Darshan
UNIVERSITY
योग: कर्मसु कौशलम्

Unit-05

Object Oriented Programming with python

Prof. Jayesh D. Vagadiya

Computer Engineering Department

Darshan Institute of Engineering & Technology, Rajkot

✉ Jayesh.vagadiya@darshan.ac.in

☎ 9537133260

✅ Outline

- ✓ Object Oriented Approach
- ✓ Custom Classes: Attributes and Methods
- ✓ Inheritance,
- ✓ Polymorphism
- ✓ Abstract class
- ✓ Abstract method

Darshan UNIVERSITY

Introduction

▶ The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

▶ Main Components of OOPs

➥ Class
➥ Objects
➥ Polymorphism
➥ Encapsulation
➥ Inheritance

Overview of OOP Terminology

▶ Class

▶ Class variable

▶ Data member

▶ Function overloading

▶ Instance variable

▶ Inheritance

▶ Instance

▶ Instantiation

▶ Method

▶ Object

Class

▶ A class is a user defined blueprint or prototype from which objects are created.

▶ It represents the set of properties and methods that are common to all objects of one type.

▶ Defines new datatype (primitive ones are not enough). For Example : **Car**

▶ A class is a template for an object .

```
class Car{



          .........

}
```

syntax

```
class className:
        # Statement-1
        .
        .
        .
        # Statement-N
```

Class

Properties (Describe)
Company
Model
Color
Mfg. Year
Price
Fuel Type
Mileage
Gear Type
Power Steering
Anti-Lock braking system

Methods (Functions)
Start
Drive
Park
On_break
On_lock
On_turn

Objects

▶ It is a basic unit of Object Oriented Programming and represents the real life entities.

▶ An object is an **instance** of a **class**.

▶ An object has a **state** and **behavior**.

▶ The **state** of an object is stored in **fields** (variables), while **methods** (functions) display the object's **behavior**.

Honda
City

Hyundai
i20

Sumo
Grand

Mercedes E
class

Swift
Dzire

The self

▸ Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it

▸ If we have a method that takes no arguments, then we still have to have one argument.

▸ When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

```
My name is = DIET
```

Example.py

```
1  class Demo:
2      # Method of Class Demo
3      def printMyName(self,name):
4          print('My name is =',name)
5
6
7  object1 = Demo() # Object created
8  object1.printMyName("DIET") # Method Called
```

Darshan UNIVERSITY

The __init__ method

▶ The __init__ method is similar to constructors in C++ and Java.

▶ It called when object of class created.

▶ Used for initialization of object data members.

OUTPUT

```
My name is = DIET
```

Example.py

```python
1  class Demo:
2      #class Attributes
3      Name = "”
4
5      #Constructor Method
6      def __init__(self,n):
7          self.Name = n
8
9      # Method of Class Demo
10     def printMyName(self):
11         print('My name is =',self.Name)
12
13
14 object1 = Demo("DIET") # Object created
15 object1.printMyName() # Method Called
```

Example

```python
 1  # Write a Program to create Circle class and
 2  define findarea method to calculate are of
 3  circle.
 4  class Circle:
 5      def __init__(self,r):
 6          self.r = r
 7
 8      def findarea(self):
 9          return 3.14 * self.r * self.r
10
11  r = int(input("Enter radius="))
12  c = Circle(r)
13  area = c.findarea()
14  print("Area = ",area)
```

```
Enter radius=1
Area =  3.14
```

Built-in class attributes

▸ Python class also contains some built-in class attributes which provide information about the class.

attributes	Description
__dict__	It provides the dictionary containing the information about the class namespace.
__doc__	It contains a string which has the class documentation
__name__	It is used to access the class name.
__module__	It is used to access the module in which, this class is defined.

