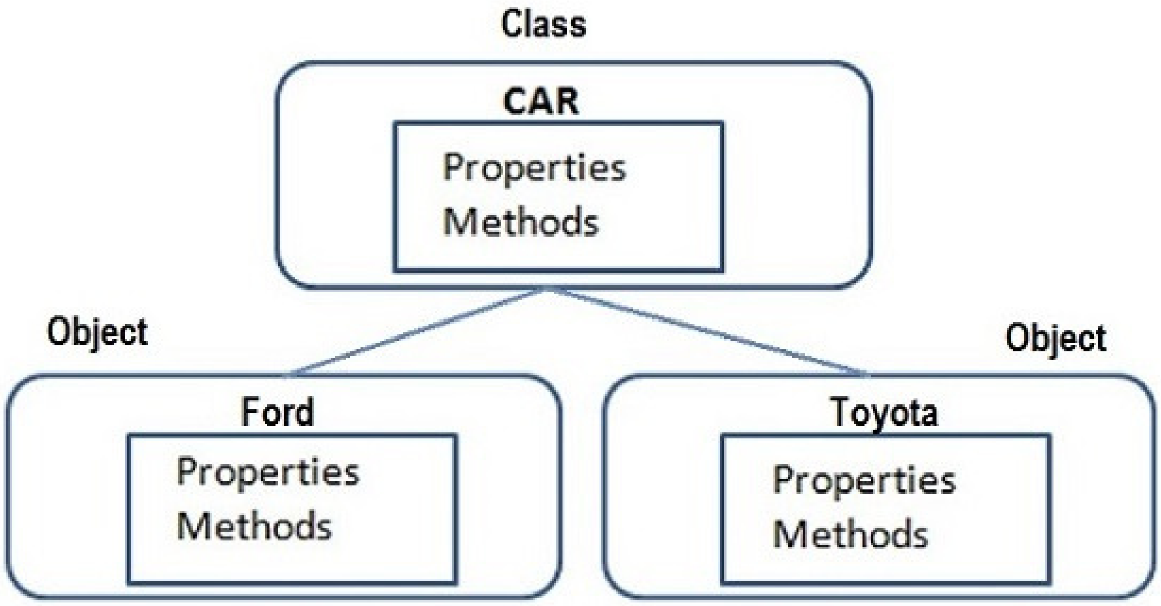
OBJECT ORIENTED PROGRAMMING

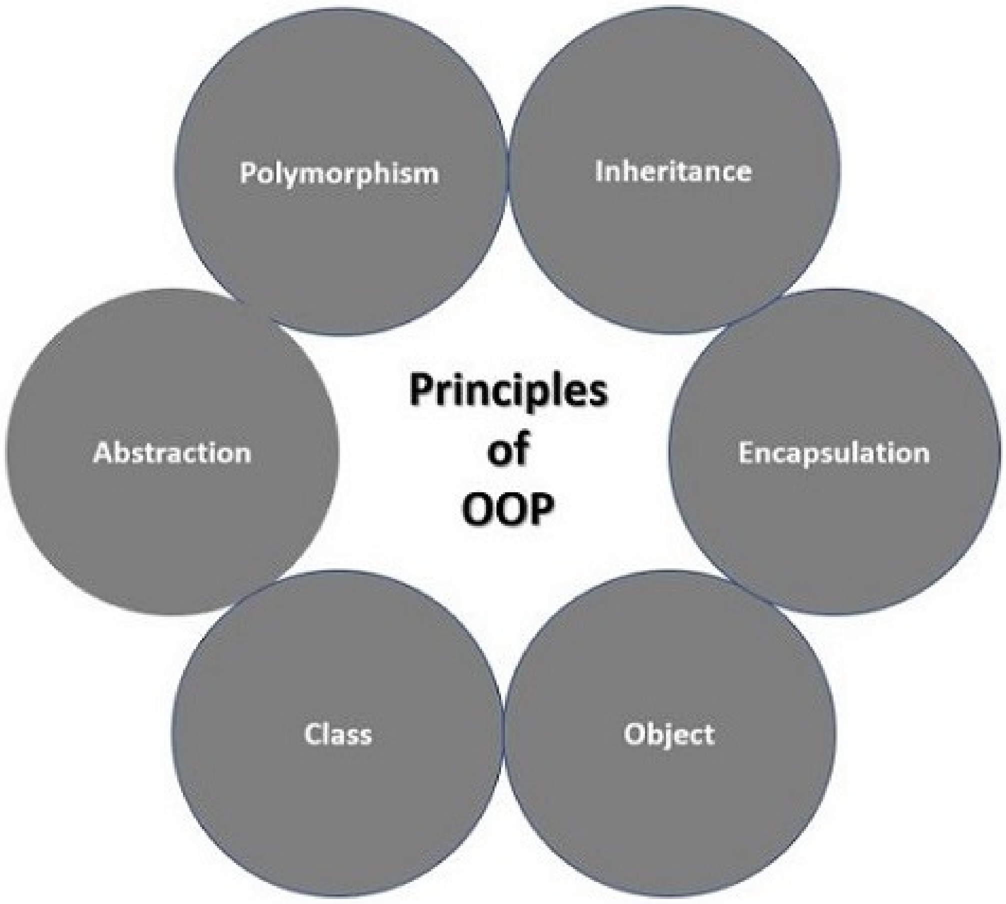
ARUN ARUNISTO

Python is a pure object oriented programming language since the time it existed. Due to this creating and using classes and objects are easy in python, you can simply create a class and object without complex coding syntax.
In the real-world, we deal with and process objects, such as student, employee, invoice, car, etc. Objects are not only data and not only functions, it's a combination of both. Each real-world object has attributes and behavior associated with it.



Each attribute will have a value associated with it. Attribute is equivalent to data.

The most important feature of object-oriented approach is defining attributes and their functionality as a single unit called class. It serves as a blueprint for all objects having similar attributes and behavior.
In OOP, class defines what are the attributes its object has, and how is its behavior. Object, on the hand, is an instance of the class.
Object oriented programming principles:



Python is a pure object-oriented programming language. In, Python each and every element is an object of one or the other class. Integer, String, List, Dictionary, etc used in a program they are objects of corresponding built-in classes.
In python the Object "**class**" is the base or parent class for all the classes, built-in as well as user defined.
The "**class**" keyword is used to define a new class.

Syntax:

```
class <class_name>:
        <class_properties>
```

So, we are going to create a class named “Calculator” with two arguments “a” and “b” and with corresponding functions like addition, subtraction, multiplication and division. lets, start,

```
#defining class
class Calculator:
    #initializing args
    def __init__(self, a, b):
        self.a = a
        self.b = b
    #functions
    def add(self):
        return self.a+self.b
    def sub(self):
        return self.a-self.b
    def mul(self):
