BASICS OF OBJECT ORIENTED PROGRAMMING

PREPARED AND COMPILED BY:

D.R. GANDHI

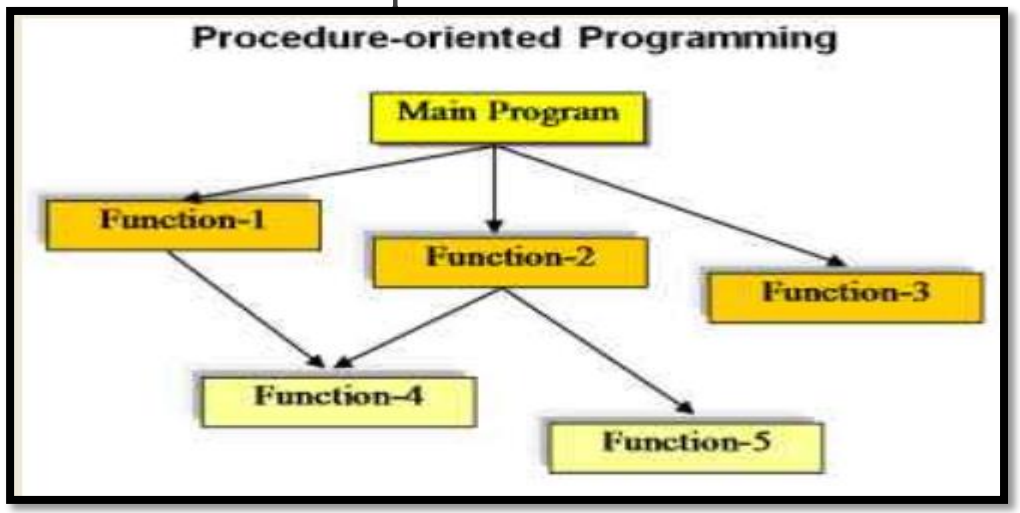
LECTURER IN INFORMATION TECHNOLOGY

LEARNING OUTCOMES:

- Oops Concepts
- Class and Object
- Constructors
- Types of methods
  - Instance method
  - Class method
  - static method
- Data Encapsulation
- Inheritance - single, multiple, multi-level, hierarchical, hybrid
- Polymorphism through inheritance
- Abstraction - abstract class

PROCEDURAL PROGRAMMING:

- It is defined as a programming language derived from the structure programming and based on calling procedures.
- The procedures are the functions, routines, or subroutines that consist of the computational steps required to be carried.
- It follows a step-by-step approach in order to break down a task into a set of variables and routines via a sequence of instructions.

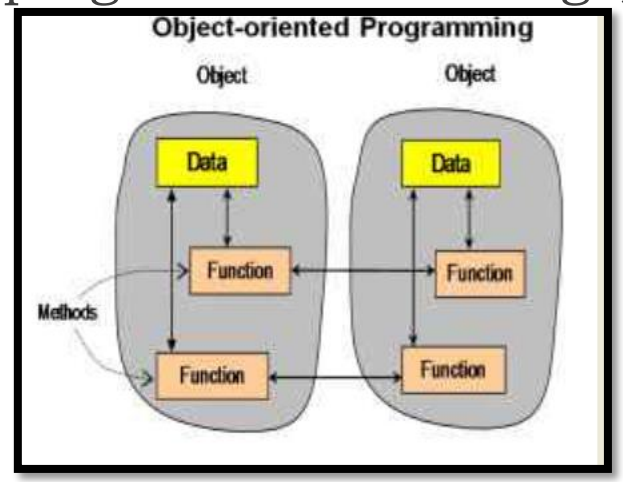


PROCEDURAL PROGRAMMING:

- **Example of make lemonade:-**
- The procedure of making lemonade involves- first taking water according to the need, then adding sugar to the water, then adding lemon juice to the mixture, and finally mixing the whole solution. And your lemonade is ready to serve.
- In a similar way, POP requires a certain procedure of steps. A procedural program consists of functions.
- This means that in the POP approach the program is divided into functions, which are specific to different tasks.
- These functions are arranged in a specific sequence and the control of the program flows sequentially.
- Procedural programming follows a top-down approach during the designing of a program.

OBJECT ORIENTED PROGRAMMING:

- Object-Oriented Programming(OOP), is all about creating “objects”.
- An object is a group of interrelated variables and functions.
- These variables are often referred to as properties of the object and functions are referred to as the behavior of the objects. These objects provide a better and clear structure for the program.
- It is well suited for programs that are large, complex, and actively updated or maintained.



OBJECT ORIENTED PROGRAMMING:

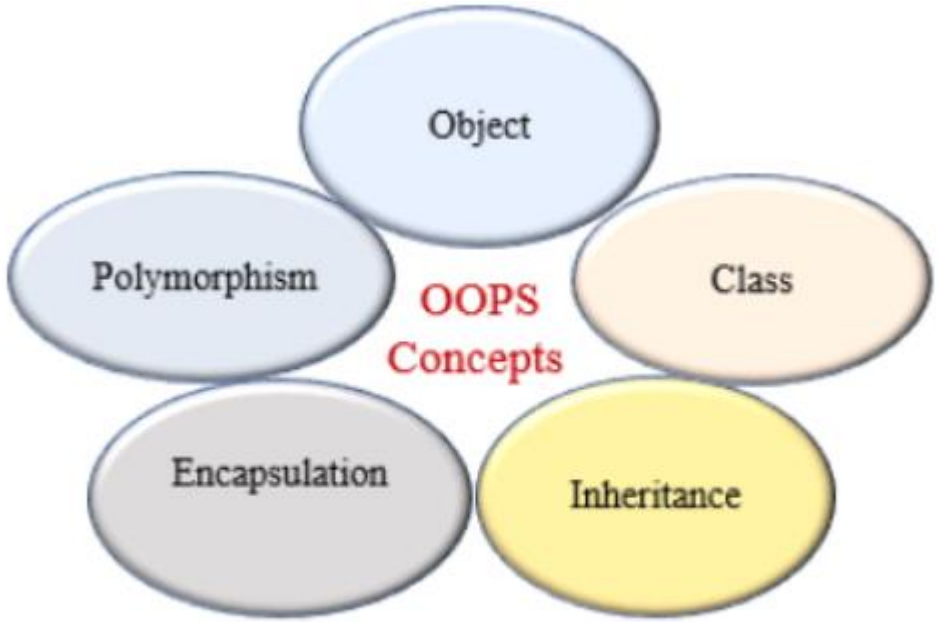
- For example, a car can be an object.
- If we consider the car as an object then its properties would be – its color, its model, its price, its brand, etc. And its behavior/function would be acceleration, slowing down, gear change.
- For example, a dog can be an object.
- If we consider a dog as an object then its properties would be- his color, his breed, his name, his weight, etc. And his behavior/function would be walking, barking, playing, etc.
- Object-Oriented programming is famous because it implements the real-world entities like objects, hiding, inheritance, etc in programming. It makes visualization easier because it is close to real-world scenarios

POP VS. OOP:

Procedural Programming	Object-oriented Programming
Procedural programming uses a list of instructions to do computation step by step.	Object-oriented programming is the problem-solving approach and used where computation is done by using objects.
It is not easy to maintain the codes when the project becomes lengthy.	It makes the development and maintenance easier.
It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.	It simulates the real world entity. So real-world problems can be easily solved through oops.
Procedural language doesn't provide any proper way for data binding, so it is less secure.	It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.
It follows a top-down approach.	It follows a bottom-up approach.
Example of procedural languages are: C, Fortran, Pascal, VB etc.	Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.

OOP CONCEPTS

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction



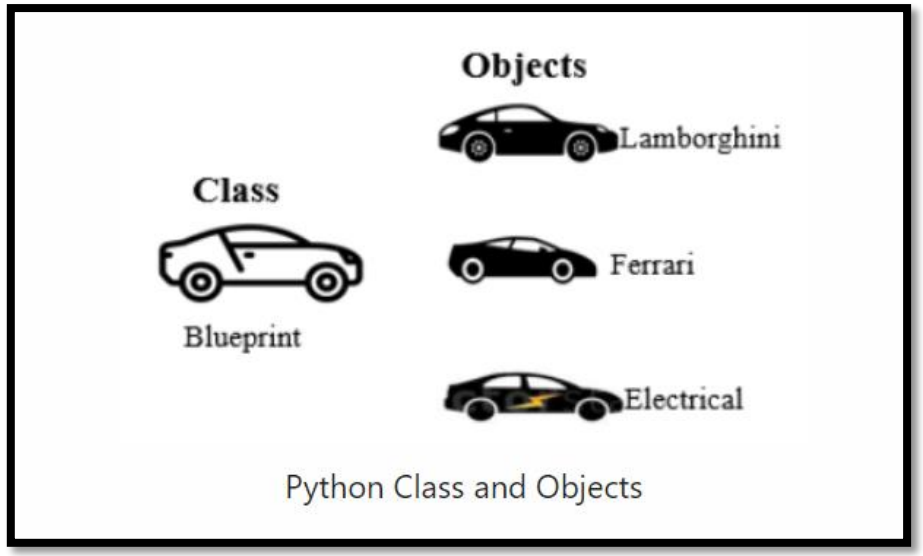
Python OOP concepts

OOP CONCEPTS

- Class:-
- The class can be defined as a collection of objects.
- It is a logical entity that has some specific attributes and methods.
- 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.
- A class contains the properties (attribute) and action (behavior) of the object.
- Properties represent variables, and the methods represent actions. Hence class includes both variables and methods.

OOP CONCEPTS

- Object:-
- Object is an instance of a class.
- 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.
- It may be any real-world object like the mouse, keyboard, laptop, etc.



Python Class and Objects

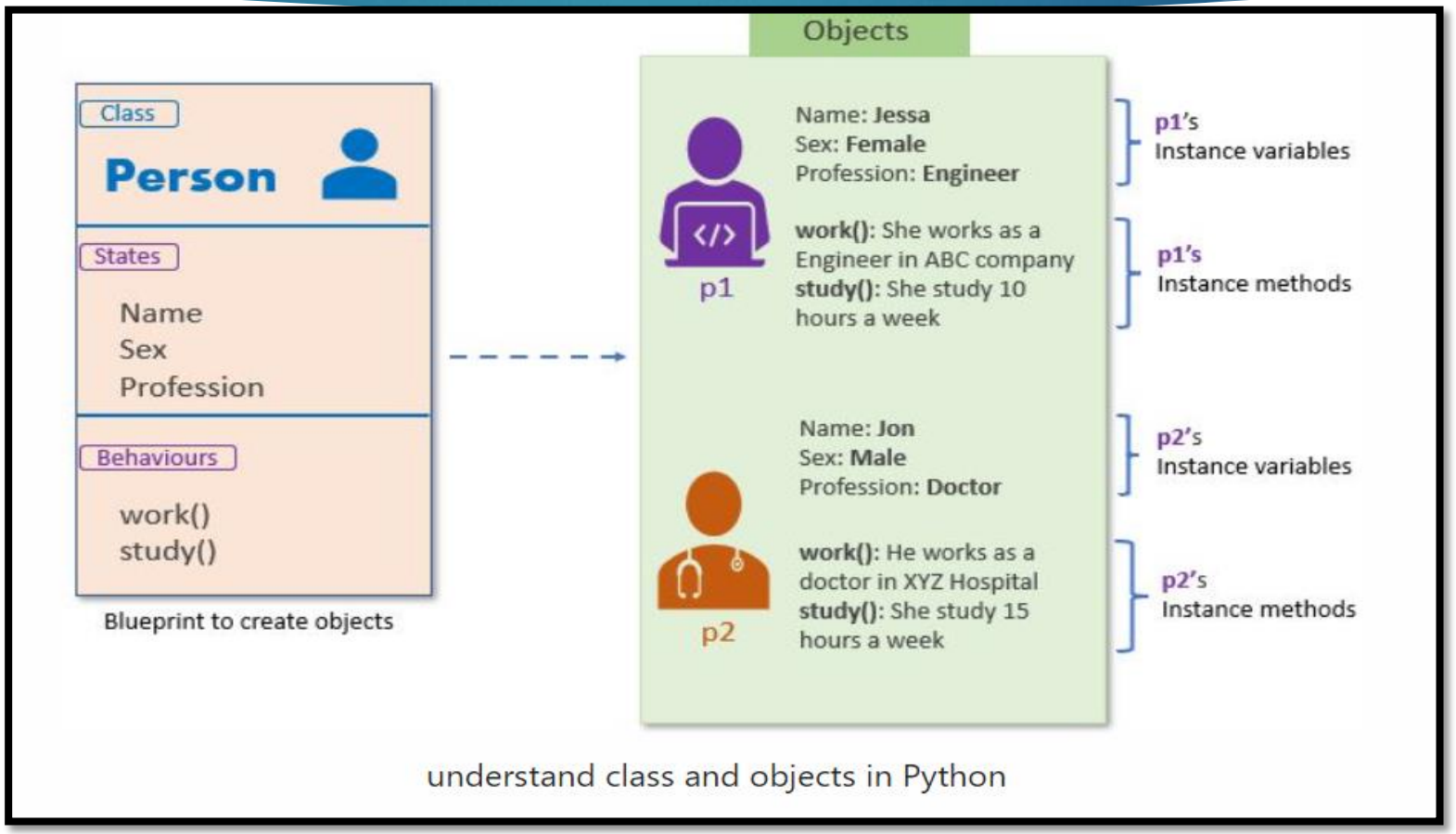
OOP CONCEPTS

- Objects have two characteristics: They have states and behaviors (object has attributes and methods attached to it) Attributes represent its state, and methods represent its behavior. Using its methods, we can modify its state.
- In short, Every object has the following property.
- **Identity:** Every object must be uniquely identified.
- **State**: An object has an attribute that represents a state of an object, and it also reflects the property of an object.
- **Behavior**: An object has methods that represent its behavior.