Example

```python
class Student:
    '''Student Class'''
    def __init__(self,rno,sname):
        self.rollno = rno
        self.studentName = sname

    def display(self):
        print(self.rollno,self.studentName)

object = Student(23,"ABC")
print(object.__doc__)
print(object.__dict__)
print(object.__module__)
```

OUTPUT

```
Student Class
{'rollno': 23, 'studentName': 'ABC'}
__main__
```

Class Attributes vs Instance Attributes

Class Attributes	Instance Attributes
Class attributes are the variables defined directly in the class that are shared by all objects of the class.	**Instance attributes** are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.
Shared across all objects.	Specific to object.
Accessed using class name as well as using object with dot notation, e.g. **classname.class_attribute or object.class_attribute**	Accessed using object dot notation e.g. **object.instance_attribute**
Changing value by using **classname.class_attribute = value** will be reflected to all the objects.	Changing value of **instance attribute will not be reflected to other** objects.

Class Attributes vs Instance Attributes

Example.py

```python
class Counter:
    #Class Attribute
    count = 0
    def __init__(self):
        Counter.count = Counter.count + 1

s1 = Counter() # Object 1 created
s2 = Counter() # Object 2 created

print(s1.count) # Counter Value
print(s2.count) # Counter Value
print(Counter.count) # Counter value

Counter.count = 100 # Updating Class Attribute

print(s1.count)
print(s2.count)
```

OUTPUT

```
2
2
2
100
100
```

Darshan UNIVERSITY

Public, Private and Protected

▶ Python uses '_' symbol to determine the access control for a specific data member or a member function of a class.

▶ **Public Access :**
  �థ The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

▶ **Protected Access Modifier:**
  ➢ The members of a class that are declared protected are only accessible to a class derived from it.
  ➢ Data members of a class are declared protected by adding a **single underscore '_' symbol** before the data member name.

▶ **Private Access Modifier:**
  ➢ The members of a class that are declared private are accessible within the class only.
  ➢ We can declared private by adding a **double underscore '__' symbol** before before the data member name.

Class Attributes vs Instance Attributes

Example.py

```python
1   class Date:
2       def __init__(self,d,m,y):
3           self.__d = d # Private
4           self.__m = m # Private
5           self.__y = y # Private
6
7       #Private Method
8       def __displaydate(self):
9           print("{}-{}-{}".format(self.__d,self.__m,self.__y))
10
11      # Public Method
12      def display(self):
13          self.__displaydate()
14
15  y1 = Date(10,10,2022)
16  y1.display()
17
18  y1.__displaydate() #errorobject has no attribute
19  y1.__m # error object has no attribute
```

OUTPUT

10-10-2022

Types of Methods

▶ We can classify the methods in the following 3 types:

▶ **Instance methods**
- ➥ Instance methods are bound to instances and hence called as: instancename.method().
- ➥ The purpose of instance methods is to set or get details about instances (objects), and that is why they're known as instance methods.
- ➥ They have one default parameter- self, which points to an instance of the class.
- ➥ Although you don't have to pass that every time. You can change the name of this parameter but it is better to stick to the convention i.e self.
- ➥ In order to call an instance method, you've to create an object/instance of the class. With the help of this object, you can access any method of the class.

▶ **Class methods**
- ➥ These methods act on class level.
- ➥ Class methods are the methods which act on the class variables.
- ➥ These methods are written using @classmethod decorator above them.

Types of Methods

▶ **Class methods:**
  - By default, the first parameter for class methods is "cls" which refers to the class itself.
  - Without creating an instance of the class, you can call the class method with – Class_name.Method_name().

▶ **Static methods:**
  - Static methods cannot access the class data.
  - They are self-sufficient and can work on their own.
  - they cannot get or set the instance state or class state.
  - In order to define a static method, we can use the @staticmethod decorator
  - we do not need to pass any special or default parameters.
  - we can call them using object/instance of the class or class name.