        return self.a*self.b
    def div(self):
        return self.a/self.b
```

As , mentioned above we created a class with arguments and functions now its time to call the function.

```
#calling functions with args
operations = Calculator(25, 45)
#calling functions
print("Add:",operations.add())
```

It will calls the function inside the class and returns the result of add function

Python class keeps the following built-in attributes and they can be accessed using dot operator

1. **__dict__** : Dictionary containing the class's namespace
2. **__doc__** : Class documentation string or none, if undefined
3. **__name__** : Class name
4. **__module__** : module name which the class is defined. This attribute is "__main__" in interactive mode
5. **__bases__** : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list

```
class PersonalDetails:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        print("Name:",self.name)
        print("Age:".self.age)
PersonalDetails.__doc__ = "Class for printing
                                            Personal Details"
```

```
print("PersonalDetails.__dict__:",
      PersonalDetails.__dict__)
print("PersonalDetails.__doc__:",
      PersonalDetails.__doc__)
print("PersonalDetails.__name__:",
      PersonalDetails.__name__)
print("PersonalDetails.__module__:",
      PersonalDetails.__module__)
print("PersonalDetails.__bases__:",
      PersonalDetails.__bases__)
“””
Output:
PersonalDetails.__dict__: {'__module__': '__main__',
'__init__': <function PersonalDetails.__init__ at
0x000001E303B4FA30>, 'display_info': <function
PersonalDetails.display_info at
0x000001E303B4FAC0>, '__dict__': <attribute '__dict__'
of 'PersonalDetails' objects>, '__weakref__': <attribute
'__weakref__' of 'PersonalDetails' objects>, '__doc__':
'Class for printing Personal Details'}
PersonalDetails.__doc__: Class for printing Personal
Details
PersonalDetails.__name__: PersonalDetails
PersonalDetails.__module__: __main__
PersonalDetails.__bases__: (<class 'object'>,)
”””
```

