INTRODUCTION TO OOPS

Procedure oriented approach

The main and sub tasks are represented by functions and procedures.
Ex: C, Pascal, FORTRAN.

Object oriented approach

The main and sub tasks are represented by classes.
Ex: C++, Java, Python.

Differences between POA and OOA

POA	*OOA*
1. There is no code reusability. For every new task, the programmer needs to develop a new function.	We can create sub classes to existing classes and reuse them.
2. One function may call on another function. Hence debugging becomes difficult as we have to check every function.	Every class is independent and hence it can be debugged without disturbing other classes.
3. This approach is not developed from human being's life and hence learning and using it is very difficult.	Developed from human being's life and hence easy to understand and handle.
4. Programmers lose control at a particular point when the code size is between 10,000 to 1,00,000 lines. Hence not suitable for bigger and complex projects.	Suitable to handle bigger and complex projects.

Features of OOPS
1. classes and objects
2. encapsulation
3. abstraction
4. inheritance
5. polymorphism

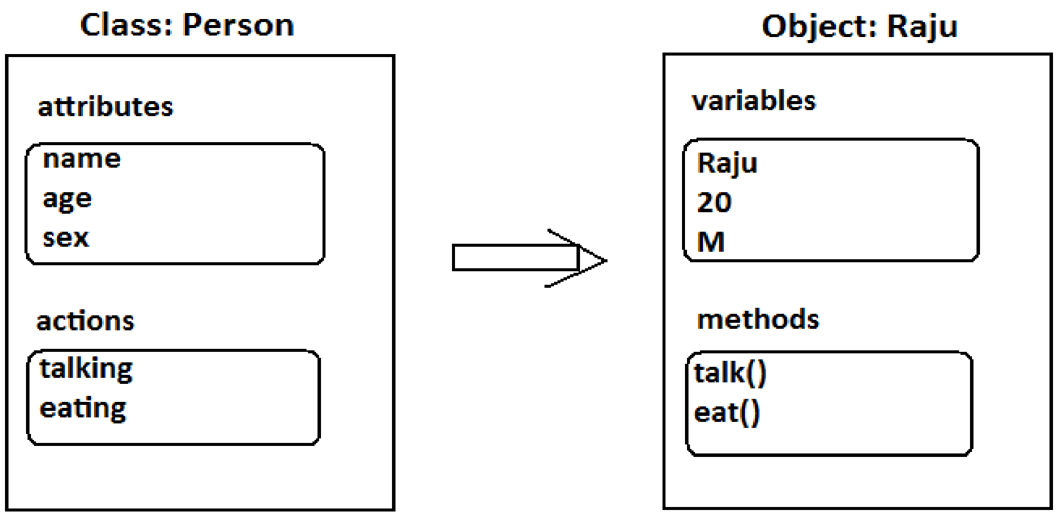
Classes and objects

An object is anything that really exists in the world. Object contains behavior -> attributes and actions -> variables and methods.

A group of objects having same behavior belong to same class or category.

A class is a model for creating objects. An object exists physically but a class does not exist physically. Class also contains variables and methods.

Def: A class is a specification of behavior of a group of objects.
Def: An object is an instance (physical form) of a class.



PROGRAMS
1. Create Person class and object to it.

Self variable

'self' is a default variable that contains the memory address of the instance of the current class. So, we can use 'self' to refer to all the instance variables and instance methods.

Constructor

A constructor is a special method that is used to initialize the instance variables of a class. In the constructor, we create the instance variables and initialize them with some starting values. The first parameter of the constructor will be 'self' variable that contains the memory address of the instance. A constructor may or may not have parameters.

Ex:
```
def __init__(self):       # default constructor
      self.name = 'Vishnu'
      self.marks = 900
```

Ex:
```
def __init__(self, n = '', m=0):       # parameterized constructor with 2 parameters
      self.name = n
      self.marks = m
```

PROGRAMS
2. Create Student class with parameterized constructor.