Types of Methods

```
1   class Student:
2       firstname = ""
3
4       #Instance Methods
5       def method1(self,lname):
6           self.lastname = lname
7
8       @classmethod
9       def classMethod(cls,fname):
10          cls.firstname = fname
11
12      @staticmethod
13      def staticMethod(fname):
14          print(fname)
15
16      #Instance Methods
17      def printData(self):
18          print(self.firstname,self.lastname)
```

Example.py

```
19  s1 = Student()
20  s1.method1("ABC")
21  Student.classMethod("XYZ")
22  Student.staticMethod("Z")
23  s1.printData()
```

OUTPUT

```
Z
XYZ ABC
```

Passing Object as function Arguments

Example.py

```
1    class Time:
2        #Constructor
3        def __init__(self,h,m):
4            self.h = h
5            self.m = m
6
7        #add Two Time Objects
8        def addTime(self,t1,t2):
9            self.h = t1.h + t2.h
10           self.m = t1.m + t2.m
11
12       # Display Time
13       def displayTime(self):
14           print(self.h,self.m)
15
16   t1 = Time(4,4)
17   t2 = Time(3,2)
18   t3 = Time(0,0)
```

Example.py

```
19   t3.addTime(t1,t2)
20   t1.displayTime()
21   t2.displayTime()
22   t3.displayTime()
```

OUTPUT

```
4 4
3 2
7 6
```

Inheritance in Python

▶ Inheritance is the capability of one class to derive or inherit the properties from another class.

▶ Advantages:
  ➥ It represents real-world relationships.
  ➥ It provides **reusability** of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
  ➥ It is transitive in nature.

▶ Parent class is the class being inherited from, also called base class.

▶ Child class is the class that inherits from another class, also called derived class.

▶ Inheritance in Java can be best understood in terms of **Parent** and **Child** relationship, also known as **Super class**(Parent) and **Sub class**(Child).

▶ Inheritance defines **IS-A** relationship between a **Super class** and its **Sub class**.
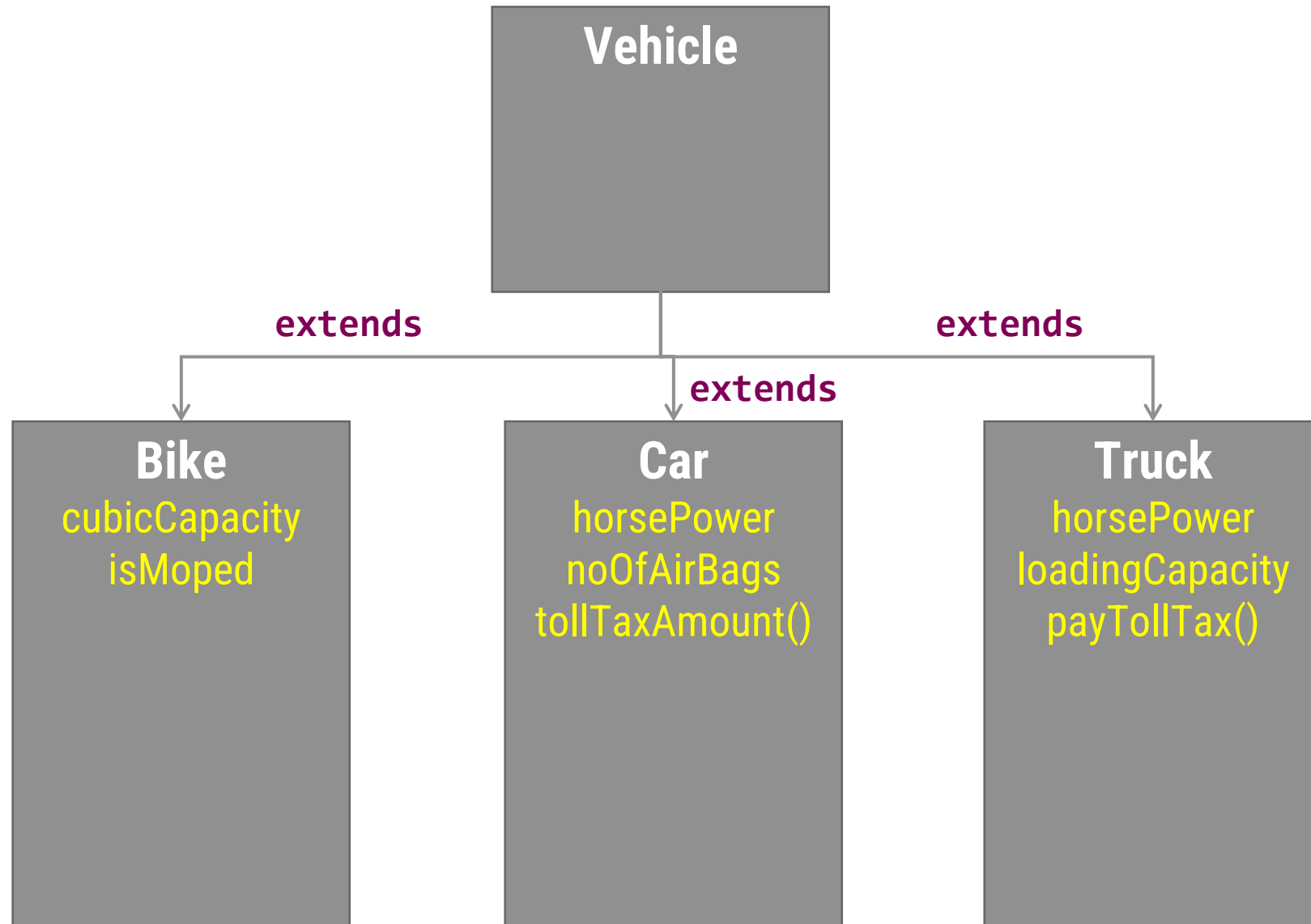
Syntax