OOP CONCEPTS

- Python is an Object-Oriented Programming language, so everything in Python is treated as an object.
- An object is a real-life entity.
- It is the collection of various data and functions that operate on those data.
- For example, If we design a class based on the states and behaviors of a Person, then States can be represented as instance variables and behaviors as class methods.

OOP CONCEPTS



OOP CONCEPTS

- A real-life example of class and objects.
- Class: Person
- State: Name, Sex, Profession
- Behavior: Working, Study

- **objects are created from the same class, but they have different states and behaviors.**

Object 1: Jessa State: Name: Jessa Sex: Female Profession: Software Engineer Behavior: Working: She is working as a software developer at ABC Company Study: She studies 2 hours a day	Object 2: Jon State: Name: Jon Sex: Male Profession: Doctor Behavior: Working: He is working as a doctor Study: He studies 5 hours a day

OOP CONCEPTS

- Create a Class in Python:
- In Python, class is defined by using the class keyword.
- Syntax:

```
class class_name:
    '''This is a docstring. I have created a new class'''
    <statement 1>
    .........
    <statement N>
```

- **class_name**: It is the name of the class
- **Docstring**: It is the first string inside the class and has a brief description of the class. Although not mandatory, this is highly recommended.
- **statements**: Attributes and methods

OOP CONCEPTS

Example:-

```
class car:
    def __init__(self,modelname, year):
     # data members (instance variables)
        self.modelname = modelname
        self.year = year
     # Behavior (instance methods)
    def display(self):
        print(self.modelname,self.year)
c1 = car("Toyota", 2016)
c1.display()
```

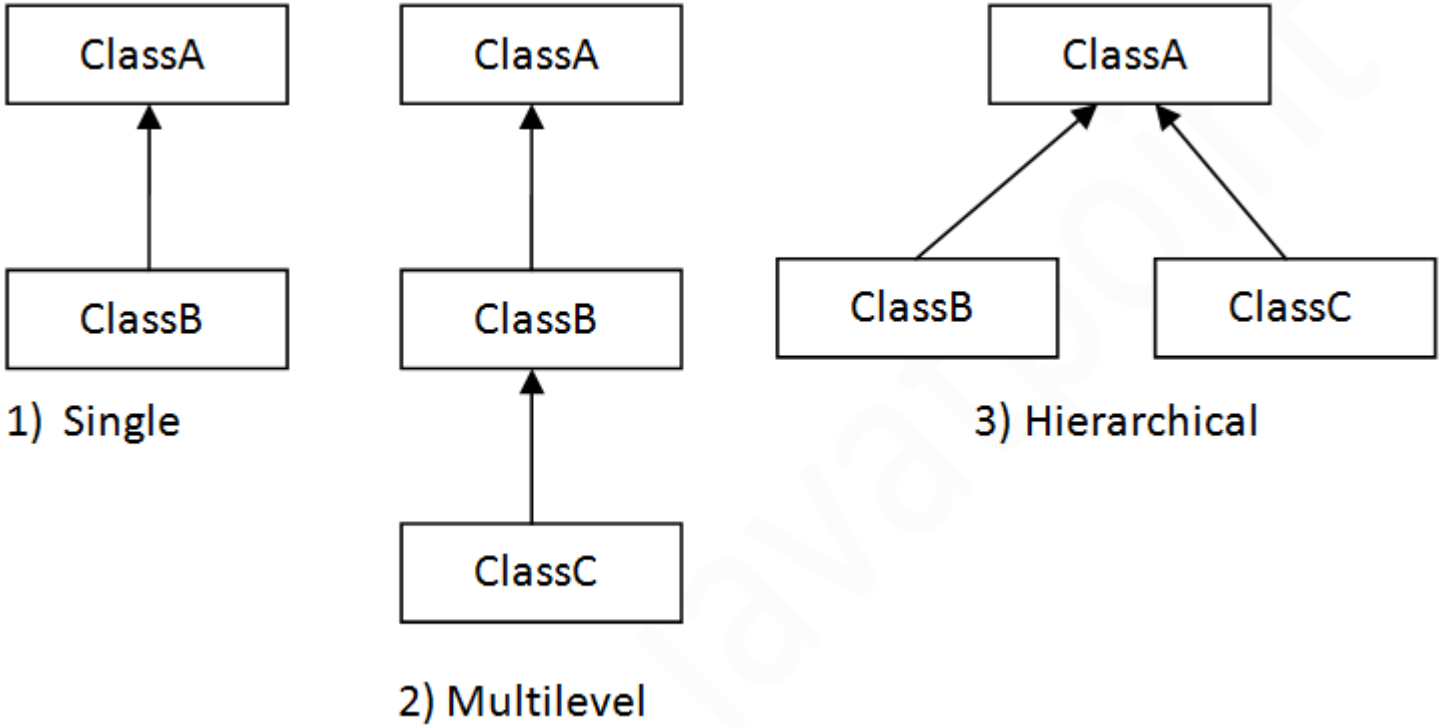
Syntax to create object:-
<object-name> = <class-name>(<arguments>)

OOP CONCEPTS

- **Inheritance:-**
- Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance.
- It specifies that the child object acquires all the properties and behaviors of the parent object.
- By using inheritance, we can create a class which uses all the properties and behavior of another class.
- The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.
- It provides the re-usability of the code.

OOP CONCEPTS

- **Inheritance:-**

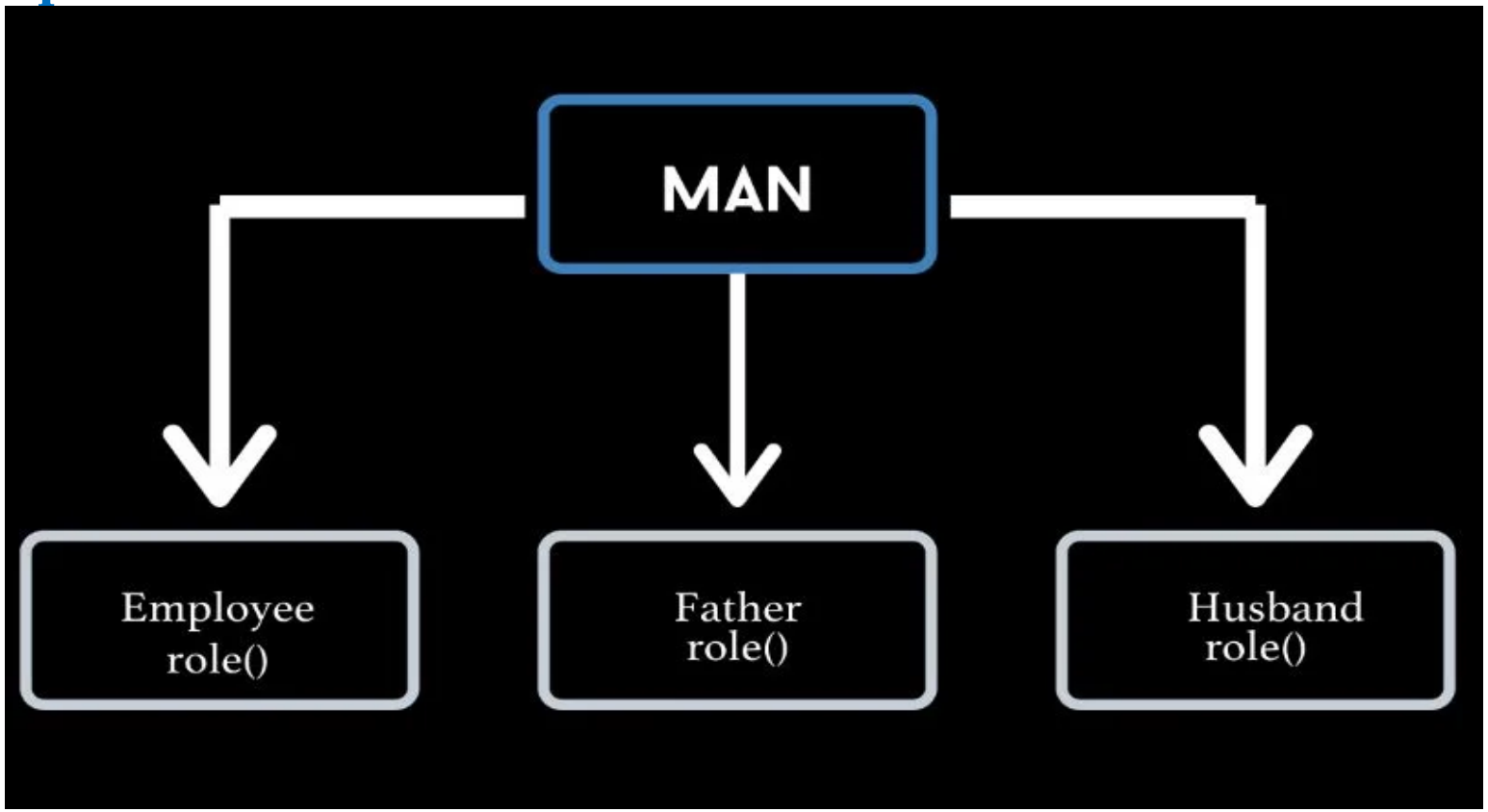


OOP CONCEPTS

- **Polymorphism:-**
- Polymorphism contains two words "poly" and "morphs".
- Poly means many, and morph means shape.
- By polymorphism, we understand that one task can be performed in different ways.
- For example - you have a class animal, and all animals speak. But they speak differently.
- Here, the "speak" behavior is polymorphic in a sense and depends on the animal.
- So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

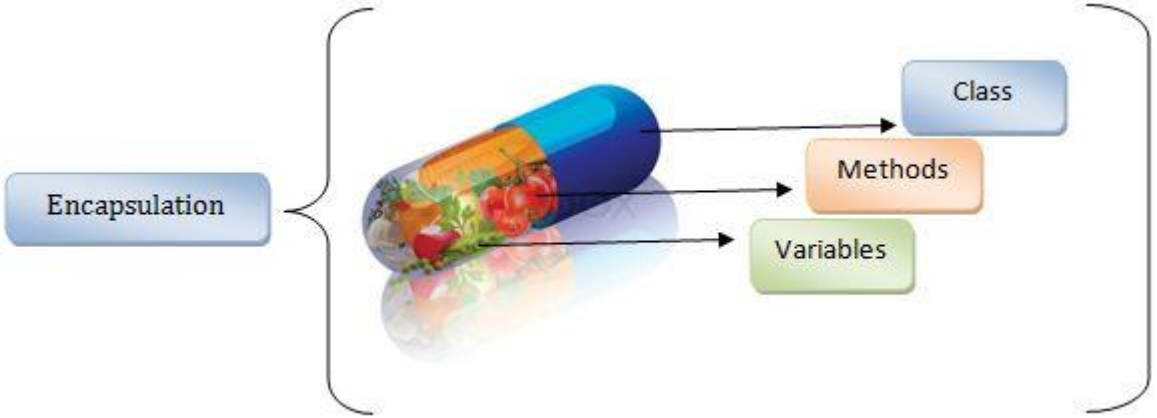
OOP CONCEPTS

- **Polymorphism:-**



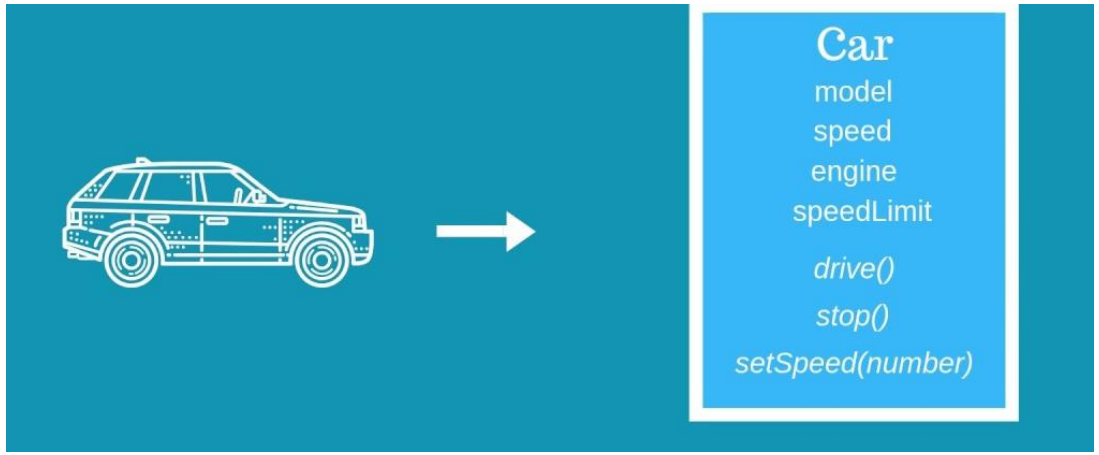
OOP CONCEPTS

- **Encapsulation:-**
- Encapsulation is also an essential aspect of object-oriented programming.
- It is used to restrict access to methods and variables.
- In encapsulation, code and data are wrapped together within a single unit from being modified by accident.



OOP CONCEPTS

- **Abstraction:-**
- Data abstraction and encapsulation both are often used as synonyms.
- Both are nearly synonyms because data abstraction is achieved through encapsulation.
- Abstraction is used to hide internal details and show only functionalities.
- Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.