Types of variables

The variables which are written inside a class are of 2 types:

- Instance variables

- Class variables or Static variables

Instance variables are the variables whose separate copy is created in every instance (or object). Instance variables are defined and initialized using a constructor with 'self' parameter. Also, to access instance variables, we need instance methods with 'self' as first parameter. Instance variables can be accessed as: obj.var

Unlike instance variables, class variables are the variables whose single copy is available to all the instances of the class. If we modify the copy of class variable in an instance, it will modify all the copies in the other instances. A class method contains first parameter by default as 'cls' with which we can access the class variables. For example, to refer to the class variable 'x', we can use 'cls.x'.

NOTE: class variables are also called 'static variables'. class methods are marked with the decorator @classmethod .
NOTE: instance variables can be accessed as: obj.var or classname.var

PROGRAMS
3. Demonstrate class variables and class methods.

Namespaces

A *namespace* represents a memory block where names are mapped (or linked) to objects. A class maintains its own namespace, called 'class namespace'. In the class namespace, the names are mapped to class variables. Similarly, every instance (object) will have its own name space, called 'instance namespace'. In the instance namespace, the names are mapped to instance variables.

When we modify a class variable in the class namespace, its modified value is available to all instances. When we modify a class variable in the instance namespace, then it is confined to only that instance. Its modified value will not be available to other instances.

Types of methods

By this time, we got some knowledge about the methods written in a class. The purpose of a method is to process the variables provided in the class or in the method. We already know that the variables declared in the class are called class variables (or static variables) and the variables declared in the constructor are called instance variables. We can classify the methods in the following 3 types:

- Instance methods
  (a) Accessor methods
  (b) Mutator methods
- Class methods
- Static methods

An instance method acts on instance variables. There are two types of methods.
1. Accessor methods: They read the instance vars. They do not modify them. They are also called getter() methods.
2. Mutator methods: They not only read but also modify the instance vars. They are also called setter() methods.

PROGRAMS
4. Create getter and setter methods for a Manager with name and salary instance variables.

Static methods

We need static methods when the processing is at class level but we need not involve the class or instances. Static methods are used when some processing is related to the class but does not need the class or its

instances to perform any work. For example, setting environmental variables, counting the number of instances of the class or changing an attribute in another class etc. are the tasks related to a class. Such tasks are handled by static methods. Static methods are written with a decorator @staticmethod above them. Static methods are called in the form of classname.method().

PROGRAMS
5. Create a static method that counts the number of instances of a class.

Inner classes

Writing a class within another class is called inner class or nested class. For example, if we write class B inside class A, then B is called inner class or nested class. Inner classes are useful when we want to sub group the data of a class.

PROGRAMS
6. Create Dob inner class in Student outer class.

Encapsulation

Bundling up of data and methods as a single unit is called 'encapsulation'. A class is an example for encapsulation.

Abstraction

Hiding unnecessary data from the user is called 'abstraction'. By default all the members of a class are 'public' in Python. So they are available outside the class. To make a variable private, we use double underscore before the variable. Then it cannot be accessed from outside of the class. To access it from outside the class, we should use: obj._Classname__var. This is called name mangling.

PROGRAMS
7. Using name mangling, access the private variable from outside of Bank class.

Inheritance

Creating new classes from existing classes in such a way that all the features of the existing classes are available to the newly created classes – is called 'inheritance'. The existing class is called 'base class' or 'super class'. The newly created class is called 'sub class' or 'derived class'.

Sub class object contains a copy of the super class object. The advantage of inheritance is 'reusability' of code. This increases the overall performance of the organization.

Syntax: class Subclass(Baseclass):

PROGRAMS
8. Create Teacher class and use it.
9. Create Student class and use it. Use inheritance to create Student class.

Constructors in inheritance

In the previous programs, we have inherited the Student class from the Teacher class. All the methods and the variables in those methods of the Teacher class (base class) are accessible to the Student class (sub class). The constructors of the base class are also accessible to the sub class.