Syntax :

```
class A {

      // code

}

class B extends A{

      // code

}
```

```
class A:
    _

    _

class B(A):
    _

    _
```

Example

```
                        ┌─────────────────┐
                        │    Vehicle      │
                        │                 │
                        │                 │
                        └─────────────────┘
          extends              │ extends            extends
     ┌───────────────┐   ┌───────────────┐   ┌───────────────┐
     │     Bike      │   │     Car       │   │    Truck      │
     │ cubicCapacity │   │  horsePower   │   │  horsePower   │
     │   isMoped     │   │ noOfAirBags   │   │loadingCapacity│
     │               │   │ tollTaxAmount()│  │ payTollTax()  │
     └───────────────┘   └───────────────┘   └───────────────┘
```

Example

```
1  class Parent():
2      def first(self):
3          print('Parent Class')
4
5  class Child(Parent):
6      def second(self):
7          print('Child Class')
8
9  ob = Child()
10 ob.first()
11 ob.second()
```

OUTPUT

```
Parent Class
Child Class
```

```
1  class Person:
2      def __init__(Self,fName,lName):
3          Self.firstName = fName
4          Self.lastName = lName
5
6      def displayPerson(self):
7          print(self.firstName,self.lastName)
8
9  class Student(Person):
10     def __init__(Self, fName, lName,gYear):
11         Self.graduateYear = gYear
12         Person.__init__(Self,fName, lName)
13
14     def displayStudent(self):
15         print(self.firstName,self.lastName,self.graduateYear)
16
17 ob = Student("jayesh","Vagadiya","2015")
18 ob.displayStudent()
```
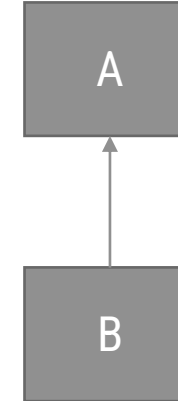
OUTPUT

```
jayesh Vagadiya 2015
```