INIT()

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

- The properties that all Human objects must have, are defined in a method called init(). Every time a new Human object is created, __init__() sets the initial state of the object by assigning the values of we provide inside the object's properties. That is, __init__() initializes each new instance of the class.
- __init__() can take any number of parameters, but the first parameter is always a variable called self.
- The self parameter is a reference to the current instance of the class. It means, the self parameter points to the address of the current object of a class, allowing us to access the data of its(the object's) variables.

class_1.py - D:\dhavalPC_backup_9-10-18\IT_15\Term-221\4331601\Python_prog\class\class_1.py (3.10.1)

File Edit Format Run Options Window Help

```
class Person:
    def __init__(self, name, gender, profession):
        # data members (instance variables)
        self.name = name
        self.gender = gender
        self.profession = profession

    # Behavior (instance methods)
    def show(self):
        print('Name:', self.name, 'Gender:', self.gender, 'Profession:', self.profession)

    # Behavior (instance methods)
    def work(self):
        print("name of Person:",self.name)
        print("Gender of a person :",self.gender)
        print("Person is working as a:", self.profession)
# create object of a class
jessa = Person('Jessa', 'Female', 'Software Engineer')

# call methods
jessa.show()
jessa.work()
```

Name: Jessa Gender: Female Profession: Software Engineer
name of Person: Jessa
Gender of a person : Female
Person is working as a: Software Engineer

CONSTRUCTOR:

- In object-oriented programming, A constructor is a special method used to create and initialize an object of a class. This method is defined in the class.
- The constructor is executed automatically at the time of object creation.
- The primary use of a constructor is to declare and initialize data member/ instance variables of a class. The constructor contains a collection of statements (i.e., instructions) that executes at the time of object creation to initialize the attributes of an object.
- For example, when we execute obj = Sample(), Python gets to know that obj is an object of class Sample and calls the constructor of that class to create an object.
- In Python, Object creation is divided into two parts in **Object Creation** and **Object initialization.**
- Internally, the **__new__** is the method that creates the object
- the **__init__()** method we can implement constructor to initialize the object.

CONSTRUCTOR:

- **Syntax of a constructor:-**

```
def __init__(self):
    # body of the constructor
```

- **def:** The keyword is used to define function.
- **__init__() Method:** It is a reserved method. This method gets called as soon as an object of a class is instantiated.
- **self:** The first argument self refers to the current object. It binds the instance to the __init__() method. It's usually named self to follow the naming convention.
- **Note:** The __init__() method arguments are optional. We can define a constructor with any number of arguments.

EXAMPLE: CREATE A CONSTRUCTOR IN PYTHON

class_constructor_1.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Py...

File Edit Format Run Options Window Help

```
class Student:
    # constructor
    # initialize instance variable
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('All variables initialized')
    # instance Method
    def show(self):
        print('Hello, my name is', self.name)
# create object using constructor
s1 = Student('Emma')
s1.show()
```

Ln: 14 Col: 0

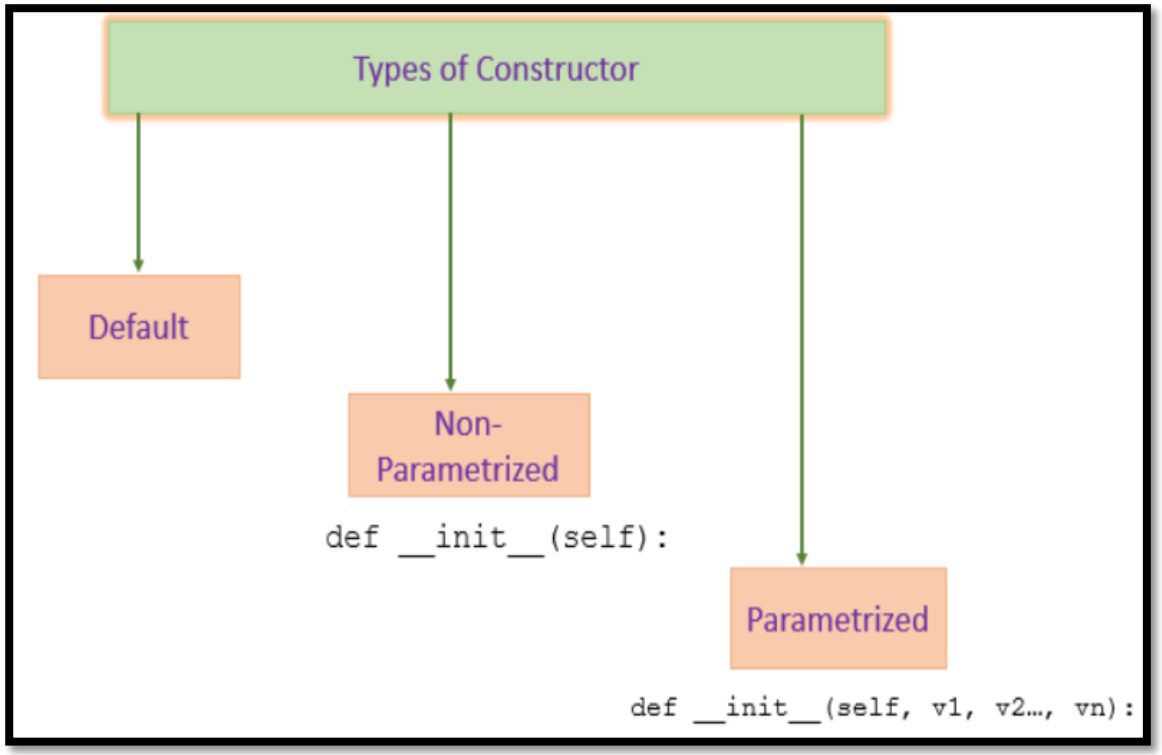
Inside Constructor
All variables initialized
Hello, my name is Emma

CONSTRUCTOR:

- **Note:**
- For every object, the constructor will be executed only once. For example, if we create four objects, the constructor is called four times.
- In Python, every class has a constructor, but it's not required to define it explicitly. Defining constructors in class is optional.
- Python will provide a default constructor if no constructor is defined.

CONSTRUCTOR:

- **Types of Constructors:-**
- In Python, we have the following three types of constructors.
- **Default Constructor**
- **Non-parametrized constructor**
- **Parameterized constructor**



CONSTRUCTOR:

- **Default Constructor:-**
- Python will provide a default constructor if no constructor is defined. Python adds a default constructor when we do not include the constructor in the class or forget to declare it. It does not perform any task but initializes the objects. It is an empty constructor without a body.
- If you do not implement any constructor in your class or forget to declare it, the Python inserts a default constructor into your code on your behalf. This constructor is known as the default constructor.
- It does not perform any task but initializes the objects. It is an empty constructor without a body.
- **Note:**
- If you implement your constructor, then the default constructor will not be added.

CONSTRUCTOR:

class_constructor_default.py - D:\dhavalPC_backup_9-10-18\IT_15\Term-221\4331601\...

File Edit Format Run Options Window Help

```
class Employee:

    def display(self):
        print('Inside Display')

emp = Employee()
emp.display()
```

Output:
Inside Display

CONSTRUCTOR:

- **Non-Parametrized Constructor:-**
- A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values.
- This constructor doesn't accept the arguments during object creation.
- Instead, it initializes every object with the same set of values.

CONSTRUCTOR:

File Edit Format Run Options Window Help

```
class Company:

    # no-argument constructor
    def __init__(self):
        self.name = "PYnative"
        self.address = "ABC Street"

    # a method for printing data members
    def show(self):
        print('Name:', self.name, 'Address:', self.address)

# creating object of the class
cmp = Company()

# calling the instance method using the object
cmp.show()
```

Name: PYnative Address: ABC Street

CONSTRUCTOR:

- **Parameterized Constructor:-**
- A constructor with defined parameters or arguments is called a parameterized constructor. We can pass different values to each object at the time of creation using a parameterized constructor.
- The first parameter to constructor is self that is a reference to the being constructed, and the rest of the arguments are provided by the programmer. A parameterized constructor can have any number of arguments.
- For example, consider a company that contains thousands of employees. In this case, while creating each employee object, we need to pass a different name, age, and salary. In such cases, use the parameterized constructor.

CONSTRUCTOR:

class_constructor_parameterised.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/class_constructor_parameterised....

File Edit Format Run Options Window Help

```
class Employee:
    # parameterized constructor
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

    # display object
    def show(self):
        print(self.name, self.age, self.salary)

# creating object of the Employee class
emma = Employee('Emma', 23, 7500)
emma.show()

kelly = Employee('Kelly', 25, 8500)
kelly.show()
```

Emma 23 7500
Kelly 25 8500

CONSTRUCTOR:

- **Constructor With Default Values:-**
- Python allows us to define a constructor with default values. The default value will be used if we do not pass arguments to the constructor at the time of object creation.

CONSTRUCTOR:

class_constructor_default_value.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/class_constructor_default_value.py (3.10.1)

File Edit Format Run Options Window Help

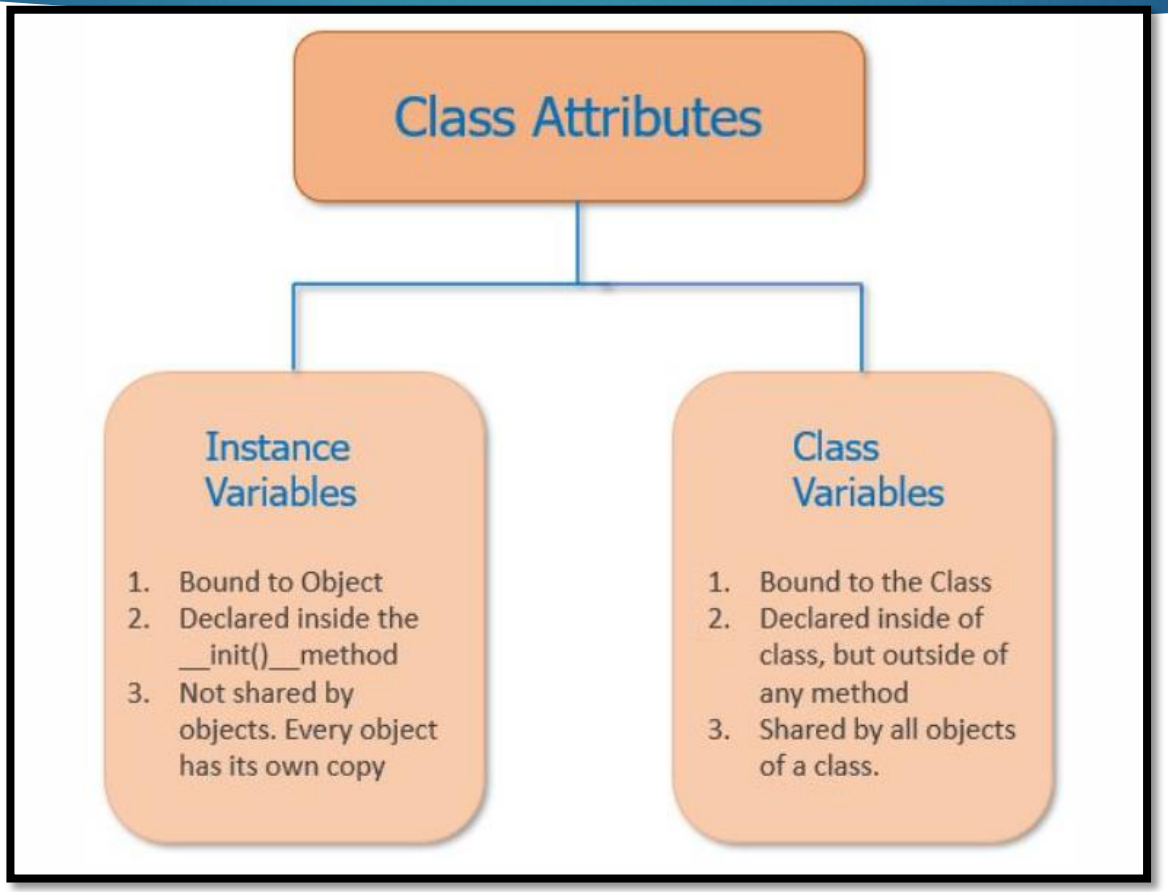
```
class Student:
    # constructor with default values age and classroom
    def __init__(self, name, age=12, classroom=7):
        self.name = name
        self.age = age
        self.classroom = classroom

    # display Student
    def show(self):
        print(self.name, self.age, self.classroom)

# creating object of the Student class
emma = Student('Emma')
emma.show()

kelly = Student('Kelly', 13)
kelly.show()
```

Emma 12 7
Kelly 13 7

CLASS ATTRIBUTES

- When we design a class, we use instance variables and class variables.
- In Class, attributes can be defined into two parts:
- **Instance variables:** The instance variables are attributes attached to an instance of a class.
- We define instance variables in the constructor (the __init__() method of a class).
- **Class Variables:** A class variable is a variable that is declared inside of class, but outside of any instance method or __init__() method.

CLASS ATTRIBUTES



CLASS ATTRIBUTES

- Objects do not share instance attributes.
- Instead, every object has its copy of the instance attribute and is unique to each object.
- All instances of a class share the class variables. However, unlike instance variables, the value of a class variable is not varied from object to object.
- Only one copy of the static variable will be created and shared between all objects of the class.
- **Accessing properties and assigning values:-**
- An instance attribute can be accessed or modified by using the dot notation: **instance_name.attribute_name.**
- A class variable is accessed or modified using the class name

*class_attributes_.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/clas...

