PES UNIVERSITY

(*Established under Karnataka Act No. 16 of 2023*)
100 ft. Ring Road, Bengaluru – 560085, Karnataka, India

Programming with Python

Unit II – Object-Oriented Programming

Prepared by,

Niteesh K R,
Assistant Professor,
Department of Computer Applications,
PES University,
Bengaluru

Contents

Chapter 1

Introduction to OOP

1.1 Introduction

- **Object** is a tangible item that can be sensed
- Software Objects are modelling of real-world objects
- Object is a collection of **data** and **behaviour**
- **Object-Oriented Programming** means programming in terms of real-world entities and objects
- Revolves around **classes** and **objects**
- **Class** is a real-world entity while **object** is created on the basis of the class
- Classes are built to represent the general behaviour of a group of objects
- Every object created using a class gets the general behaviour defined in the class
- Creation of an object – Instantiation
- In OOP, a class is a template based on which an object (Instance) is created

1.1.1 Definitions

- **Object-Oriented Programming**
  - A programming paradigm in which the programs are written using classes and objects
  - Each class has data and associated behaviour
  - OOP is an effective representation and modelling of real-world entities
- **Class**
  - A group of real-world entities sharing similar attributes and behaviour
  - Each class has attributes in terms of data and behaviour in terms of functions
  - Can be seen as a template or a prototype
- **Object**

  - An instance of the class
  - Class is a template while object is a concrete implementation of the same

- **Attribute**
  - The general set of features shared by objects of the class
  - Defined in terms of variables in the functions of the class
  - Different objects have same set of attributes but, the values of the attributes may change from one object to another
  - Terms like **Data Member**, **Data Variable**, **Properties**, **Features of a class** can be used interchangeably

- **Function**
  - Each class definition has a set of functions
  - Each function is a behaviour that can be exhibited by the objects of the class
  - While data members add attributes to the class, member functions add functionality
  - Objects are necessary to use member functions

Note:
A class is a combination of both data members and member functions in a single entity

1.1.2 Analogies

- A class can be seen as a blueprint
- An object can be seen as an implementation of the blueprint
- Without an object, the class has no usage
- Without class, the object can not exist
- A **class** can be seen as a **blueprint** of the house
- The same blueprint can be used to construct multiple houses

- Each **house** can be seen as an **object**, which is built based on the blueprint i.e., the class
- The houses built with the same blueprint will be **similar**, not the same.
  - They may differ in terms of the paint, door pattern etc. while the overall structure remains the same.
  - Paint, door pattern etc. can be seen as the **attributes** of the **house** object
- In the same way, different **similar** objects can be created using one class

1.2 Advantages and Features of OOP

1.2.1 Advantages

- **Modularity**
  - Objects are created and managed independently
- **Code Reusablity**
  - Once defined, the same class can be used to create several objects
  - With inheritance, properties of one class can be made available in other classes

1.2.2 Features

- **Encapsulation**
  - Data members and member functions are put inside a single entity called **class**
  - Only the objects of the class can access the contents of the class
  - Provides security to data and functions from unauthorised access
- **Inheritance**
  - Properties of one class (Parent) can be made available to other classes (Children)

  - Ensures code reusability
  - Reduces duplication

- **Polymorphism**
  - Same entity (Functions or Operators) can perform multiple tasks
  - Objects can respond to different messages in different ways, which makes code more flexible and adaptable
  - Ensures ease of writing the programs
  - With inheritance, parent and child classes can have functions of the same name and the implementation depends on which class has been instantiated

Chapter 2

Classes and Objects

2.1 Class

- A blueprint or a template
- Defines the structure and behaviour of objects, including their properties and methods
- Includes different attributes as **data members** and behaviour as **member functions**
- Can be considered as a prototype for the category in general
- Used to create objects
- Encapsulates data and functions
- Can be seen as user-defined datatypes
- **Example**

  All cars have a brand name, model, and the year of production

  Car becomes the class and properties like brand name, model, and the year if production becomes data members of the class

2.1.1 Usage of Class, Object, and Instance Variables in Python

- **Defining class**
  - A class is defined using the keyword, **class** with an indented code block for definition of the class

    Syntax

    class ClassName:
    # BODY OF THE CLASS
  - General notation is to capitalise the first letter of the class
- **Functions**
  - Can be defined using the **def** keyword

- Not mandatory to have functions
- Syntax

```
class ClassName:
    def functionName(Parameters):
        # Function Body
    # Rest of the Class
```

- **Constructor**
  - When an object is created, the data members must be assigned some values
  - A special function called **__init__()** is executed when an object is created. This is called the **constructor**
  - Constructors in Python **DO NOT HAVE THE NAME OF THE CLASS**
  - Constructor function returns the object of the class implicitly, i.e., without a return statement
    Eg.,

```
class ClassName:
    def __init__(Parameters):
        # Body of the constructor
        # Run automatically when an object of
        # ClassName is created
        # Returns an object of type ClassName

    def someFunction(Parameters):
        # A Member Function
        # Body of someFunction()

    #Rest of the class
```