The variables inside __init__() are instance variables. A class attributes are not initialized inside __init__() constructor, they're defined in the class, but outside any method. They can be accessed by name of class in addition to object. A class attribute represents common attribute of all objects of a class.

```
class PersonalDetails:
    company_name = "CyberWorld"
    __doc__ = "PersonalDetails of employee"
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        print("Company:",
                PersonalDetails.company_name)
        print("Name:",self.name)
        print("Age:",self.age)
```

In the above example "**name**" and "**age**" is instance of class and "**company_name**" and "**__doc__**" are the attributes of the class.

Python has a built-in function **classmethod()** which transforms an instance method to a class method which can be called with the refrence to the class only and not the object.

```
class PersonalDetails:
    company_name = "CyberWorld"
    __doc__ = "PersonalDetails of employee"
    employee_count = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        PersonalDetails.employee_count+=1

    def display_info(self):
        print("Company:",
                     PersonalDetails.company_name)
        print("Name:",self.name)
        print("Age:",self.age)

    def show_count(self):
        print(self.employee_count)
    total_employee = classmethod(show_count)
```

In the above example **total_employee** we declared it as classmethod so the **total_employee** we can only access using the class not with the object

```
per1 = PersonalDetails("Arun", 27)
per2 = PersonalDetails("Arunisto", 27)
per3 = PersonalDetails("Jake", 25)

per1.show_count()
PersonalDetails.total_employee()

""""
Output:
3
3
""""
```

using **@classmethod** decorator is the best way to define a class method as it is more convenient than first declaring an instance method and then transforming to a class method.

```
class PersonalDetails:
    company_name = "CyberWorld"
    __doc__ = "PersonalDetails of employee"
    employee_count = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        PersonalDetails.employee_count+=1

    def display_info(self):
        print("Company:",
                    PersonalDetails.company_name)
        print("Name:",self.name)
        print("Age:",self.age)

    @classmethod
    def show_count(cls):
        print(cls.employee_count)

    @classmethod
    def add_employee(cls, name, age):
        return cls(name, age)
```

```
per1 = PersonalDetails("Arun", 27)
per2 = PersonalDetails("Arunisto", 27)
per3 = PersonalDetails("Jake", 25)
PersonalDetails.show_count() #3
per4 = PersonalDetails.add_employee("Arun", 25)
PersonalDetails.show_count() #4
```

And the other one is **staticmethod()**, and it doesn't have a mandatory argument like reference to the object - self or reference to the class - cls. Python's standard library function **staticmethod()** returns a static method

```
class PersonalDetails:
    ...
    def show_count():
        print(PersonalDetails.employee_count)

    total_count = staticmethod(show_count)

per1 = PersonalDetails("Arun", 27)
per2 = PersonalDetails("Arunisto", 27)
per3 = PersonalDetails("Jake", 25)
PersonalDetails.total_count() #3
```

using **@staticmethod** decorator

```
class PersonalDetails:
    ...
    @staticmethod
    def show_count():
        print(PersonalDetails.employee_count)

per1 = PersonalDetails("Arun", 27)
per2 = PersonalDetails("Arunisto", 27)
per3 = PersonalDetails("Jake", 25)
PersonalDetails.show_count() #3
```