Inheritance Types

▶ Single:
  ➥ When a child class inherits only a single parent class.
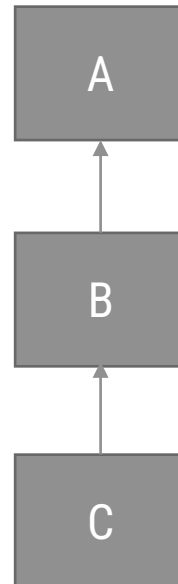
A

B

syntax

```
class ParentClass:
         -
         -
class ChildClass(ParentClass):
         -
         -
```

▶ Multilevel:
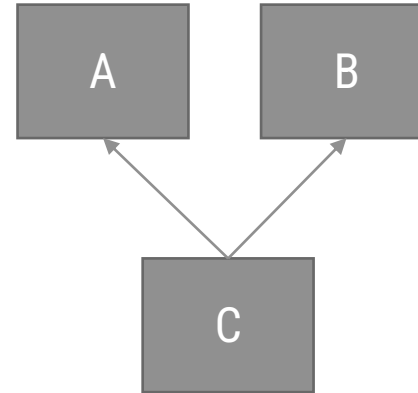 ➥ When a child class becomes a parent class for another child class.

A

B

C

syntax

```
class ParentClass:
         -
         -
class Child1(ParentClass):
         -
         -
class Child2(Child1):
         -
         -
```

Darshan UNIVERSITY

Inheritance Types (cont.)

▶ Multiple:
➥ When a child class inherits from more than one parent class.
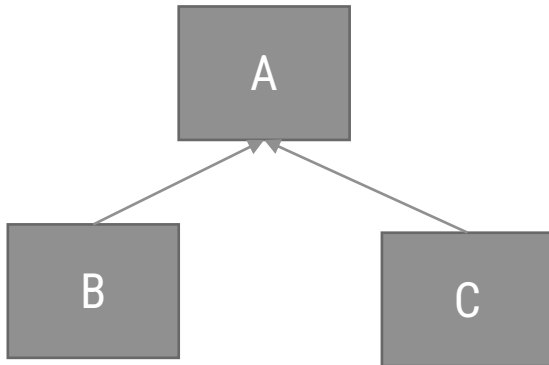
```
class ParentClass1:
        -
        -
class ParentClass2:
        -
        -


class ChildClass(ParentClass1,
ParentClass2):
        -
        -
```

▶ Hierarchical :
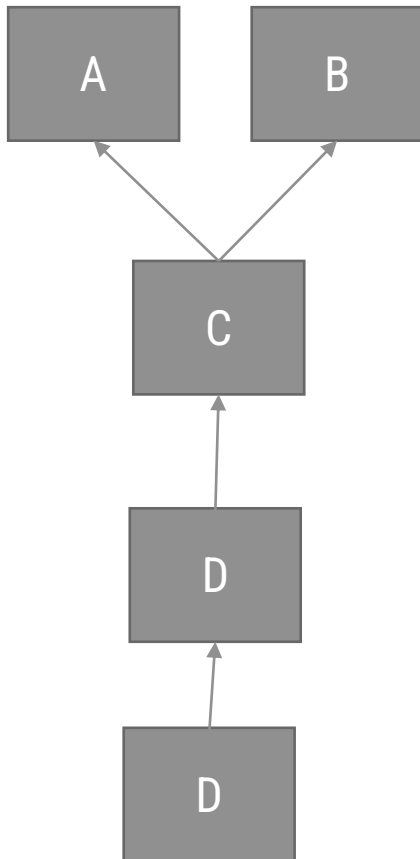➥ When a more then one chid is derived from parent class.

```
class ParentClass:
        -
        -
class Child1(ParentClass):
        -
        -
class Child2(ParentClass):
        -
        -
```

Inheritance Types (cont.)

