__.species}')

# access the instance attributes:
print(f'{blu.name} is {blu.age} years old')
print(f'{woo.name} is {woo.age} years old')
```

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

3 Methods

Methods – functions defined inside a class used to specify behavior of an object.

Example 2: Creating Methods in Python

```
class Parrot:

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

    # instance method
    def sing(self, song):
        return f'{self.name} sings {song}'

    def dance(self):
        return f'{self.name} is now dancing'

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
blu.sing("'Happy'")
blu.dance()
```

```
Blu sings 'Happy'
Blu is now dancing
```

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

4 Inheritance, super( ).__init__( )

Inheritance is a way of creating new class 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()
```

```
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. Also, **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.

5 Encapsulation, setter function

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(f'Selling Price: {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()
```

```
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.

Then, 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.

6 Polymorphism

Polymorphism is an ability to use common interface for multiple 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)
```

```
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.

Key points to remember

- The programming gets easy and efficient;
- Class is sharable, so codes can be reused;
- Data is safe and secure with data abstraction.