File Edit Format Run Options Window Help

```
class Student:
    # class variables
    school_name = 'ABC School'

    # constructor
    def __init__(self, name, age):
        # instance variables
        self.name = name
        self.age = age
s1 = Student("Harry", 12)
# access instance variables
print('Student:', s1.name,

# access class variable
print('School name:', Stude
# Modify instance variables
s1.name = 'Jessa'
s1.age = 14
print('Student:', s1.name, s1.age)

# Modify class variables
Student.school_name = 'XYZ School'
print('School name:', Student.school_name)
```

```
Student: Harry 12
School name: ABC School

Student: Jessa 14
School name: XYZ School
```

EXAMPLE

- Define a Book class with the following attributes: Title, Author (Full name),Price.
- Define a constructor used to initialize the attributes of the method with values entered by the user.
- Set the View() method to display information for the current book.

class_user_input_book.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/class_user_input_book.py (3.10.1)

File Edit Format Run Options Window Help

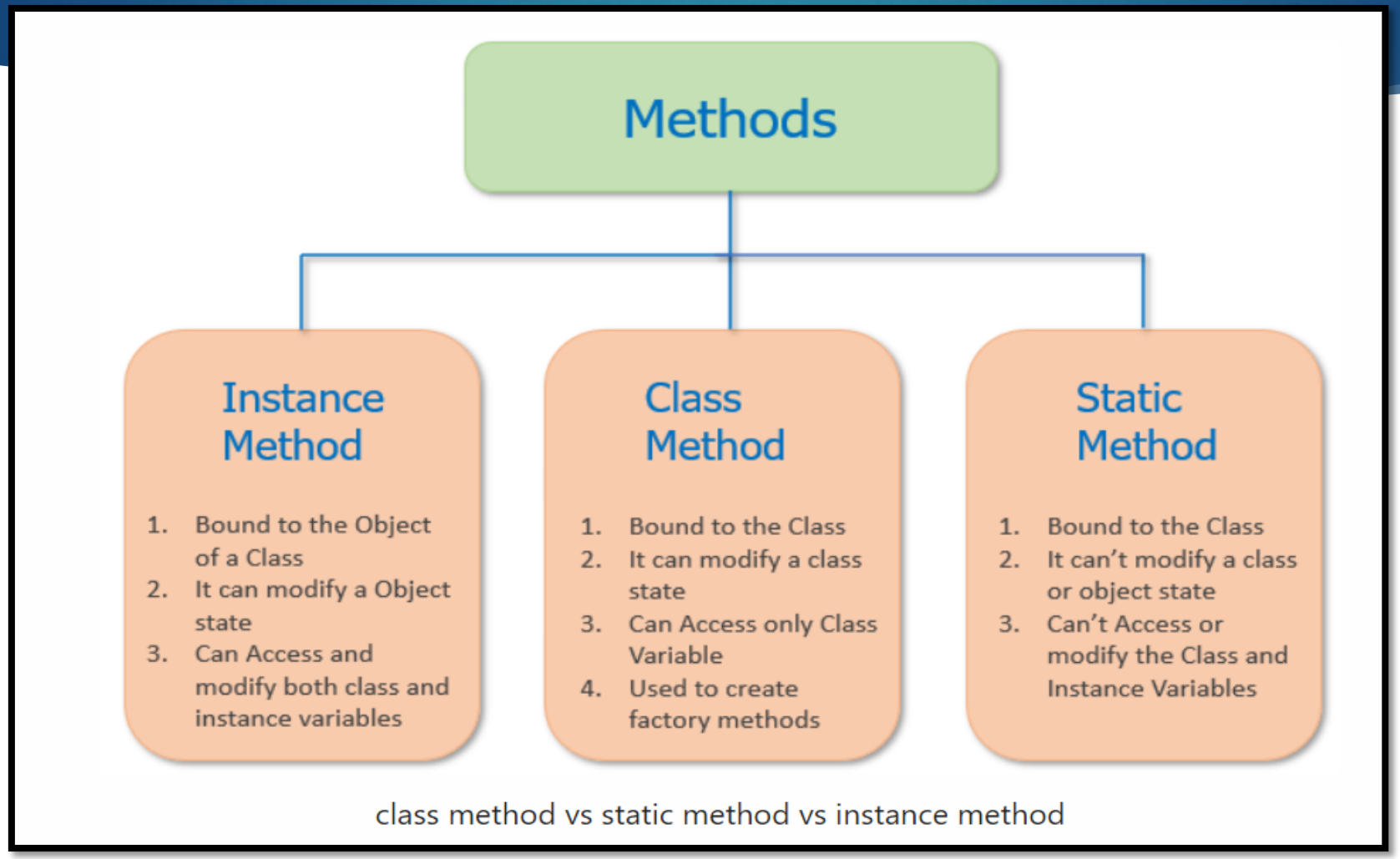
```
class book:

    def __init__(self,title,author,price):
        self.title=title;self.author=author;self.price=price

    def view(self):
        print("Name of book is:",self.title)
        print( "Name of author is:",self.author)
        print("Name of price is:",self.price)

title=input("Enter Title of book:")
author=input("Enter Author name of t
price=float(input("Enter Price of bo

b1=book(title,author,price)
b1.view()
```

Enter Title of book: DS with python
Enter Author name of the book: Dhaval Gandhi
Enter Price of book:250
Name of book is: DS with python
Name of author is: Dhaval Gandhi
Name of price is: 250.0

CLASS METHODS

- In Object-oriented programming, Inside a Class, we can define the following three types of methods.
- **Instance method:** Used to access or modify the object state. If we use instance variables inside a method, such methods are called instance methods.
- **Class method:** Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method.
- **Static method:** It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this static method doesn't have access to the class attributes.

CLASS METHODS



class method vs static method vs instance method

INSTANCE METHOD:

- If we use instance variables inside a method, such methods are called instance methods.
- The instance method performs a set of actions on the data/value provided by the instance variables.
- A instance method is bound to the object of the class.
- It can access or modify the object state by changing the value of a instance variables
- When we create a class in Python, instance methods are used regularly. To work with an instance method, we use the self keyword.
- We use the self keyword as the first parameter to a method. The self refers to the current object.
- Any method we create in a class will automatically be created as an instance method unless we explicitly tell Python that it is a class or static method.

INSTANCE METHOD:

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

    def show(self):
        print('Name:', self.name, 'Age:', self.age)


emma = Student("Jessa", 14)
emma.show()
```

Constructor to initialize Instance variables

Instance variables

Self refers to the calling object

Instance method

Call instance method

class_instance_method.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/c...

File Edit Format Run Options Window Help

```
class Student:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

    # instance method access instance variable
    def show(self):
        print('Name:', self.name, 'Age:', self.age)

# create first object
print('First Student')
emma = Student("Jessa", 14)
# call instance method
emma.show()

# create second object
print('Second Student')
kelly = Student("Kelly", 16)
# call instance method
kelly.show()
```

First Student
Name: Jessa Age: 14

Second Student
Name: Kelly Age: 16

INSTANCE METHOD:

- **Notes:**
- Inside any instance method, we can use self to access any data or method that reside in our class. We are unable to access it without a self parameter.
- An instance method can freely access attributes and even modify the value of attributes of an object by using the self parameter.
- By Using self.__class__ attribute we can access the class attributes and change the class state. Therefore instance method gives us control of changing the object as well as the class state.

INSTANCE METHOD:

- Instance method has two types:
- **Accessor method:-**
- It simply access or read data of the variables. They do not modify the data in the variables.

```
def getname(self)
return self.name
```

- **Mutator method:-**
- It not only read the data but also modify them.

```
def getname(self)
return self.name=name
```

class_instance method.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/class_instance method.py (3.10.1)

File Edit Format Run Options Window Help

```
class Student:
    def __init__(self, roll_no, name, age):
        # Instance variable
        self.roll_no = roll_no
        self.name = name
        self.age = age

    # instance method access instance variable
    def show(self):
        print('Roll Number:', self.roll_no, 'Name:', self.name, 'Age:', self.age)

    # instance method to modify instance variable
    def update(self, roll_number, age):
        self.roll_no = roll_number
        self.age = age

# create object
print('class VIII')
stud = Student(20, "Emma", 14)
# call instance method
stud.show()

# Modify instance variables
print('class IX')
stud.update(35, 15)
stud.show()
```

```
class VIII
Roll Number: 20 Name: Emma Age: 14
class IX
Roll Number: 35 Name: Emma Age: 15
```

Create Instance Variables in Instance Method:

- Till the time we used constructor to create instance attributes.
- instance attributes are not specific only to the __init__() method; they can be defined elsewhere in the class.

class_instance_method_aatribute.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/class_instance_method_aatribute.py (3.10.1)

File Edit Format Run Options Window Help

```
class Student:
    def __init__(self, roll_no, name, age):
        # Instance variable
        self.roll_no = roll_no
        self.name = name
        self.age = age

    # instance method to add instance variable
    def add_marks(self, marks):
        # add new attribute to current object
        self.marks = marks

# create object
stud = Student(20, "Emma", 14)
# call instance method
stud.add_marks(75)

# display object
print('Roll Number:', stud.roll_no, 'Name:', stud.name, 'Age:', stud.age, 'Marks:', stud.marks)
```

Roll Number: 20 Name: Emma Age: 14 Marks: 75

Ln: 19 Co

CLASS METHOD:

- Class methods are methods that are called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.
- A class method is bound to the class and not the object of the class. It can access only class variables.
- It can modify the class state by changing the value of a class variable that would apply across all the class objects.
- In method implementation, if we use only class variables, we should declare such methods as class methods. The class method has a **cls** as the first parameter, which refers to the class.
- The class method can be called using **ClassName.method_name()** as well as by using an object of the class.

Create Class Method Using @classmethod Decorator

- We must explicitly tell Python that it is a class method using the **@classmethod decorator or classmethod() function**.
- Class methods are defined inside a class, and it is pretty similar to defining a regular function.
- inside the class method, we use the cls keyword as a first parameter to access class variables. Therefore the class method gives us control of changing the class state.
- The class method can only access the class attributes, not the instance attributes.
- Class methods are used when we are dealing with **factory methods.** Factory methods are those methods that **return a class object for different use cases**. Thus, factory methods create concrete implementations of a common interface.

*class_method.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/class_method....

File Edit Format Run Options Window Help

```
from datetime import date
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def calculate_age(cls, name, birth_year):
        # calculate age an set it as a age
        # return new object
        return cls(name, date.today().year - birth_year)

    def show(self):
        print(self.name + "'s age is: " + str(self.age))

jessa = Student('Jessa', 20)
jessa.show()

# create new object using the factory method
joy = Student.calculate_age("Joy", 1995)
joy.show()
```

Jessa's age is: 20
Joy's age is: 27

Ln: 2 Col: 0

Create Class Method Using @classmethod Decorator

- To make a method as class method, add @classmethod decorator before the method definition, and add cls as the first parameter to the method.
- The @classmethod decorator is a built-in function decorator. In Python, we use the @classmethod decorator to declare a method as a class method. The @classmethod decorator is an expression that gets evaluated after our function is defined.
- The @classmethod decorator is used for converting calculate_age() method to a class method.
- The calculate_age() method takes Student class (cls) as a first parameter and returns constructor by calling Student(name, date.today().year - birthYear), which is equivalent to Student(name, age)

Create Class Method Using classmethod() Function

- Apart from a decorator, the built-in function classmethod() is used to convert a normal method into a class method.
- The classmethod() is an inbuilt function in Python, which returns a class method for a given function.
- **Syntax:**
- **classmethod(function)**
- **function:** It is the name of the method you want to convert as a class method.
- It returns the converted class method.
- A classmethod() function is the older way to create the class method in Python. In a newer version of Python, we should use the @classmethod decorator to create a class method.

Create Class Method Using Classmethod() Function

```
class School:
    # class variable
    name = 'ABC School'

    def school_name(cls):
        print('School Name is :', cls.name)

# create class method
School.school_name = classmethod(School.school_name)

# call class method
School.school_name()
```

School Name is : ABC School

class_method2.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/class_method2.py (3.10.1)

File Edit Format Run Options Window Help

```
class Student:
    school_name = 'ABC School'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def change_school(cls, school_name):
        # class_name.class_variable
        cls.school_name = school_name

    # instance method
    def show(self):
        print(self.name, self.age, 'School:', Student.school_name)

jessa = Student('Jessa', 20)
jessa.show()

# change school_name
Student.change_school('XYZ School')
jessa.show()
```

Jessa 20 School: ABC School
Jessa 20 School: XYZ School

```
class Student:
    school_name = 'ABC School'
    def __init__(self, name, age):
        # instance variables
        self.name = name
        self.age = age
    # instance method
    def show(self):
        # access instance variables and class variables
        print('Student:', self.name, self.age, Student.school_name)
    # instance method
    def change_age(self, new_age):
        # modify instance variable
        self.age = new_age
    # class method
    @classmethod
    def modify_school_name(cls, new_name):
        # modify class variable
        cls.school_name = new_name
s1 = Student("Harry", 12)
# call instance methods
s1.show()
s1.change_age(14)
# call class method
Student.modify_school_name('XYZ School')
```

Student: Harry 12 ABC School
Student: Harry 14 XYZ School

STATIC METHOD:

- A static method is a general utility method that performs a task in isolation.
- A static method is bound to the class and not the object of the class. Therefore, we can call it using the class name.
- A static method doesn't have access to the class and instance variables because it does not receive an implicit first argument like self and cls. Therefore it cannot modify the state of the object or class.
- To make a method a static method, add **@staticmethod decorator** before the method definition.
- The **@staticmethod decorator** is a built-in function decorator in Python to declare a method as a static method.
- It is an expression that gets evaluated after our function is defined.