  - Not mandatory to have a constructor. In cases with no constructor function, a default constructor will be invoked implicitly
  - Constructors can not be overloaded in Python. If multiple constructor definitions are found, the latest one will be considered
    Eg.,

```
class ClassName:
    def __init__(Parameters):
        # Body of constructor 1

    def __init__(Parameters):
        # Body of constructor 2
        # This Constructor is executed

    def someFunction(Parameters):
        # A Member Function
        # Body of someFunction()

    #Rest of the class
```

- **Object**
  - An object is created using the class name
  - When an object is created, the constructor function will be executed
    Eg.,

```
class ClassName:
    def __init__(Parameters):
        print('Constructor')

objectName = ClassName()
# Prints 'Constructor' since __init__() is called
```

  - If no constructor is found, default constructor will be called
    Eg.,

```
class ClassName:
    pass

objectName = ClassName()
# Calls default constructor
```

  - In both the examples, type of ***objectName*** will be ***class ClassName***
  - In Python, all variables are objects of some class

- **Instance Variables**
  - Each object has its own set of properties
  - These properties are called **Instance Variables**
  - Instance variables differ from one object to another
  - The objects of the same class need not have same instance variables
  - While the name of the instance variables will be same for all the objects of a class, the values of the instance variables will be different
  - Instance variables are initialised with initial values while creating the object using the constructor of the class
  - **Accessing Instance Variables**
    - The values to be assigned to the instance variables are passed as parameters to the __init__()
    - Since the values of instance variables are different, they should be accessed using the object address
    - Each object is uniquely identified by the address of the object
    - This address needs to be passed as a parameter to the functions of the class which needs to access instance variables
    - Most of the functions of a class are instance functions and can access instance variables
    - To identify which object is being used to call the function or access an instance variable, a special parameter is to be passed to all the instance functions, including __init__()
    - This parameter is called as the **object reference parameter** and is usually denoted by **self**
    - While *self* is NOT A KEYWORD, it is a common practice to use *self* as the object reference parameter
    - Every instance function has **at least one parameter**, i.e., the object reference parameter
    - The first parameter of any instance function is the object reference parameter
    - The object reference parameter can be seen as an equivalent of **this** keyword in Java
    - The object reference parameter holds the **address of the object**

* While calling any function, it is not needed to pass the object as a parameter
* Instance functions and variables are accessed using the **.** notation

  Syntax

  objectName = ClassName(Parameters)
  objectName.someFunction(Parameters)
* If an instance function or a constructor has *n* parameters defined in the class while calling, they should be passed with *n - 1* parameters
* This is because the first parameter will be implicitly passed while calling the functions
* Example

  Create a class **Car** with the brand, model, and year of production as the parameters
  Create a constructor function and a display function
  Create 2 objects and call the display function

```
class Car:
    def __init__(self, brand, model, year):
        # self: Object Reference Parameter
        # brand, model, and year are to be assigned
        # to the instance variables
        # using . notation

        self.brand = brand
        self.model = model
        self.year = year
        print('Address: '+str(self))
        # Prints the hexadecimal address of the object
       # self.brand, self.model, self.year are instance variables

    def display(self):
        # self – Object Reference Parameter
        # Used to access instance variables
```

```
        print('Brand: '+self.brand)
        print('Model: '+self.model)
        print('Year: '+str(self.year))

car1 = Car('Tata', 'Harrier', 2023)
# Calls the __init__() with 3 + 1 (Object Ref) parameters

car2 = Car('Mahindra', 'XUV700', 2023)
# Calls the __init__() with 3 + 1 (Object Ref) parameters

car1.display()
# Calls the display() of Car using one parameter
# (Object Ref of car1)

car2.display()
# Calls the display() of Car using one parameter
# (Object Ref of car2)
```

  - Instance variables can also be accessed using objectName.variable outside of the class

    In the previous example, after creating objects:

    ```
    print(car1.brand) # Prints brand of car1 object
    ```

- Setting default values to instance variables
  - Default values can be set to instance variables
  - While creating an object, each object will have a copy of the default instance variable with same value
  - The values can then be modified using functions or the objects
  - Default values are set to the instance variables inside the constructor
  - Example

    Create a class **Car** with the brand, model, and year of production as the parameters

