*All the IPython Notebooks in **Python Object Class** lecture series by **[Dr. Milaan Parmar](https://www.linkedin.com/in/milaanparmar/)** are available @ **[GitHub](https://github.com/milaan9/06_Python_Object_Class)**** </small>

# Python OOPs Concepts

In this class, you'll learn about Object-Oriented Programming (OOP) in Python and its fundamental concept with the help of examples.

## Object Oriented Programming

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

Object-oriented programming (OOP) is a programming paradigm based on the concept of **"objects"**. The object contains both data and code: Data in the form of properties (often known as attributes), and code, in the form of methods (actions object can perform).

An object has two characteristics:

- attributes
- behavior

For example:

A parrot is can be an object,as it has the following properties:

- name, age, color as attributes
- singing, dancing as 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:

## Class

In Python, everything is an object. A class is a blueprint for the object. To create an object we require a model or plan or blueprint which is nothing but class.

We create class to create an object. A class is like an object constructor, or a "blueprint" for creating objects. We instantiate a class to create an object. The class defines attributes and the behavior of the object, while the object, on the other hand, represents the class.

**Class represents the properties (attribute) and action (behavior) of the object. Properties represent variables, and actions are represented by the methods. Hence class contains both variables and methods.**

We can think of class as a 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, a parrot is an object.

**Syntax:**

```
class classname:
    '''documentation string'''
    class_suite
```

- **Documentation string:** represent a description of the class. It is optional.
- **class_suite:** class suite contains component statements, variables, methods, functions, attributes.

The example for class of parrot can be :

```
class Parrot:
    pass
```

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

```
class Person:
    pass
print(Person)
```

In [1]:
```
# Creating a class

class Person:
    pass
print(Person)
```

```
<class '__main__.Person'>
```

## Object

The physical existence of a class is nothing but an object. In other words, the object is an entity that has a state and behavior.

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

**Syntax:**

```
reference_variable = classname()
```

The example for object of parrot class can be:

```
obj = Parrot()
```

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

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

```
p = Person()
print(p)
```

In [2]:
```
# Example 1: We can create an object by calling the class

p = Person()
print(p)
```

```
<__main__.Person object at 0x0000021E45765B20>
```

In [3]:
```
# Example 2: Creating Class and Object in Python

class Student:
    """This is student class with data"""
    def learn(self):    # A sample method
        print("Welcome to Dr. Milaan Parmar's class on Python Programming")

stud = Student()        # creating object
stud.learn()            # Calling method

# Output: Welcome to Dr. Milaan Parmar's class on Python Programming
```

```
Welcome to Dr. Milaan Parmar's class on Python Programming
```

# Class Constructor

In the examples above, we have created an object from the **`Person`** class. However, a class without a constructor is not really useful in real applications. Let us use constructor function to make our class more useful. Like the constructor function in Java or JavaScript, Python has also a built-in **`__init__()`** constructor function. The **`__init__()`** constructor function has **`self`** parameter which is a reference to the current instance of the class.

In [4]:
```
class Person:
      def __init__ (self, name):
        # self allows to attach parameter to the class
          self.name =name

p = Person('Milaan')
print(p.name)
print(p)
```

```
Milaan
<__main__.Person object at 0x0000021E45765DC0>
```

Let us add more parameters to the constructor function.

In [5]:
```
# Example 1: add more parameters to the constructor function.

class Person:
      def __init__(self, firstname, lastname, age, country, city):
            self.firstname = firstname
            self.lastname = lastname
            self.age = age
            self.country = country
            self.city = city

p = Person('Milaan', 'Parmar', 96, 'England', 'London')
print(p.firstname)
print(p.lastname)
print(p.age)
print(p.country)
print(p.city)
```

```
Milaan
Parmar
96
England
London
```

## Instance Variables and Methods

If the value of a variable varies from object to object, then such variables are called instance variables. For every object, a separate copy of the instance variable will be created.

When we create classes in Python, instance methods are used regularly. we need to create an object to execute the block of code or action defined in the instance method.

We can access the instance variable and methods using the object. Use dot ( **.** ) operator to access instance variables and methods.

In Python, working with an instance variable and method, we use the **`self`** keyword. When we use the **`self`** keyword as a parameter to a method or with a variable name is called the instance itself.

> **Note:** Instance variables are used within the instance method

In [6]:
```
# Example 2: Creating Class and Object in Python

class Student:
    def __init__(self, name, percentage):
        self.name = name
        self.percentage = percentage

    def show(self):
        print("Name is:", self.name, "and percentage is:", self.percentage)


stud = Student("Arthur", 90)
stud.show()

# Output Name is: Arthur and percentage is: 90
```

```
Name is: Arthur and percentage is: 90
```

In [7]:
```python
# Example 3: Creating Class and Object in Python

class Parrot:
    species = "bird"                    # class attribute
    def __init__(self, name, age):      # instance attribute
        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))
```

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

**Explanation**:

In the above program, we created a class with the name **`Parrot`** . Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the **`__init__`** method of the class. It is the initializer method that is first run as soon as the object is created.