```python
class Employee(object):
    def __init__(self, name, salary, project_name):
        self.name = name
        self.salary = salary
        self.project_name = project_name
    @staticmethod
    def gather_requirement(project_name):
        if project_name == 'ABC Project':
            requirement = ['task_1', 'task_2', 'task_3']
        else:
            requirement = ['task_1']
        return requirement
    # instance method
    def work(self):
        # call static method from instance method
        requirement = self.gather_requirement(self.project_name)
        for task in requirement:
            print('Completed', task)

emp = Employee('Kelly', 12000, 'ABC Project')
emp.work()
emp1 = Employee('Kelly', 12000, 'XYZ Project')
emp1.work()
```

```
Completed task_1
Completed task_2
Completed task_3
Completed task_1
```

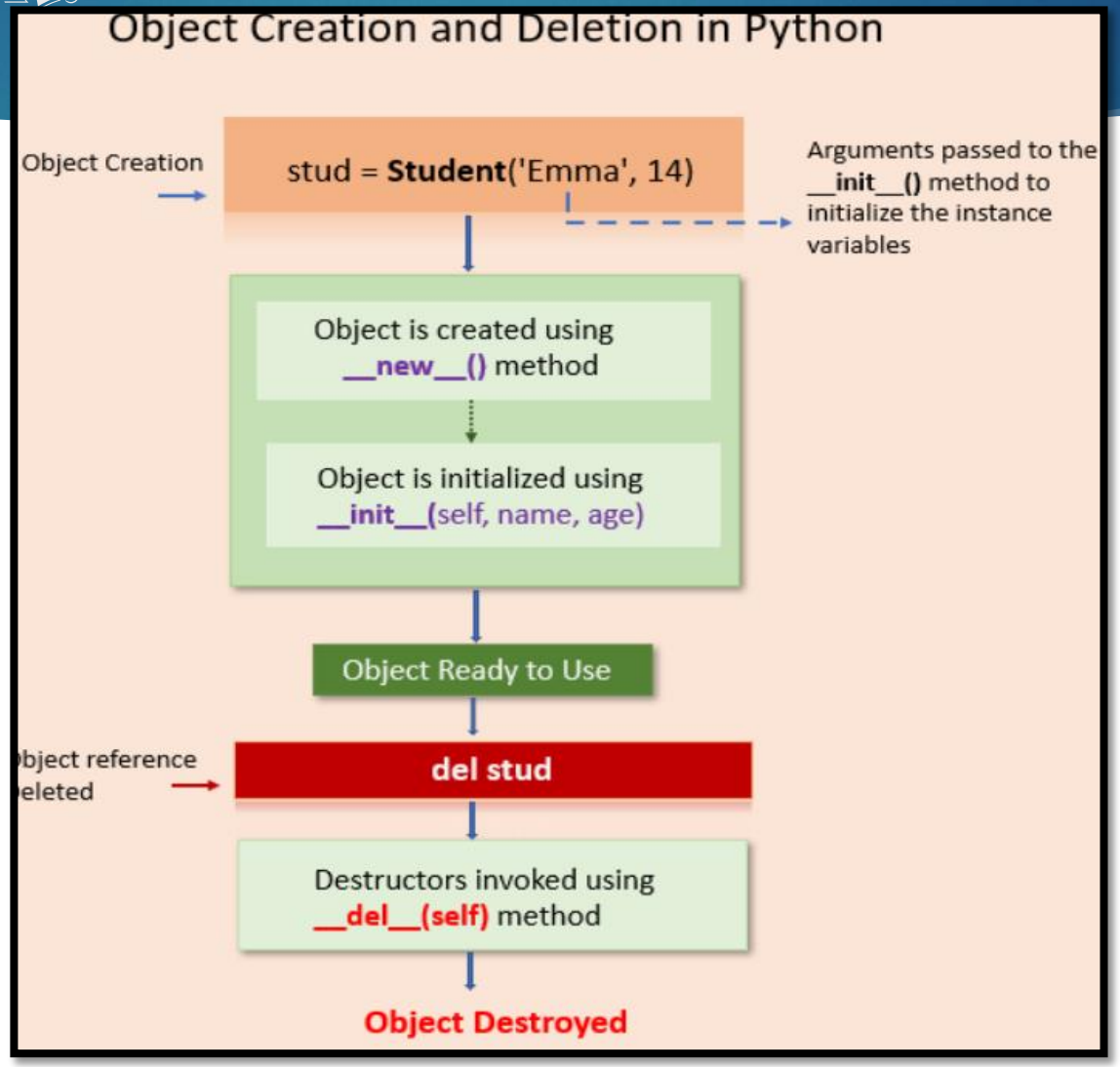
DESTRUCTOR:

- Destructor is a special method that is called when an object gets destroyed.
- Destructor is used to perform the clean-up activity before destroying the object.
- Python has a garbage collector that handles memory management automatically. For example, it cleans up the memory when an object goes out of scope.
- But it's not just memory that has to be freed when an object is destroyed. We must release or close the other resources object were using, such as open files, database connections, cleaning up the buffer or cache.

DESTRUCTOR:

- Destructor gets called in the following two cases
- **When an object goes out of scope.**
- **The reference counter of the object reaches 0.**
- In Python, The special method **__del__()** is used to define a destructor.
- when we execute **del object_name** destructor gets called automatically and the object gets garbage collected.

DESTRUCTOR:


Object Creation and Deletion in Python
Object Creation
stud = Student('Emma', 14)
Arguments passed to the __init__() method to initialize the instance variables
Object is created using __new__() method
Object is initialized using __init__(self, name, age)
Object Ready to Use
bject reference eleted
del stud
Destructors invoked using __del__(self) method
Object Destroyed

DESTRUCTOR:

- __del__() method is automatically called by Python when the instance is about to be destroyed. It is also called a finalizer or (improperly) a destructor.
- **Syntax of destructor declaration:-**

```
def __del__(self):
    # body of a destructor
```

- **def:** The keyword is used to define a method.
- **__del__() Method:** It is a reserved method. This method gets called as soon as all references to the object have been deleted
- **self:** The first argument self refers to the current object.

destructor.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/...

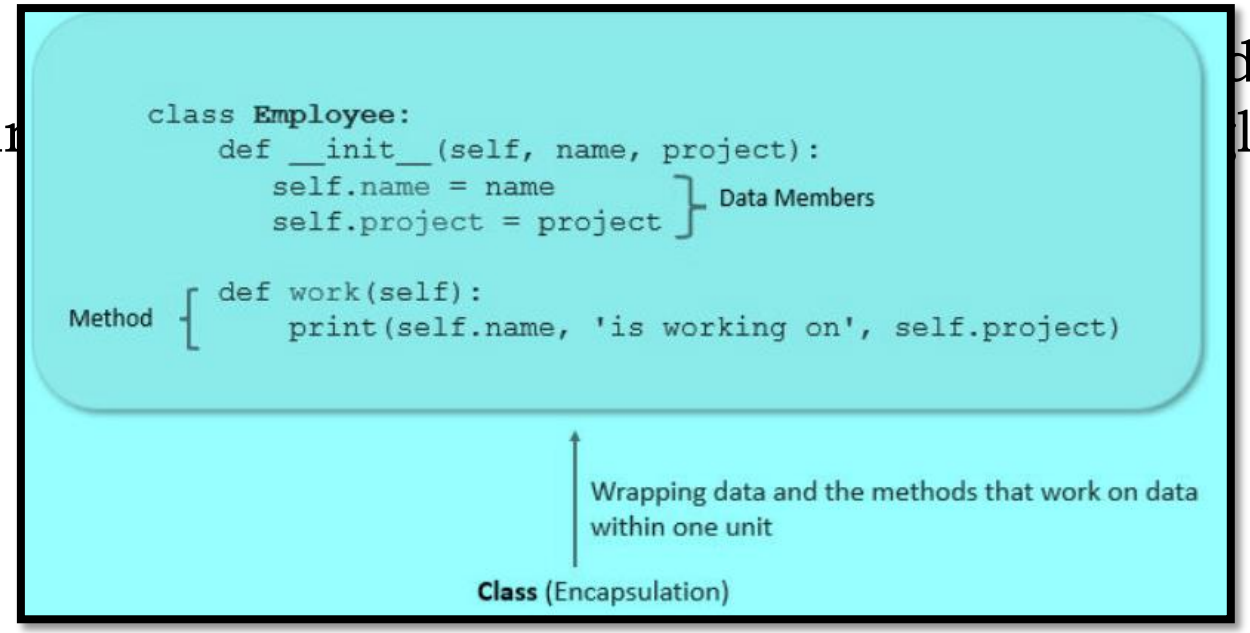
File Edit Format Run Options Window Help

```
class Student:
    # constructor
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('Object initialized')

    def show(self):
        print('Hello, my name is', self.name)

    # destructor
    def __del__(self):
        print('Inside destructor')
        print('Object destroyed')

# create object
s1 = Student('Emma')
s1.show()

# delete object
del s1
```

Inside Constructor
Object initialized
Hello, my name is Emma
Inside destructor
Object destroyed

ENCAPSULATION:

- Encapsulation in Python describes the concept of bundling data and methods within a single unit.
- For example, when you create a class, it means you are implementing encapsulation.
- A class is ds all the data members (i le unit.



ENCAPSULATION:

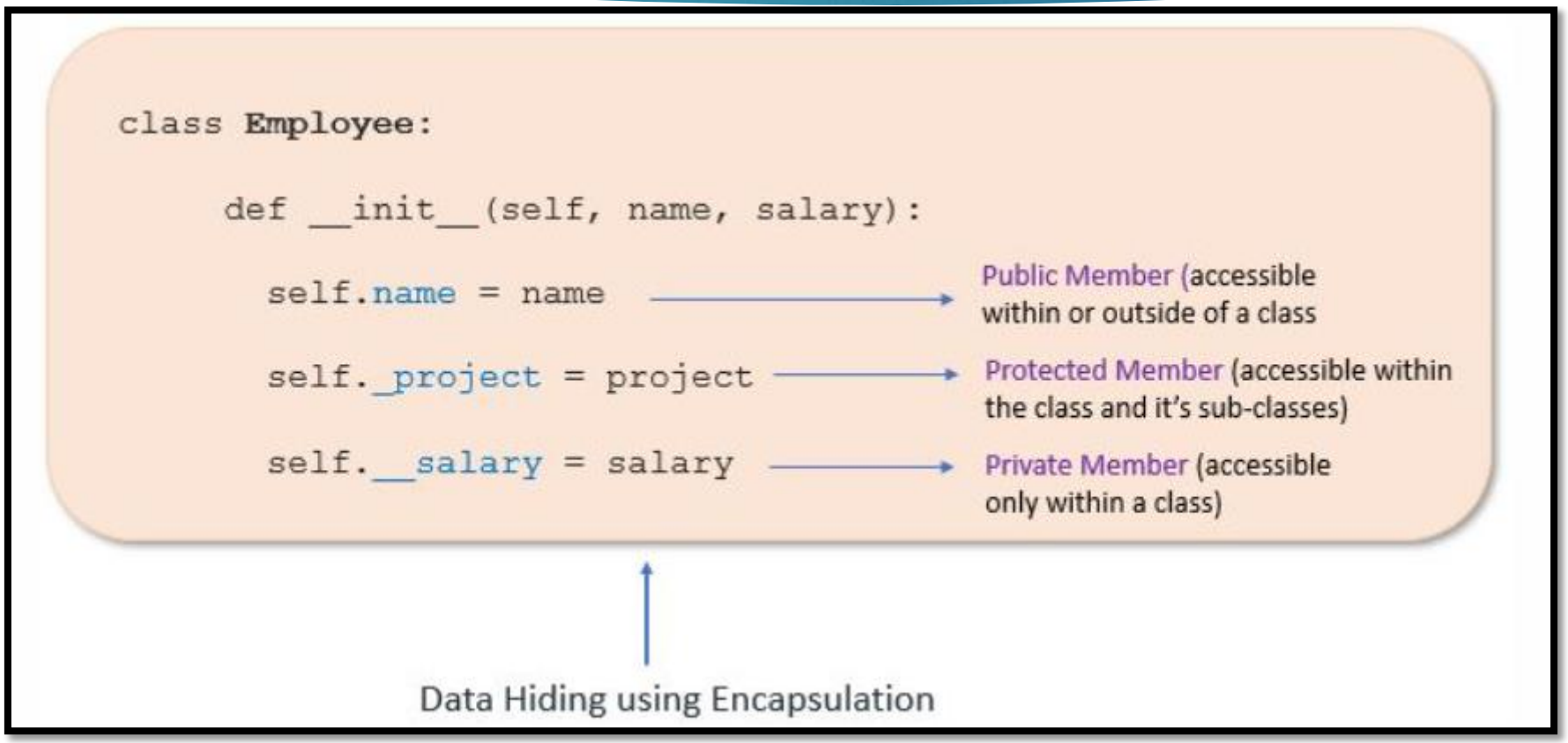
- Using encapsulation, we can hide an object's internal representation from the outside. This is called information hiding.
- Encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.
- Encapsulation is a way to can restrict access to methods and variables from outside of class.
- Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice.

ACCESS SPECIFIER:

- Access specifiers or access modifiers in python programming are used to limit the access of class variables and class methods outside of class while implementing the concepts of inheritance.
- This can be achieved by: Public, Private and Protected keyword.
- There are three types of access specifiers or access modifiers
- 1). Public access modifier
- 2). Private access modifier
- 3). Protected access modif

Access Specifiers	Same Class	Same Package	Derived Class	Other Class
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Private	Yes	No	No	No

ACCESS SPECIFIER:



ACCESS SPECIFIER:

```
#defining class Student
class Student:
    #constructor is defined
    def __init__(self, name, age, salary):
        self.age = age              # public Attribute
        self._name = name           # protected Attribute
        self.__salary = salary      # private Attribute

    def _funName(self):             # protected method
        pass

    def __funName(self):            # private method
        pass

# object creation
obj = Student("pythonlobby",21,45000)
```

ACCESS SPECIFIER:

- **Public Access Modifier:-**
- All the variables and methods (member functions) in python are by default public.
- Any instance variable in a class followed by the ‘self’ keyword ie. self.var_name are public accessed.
- Public data members are accessible within and outside of a class.

encapsulation_public.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/encapsula...

File Edit Format Run Options Window Help

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class
emp.show()
```

Ln: 21 Col:

```
Name:  Jessa Salary: 10000
Name:  Jessa Salary: 10000
```

ACCESS SPECIFIER:

- **Private Access Modifier:-**
- Private members of a class (variables or methods) are those members which are only accessible inside the class.
- We cannot use private members outside of class.
- To define a private variable add two underscores as a prefix at the start of a variable name. **ex: self.__varname**
- It is also not possible to inherit the private members of any class (parent class) to derived class (child class).
- Any instance variable in a class followed by self keyword and the variable name starting with double underscore ie. self.__varName are the private accessed member of a class.

