INSTANCE VARIABLE: OBJECT LEVEL VARIABLE

The value of the variables varies from object to object, such type of variables are called as 'instance variable'.

STATIC VARIABLE: CLASS LEVEL VARIABLE.

These variables are same for all the objects.

LOCAL VARIABLE: METHOD LEVEL VARIABLE.

```
I/p - class Abc:
        def __init__(self,n,a):
            self.name = n
            self.age = a

        def display(self):
            Print("*" * 5)
            Print("name: ", self.name)
            Print("age: ", self.age)
            Print('*' * 5)
```

```
@staticmethod
def utility():
    Print("Hello world, this is utility
                                method!")

obj1 = Abc('xyz', 34)
obj1.name
```

o/p - 'xyz'

```
obj1.display()
```

o/p - *****
name : xyz
age : 34
*****

```
obj.utility()
```

o/p - Hello world, this is utility method!

NOTE:

@staticmethod : It is used when 'self' is not called (or) mentioned

The four pillars of Object oriented program are:

1. Abstraction.
2. Encapsulation
3. Inheritance
4. Polymorphism.

1. ABSTRACTION : HIDING

Abstraction means hiding the complexity and only showing the essential features of the object.

So, in a way, Abstraction means hiding the real implementation and we, as a user, knowing only how we use it.

(or)

A process of highlighting set of services and hiding the implementation.

↳ Abstraction in Python is achieved by using abstract classes and interfaces.

While defining an abstract class, the following are to be noted:

↳ An abstract class can have both a normal method and an abstract method.

↳ An abstract class can't be instantiated i.e., we can't create objects for the abstract class.

2. ENCAPSULATION :

Encapsulation is a process of wrapping up variables and methods into a single entity.

↳ In programming, a class is an example that wraps all the variables and methods defined inside it.

↳ This can be achieved by using private and protected Access members.

↳ Private variables are preceded by using two underscores.

↳ protected variables are preceded by using a single underscore.

SETTERS - Takes each and every variable at a time.

GETTERS - To get the ~~value~~ variables.

I/P - class Friend:

```
    def __init__(self):
        self.job = "None"
    def getjob(self):
        return self.job
    def setjob(self, job):
        self.job = job

SITA = Friend()
RAMA = Friend()

SITA.setjob("Muslcian")
RAMA.setjob("Manager")

Print(SITA.job)
Print(RAMA.job)
```

O/P

O/P - Muslcian
Manager

I/p -
```
class Abc():
    def __init__(self,n,a):
        def setname(self,n):
            self._name = n
        def setage(self,a):
            self.__age = a
        def getname(self):
            return self._name
        def getage(self):
            return self._age

a = Abc()
a.setage(24)
a.setname(PAVANI)
Print(a.getname(), a.getage())
```

O/p - PAVANI , 24

3. INHERITANCE:

It allows us to define a class that inherits all the methods and properties from another class.

PARENT CLASS: Parent class is the class being inherited from, also called base class.

CHILD CLASS: Child class is the class that inherits from another class, also called derived class.

WITHOUT INHERITANCE:

```
I/P - class A:
        def m1():
            pass
        def m2():
            pass
      class B:
        def m3():
            pass
```

```
    def m4():
        pass

class C:-
    def m5():
        pass
    def m6():
        pass.
```

WITH INHERITANCE :

I/P -

```
class A:
    def m1(self):
        Print("class-A : Method-1")
    def m2(self):
        Print("class-A : Method-2")

class B:
    def m3(self):
        Print("class B : Method-3")
    def m4(self):
        Print("class B : Method-4")

class C:
    def m5(self):
```

```
        Print("class-C : Method-5")

    def m6(self):
        Print("class-C : Method-6")

a = A()
b = B()
c = C()

a.m1()
b.m4()
b.m1()
```

o/p - class-A : Method-1

class-B : Method-4

class-A : Method-1

There are 2 types of Inheritances. they are:

1. Multiple Inheritance
2. Multilevel Inheritance.

1. Multiple Inheritance: A class can be derived from more than one base class. This is called as Multiple Inheritance.

↳ In this, the features of all the base classes are inherited into the derived class.