Then, we create instances of the **`Parrot`** class. Here, **`blu`** and **`woo`** are references (value) to our new objects.

We can access the class attribute using **`__class__.species`** . Class attributes are the 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.

To learn more about classes and objects, go to **Python Classes and Objects** (https://github.com/milaan9/06_Python_Object_Class/blob/main/002_Python_Classes_and_Objects.ipynb)

## Object Method

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

Objects can have methods. The methods are functions which belong to the object.

In [8]:
```
# Example 1:

class Person:
      def __init__(self, firstname, lastname, age, country, city):
            self.firstname = firstname
            self.lastname = lastname
            self.age = age
            self.country = country
            self.city = city
      def person_info(self):
        return f'{self.firstname} {self.lastname} is {self.age} years old. He lives in {self.
city}, {self.country}'

p = Person('Milaan', 'Parmar', 96, 'England', 'London')
print(p.person_info())
```

```
Milaan Parmar is 96 years old. He lives in London, England
```

In [9]:
```
# Example 2: Creating Object 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("'Happy'"))
print(blu.dance())
```

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

**Explanation**:

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`** .

## Object Default Methods

Sometimes, you may want to have a default values for your object methods. If we give default values for the parameters in the constructor, we can avoid errors when we call or instantiate our class without parameters. Let's see how it looks:

In [10]:
```
class Person:
      def __init__(self, firstname='Milaan', lastname='Parmar', age=96, country='England', ci
ty='London'):
            self.firstname = firstname
            self.lastname = lastname
            self.age = age
            self.country = country
            self.city = city

      def person_info(self):
        return f'{self.firstname} {self.lastname} is {self.age} years old. He lives in {self.
city}, {self.country}.'

p1 = Person()
print(p1.person_info())
p2 = Person('Ben', 'Doe', 30, 'Finland', 'Tampere')
print(p2.person_info())
```

```
Milaan Parmar is 96 years old. He lives in London, England.
Ben Doe is 30 years old. He lives in Tampere, Finland.
```

## Method to Modify Class Default Values

In the example below, the **`Person`** class, all the constructor parameters have default values. In addition to that, we have skills parameter, which we can access using a method. Let us create **`add_skill`** method to add skills to the skills list.

In [11]:
```
class Person:
      def __init__(self, firstname='Milaan', lastname='Parmar', age=96, country='England', ci
ty='London'):
            self.firstname = firstname
            self.lastname = lastname
            self.age = age
            self.country = country
            self.city = city
            self.skills = []

      def person_info(self):
        return f'{self.firstname} {self.lastname} is {self.age} years old. He lives in {self.
city}, {self.country}.'
      def add_skill(self, skill):
            self.skills.append(skill)

p1 = Person()
print(p1.person_info())
p1.add_skill('Python')
p1.add_skill('MATLAB')
p1.add_skill('R')
p2 = Person('Ben', 'Doe', 30, 'Finland', 'Tampere')
print(p2.person_info())
print(p1.skills)
print(p2.skills)
```

```
Milaan Parmar is 96 years old. He lives in London, England.
Ben Doe is 30 years old. He lives in Tampere, Finland.
['Python', 'MATLAB', 'R']
[]
```

# Inheritance

In Python, [inheritance (https://github.com/milaan9/06_Python_Object_Class/blob/main/003_Python_Inheritance.ipynb)](https://github.com/milaan9/06_Python_Object_Class/blob/main/003_Python_Inheritance.ipynb) is the process of inheriting the properties of the base class (or parent class) into a derived class (or child class).

In an Object-oriented programming language, inheritance is an important aspect. Using inheritance we can reuse parent class code. Inheritance allows us to define a class that inherits all the methods and properties from parent class. The parent class or super or base class is the class which gives all the methods and properties. Child class is the class that inherits from another or parent class.

In inheritance, the child class acquires and access all the data members, properties, and functions from the parent class. Also, a child class can also provide its specific implementation to the functions of the parent class.

## Use of inheritance

The main purpose of inheritance is the reusability of code because we can use the existing class to create a new class instead of creating it from scratch.

**Syntax:**

```
class BaseClass:
    Body of base class
class DerivedClass(BaseClass):
    Body of derived class
```

In [12]:
```
# Example 1: Use of Inheritance in Python

class ClassOne:              # Base class
    def func1(self):
        print('This is Parent class')

class ClassTwo(ClassOne):    # Derived class
    def func2(self):
        print('This is Child class')

obj = ClassTwo()
obj.func1()
obj.func2()
```

```
This is Parent class
This is Child class
```

Let us create a student class by inheriting from **`Person`** class.