Set odometer to 0 while creating object (Default value to instance variable)
Create a constructor function and a display function
Create 2 objects and call the display function

```
class Car:
    def __init__(self, brand, model, year):
        # self: Object Reference Parameter

        self.brand = brand
        self.model = model
        self.year = year
        self.odometer = 0
        # Default value to the instance variable
        # self.brand, self.model, self.year are instance variables

    def display(self):
        # self – Object Reference Parameter
        # Used to access instance variables

        print('Brand: '+self.brand)
        print('Model: '+self.model)
        print('Year: '+str(self.year))
        print('Odometer: '+str(self.odometer))

car1 = Car('Tata', 'Harrier', 2023)
# Calls the __init__() with 3 + 1 (Object Ref) parameters

car2 = Car('Mahindra', 'XUV700', 2023)
# Calls the __init__() with 3 + 1 (Object Ref) parameters

car1.display()
# Calls the display() of Car using one parameter
# (Object Ref of car1)

car2.display()
# Calls the display() of Car using one parameter
# (Object Ref of car2)
```

car1.odometer = 100
 # Modifies odometer of car1 to 100
 # Does not affect car2

* Instance variables can also be accessed using objectName.variable outside of the class

  In the previous example, after creating objects:

  print(car1.brand) ***# Prints brand of car1 object***

– Attributes can be modified using
  * A function
  * Object

Chapter 3

Inheritance

3.1 Introduction to Inheritance

- Inheritance is an OOP feature in which a class **inherits** the properties of another class
- A class (Child class) can be built such that it gets properties of another class (Parent class)
- In inheritance, all the attributes and methods of the parent class become available in the child class
- Child class can also have additional attributes of its own
- Inheritance is a **strong form of association**
- Child classes can call parent class functions and access attributes

3.1.1 Types

Types of Inheritance include:

1. **Single Inheritance** – A parent class is inherited into a child class. Simplest form of inheritance
2. **Multi-level Inheritance** – A parent class is inherited into a child class, which is then inherited into another child class
3. **Multiple Inheritance** – Multiple parent classes are inherited into a child class. Child class inherits **all** the properties of the parent classes. Made possible in Python with the help of MRO (Method Resolution Order)
4. **Hierarchical Inheritance** – One parent class is inherited into multiple child classes
5. **Hybrid Inheritance** – Combinational inheritance of any two kinds of inheritance

3.2 Inheritance in Python

- In Python, different kinds of inheritance can be implemented
- While creating an object of the child class, it is necessary to initialise the instance variables of the parent class
- This can be done using **super()**
- super() provides access to the attributes of the parent class

3.2.1 Calling parent class Constructor

- The parent class constructor has to be called using the super()
- When this is done, the child class creates a copy of the instance variables of the parent class inside its object
- super() can also be used to access the members of the parent class
- Example

  Create a class called **Car** with properties brand, model, and year
  Create a child class called **EV** with an additional property called battery
  Using the constructor method of EV, call the constructor for Car
  Create a function called display() in Car which displays the instance variables
  Create a function displayEV() in EV which calls the display() of Car and also prints battery attribute

```
class Car:
    def __init__(self, brand, model, year):
        self.brand, self.model, self.year = brand, model, year

    def display(self):
        print('Brand:'+self.brand)
        print('Model:'+self.model)
        print('Year:'+str(self.year))

class EV(Car):
```

```
    # class Child(Parent) means Child is inheriting Parent
        def __init__(self, brand, model, year, battery):
            super().__init__(brand, model, year)
            # Calls the parent class constructor with 3 parameters
            self.battery = battery
            # Additional attribute of class EV

        def displayEV(self):
            super().display()
            # Calls the display() in Car
            print('Battery: '+str(self.battery))

    ev = EV('Tata', 'Nexon.ev', 2024)
    # Creates an object of EV class
    # Calls the constructor of EV class
    # Which internally calls the constructor of Car
    ev.displayCar()
```

3.2.2 Method Overriding

- It is possible for methods of the child class to have the same name as that of the parent class
- In such cases, method overriding will be effected
- Method overriding happens when the parent and child class share the same function signature (Function name and the number of parameters)
- When method overriding happens and a method is called using the parent class object, it executes the method in the parent class
- If the child class object is used to call the method, it will call the child class method
- Note that both the methods will share the same name
- Example

  Create a class called **Car** with properties brand, model, and year

Create a child class called **EV** with an additional property called battery
Using the constructor method of EV, call the constructor for Car
Create a function called display() in both the classes which displays the instance variables

Create objects of both the classes and call display()

```
class Car:
    def __init__(self, brand, model, year):
        self.brand, self.model, self.year = brand, model, year

    def display(self):
        print('Brand:'+self.brand)
        print('Model:'+self.model)
        print('Year:'+str(self.year))

class EV(Car):
# class Child(Parent) means Child is inheriting Parent
    def __init__(self, brand, model, year, battery):
        super().__init__(brand, model, year)
        # Calls the parent class constructor with 3 parameters
        self.battery = battery
        # Additional attribute of class EV

    def display(self):
        super().display()
        # Calls the display() in Car
        print('Battery: '+str(self.battery))

ev = EV('Tata', 'Nexon.ev', 2024)
# Creates an object of EV class
# Calls the constructor of EV class
# Which internally calls the constructor of Car
ev.display()
# Calls the display() function of EV class
car = Car('Tata', 'Nexon', 2023)
# Creates an object of class Car
```

car.display()
 # Executes the display() in class Car

3.2.3 Method Resolution Order (MRO)

- MRO defines the order in which the methods are **resolved** in Python
- MRO is used to search for an instance function or a data member
- MRO in Python is ***BOTTOM – TOP and LEFT – RIGHT***
- MRO tells the interpreter where to search for a method in case of inheritance
- For all classes, *object* is the base class
- Example