When the programmer writes a constructor in the sub class, then only the sub class constructor will get executed. In this case, super class constructor is not executed. That means, the sub class constructor is replacing the super class constructor. This is called *constructor overriding.*
PROGRAMS
10. Override the constructor and method of Father class in Son's class.

super() method

super() is a built-in method which is useful to call the super class constructor or methods from the sub class.

```
super().__init__()     # call super class constructor
super().__init__(arguments)    # call super class constructor and pass arguments
super().method()   # call super class method
```

PROGRAMS
11. Derive Rectangle class from Square class. Access the constructor and method of super class in the sub class using super().

Types of inheritance

There are two types:
1. Single inheritance: deriving sub class from a single super class.
Syntax: class Subclass(Baseclass):

2. Multiple inheritance: deriving sub class from more than one super class.
Syntax: class Subclass(Baseclass1, Baseclass2, …):

NOTE: 'object' is the super class for all classes in Python.

PROGRAMS
12. Derive Child class from both Father and Mother class.

Polymorphism

poly + morphos = many + forms
If something exists in various forms, it is called 'Polymorphism'. If an operator or method performs various tasks, it is called polymorphism.

Ex:

Duck typing: Calling a method on any object without knowing the type (class) of the object.
Operator overloading: same operator performing more than one task.
Method overloading: same method performing more than one task.
Method overriding: executing only sub class method in the place of super class method.

PROGRAMS
13. Example of how to call a method on various objects.
14. Show how + operator can be overloaded.
15. Write a method to perform sum of two or three numbers.
16. Override base class method in sub class.

ABSTRACT CLASSES AND INTERFACES

An abstract method is a method whose action is redefined in the sub classes as per the requirement of the objects. Generally abstract methods are written without body since their body will be defined in the sub classes

anyhow. But it is possible to write an abstract method with body also. To mark a method as abstract, we should use the decorator @abstractmethod. On the other hand, a concrete method is a method with body.

An abstract class is a class that generally contains some abstract methods. PVM cannot create objects to an abstract class.
Once an abstract class is written, we should create sub classes and all the abstract methods should be implemented (body should be written) in the sub classes. Then, it is possible to create objects to the sub classes.

A meta class is a class that defines the behavior of other classes. Any abstract class should be derived from the meta class ABC that belongs to 'abc' module. So import this module, as:

```
from abc import ABC, abstractmethod
(or) from abc import *
```

PROGRAMS
17. A program to create abstract class and sub classes which implement the abstract method of the abstract class.

Interfaces in Python

We learned that an abstract class is a class which contains some abstract methods as well as concrete methods also. Imagine there is a class that contains only abstract methods and there are no concrete methods. It becomes an interface. This means an interface is an abstract class but it contains only abstract methods. None of the methods in the interface will have body. Only method headers will be written in the interface. So an interface can be defined as a specification of method headers. Since, we write only abstract methods in the interface, there is possibility for providing different implementations (body) for those abstract methods depending on the requirements of objects. In Python, we have to use abstract classes as interfaces.

Since an interface contains methods without body, it is not possible to create objects to an interface. In this case, we can create sub classes where we can implement all the methods of the interface. Since the sub classes will have all the methods with body, it is possible to create objects to the sub classes. The flexibility lies in the fact that every sub class can provide its own implementation for the abstract methods of the interface.

PROGRAMS
18. Create an interface that connects to any database.