ACCESS SPECIFIER:

encapsulation_private.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/encapsula

File Edit Format Run Options Window Help

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing private data members
print('Salary:', emp.__salary)
```

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

Access Private Member Outside of a Class Using an Instance Method

encapsulation_private_2.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/encapsulatio...

File Edit Format Run Options Window Help

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

    # public instance methods
    def show(self):
        # private members are accessible from a class
        print("Name: ", self.name, 'Salary:', self.__salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# calling public method of the class
emp.show()
```

Ln: 19 Col: 0

Name: Jessa Salary: 10000

ACCESS SPECIFIER:

- **Protected Access Modifier :-**
- Protected variables or we can say protected members of a class are restricted to be used only by the member functions and class members of the same class. And also it can be accessed or inherited by its derived class (child class).
- Protected members are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member name with a single underscore _.
- **Ex: _varname**

encapsulation_protected.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/c...

File Edit Format Run Options Window Help

```
# base class
class Company:
    def __init__(self):
        # Protected member
        self._project = "NLP"

# child class
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)

    def show(self):
        print("Employee name :", self.name)
        # Accessing protected member in child class
        print("Working on project :", self._project)

c = Employee("Jessa")
c.show()

# Direct access protected data member
print('Project:', c._project)
```

Ln: 22 Col: 2

Employee name : Jessa
Working on project : NLP
Project: NLP

```
class Student:
    def __init__(self, name, age):
        # private member
        self.name = name
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Jessa', 14)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())

# changing age using setter
stud.set_age(16)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```

Name: Jessa 14
Name: Jessa 16

ACCESS SPECIFIER:

```
#Syntax_protected_access_modifiers
class Student:
    def __init__(self):
        self._name = "PythonLobby.com"

    def _funName(self):
        return "Method Here"

class Subject(Student):
    pass

obj = Student()
obj1 = Subject()

# calling by obj. ref. of Student class
print(obj._name)        # PythonLobby.com
print(obj._funName())       # Method Here
# calling by obj. ref. of Subject class
print(obj1._name)       # PythonLobby.com
print(obj1._funName())      # Method Here
```

ADVANTAGES OF ENCAPSULATION

- **Security:** The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
- **Data Hiding:** The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
- **Simplicity:** It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
- **Aesthetics:** Bundling data and methods within a class makes code more readable and maintainable

INHERITANCE

- The process of inheriting the properties of the parent class into a child class is called inheritance.
- The existing class is called a base class or parent class and the new class is called a subclass or child class or derived class.
- 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.
- In inheritance, the child class acquires all the data members, properties, and functions from the parent class.
- A child class can also provide its specific implementation to the methods of the parent class.

INHERITANCE

- **Syntax :-**

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

INHERITANCE

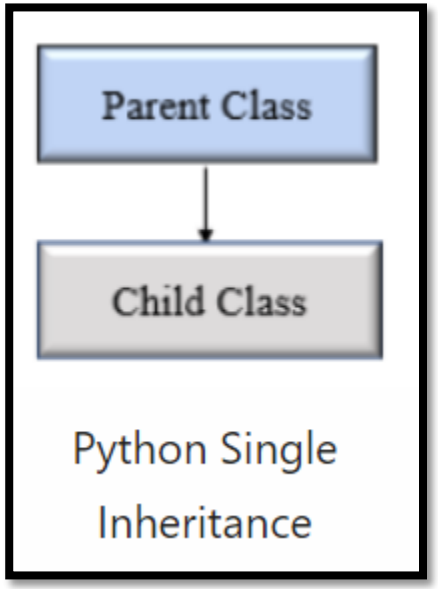
Types Of Inheritance :-

- **Single inheritance**
- **Multiple Inheritance**
- **Multilevel inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**

INHERITANCE

Single Inheritance:-

- In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class.



INHERITANCE

inheritance_single.py - D:/dhavalPC_backup_9-10-18/IT_13/Term-2/433160...

File Edit Format Run Options Window Help

```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Create object of Car
car = Car()

# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```

Ln: 17 Co

Inside Vehicle class
Inside Car class

inheritance_single_2.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/inheritance_single_2.py (3.10.1)

File Edit Format Run Options Window Help

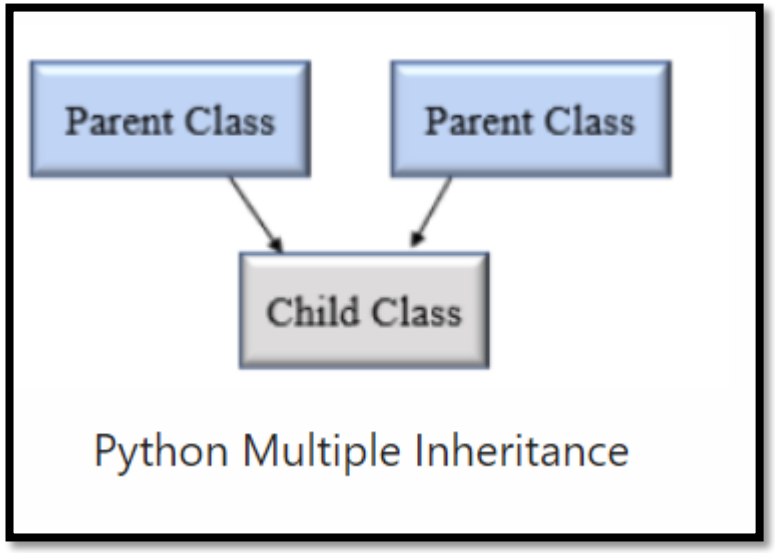
```
class Parent_class(object):
    # Constructor
    def __init__(self, value1,value2):
        self.value1 = value1
        self.value2 = value2
    # To perform addition
    def Addition(self) :
        print(" Addition value1 : " , self.value1)
        print(" Addition value2 : " , self.value2)
        return self.value1 + self.value2
    def multiplication(self) :
        print(" multiplication value1 : " , self.value1)
        print(" multiplication value2 : " , self.value2)
        return self.value1 * self.value2
    def subraction(self) :
        print(" subraction value1 : " , self.value1)
        print(" subraction value2 : " , self.value2)
        return self.value1 - self.value2
# derived class or the sub class
class Child_class(Parent_class):

    pass

# Driver code
Object1 = Child_class(10,15)  # parent class object
print(" Added value :" , Object1.Addition() )
print( " " )
Object2 = Child_class(20,30)  # parent class object
print(" Multiplied value :" , Object2.multiplication() )
print( " " )
Object3 = Child_class(50,30)  # parent class object
```

```
Addition value1 :  10
 Addition value2 :  15
 Added value : 25

 multiplication value1 :  20
 multiplication value2 :  30
 Multiplied value : 600

 subraction value1 :  50
 subraction value2 :  30
Subracted value : 20
```

INHERITANCE

Multiple Inheritance:-

- In multiple inheritance, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.



inheritance_multiple.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/inheritance_multi...

File Edit Format Run Options Window Help

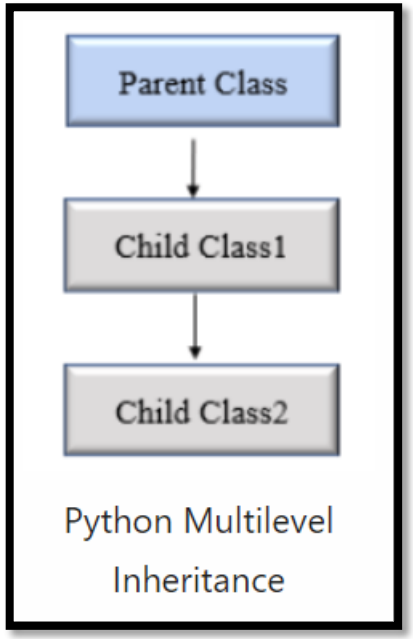
```
# Parent class 1
class Person:
    def person_info(self, name, age):
        print('Inside Person class')
        print('Name:', name, 'Age:', age)

# Parent class 2
class Company:
    def company_info(self, company_name, location):
        print('Inside Company class')
        print('Name:', company_name, 'location:', location)

# Child class
class Employee(Person, Company):
    def Employee_info(self, salary, skill):
        print('Inside Employee class')
        print('Salary:', salary, 'Skill:', skill)

# Create object of Employee
emp = Employee()

# access data
emp.person_info('Jessa', 28)
emp.company_info('Google', 'Atlanta')
emp.Employee_info(12000, 'Machine Learning')
```

Inside Person class
Name: Jessa Age: 28
Inside Company class
Name: Google location: Atlanta
Inside Employee class
Salary: 12000 Skill: Machine Learning

INHERITANCE

Multilevel Inheritance:-

- In multilevel inheritance, a class inherits from a child class or derived class.
- Suppose three classes A, B, C.
- A is the superclass,
- B is the child class of A,
- C is the child class of B.



Python Multilevel Inheritance

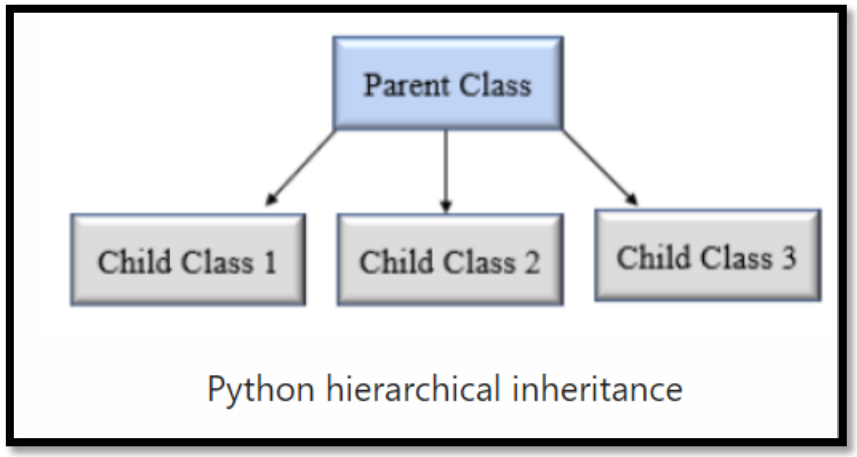
```python
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Child class
class SportsCar(Car):
    def sports_car_info(self):
        print('Inside SportsCar class')

# Create object of SportsCar
s_car = SportsCar()

# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

Inside Vehicle class
Inside Car class
Inside SportsCar class

INHERITANCE

Hierarchical Inheritance:-

- In Hierarchical inheritance, more than one child class is derived from a single parent class.
- In other words, we can say one parent class and multiple child classes.



Python hierarchical inheritance

inheritance_herichical.py - D:/dhavalPC_backup_9-10-18/IT_15/Ter...

File Edit Format Run Options Window Help

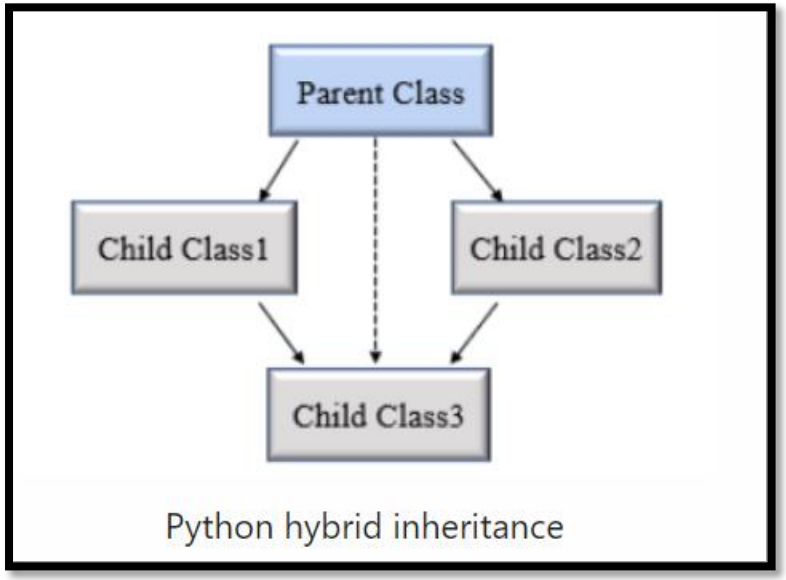
```
class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)

class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)

obj1 = Car()
obj1.info()
obj1.car_info('BMW')

obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

This is Vehicle
Car name is: BMW
This is Vehicle
Truck name is: Ford

INHERITANCE

Hybrid Inheritance:-

- When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.