In [13]:
```
# Example 2: Use of Inheritance in Python

class Student(Person):
    pass

s1 = Student('Arthur', 'Curry', 33, 'England', 'London')
s2 = Student('Emily', 'Carter', 28, 'England', 'Manchester')
print(s1.person_info())
s1.add_skill('HTML')
s1.add_skill('CSS')
s1.add_skill('JavaScript')
print(s1.skills)

print(s2.person_info())
s2.add_skill('Organizing')
s2.add_skill('Marketing')
s2.add_skill('Digital Marketing')
print(s2.skills)
```

```
Arthur Curry is 33 years old. He lives in London, England.
['HTML', 'CSS', 'JavaScript']
Emily Carter is 28 years old. He lives in Manchester, England.
['Organizing', 'Marketing', 'Digital Marketing']
```

**Exaplanation:**

We did not call the **`__init__()`** constructor in the child class. If we didn't call it then we can still access all the properties from the parent. But if we do call the constructor we can access the parent properties by calling **`super()`** .
We can add a new method to the child or we can override the parent class methods by creating the same method name in the child class. When we add the **`__init__()`** function, the child class will no longer inherit the parent's **`__init__()`** function.

## Overriding parent method

In [14]:
```
# Example 2: Overriding parent method from above example

class Student(Person):
    def __init__ (self, firstname='Milaan', lastname='Parmar',age=96, country='England', city
='London', gender='male'):
        self.gender = gender
        super().__init__(firstname, lastname,age, country, city)

    def person_info(self):
        gender = 'He' if self.gender =='male' else 'She'
        return f'{self.firstname} {self.lastname} is {self.age} years old. {gender} lives in
{self.city}, {self.country}.'

s1 = Student('Arthur', 'Curry', 33, 'England', 'London','male')
s2 = Student('Emily', 'Carter', 28, 'England', 'Manchester','female')
print(s1.person_info())
s1.add_skill('HTML')
s1.add_skill('CSS')
s1.add_skill('JavaScript')
print(s1.skills)

print(s2.person_info())
s2.add_skill('Organizing')
s2.add_skill('Marketing')
s2.add_skill('Digital Marketing')
print(s2.skills)
```

```
Arthur Curry is 33 years old. He lives in London, England.
['HTML', 'CSS', 'JavaScript']
Emily Carter is 28 years old. She lives in Manchester, England.
['Organizing', 'Marketing', 'Digital Marketing']
```

**Exaplanation**:

We can use **`super()`** built-in function or the parent name Person to automatically inherit the methods and properties from its parent. In the example above we override the parent method. The child method has a different feature, it can identify, if the gender is male or female and assign the proper pronoun(He/She).

In [15]:
```
# 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
```

**Exaplanation**:

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 the **`swim()`** method.

Again, the child class modified the behavior of the parent class. We can see this from the **`whoisThis()`** method. Furthermore, we extend the functions of the parent class, by creating a new **`run()`** method.

Additionally, we use the **`super()`** function inside the **`__init__()`** method. This allows us to run the **`__init__()`** method of the parent class inside the child class.

# Encapsulation

n Python, encapsulation is a method of wrapping data and functions into a single entity. For example, A class encapsulates all the data (methods and variables). Encapsulation means the internal representation of an object is generally hidden from outside of the object's definition.

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

## Need of Encapsulation

Encapsulation acts as a protective layer. We can restrict access to methods and variables from outside, and It can prevent the data from being modified by accidental or unauthorized modification. Encapsulation provides security by hiding the data from the outside world.

In Python, we do not have access modifiers directly, such as public, private, and protected. But we can achieve encapsulation by using single prefix underscore and double underscore to control access of variable and method within the Python program.

In [16]:
```
# Example 1: 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()
```

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

**Explanation**:

In the above program, we defined a **`Computer`** class.

We used **`__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.

As shown, to change the value, we have to use a setter function i.e **`setMaxPrice()`** which takes price as a parameter.

In [17]:
```
# Example 2: Data Encapsulation in Python

class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary

    def show(self):
        print("Name is ", self.name, "and salary is", self.__salary)

# Outside class
E = Employee("Bella", 60000)
E.PrintName()
print(E.name)
print(E.PrintName())
print(E.__salary)

# AttributeError: 'Employee' object has no attribute '__salary'
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-17-b977fb738e16> in <module>
     11 # Outside class
     12 E = Employee("Bella", 60000)
---> 13 E.PrintName()
     14 print(E.name)
     15 print(E.PrintName())

AttributeError: 'Employee' object has no attribute 'PrintName'
```

**Explanation**:

In the above example, we create a class called `Employee` . Within that class, we declare two variables `name` and `__salary` . We can observe that the name variable is accessible, but `__salary` is the **private variable**. We cannot access it from outside of class. If we try to access it, we will get an error.