▶ Hybrid:
  ➥ This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.

syntax

```
class ParentClass1:
        -
        -
class ParentClass2:
        -
        -

class ChildClass1(ParentClass1,
ParentClass2):
        -
        -
class ChildClass2(ChildClass1):
        -
        -
class ChildClass3(ChildClass2):
        -
        -
```

Examples

```python
1  class A:
2      def diplayA(self):
3          print("Class A")
4
5  class B(A):
6      def diplayB(self):
7          print("Class B")
8
9  object = B()
10 object.diplayA()
11 object.diplayB()
```

OUTPUT

```
Class A
Class B
```

```python
1  class A:
2      def diplayA(self):
3          print("Class A")
4
5  class B(A):
6      def diplayB(self):
7          print("Class B")
8
9  class C(B):
10     def diplayC(self):
11         print("Class C")
12
13 object = C()
14 object.diplayA()
15 object.diplayB()
16 object.diplayC()
```

OUTPUT

```
Class A
Class B
Class C
```

Examples

```
1  class A:
2      def diplayA(self):
3          print("Class A")
4
5  class B:
6      def diplayB(self):
7          print("Class B")
8
9  class C(A,B):
10      def diplayC(self):
11          print("Class C")
12
13  object = C()
14  object.diplayA()
15  object.diplayB()
16  object.diplayC()
```

OUTPUT

```
Class A
Class B
Class C
```

Hierarchical.py

```
1  class A:
2      def diplayA(self):
3          print("Class A")
4
5  class B(A):
6      def diplayB(self):
7          print("Class B")
8
9  class C(A):
10      def diplayC(self):
11          print("Class C")
12
13  object1 = B()
14  object2 = C()
15  object1.diplayA()
16  object1.diplayB()
17  object2.diplayA()
18  object2.diplayC()
```

OUTPUT

```
Class A
Class B
Class A
Class C
```

Darshan UNIVERSITY

Polymorphism

- If one task is performed in different ways, it is known as polymorphism.

- OOP concept lets programmers use the same word to mean different things in different contexts.

- Polymorphism is taken from the Greek words Poly (many) and morphism (forms).

Example1.py
```
1  num1 = 3
2  num2 = 4
3  print(num1+num2)
4
5  str1 = "Python "
6  str2 = "Programming"
7  print(str1+str2)
```

Example2.py
```
1  print(len("DIET"))
2  print(len([1,2,3,4,5,6]))
3  print(len((1,2,3,4,5,6,7,)))
```

Method Overloading

▸ The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.

```python
Example1.py
1  def display():
2   print("Hello")
3
4  def display(name):
5   print(name)
6
7  #display() Error Message
8  display("Name")
```

▸ In the above code, we have defined two display method, but we can only use the second display method, as python does not support method overloading.

Method Overloading

▶ To overcome the above problem, we can use different ways to achieve the method overloading.

▶ We need to write the method's logic so that different code executes inside the function depending on the parameter passes.

Example1.py
```
1  class Shape:
2      # function with two default parameters
3      def area(self, a, b=0):
4          if b > 0:
5              print('Area of Rectangle is:', a * b)
6          else:
7              print('Area of Square is:', a ** 2)
8
9  square = Shape()
10 square.area(5)
11 square.area(1,2)
```

OUTPUT
```
Area of Square is: 25
Area of Rectangle is: 2
```

Polymorphism with Function and Objects

▶ 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.

Example1.py

```python
1   class Honda:
2       def fuel_type(self):
3           print("Petrol")
4
5       def max_speed(self):
6           print("Max speed 180")
7
8   class Skoda:
9       def fuel_type(self):
10          print("Diesel")
11
12      def max_speed(self):
13          print("Max speed is 220")
```

Example1.py

```python
14  # normal function
15  def car_details(obj):
16      obj.fuel_type()
17      obj.max_speed()
18
19  honda = Honda()
20  skoda = Skoda()
21  car_details(honda)
22  car_details(skoda)
```

OUTPUT

```
Petrol
Max speed 180
Diesel
Max speed is 220
```

Method Overriding

▶ Method overriding allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.