  A class *A* is the parent class for *B*
  Create empty classes, without implementation
  Display MRO of class B
  class A: pass
  class B(A): pass
  # B is the child class of A

  print(B.mro())
  # Prints [class __main__.B, class __main__.A, class object]

- The example demonstrates that the MRO is **(B, A, object)**
- If an object of class *B* is created and a method is called using the same, Python MRO first searches for the method in class *B*, if it is found, the search will stop and method will be executed. If the method is not found in class *B*, it will be searched for in class *A* and executed if found in class *A*.
- **Case 1: Parent and child class with the same function signature**
  - If parent and child class have the same function signature and the method is called using the child class object, the method will be overridden

- If the method is called with the child class object, the method in the child class gets called
- If the method is called with the parent class object, the method in the parent class gets called
- Example

```
class A:
    def foo(self): print('class A')

class B(A):
    def foo(self): print('class B')

a = A()
# a is an object of class A
b = B()
# b is an object of class B
b.foo()
# Prints class B
a.foo()
# Prints class A
```

- **Case 2: Method present only in child class**
  - If the method is present only in the child class but, not in parent class, the method can be accessed only with the child class object
  - Throws an AttributeError if tried to access with the parent class object
  - Example

```
class A: pass
class B(A):
    def foo(self): print('class B')

a = A()
b = B()
b.foo()
# Prints class B
a.foo()
```

 # AttributeError: No attribute called foo found in A

- **Case 3: Method found in parent class only**
  - If the method is present only in the parent class, it can be accessed by objects of both child and parent classes
  - Since the method is a property of parent class, it becomes available in child class as well
  - When the method is called with the child class object, the method will be searched for in the child class
  - Since the method will not be found in the child class, MRO searches for the method in the immediate parent class and executes the method if found in the parent class
  - Example

```
class A:
    def foo(self): print('class A')

class B(A): pass

a = A()
b = B()
b.foo()
# Prints class A
a.foo()
# Also prints class A
```

- **Case 4: Method absent in both classes**
  - If the method is accessed through objects of child and parent classes but, the method is absent in both the classes, AttributeError will be thrown
  - Example

```
class A: pass

class B(A): pass

a = A()
b = A()
a.foo()
# AttributeError b.foo() # AttributeError
```

3.2.4 Multiple Inheritance in Python

- Multiple Inheritance is possible in Python because of MRO
- Example

  Consider that class *C* inherits classes *A* and *B*

```
class A: pass
class B: pass
class C(A, B): pass

print(C.mro())
# Prints [class __main__.C, class __main__.A, class __main__.B, class object]
```

- In the example, MRO for class C would be **(C, A, B, object)**
- If class C(A, B) is changed as class C(B, A), this will change the MRO as well to **(C, B, A, object)**
- Calling constructors of both the super classes
  Example

  Consider class *C* inheriting from classes *A* and *B*
  Call the constructors of both the classes in the constructor of class C

```
class A:
    def __init__(self): print('A')
```

```
class B:
    def __init__(self): print('B')

class C(A, B):
    def __init__(self):
        super().__init__()
        # Calls the constructor of A

        super(A, self).__init__()

        # Specifies that the MRO should start from class A
        # Not class C
        # super() calls the constructor of the next class in the MRO list
        # which is class B

c = C()
# Prints A, B, C in 3 lines
```

3.2.5 Instances as Attributes

- When modelling something from the real world in code, the user may find that more and more detail is being added to a class
- This causes a growing list of attributes and methods and the files are becoming lengthy
- In these situations, recognize that part of one class can be written as a separate class
- Break the large class into smaller classes that work together
- Example

  Create a class Car with brand, model, year
  Inherit Car into EV with battery and range as additional properties
  Create a class EVProps with battery and range as properties
  Create an object of EVProps in the constructor of EV

```
class Car:
    def __init__(self, b, m, y): self.b, self.m, self.y = b, m, y

class EVProps:
    def __init__(self, bt, r): self.bt, self.r = bt, r

class EV(Car):
    def __init__(self, b, m, y, bt, r):
        super().__init__(b, m, y)
        # Calls constructor of Car

        self.evpropprs = EVProps(bt, r)
        # Creates an object of EVProps as an attribute of EV

    def display(self):
        print('Brand: '+self.b)
        print('Model: '+self.m)
        print('Year: '+str(self.y))
        print('Battery: '+str(self.evprops.bt))
        print('Range: '+str(self.evprops.r))

        # Access bt and r attributes of EVProps using self.evprops
        # self.evprops is an object of EVProps

ev = EV('Tata', 'Nexon.ev', 2024)
ev.display()
```