↳ The syntax for multiple inheritance is similar to the single inheritance.

I/p -

Father class created

O/p - class Father:

fathername = " "

def show_father(self):

Print(self.fathername)

Mother class created

class Mother:

mothername = " "

def show_mother(self):

Print(self.mothername)

Son class inherits father and mother classes

class Son(Father, Mother):

def show_parent(self):

Print("Father : ", self.fathername)

```
        Print("Mother : ", self.mothername)

S1 = son()  # Object of son class
S1.fathername = 'N.E.K'
S1.mothername = 'N.D.K.S'
S1.show_parent()
```

O/p - Father : N.E.K
Mother : N.D.K.S

2. Multilevel Inheritance:

We can also inherit from a derived class. This is called multilevel inheritance.

4 In this, features of the base class and the derived class are inherited into the new derived class.

```
I/p - class Family:
        def show_family(self):
            Print("This is our family:")

# Father class inherited from family
```

(113)

```
Class Father(family):
    fathername = " "
    def show_father(self):
        Print(self.fathername)

# Mother class inherited from family
class Mother(family):
    mothername= ' '
    def show_mother(self):
        Print(self.mothername)

#Son class inherited from father and
mother classes:
class son(Father, Mother):
    def show_parent(self):
        Print('Father: ', self.fathername)
        Print('Mother: ', self.mothername)

S1 = son( ) # Object of son class
S1. fathername = "N.E.K"
S1. mothername = "N.D.K.S"
```

S1. show_family()
S2. show_parent()

O/p - This is our family:

Father : N.E.K

Mother : N.D.K.S

4. POLYMORPHISM: ~~The~~

The condition of occurance in different forms is called as polymorphism.

↳ It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

↳ It allows us to define methods in the class of child with the same name as defined in their parent class.

I/p - class A:

```
    def m1(self):
        Print('Hi')
    def m2(self, name):
        Print("Hi", name)

a = A()
a.m1('PAVANI')
```

O/p - Hi PAVANI

Polymorphism is classified into 2 categories.

1. Overloading
2. Overriding

1. Overloading:

The ability of a function or an operator to behave in different ways depending on the parameters that are passed to the function, or the operands that the operator acts on.

The Overloading is classified into 3 sub-categories:

(i) Function Overloading

(ii) Operator Overloading

(iii) Constructor Overloading.

* In the previous example, the last function overloading given will be executed and is valid. This is same for the constructor overloading.

* For operators, overloading is possible.

↳ '+' operator adds two numbers and concatenate two strings.

↳ '*' operator multiplies two numbers and when used with a string and int, repeats the string given int times and concatenate them.

Constructor Overloading

I/P - class myclass():

```
    def __init__(self, a, b):
        Print("parameterized")
    def __init__(self):
        Print("Default constructor")
M1 = myclass()  # this is valid & executed.
M2 = myclass(1, 2)
```

O/P - Default constructor.

2. Overriding:

It means having two methods with the same name but doing different tasks. It means that one of the methods overrides the other.

I/P -

```
class A:
    def m1(self):
        Print("class-A: Method-1")
    def m2(self):
        Print("class-A: Method-2")
```

```
class B(A):
    def m1(self):
        Print("class - B : Method - 1")

b = B()
b.m1()
```

O/p - class - B : Method - 1

The Overriding is classified into 2 categories:

(i) Function overriding

(ii) Constructor overriding.

•

Super() Function:

This function is used to give access to methods and properties of a parent or sibling class.

↳ This returns an object that represents the parent class.

SYNTAX: super() # Goes to parents call.

I/p -

```
class A:
    def __init__(self):
        Print("parent constructor")

class B(A):
    def __init__(self):
        super().__init()
        Print("child constructor")

b = B()
```

O/p - parent constructor
child constructor.

Overriding is possible in python but Overloading is not possible in python except for operators

ADVANTAGES OF POLYMORPHISM: (120)

↳ The code and classes written once can be reused and implemented multiple times.

↳ It helps in reducing the coupling between different functionalities and behaviour of objects.