▶ When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

▶ If an object of a parent class is used to invoke the method, then the version in the parent class will be executed.

▶ if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

Darshan UNIVERSITY

Example

```
1   class A:
2       def display(self):
3           print("Inside class A")
4
5   class B(A):
6     # Method override by Sublcass
7       def display(self):
8           print("Inside class B")
9
10
11  object1 = B()
12  object1.display()
```

OUTPUT

```
Inside class B
```

Example1.py

```
1   class A:
2       def display(self):
3           print("Inside class A")
4
5   class B(A):
6       # Method override by Sublcass
7       def display(self):
8           print("Inside class B")
9
10
11  object1 = A()
12  object1.display()
13  object2 = B()
14  object2.display()
```

OUTPUT

```
Inside class A
Inside class B
```

Calling the Parent's method

▸ Parent class methods can also be called within the overridden methods. This can generally be achieved by two ways.

▸ **Using Classname**
  ➥ Parent's class methods can be called by using the Parent classname.method inside the overridden method.

Example1.py
```python
1  class A:
2      def display(self):
3          print("Inside class A")
4
5  class B(A):
6      # Method override by Sublcass
7      def display(self):
8          #Calling Super Class Method
9          A.display(self)
10         print("Inside class B")
11
12 object1 = B()
13 object1.display()
```

OUTPUT
```
Inside class A
Inside class B
```

Calling the Parent's method

▶ **Using Super()**

➥ Python super() function provides us the facility to refer to the parent class explicitly. It is basically useful where we have to call superclass functions.

Example1.py

```python
1   class A:
2       def display(self):
3           print("Inside class A")
4
5   class B(A):
6       # Method override by Sublcass
7       def display(self):
8           #Calling Super Class Method
9           super().display()
10          print("Inside class B")
11
12
13  object1 = B()
14  object1.display()
```

OUTPUT

```
Inside class A
Inside class B
```

Abstract class

▶ python supports all the features of object-oriented programming including abstraction and abstract classes.

▶ We cannot create an abstract class in Python directly. However, Python does provide a module that allows us to define abstract classes.

▶ The module we can use to create an abstract class in Python is abc(abstract base class) module.

▶ Abstract methods force the child classes to give the implementation of these methods in them and thus help us achieve abstraction as each subclass can give its own implementation.

▶ A class containing one or more than one abstract method is called an abstract class.

Syntax

```
from abc import ABC
class <Abstract_Class_Name>(ABC):
            # body of the class
```

Abstract class

▸ we cannot create an instance or object of an abstract class in Python.

▸ abstract classes are used to create a blueprint of our classes as they don't contain the method implementation.

▸ This is a very useful capability, especially in situations where child classes should provide their own separate implementation.

Abstract Method

▶ To define an abstract method we use the @abstractmethod decorator of the abc module.

▶ It tells Python that the declared method is abstract and should be overridden in the child classes.

▶ We just need to put this decorator over any function we want to make abstract, and the abc module takes care of the rest.

Syntax

```python
from abc import ABC, abstractmethod
class <Abstract_Class_Name>(ABC):
        @abstractmethod
        def <abstract_method_name>(self,other_parameters):
            pass
```

Example.py

```python
from abc import ABC, abstractmethod
class Demo(ABC):
        @abstractmethod
        def printName(self):
            pass
```

Example

Example1.py

```python
1  from abc import ABC,abstractmethod
2
3  class Shape(ABC):
4      @abstractmethod
5      def drawShape(self):
6          pass
7
8  class Circle(Shape):
9      def drawShape(self):
10         print("Drawing a Circle")
11
12 a = Circle()
13 a.drawShape()
14 B = Shape()
```

OUTPUT

```
Drawing a Circle

TypeError: Can't instantiate abstract class Shape with abstract
method drawShape
```