Chapter 4

Polymorphism

4.1 Introduction to Polymorphism

- One entity can take different forms
- Python has different types of Polymorphism

4.1.1 Types of Polymorphism in Python

- **Operator Polymorphism**
  - Same operator can be used with different types to generate different results
  - **+** with numeric types returns the sum while **+** with string types returns the concatenated string and with lists, returns an extended list

    print(1+2) ***# Prints 3***
    print('a'+'b') ***# Prints 'ab'***
    print([1, 2]+[3, 4]) ***# Prints [1, 2, 3, 4]***
  - ***** with numeric types returns the product while ***** with string type and an integer (Say *n*) returns the string repeated *n* times and with lists, returns an extended list with the items repeated *n* times

    print(1*2) ***# Prints 2***
    print('a'*5) ***# Prints 'aaaaa'***
    print([1, 2]*3) ***# Prints [1, 2, 1, 2, 1, 2]***
- **Function Polymorphism**
  - Same function works with different datatypes
  - Example
    len(string) returns the length of the string
    len(List) returns the number of items in the List
    len(Tuple) returns the number of items in the Tuple
    len(Dictionary) returns the number of *key:value* pairs in the Dictionary
  - User defined function polymorphism can be of 2 types
    - Method Overloading – Multiple Functions having same method name but, different number and type of parameters with different behaviour

    - Method Overriding – Parent and child classes having the same method signature but, different behaviour **(Refer to Method Overriding of Inheritance in Python)**

- **Class Polymorphism**
  - Different classes with the same function signatures and same instance variables but, different function body and different instance variable values
  - Classes share similar structure
  - Classes must be independent, i.e., not inherited from one another
  - Example

```
class Cat:
    def __init__(self): print('Cat')

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

class Dog:
    def __init__(self): print('Dog')

    def display(self): print('Dog Display')
```

    Here, classes Cat and Dog have a constructor with one parameter and a method called display()

4.2 Duck Typing

- Term commonly related to dynamically typed programming languages and polymorphism
- With Duck Typing, type or the class does not matter
- Instead, the presence of a given method or attribute matters
- Used in iteration
  - The same for-in loop works with all kinds of iterables (Lists, Tuples, Sets, Strings, Dictionaries)

  - This is because of Duck Typing
  - No matter how different these objects are in terms of application, they are treated equally because of Duck Typing and the internal working of the iterator function

- With Duck Typing, as long as a class implements a method, it will be executed regardless of what class it is

Chapter 5

Encapsulation

5.1 Introduction

- Putting together data members and member functions into an entity called class
- Enhances data security by preventing unauthorised access
- Data members and member functions can only be accessed using the object of the class
- Enhances code modularity
- Encapsulation allows programmers better control of how data flows in their programs, and it protects that data
- In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class

5.2 Access Modifiers

- Purpose of Encapsulation is to provide security to data
- Access Modifiers allow different levels of security to the data members and member functions
- Python has 3 Access Modifiers
  - Public
  - Private
  - Protected

5.2.1 Public Access Modifier

- Data members and member functions that are declared as **Public** if they do not have any trailing underscores in their identifier names
- Once an attribute is set as public, it can be accessed by the object of the class directly

- Public attributes are also inherited, i.e., public attributes of parent class are available to the child class as well
- Example

  Create class A and inherit class B from it
  class A to have a public data member
  Access the public data member of class A using the object of A as well as the object of B

```
class A:
    def __init__(self, a): self.a = a
    # self.a is public

class B(A):
    def __init__(self, a, b):
        super().__init__(a)
        # Calling constructor of A

        self.b = b
        # self.b is public in B

a = A(10)
print('Accessing using object of A: '+str(a.a))
b = B(100, 200)
print('Accessing using object of B: '+str(b.a)+' and '+str(b.b))
```

5.2.2 Private Access Modifier

- Data members and member functions that are declared as **Private** if they have **2 trailing underscores** (__) in their identifier names
- Once an attribute is set as private, it can be accessed by the functions of the class only
- Private attributes are not inherited, i.e., private attributes of the parent class are not available to the child class

- AttributeError is thrown when an object tries to access a private variable
- Example