LIST OF PROGRAMS (ADVANCED PYTHON)

```
# 1.a class and object
class Person(object):
    # special method to initialize the vars
    # this is called constructor
    def __init__(self):
        self.name = 'Vishnu'
        self.age = 20
        self.sex = 'M'

    # instance method
    def talk(self):
        print('Hi, iam ', self.name)
        print('My age is ', self.age)

# create an instance
p1 = Person()

# call the method
p1.talk()

# to display instance variables through object
print(p1.name)
print(p1.age)
print(p1.sex)
```

```
# 2.student class example
class Student(object):
    # parameterized constructor
    def __init__(self, n='', m=[]):
        self.name = n
        self.marks = m

    # instance method
    def total(self):
        print('Name: ', self.name)
        tot = sum(self.marks)
        print('Total marks: ', tot)

# create two instances
s1 = Student('Lakshmi', [10, 20, 30])
s1.total()
```

```
s2 = Student()
s2.total()
```

```
# 3.class vars and class methods
class Sample:
    x = 10

    @classmethod
    def modify(cls):
        cls.x+=1

# create two instances
s1 = Sample()
s2 = Sample()
print(s1.x, s2.x)

# modify s1 in the class namespace
s1.modify()
print(s1.x, s2.x)

# modify s1 in the instance namespace
s1.x+=1
print(s1.x, s2.x)
```

```
# 4.accessor and mutator methods
class Manager:
    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

    def setSal(self, sal):
        self.sal = sal

    def getSal(self):
        return self.sal

# create object to Manager class
m = Manager()
m.setName('Raju')
print('Manager name= %s'%m.getName())

m.setSal(12000.50)
print('His salary= %.2f'% m.getSal())
```

```
# 5.understanding static methods
class Myclass:
    # this is class var or static var
    n=0

    # constructor that increments n when an instance is created
    def __init__(self):
        Myclass.n +=1

    # this is a static method to display the no. of instances
    @staticmethod
    def noObjects():
        print('No. of instances created: ', Myclass.n)

# create 3 instances
obj1 = Myclass()
obj2 = Myclass()
obj3 = Myclass()
Myclass.noObjects()
```

```
# 6.this is outer class
class Student:
    def __init__(self):
        self.name = 'Srinu'

    def display(self):
        print('Name= ', self.name)
    # inner class
    class Dob:
        def __init__(self):
            self.dd = 10
            self.mm = 5
            self.yy = 1990

        def display(self):
            print('Dob= {}/{}/{}'.format(self.dd, self.mm, self.yy))

# create outer class instance
s = Student()
s.display()

# create inner class instance
x = Student().Dob()
x.display()
```

```
# 7.Bank class with a private variable 'loan'.
class Bank:
    def __init__(self):
        self.accno=1001
        self.name='Ganesh'
        self.bal = 5000.0
        self.__loan = 1500000.00

    def display_to_clerk(self):
        print('Accno= ', self.accno)
        print('Name= ', self.name)
        print('Balance= ', self.bal)


# create instance to the class
b = Bank()
b.display_to_clerk()

print(b.accno)
print(b.name)
print(b.bal)
print(b._Bank__loan)  # name mangling
```

```
# 8. save this as teacher.py
class Teacher:
    def setId(self, id):
        self.id = id

    def getId(self):

        return self.id

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

    def setAddress(self, addr):
        self.addr = addr

    def getAddress(self):
        return self.addr
    def setSal(self, sal):
        self.sal = sal

    def getSal(self):
        return self.sal

# use the Teacher class
from teacher import Teacher
t = Teacher()

t.setId(10)
print(t.getId())

t.setName('Ramu')
print(t.getName())

t.setAddress('HNO-15, Arandal pet, Guntur')
print(t.getAddress())

t.setSal(5000.55)
print(t.getSal())
```

```
# 9.save this as student.py
class Student:
    def setId(self, id):
        self.id = id

    def getId(self):
        return self.id

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

    def setAddress(self, addr):
        self.addr = addr

    def getAddress(self):
        return self.addr

    def setMarks(self, marks):
        self.marks = marks

    def getMarks(self):
        return self.marks

# use the Student class

from student import Student
s = Student()

s.setId(10)