# Polymorphism

Polymorphism is based on the greek words **Poly** (many) and **morphism** (forms). We will create a structure that can take or use many forms of objects.

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

**Example 1:** The student can act as a student in college, act as a player on the ground, and as a daughter/brother in the home.

**Example 2:** In the programming language, the `+` operator, acts as a concatenation and arithmetic addition.

**Example 3:** If we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape.

In Python, polymorphism allows us to define the child class methods with the same name as defined in the parent class.

In [18]:
```
# Example 1: 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
```

**Exaplanation**:

In the above program, we defined two classes **`Parrot`** and **`Penguin`** . Each of them have a common **`fly()`** method. However, their functions are different.

To use polymorphism, we created a common interface i.e **`flying_test()`** function that takes any object and calls the object's **`fly()`** method. Thus, when we passed the **`blu`** and **`peggy`** objects in the **`flying_test()`** function, it ran effectively.

In [19]:
```
# Example 2: Using Polymorphism in Python

class Circle:
    pi = 3.14

    def __init__(self, redius):
        self.radius = redius

    def calculate_area(self):
        print("Area of circle:", self.pi * self.radius * self.radius)

class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        print("Area of Rectangle:", self.length * self.width)

cir = Circle(9)
rect = Rectangle(9, 6)
cir.calculate_area()   # Output Area of circle: 254.34

rect.calculate_area()  # Output Area od Rectangle: 54
```

```
Area of circle: 254.34
Area of Rectangle: 54
```

**Exaplanation**:

In the above example, we created two classes called `Circle` and `Rectangle`. In both classes, we created the same method with the name `calculate_area`. This method acts differently in both classes. In the case of the `Circle` class, it calculates the area of the circle, whereas, in the case of a `Rectangle` class, it calculates the area of a rectangle.

# Key Points to Remember:

- Object-Oriented Programming makes the program easy to understand as well as efficient.
- Since the class is sharable, the code can be reused.
- Data is safe and secure with data abstraction.
- Polymorphism allows the same interface for different objects, so programmers can write efficient code.

# 💻 Exercises → Objects and Classes

## Exercises → Level 1

1. Python has the module called **statistics** and we can use this module to do all the statistical calculations. However, to learn how to make function and reuse function let us try to develop a program, which calculates the measure of central tendency of a sample ( **mean** , **median** , **mode** ) and measure of variability ( **range** , **variance** , **standard deviation** ). In addition to those measures, find the **min** , **max** , **count** , **percentile** , and **frequency distribution** of the sample. You can create a class called **statistics** and create all the functions that do statistical calculations as methods for the **statistics** class. Check the output below.

- ```
  ages = [31, 26, 34, 37, 27, 26, 32, 32, 26, 27, 27, 24, 32, 33, 27, 25, 26, 38, 37, 31, 34, 24, 33, 29, 26]
  print('Count:', data.count()) # 25
  print('Sum: ', data.sum()) # 744
  print('Min: ', data.min()) # 24
  print('Max: ', data.max()) # 38
  print('Range: ', data.range() # 14
  print('Mean: ', data.mean()) # 30
  print('Median: ', data.median()) # 29
  print('Mode: ', data.mode()) # {'mode': 26, 'count': 5}
  print('Standard Deviation: ', data.std()) # 4.2
  print('Variance: ', data.var()) # 17.5
  print('Frequency Distribution: ', data.freq_dist()) # [(20.0, 26), (16.0, 27), (12.0, 32), (8.0, 37), (8.0, 34), (8.0, 33), (8.0, 31), (8.0, 24), (4.0, 38), (4.0, 29), (4.0, 25)]
  ```

```sh
# you output should look like this:
print(data.describe())
Count: 25
Sum:  744
Min:  24
Max:  38
Range:  14
Mean:  30
Median:  29
Mode:  (26, 5)
Variance:  17.5
Standard Deviation:  4.2
Frequency Distribution: [(20.0, 26), (16.0, 27), (12.0, 32), (8.0, 37), (8.0, 34), (8.0, 33), (8.0, 31), (8.0, 24), (4.0, 38), (4.0, 29), (4.0, 25)]
```

## Exercises → Level 2

1. Create a class called **PersonAccount** . It has **firstname** , **lastname** , **incomes** , **expenses** properties and it has **total_income** , **total_expense** , **account_info** , **add_income** , **add_expense** and **account_balance** methods. Incomes is a set of incomes and its description. The same goes for expenses.

In [ ]:
```