  Create class A and inherit class B from it
  class A to have a private data member
  Access the private data member of class A using the object of A as well as the object of B

```
class A:
    def __init__(self, a): self.__a = a
    # self.__a is private

class B(A):
    def __init__(self, a, b):
        super().__init__(a)
        # Calling constructor of A

        self.b = b
        # self.b is public in B

a = A(10)
print('Accessing using object of A: '+str(a.__a))
# Throws AttributeError: A has no attribute __a b = B(100, 200)
print('Accessing b using object of B: '+str(b.b))
# Works fine
print(' and '+str(b.__a))
# Throws AttributeError: B has no attribute __a
```

Accessing Private Data Members using Getters and Setters

- Private data members can only be accessed by the functions of the class
- To access private members, public methods are required in the class
- **Getter** – A public method which returns the private instance variable/data member
  Structure of Getters

def get_var(self):
return self.__var
 # Returns __var which is private

- **Setter** – A public method which updates the value of the private data member using the value passed as a parameter
  Structure of Setters

  def set_var(self, newValue):
  self.__var = newValue
  # Modifies __var with newValue

- Example

  Create a class Car with brand, model, and year as private members
  Create an object with brand = Tata, model = Nexon, year = 2023
  Using getters, display the private data members using objects
  Using setters, modify brand to 'Mahindra', model to 'XUV700', and year to 2022
  Display using getters

```
class Car:
    def __init__(self, b, m, y): self.__b, self.__m, self.__y = b, m, y

    # Getters

    def getBrand(self): return self.__b
    def getModel(self): return self.__b
    def getYear(self): return self.__b

    # Setters

    def setBrand(self, nv): self.__b = nv
    def setModel(self): self.__b = nv
    def setYear(self): self.__b = nv
```

```
car = Car('Tata', 'Nexon', 2023)
print('Before Modification: ')
print(car.getBrand())
print(car.getModel())
print(car.getYear())

car.setBrand('Mahindra')
car.setModel('XUBV700')
car.setYear(2022)

print('After Modification: ')
print(car.getBrand())
print(car.getModel())
print(car.getYear())
```

- Setters may also check a certain condition before modifying the value of the private data member
- Example

Create a class Student and create an object
Let the private data members be name and year of birth
Display using getters
Define setters. Define setYOB() it only modifies if the new value is between 1995 and 2005. Display 'Invalid value. No modification' if the condition fails
Display after modification using getters

```
class Student:
    def __init__(self, n, yob): self.__n, self.__yob = n, y

    # Getters
    def getName(self): return self.__n
    def getYOB(self): return self.__yob


    # Setters
    def setName(self, n): self.__n = n
```

```
    def setYOB(self, y):
        if y in range(1995, 2006):
            self.__yob = y
        else:
            print('Invalid value. No modification')

s = Student('Name', 2002)
print('Before Modification: ')
print(s.getName())
print(s.getYOB())

s.setName('Name2')
s.setYOB(2000)
print('After Modification: ')
print(s.getName())
print(s.getYOB())

s.setName('Name3')
s.setYOB(2010)
# Error message printed
print('After Modification: ')
print(s.getName())
print(s.getYOB())
# YOB will not be changed
```

Accessing Private Data Members using Name Mangling

- Another way of accessing private members is through Name Mangling
- If __var is a private data member in class A and is to be accessed using an object of A, a:

  Name Mangling format: **a._A.__var**
- It is not advisable to use Name Mangling

5.2.3 Protected Access Modifier

- Data members and member functions that are declared as **Protected** if they have **1 trailing underscore (_)** in their identifier names
- Once an attribute is set as protected, it can be accessed by the functions of the class only
- Protected attributes are inherited, i.e., protected attributes of the parent class are available to the child class
- AttributeError is thrown when an object tries to access a protected variable

5.2.4 Public vs. Private vs. Protected

Benchmark	**Public**	**Private**	**Protected**
Access From Functions of the class	Allowed	Allowed	Allowed
Access From Child class	Allowed	Not Allowed	Allowed
Access From Object	Allowed	Not Allowed	Not Allowed
Declaration	No underscore before identifier	__before identifier	_ before identifier

Table 5.1: Public vs. Private vs. Protected

Note:

It is always a good practice to define data members as *private* and instance/member functions as *public*

5.2.5 Example Notebook: Encapsulation

Encapsulation #1

March 22, 2024

[3]:
```python
# Public data members
# Accessed inside class, inside child class, and using object
# No underscore behind the name of the data member

class A:
    # Constructor
    def __init__(self, publicVar):
        self.publicVar = publicVar # self.publicVar is a public data member
    def display(self):
        print('Accessing using function: '+str(self.publicVar))

class B(A):
    def __init__(self, parentPublicVar):
        super().__init__(parentPublicVar)

# Object of A
a = A(1000)
# Access public data member using object
print('Accessing using object: '+str(a.publicVar))
# Access public data member using function
a.display()
# Access public data member using object of the child class
b = B(100)
print('Accessing using child class object: '+str(b.publicVar))
```

```
Accessing using object: 1000
Accessing using function: 1000
Accessing using child class object: 100
```