print(s.getId())

s.setName('Ramu')
print(s.getName())

s.setAddress('HNO-15, Arandal pet, Guntur')
print(s.getAddress())

s.setMarks(988)
print(s.getMarks())

# derive Student class from Teacher class - v2.0
from teacher import Teacher
class Student(Teacher):
    def setMarks(self, marks):
        self.marks = marks

    def getMarks(self):
        return self.marks
```

```
# 10.overring the base class constructor and method in sub class
# cut the contructor in Son class and re-execute this program
class Father:
    def __init__(self):
        self.property = 800000.00

    def display_property(self):
        print('Father\'s property= ', self.property)

class Son(Father):
    def __init__(self):
        self.property = 200000.00

    def display_property(self):
        print('Child\'s property= ', self.property)

# create sub class instance and display father's property
s = Son()
s.display_property()
```

```
# 11.Accessing base class constructor and method in the sub class
class Square:
    def __init__(self, x):
        self.x = x
    def area(self):
        print('Area of square= ', self.x*self.x)

class Rectangle(Square):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y

    def area(self):
        super().area()

print('Area of rectange= ', self.x*self.y)

# find areas of square and rectanlge
a, b = [float(x) for x in input("Enter two measurements: ").split()]
r = Rectangle(a,b)
r. area()
```

```
# 12.multiple inheritance
class Father:
    def height(self):
        print('Height is 6.0 foot')

class Mother:
    def color(self):
        print('Color is brown')

class Child(Father, Mother):
    pass

c = Child()
print('Child\'s inherited qualities: ')
c.height()
c.color()
```

```
# 13.duck typing example
class Duck:
    def talk(self):
        print('Quack, quack!')

class Human:
    def talk(self):
        print('Hello, hai!')

# this method accepts an object and calls talk() method
def call_talk(obj):
    obj.talk()

# call call_talk() method and pass an object
# depending on type of object, talk() method is executed
x = Duck()
call_talk(x)
x = Human()
call_talk(x)
```

```
# 14.+ operator is already overloaded on numbers, strings and lists
a = 10
```

```
b = 20
print(a+b)  # addition

a = "Hello"
b = "Student"
print(a+b)   # concatenation

a = [10, 20, 30]
b = [ -5, 10, 3]
print(a+b)   # making a single list
```

```
# 15.method overloading
class Myclass:
    def sum(self, a=0, b=0, c=0):
        print('Sum= ', a+b+c)
# overload the sum() method
m = Myclass()
m.sum(10, 15)
m.sum(10, 15, 22.5)
```

```
# 16.method overriding
class A:
    def display(self):
        print('Iam from base class')

class B(A):
    # override the display() method of A
    def display(self):
        print('Iam from sub class')

b = B()
b.display()
```

```
# 17.abstract class example
from abc import ABC, abstractmethod

class Myclass(ABC):
    @abstractmethod
    def calculate(self, x):
        pass   # empty body, no code

# this is sub class of Myclass
class Sub1(Myclass):
    def calculate(self, x):
        print('Square value= ', x*x)

# this is another sub class for Myclass
import math
class Sub2(Myclass):
    def calculate(self, x):
        print('Square root= ', math.sqrt(x))
# third sub class for Myclass
class Sub3(Myclass):
    def calculate(self, x):
        print('Cube value= ', x**3)

# create Sub1 class object and call calculate() method
obj1 = Sub1()
obj1.calculate(16)
```

```
# create Sub2 class object and call calculate() method
obj2 = Sub2()
obj2.calculate(16)

# create Sub3 class object and call calculate() method
obj3 = Sub3()
obj3.calculate(16)
```

```
# 18.abstract class works like an interface
from abc import *
class Myclass(ABC):
    def connect(self):
        pass

# this is a sub class
class Oracle(Myclass):
    def connect(self):
        print('Connecting to Oracle database...')

# this is another sub class
class Sybase(Myclass):
    def connect(self):
        print('Connecting to Sybase database...')

class Database:
    # accept database name as a string
    str = input('Enter database name: ')

    # convert the string into classname
    classname = globals()[str]

    # create an object to that class
    x = classname()

    # call the connect() method
    x.connect()
```