You can **add**, **remove**, or **modify** attributes of classes and objects at any time.

```
class PersonalDetails:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def displayInfo(self):
        print("Name:",self.name)
        print("Age:",self.age)
```

```
per1 = PersonalDetails("Arun", 25)
#adding attribute
per1.salary = "100$"
#modifying
per1.name = "Arunisto"
#displaying
per1.displayInfo()
print("Salary:",per1.salary)
#deleting
del per1.salary
```

Instead of using normal statements to access the attributes, we can use the following functions

1. **getattr(object, name)** - to access attribute
2. **hasattr(object, name)** - to check if an attribute exist
3. **setattr(object, name, value)** - to set an attribute, if attribute not exist, it will create
4. **delattr(object, name)** - to delete an attribute

```
class PersonalDetails:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def displayInfo(self):
        print("Name:",self.name)
        print("Age:",self.age)

per1 = PersonalDetails("Arun", 25)
#accessing attribute
print(getattr(per1, 'name')) #Arun
#setting attribute
setattr(per1, 'name', 'Arunisto')
#setting not existing attribute
setattr(per1, 'salary', '100$')
#checking attribute
print(hasattr(per1, 'salary')) #True
#deleting attribute
delattr(per1, 'salary')
#checking after deleting
print(hasattr(per1, 'salary')) #False
```

The languages such as C++ and java use access modifiers to restrict access to class members (i.e., variables and methods). They have keywords like public, protected, and private to specify the type of access.
Python has no provision to specify the type of access that class member may have. by default, all the variables and methods in a class are public.
Python doesn't enforce any restrictions on accessing any instance variable or method. python use underscores to differentiate it. like, for private variable we prefix with double underscore for protected we use single underscore.

```
class PersonalDetails:
   def __init__(self, name, age):
      self._name = name #protected
      self.__age = age #private
   def displayInfo(self):
      print("Name:",self._name)
      print("Age:",self.__age)
per1 = PersonalDetails("Arun", 25)
per1.displayInfo()
```

In python private variable not available for use outside the class, so we are not able to access using object it will raise an error.

```
per1 = PersonalDetails("Arun", 25)
print(per1.__age)
```

```
AttributeError: 'PersonalDetails' object has no attribute '__age'. Did you mean: '_name'?
```

Dont worry we can still access the private members by python's **name managling** technique. **Name mangling** is the process of changing name of a member with double underscore to the form **object._class__variable**. using this technique we can still access outside the class.

```
per1 = PersonalDetails("Arun", 25)
print(per1._PersonalDetails__age) #25
```

Python has a built-in function **property()** it act as an interface to the instance variables of a python class. The encapsulation of oops requires that the instance variables should have restricted private access. Python doesn't have an efficient mechanism for that purpose. The **property()** function provides an alternative. The **property()** functions uses the getter, setter and delete methods defined in a class to define a property object for the class.

Syntax:

```
property(fget=None, fset=None,
              fdel=None, doc=None)
```

1. **fget** - retrieves value of an instance variable
2. **fset** - assigns value to an instance variable
3. **fdel** - method that removes an instance variable
4. **doc** = documentation string for the property

```python
class PersonalDetails:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
    #getters and setters
    def get_name(self):
        return self.__name
    def set_name(self, name):
        self.__name = name
    def get_age(self):
        return self.__age
    def set_age(self, age):
        self.__age = age

per1 = PersonalDetails("Arun", 27)
print("Name:",per1.get_name()) #Arun
per1.set_name("Arunisto")
print("Name:",per1.get_name()) #Arunisto
```

The getter and setter methods can retrieve or assign value to instance variables. The **property()** function uses them to add property objects as class attributes.

```
#The property defined as
class PersonalDetails:
  def __init__(self, name, age):
    self.__name = name
    self.__age = age
  #getters and setters
  def get_name(self):
    return self.__name
  def set_name(self, name):
    self.__name = name
  def get_age(self):
    return self.__age
  def set_age(self, age):
    self.__age = age

  name = property(get_name, set_name, "name")
  age = property(get_age, set_age, "age")

per1 = PersonalDetails("Arun", 27)
print("Name:",per1.name) #Arun
per1.name = "Arunisto"
print("Name:",per1.name) #Arunisto
```