[6]:
```python
# Private data members
# Private data members can be accessed inside the class only
# Not accessed using object or child class object
# Can be accessed using name mangling or using member functions
# Private data members have __ behind the varname

class A:
    def __init__(self, a, b):
        self.a = a # Public data member
```

```
        self.__b = b # self.__b is a private data member
    def getB(self): # Getter function for private data member, 'b'
        return self.__b
    def setB(self, newValue): # Setter function
        self.__b = newValue

a = A(100, 200)
# Access private member using function
print('Accessing private member using function: '+str(a.getB()))
# Modify private member using function
a.setB(300)
# Access private member using function after modification
print('Accessing private member using function after modification: ')
print(a.getB())
# Access private member using object
print('Accessing private member using object: ')
print(a.__b) # Throws an AttributeError, since __b is private
```

```
Accessing private member using function: 200
Accessing private member using function after modification:
300
Accessing private member using object:
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[6], line 26
     24 # Access private member using object
     25 print('Accessing private member using object: ')
---> 26 print(a.__b) # Throws an AttributeError, since __b is private

AttributeError: 'A' object has no attribute '__b'
```

[8]:
```
# Private data members
# Private data members can be accessed inside the class only
# Not accessed using object or child class object
# Can be accessed using name mangling or using member functions
# Private data members have __ behind the varname

class A:
    def __init__(self, a, b):
        self.a = a # Public data member
        self.__b = b # self.__b is a private data member
    def getB(self): # Getter function for private data member, 'b'
        return self.__b
    def setB(self, newValue): # Setter function
        self.__b = newValue
```

```
class B(A):
    def __init__(self, a, b):
        super().__init__(a, b)
a = A(100, 200)
# Access private member using function
print('Accessing private member using function: '+str(a.getB()))
# Modify private member using function
a.setB(300)
# Access private member using function after modification
print('Accessing private member using function after modification: ')
print(a.getB())
# Access private member using object of the child class
b = B(100, 200)
print('Accessing private member using object of the child class: ')
# print(b.__b) # Throws an AttributeError, since __b is private in class A
```

```
Accessing private member using function: 200
Accessing private member using function after modification:
300
Accessing private member using object of the child class:
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[8], line 30
     28 b = B(100, 200)
     29 print('Accessing private member using object of the child class: ')
---> 30 print(b.__b) # Throws an AttributeError, since __b is private

AttributeError: 'B' object has no attribute '__b'
```

[1]:
```
# Name mangling for private data members
class A:
    def __init__(self, privateMember):
        self.__privateMember = privateMember
# Create an object of class A
a = A(10)
# Access using object
# print(a.__privateMember) # AttributeError
print('Name mangling: ')
# objectName._ClassName__PrivateVariableName
print(a._A__privateMember)
```

```
Name mangling:
10
```

Chapter 6

Class and Instance Variables

6.1 Instance Variables

- Instance variables are the variables which are defined in the class but, each object has a different copy of these instance variables
- Instance variables can be accessed using object outside the class
- Instance variables can be accessed using the object reference parameter *(self)* inside the functions of a class
- Functions that can access instance variables is an instance method
  Eg., __init__(), other methods seen so far
- Even though the name of the instance variables is same in different objects, the values stored in them will be different
- Example

```
class A:
    def __init__(self, a): self.instance_var = a

a = A(100)
b = A(200)
# a and b are the objects of the same class and have an attribute instance_var print('Object a: ' +str(a.instance_var)
# Prints 100

print('Object b: ' +str(b.instance_var)
# Prints 200
```

- Instance variables are modified using the object or methods of the class
- If the instance variables are private, they can only be modified using public methods of the class

6.2 Class Variables

- Data Members that are shared between different objects of the class are Class Variables

- Shared variable
- Modification to class variables will reflect in all the objects
- Accessed using the object or the class name
- Modified using class name (ClassName.class_var = 100)

  Example

```
class A:
    cv = 100
    # cv is the class variable

a = A()
b = A()
print(A.cv)
print(a.cv)
print(b.cv)
# Prints the same value, 100

A.cv = 200
print(A.cv)
print(a.cv)
print(b.cv)
# Prints the same value, 200
```

- If modification of class variables is attempted using the object, it will not modify the class variable, instead, it creates a new instance variable in that object only

  Example

```
class A:
    cv = 100
    # cv is the class variable

a = A()
b = A()
```

```
print(A.cv)
print(a.cv)
print(b.cv)
# Prints the same value, 100

a.cv = 200
print(A.cv)
# Prints 100
print(a.cv)
# Prints 200, which is an instance variable
print(b.cv)
# Prints 100
```

6.3 Instance Variables vs. Class Variables

Instance Variables	Class Variables
Not shared across objects	Shared across objects
Modification in one object does not affect another	Modification reflects in all the objects
Accessed and modified using object and its reference inside the class Defined inside __init__()	Accessed using object but, modified using class only Defined outside methods, inside class
Created when an object is created	Created when the class part is interpreted