Python hybrid inheritance

```python
class Vehicle:
    def vehicle_info(self):
        print("Inside Vehicle class")

class Car(Vehicle):
    def car_info(self):
        print("Inside Car class")

class Truck(Vehicle):
    def truck_info(self):
        print("Inside Truck class")

# Sports Car can inherits properties of Vehicle and Car
class SportsCar(Car, Vehicle):
    def sports_car_info(self):
        print("Inside SportsCar class")

# create object
s_car = SportsCar()

s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

Inside Vehicle class
Inside Car class
Inside SportsCar class

INHERITANCE

super() function:-

- When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, the inherited class is a subclass and the latter class is the parent class.
- In child class, we can refer to parent class by using the super() function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.
- **Benefits of using the super() function:-**
- We are not required to remember or specify the parent class name to access its methods.
- We can use the super() function in both single and multiple inheritances.
- The super() function support code reusability as there is no need to write the entire function

inheritance_super.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/inheritance_super.py (...

File Edit Format Run Options Window Help

```
class Company:
    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self):
        # Calling the superclass method using super()function
        c_name = super().company_name()
        print("Jessa works at", c_name)

# Creating object of child class
emp = Employee()
emp.info()
```

Jessa works at Google

Ln: 14 Col: 0

INHERITANCE

inheritance_super_2.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/inheritance_super_2.py (3.10.1)

File Edit Format Run Options Window Help

```
class Mammal(object):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')

class Dog(Mammal):
  def __init__(self):
    print('Dog has four legs.')
    super().__init__('Dog')

d1 = Dog()
```

Dog has four legs.
Dog is a warm-blooded animal.

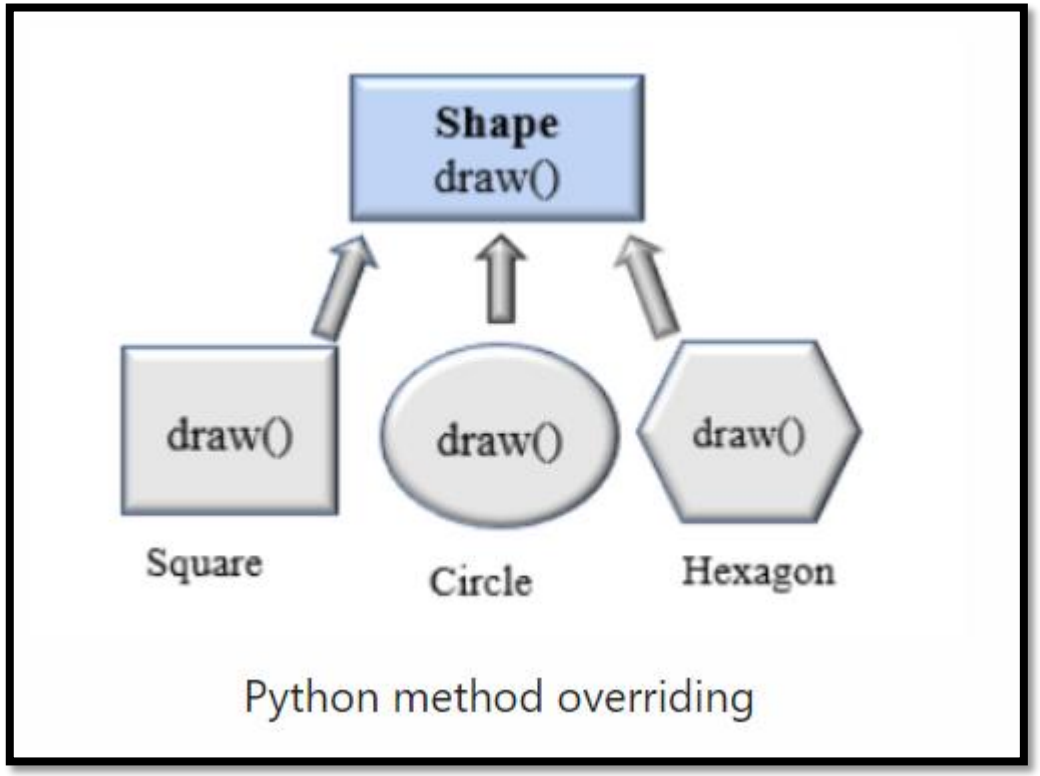
INHERITANCE

Method Overriding

- In inheritance, all members available in the parent class are by default available in the child class.
- If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class.
- This concept is called **method overriding**.
- When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to override the method in the parent class.

INHERITANCE

Method Overriding



Python method overriding

INHERITANCE

inheritance_super.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/inheritance_super.py (3.10.1)

File Edit Format Run Options Window Help

```
class Company:
    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self):
        # Calling the superclass method using super()function
        c_name = super().company_name()
        print("Jessa works at", c_name)

# Creating object of child class
emp = Employee()
emp.info()
```

Ln: 13 Col: 0

max speed is 200 Km/Hour

INHERITANCE

Method Resolution Order:-

- In Python, Method Resolution Order(MRO) is the order by which Python looks for a method or attribute.
- First, the method or attribute is searched within a class, and then it follows the order we specified while inheriting.
- This order is also called the Linearization of a class, and a set of rules is called MRO (Method Resolution Order).
- The MRO plays an essential role in multiple inheritances as a single method may found in multiple parent classes.

INHERITANCE

Method Resolution Order:-

- In multiple inheritance, the following search order is followed.
- First, it searches in the current parent class if not available, then searches in the parents class specified while inheriting (that is left to right.)
- We can get the MRO of a class. For this purpose, we can use either the mro attribute or the mro() method.

inheritance_MRO.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/inheritance_MRO.py (3.10.1)

File Edit Format Run Options Window Help

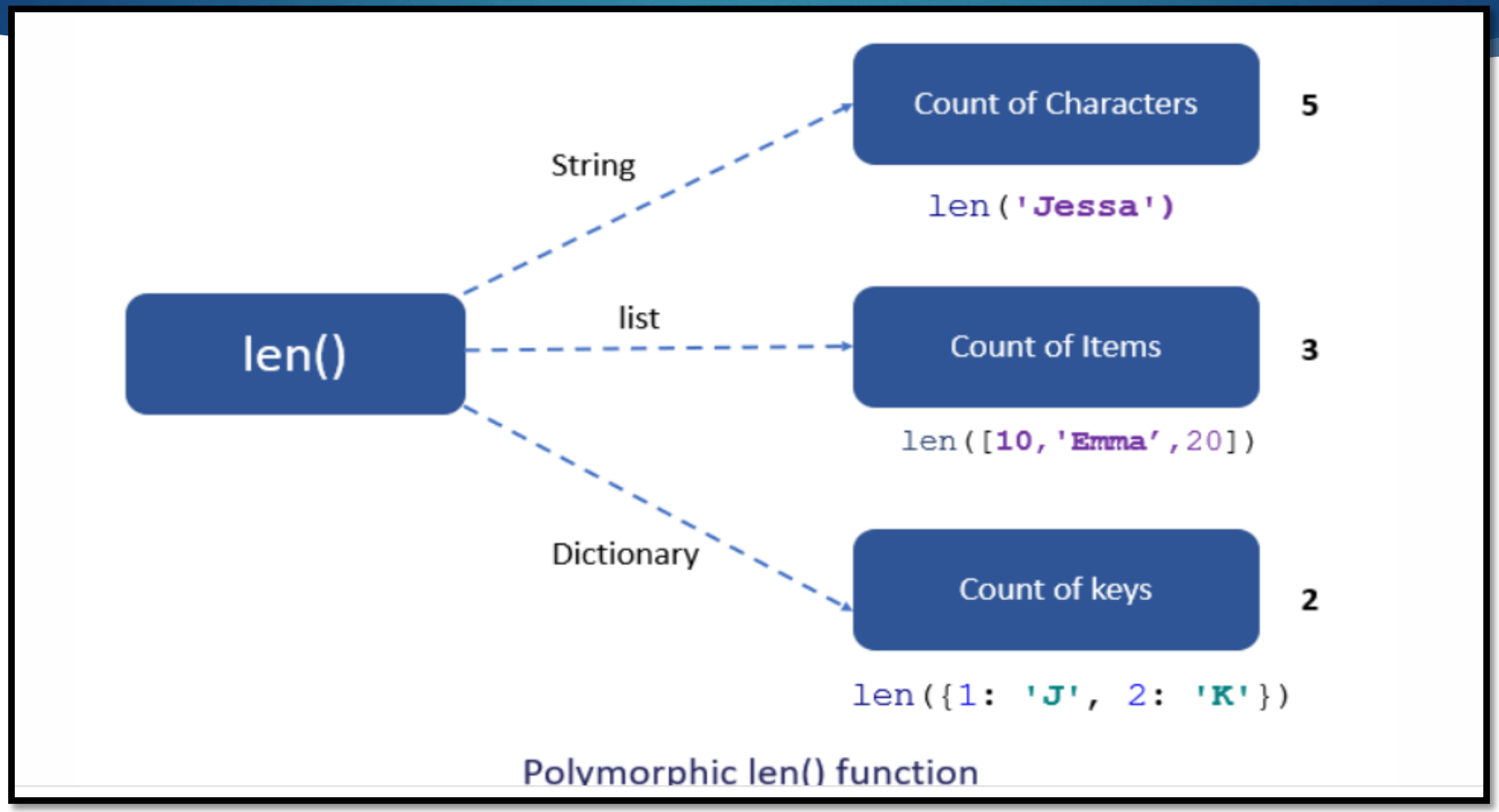
```
class A:
    def process(self):
        print(" In class A")

class B(A)
    def pr
        pr

class C(B, A):
    def process(self):
        print(" In class C")

# Creating object of C class
C1 = C()
C1.process()
print(C.mro())
B1=B()
B1.process()
print(B.mro())
# In class C
# [<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

```
In class C
[<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
 In class B
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

POLYMORPHISM

- Polymorphism in Python is the ability of an object to take many forms.
- Polymorphism allows us to perform the same action in many different ways.
- In polymorphism, a method can process objects differently depending on the class type or data type.

POLYMORPHISM



Polymorphic len() function

POLYMORPHISM

Polymorphism With Inheritance:-

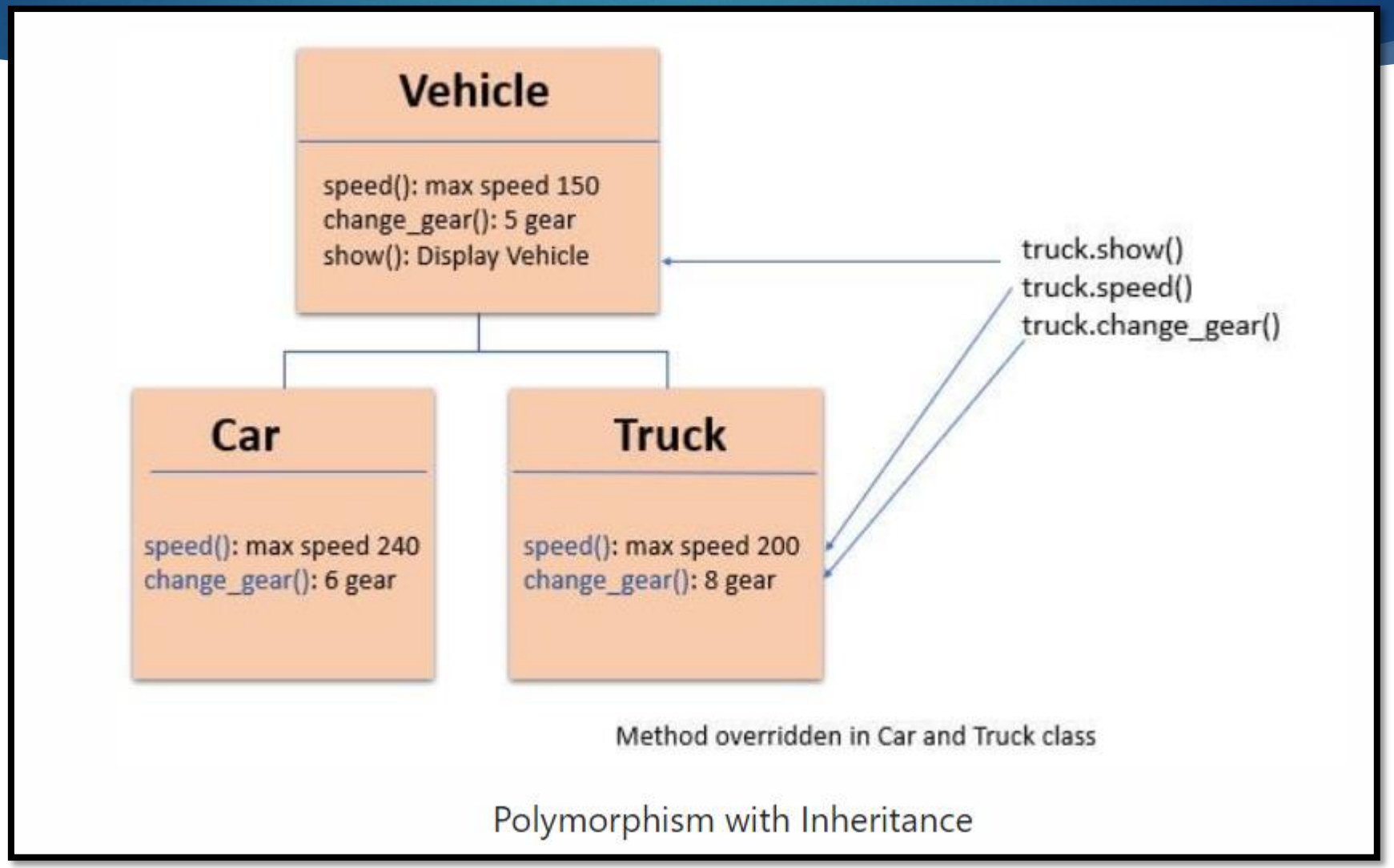
- Polymorphism is mainly used with inheritance. In inheritance, child class inherits the attributes and methods of a parent class.
- The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class.
- Using method overriding polymorphism allows us to defines methods in the child class that have the same name as the methods in the parent class.
- This process of re-implementing the inherited method in the child class is known as **Method Overriding.**

POLYMORPHISM

Advantage of method overriding:-

- It is effective when we want to extend the functionality by altering the inherited method. Or the method inherited from the parent class doesn't fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way.
- Method overriding is useful when a parent class has multiple child classes, and one of that child class wants to redefine the method. The other child classes can use the parent class method. Due to this, we don't need to modification the parent class code
- In polymorphism, Python first checks the object's class type and executes the appropriate method when we call the method.

POLYMORPHISM



POLYMORPHISM

Advantage of method overriding:-

- It is effective when we want to extend the functionality by altering the inherited method. Or the method inherited from the parent class doesn't fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way.
- Method overriding is useful when a parent class has multiple child classes, and one of that child class wants to redefine the method. The other child classes can use the parent class method. Due to this, we don't need to modification the parent class code
- In polymorphism, Python first checks the object's class type and executes the appropriate method when we call the method.

polymorphism_1.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/polymorphism_1.py (3.10.1)

File Edit Format Run Options Window Help

```
class Vehicle:
    def __init__(self, name, color, price):
        self.name = name
        self.color = color
        self.price = price
    def show(self):
        print('Details:', self.name, self.color, self.price)
    def max_speed(self):
        print('Vehicle max speed is 150')
    def change_gear(self):
        print('Vehicle change 6 gear')

# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')
    def change_gear(self):
        print('Car change 7 gear')
# Car Object
car = Car('Car x1', 'Red', 20000)
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()
# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.change_gear()
```

Details: Car x1 Red 20000
Car max speed is 240
Car change 7 gear
Details: Truck x1 white 75000
Vehicle max speed is 150
Vehicle change 6 gear

```
from math import pi
class Shape:
    def __init__(self, name):
        self.name = name
    def area(self):
        pass
    def fact(self):
        return "I am a two-dimensional shape."
    def __str__(self):
        return self.name
class Square(Shape):
    def __init__(self, length):
        super().__init__("Square")
        self.length = length
    def area(self):
        return self.length**2
    def fact(self):
        return "Squares have each angle equal to 90 degrees."
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius
    def area(self):
        return pi*self.radius**2
a = Square(4)
b = Circle(7)
print(b)
print(b.fact())
print(a.fact())
print(b.area())
```

Circle
I am a two-dimensional shape.
Squares have each angle equal to 90 degrees.
153.93804002589985

POLYMORPHISM

Polymorphism In Class methods:-

- Polymorphism with class methods is useful when we group different objects having the same method.
- we can add them to a list or a tuple, and we don't need to check the object type before calling their methods.
- Instead, Python will check object type at runtime and call the correct method.
- Thus, we can call the methods without being concerned about which class type each object is. We assume that these methods exist in each class.

```
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

ferrari = Ferrari()
bmw = BMW()

# iterate objects of same type
for car in (ferrari, bmw):
    # call methods without checking class of object
    car.fuel_type()
    car.max_speed()
```

Petrol
Max speed 350
Diesel
Max speed is 240

POLYMORPHISM

Polymorphism In Class methods:-

- As you can see, we have created two classes Ferrari and BMW. They have the same instance method names fuel_type() and max_speed(). However, we have not linked both the classes nor have we used inheritance.
- We packed two different objects into a tuple and iterate through it using a car variable.
- It is possible due to polymorphism because we have added the same method in both classes Python first checks the object's class type and executes the method present in its class.

POLYMORPHISM

Polymorphism In Function and Object:-

- We can create polymorphism with a function that can take any object as a parameter and execute its method without checking its class type.
- Using this, we can call object actions using the same function instead of repeating method calls.

```python
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

# normal function
def car_details(obj):
    obj.fuel_type()
    obj.max_speed()

ferrari = Ferrari()
bmw = BMW()

car_details(ferrari)
car_details(bmw)
```

Petrol
Max speed 350
Diesel
Max speed is 240

POLYMORPHISM

Polymorphism In Built-in Methods:-

- It means a method can process objects differently depending on the class type or data type.
- The built-in function reversed(obj) returns the iterable by reversing the given object.
- For example, if you pass a string to it, it will reverse it. But if you pass a list of strings to it, it will return the iterable by reversing the order of elements (it will not reverse the individual string).

POLYMORPHISM

Polymorphism In In Built-in Methods:-

```
students = ['Emma', 'Jessa', 'Kelly']
school = 'ABC School'

print('Reverse string')
for i in reversed('PYnative'):
    print(i, end=' ')

print('\nReverse list')
for i in reversed(['Emma', 'Jessa', 'Kelly']):
    print(i, end=' ')
```

Reverse string
e v i t a n Y P
Reverse list
Kelly Jessa Emma

POLYMORPHISM

Method Overloading:-

- The process of calling the same method with different parameters is known as method overloading.
- Python does not support method overloading.
- Python considers only the latest defined method even if you overload the method.
- Python will raise a TypeError if you overload the method.

POLYMORPHISM

Method Overloading:-

```
def addition(a, b):
    c = a + b
    print(c)

def addition(a, b, c):
    d = a + b + c
    print(d)

# the below line shows an error
# addition(4, 5)

# This line will call the second product method
addition(3, 7, 5)
```

POLYMORPHISM

Method Overloading:-

File Edit Format Run Options Window Help

```
class Shape:
    # function with two default parameters
    def area(self, a, b=0):
        if b > 0:
            print('Area of Rectangle is:', a * b)
        else:
            print('Area of Square is:', a ** 2)

square = Shape()
square.area(5)

rectangle = Shape()
rectangle.area(5, 3)
```

Area of Square is: 25
Area of Rectangle is: 15

POLYMORPHISM

Operator Overloading in Python:-

- Operator overloading means changing the default behavior of an operator depending on the operands (values) that we use. In other words, we can use the same operator for multiple purposes.
- For example, the + operator will perform an arithmetic addition operation when used with numbers. Likewise, it will perform concatenation when used with strings.
- The operator + is used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

POLYMORPHISM

```
# add 2 numbers
print(100 + 200)

# concatenate two strings
print('Jess' + 'Roy')

# merger two list
print([10, 20, 30] + ['jessa', 'emma', 'kelly'])
```

300
JessRoy
[10, 20, 30, 'jessa', 'emma', 'kelly']

POLYMORPHISM

Overloading + operator for custom objects:-

- Suppose we have two objects, and we want to add these two objects with a binary + operator. However, it will throw an error if we perform addition because the compiler doesn't add two objects

```
class Book:
    def __init__(self, pages):
        self.pages = pages

# creating two objects
b1 = Book(400)
b2 = Book(300)

# add two objects
print(b1 + b2)
```

POLYMORPHISM

Overloading + operator for custom objects:-

- Python provides some special or magic function that is automatically invoked when associated with that particular operator.
- For example, when we use the + operator, the magic method __add__() is automatically invoked.
- Internally + operator is implemented by using __add__() method. We have to override this method in our class if you want to add two custom objects.

POLYMORPHISM

Overloading + operator for custom objects:-

File Edit Format Run Options Window Help

```
class Book:
    def __init__(self, pages):
        self.pages = pages

    # Overloading + operator with magic method
    def __add__(self, other):
        return self.pages + other.pages

b1 = Book(400)
b2 = Book(300)
print("Total number of pages: ", b1 + b2)
```

Total number of pages: 700

overlaoding_magic method_mul.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/overlaoding_

File Edit Format Run Options Window Help

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

    def __mul__(self, timesheet):
        print('Worked for', timesheet.days, 'days')
        # calculate salary
        return self.salary * timesheet.days

class TimeSheet:
    def __init__(self, name, days):
        self.name = name
        self.days = days

emp = Employee("Jessa", 800)
timesheet = TimeSheet("Jessa", 50)
print("salary is: ", emp * timesheet)
```

Worked for 50 days
salary is: 40000

ABSTRACTION

- The process by which data and functions are defined in such a way that only essential details can be seen and unnecessary implementations are hidden is called **Data Abstraction**.
- In Python, abstraction can be achieved by using abstract classes and interfaces.
- A class that consists of one or more abstract method is called the abstract class. Abstract methods do not contain their implementation.
- Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass.

ABSTRACTION

- Abstraction classes are meant to be the blueprint of the other class. An abstract class can be useful when we are designing large functions.
- An abstract class is also helpful to provide the standard interface for different implementations of components.

ABSTRACTION

Abstract Class:

- An Abstract class can contain the both method normal and abstract method.
- An Abstract cannot be instantiated; we cannot create objects for the abstract class.
- An object of the derived class is used to access the features of the base class.
- The abstract class is an interface.
- Interfaces in OOP enable a class to inherit data and functions from a base class by extending it.

ABSTRACTION

Abstract Class:

- Python provides the abc module to use the abstraction in the Python program.

Syntax :-

from abc import ABC

class ClassName(ABC):

- We import the ABC class from the abc module.

ABSTRACTION

Abstract Base Classes:

- An abstract base class is the common application program of the interface for a set of subclasses.
- It can be used by the third-party, which will provide the implementations such as with plugins.
- It is also beneficial when we work with the large code-base hard to remember all the classes.

ABSTRACTION

Working of the Abstract Classes:

- Unlike the other high-level language, Python doesn't provide the abstract class itself.
- We need to import the abc module, which provides the base for defining Abstract Base classes (ABC).
- The ABC works by decorating methods of the base class as abstract. It registers concrete classes as the implementation of the abstract base.
- We use the @abstractmethod decorator to define an abstract method or if we don't provide the definition to the method, it automatically becomes the abstract method.

abstarction_1.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/abstarction_1.py (3.10.1)

File Edit Format Run Options Window Help

```
from abc import ABC, abstractmethod
class Car(ABC):
    def mileage(self):
        pass

class Tesla(Car):
    def mileage(self):
        print("The mileage is 30kmph")
class Suzuki(Car):
    def mileage(self):
        print("The mileage is 25kmph ")
class Duster(Car):
     def mileage(self):
          print("The mileage is 24kmph ")
t= Tesla ()
t.mileage()
s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
```

The mileage is 30kmph
The mileage is 25kmph
The mileage is 24kmph

abstarction_3.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/abstarction_3.py (3.10.1)

File Edit Format Run Options Window Help

```
from abc import ABC
class Polygon(ABC):
    # abstract method
    def sides(self):
        pass
class Triangle(Polygon):

    def sides(self):
        print("Triangle has 3 sides")

class Pentagon(Polygon):
    def sides(self):
        print("Pentagon has 5 sides")

class square(Polygon):
    def sides(self):
        print("I have 4 sides")

t = Triangle()
t.sides()

s = square()
s.sides()

p = Pentagon()
p.sides()
```

Triangle has 3 sides
I have 4 sides
Pentagon has 5 sides

ABSTRACTION

Working of the Abstract Classes:

- The methods where the implementation may vary for any other subclass are defined as **abstract methods** and need to be given an implementation in the subclass definition.
- There are methods that have the same implementation for all subclasses as well. There are characteristics that exhibit the properties of the abstract class and so, must be implemented in the abstract class itself. Otherwise, it will lead to repetitive code in all the inherited classes. These methods are called **concrete methods**.

ABSTRACTION

Working of the Abstract Classes:

- The methods where the implementation may vary for any other subclass are defined as **abstract methods** and need to be given an implementation in the subclass definition.
- There are methods that have the same implementation for all subclasses as well. There are characteristics that exhibit the properties of the abstract class and so, must be implemented in the abstract class itself. Otherwise, it will lead to repetitive code in all the inherited classes. These methods are called **concrete methods**.

abstarction_2.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/class/abstarction_2.py (3.10.1)

File Edit Format Run Options Window Help

```
from abc import ABC
class Animal(ABC):
    #concrete method
    def sleep(self):
        print("I am going to sleep in a while")

    #@abstractmethod
    def sound(self):
        print("This function is for defining the sound by any animal")
        pass

class Snake(Animal):
    def sound(self):
        print("I can hiss")

class Dog(Animal):
    def sound(self):
        print("I can bark")

class Cat(Animal):
    def sound(self):
        print("I can meow")

c = Cat()
c.sleep()
c.sound()

c = Snake()
c.sound()
```

I am going to sleep in a while
I can meow
I can hiss

THANK YOU