Table 6.1: Instance Variables vs. Class Variables

Chapter 7

Static and Class Methods

7.1 Decorators

- Decorators add additional functionality to a function
- Decorators modify a function by wrapping it with additional functionality
- Decorators are of the form @decoratorName and are mentioned above the function to be *decorated*
- Decorator is also a function which accepts another function as a parameter, returns a modified version of the function
- Example

```
def decoratorFunction(func):
    # This is a decorator function

    def wrap():
        print('Before')
        func()
        print('After')

    return wrap

@decoratorFunction
def foo():
    print('Hello!')

foo()
# Prints Before, Hello!, After in 3 lines
```

7.2 Class Methods

- Class Methods can access the state of the class
- Defined with decorator, **@classmethod**
- Bound to the class, not objects

- Do not have access to individual objects but, have access to the state of the class
- The first parameter of a class method will be the state of the class, often denoted by **cls**. Note that cls, like self, is not a keyword
- The class state parameter, cls, has the name of the class
- Class Methods can be used to modify the class variables
- Can be accessed using the class name and object of the class
- Example

  Create a class with a class variable, cv
  Create 2 objects and display the value of cv
  Modify the value of cv using a class method
  Display the modification

```
class A:
    cv = 100

    @classmethod
    def modify(cls, newValue):
        # cls is the class state parameter
        cls.cv = newValue

a, b = A(), A()
print(a.cv)
print(b.cv)
# Both print 10

A.modify(20)
print(a.cv)
print(b.cv)
# Both print 20
a.modify(30)
print(a.cv)
print(b.cv)
# Both print 30
```

7.3 Static Methods

- Static Methods are not dependent on class or the objects of the class
- Declared with **@staticmethod** decorator
- They do not have access to the state of the class or the object
- No *self* or *cls* parameter
- Do not operate on class or instance variables
- Can be used an utility or helper functions (Supplying a constant value or checking some condition)
- Can be accessed by the name of the class or object

7.4 Class Method vs. Static Method vs. Instance Method

Benchmark	**Class Method**	**Static Method**	**Instance Method**
Able to access	State of the class	Neither state of the class or object	Object
Called using	Name or object of the class	Name or object of the class	Object of the class only
Primary usage	To modify class variables	As helper functions	To access and modify instance variables
Decorator	@classmethod	@staticmethod	No Decorator
Min. No. of Parameters	1 (cls)	0	1 (self)
Bound to	Class, not objects	Neither class, nor objects	Objects only

Table 7.1: Class Method vs. Static Method vs. Instance Method

7.5 Example Notebook: Class and Static Methods

Class Methods #1

March 22, 2024

[ ]:
```
# Class Method
```

[3]:
```
class Student:
    uni = 'PES'
    def __init__(self, srn, name):
        self.srn, self.name = srn, name # Instance Variables
    @classmethod
    def set_uni(c, newname):
        print('Inside class method')
        print('State: '+str(c))
        c.uni = newname

s1 = Student(101, 'Student1')
s2 = Student(102, 'Student2')
print('First Object: ')
print(s1.srn)
print(s1.name)
print(s1.uni)
print('Second Object: ')
print(s2.srn)
print(s2.name)
print(s2.uni)
print('After modification of class variable using class method: ')
# Call class method using object of the class (Any object)
s1.set_uni('PES University')
print('First Object: ')
print(s1.srn)
print(s1.name)
print(s1.uni)
print('Second Object: ')
print(s2.srn)
print(s2.name)
print(s2.uni)
s3 = Student(103, 'Student3')
print('Third Object: ')
print(s3.srn)
print(s3.name)
print(s3.uni)
```

```
First Object:
101
Student1
PES
Second Object:
102
Student2
PES
After modification of class variable using class method:
Inside class method
State: <class '__main__.Student'>
First Object:
101
Student1
PES University
Second Object:
102
Student2
PES University
Third Object:
103
Student3
PES University
```

[ ]:
```
# Static Method
```

[3]:
```
class Circle:
    def __init__(self, r):
        self.r = r

    # Define a static method returning the value of pi whenever needed
    @staticmethod
    def pi():
        return 22/7

    def area(self):
        return Circle.pi() * self.r ** 2

    def circum(self):
        return Circle.pi() * self.r * 2

c = Circle(7)
print('Pi: ')
print(c.pi())
print(Circle.pi())
print('Area: ')
print(c.area())
```

```
print('Circum: ')
print(c.circum())
```

```
Pi:
3.142857142857143
3.142857142857143
Area:
154.0
Circum:
44.0
```

Chapter 8

Composition and Aggregation

8.1 Composition

- A form of association between classes
- Represents **part-of** relationship between classes
- Multiple **Components** put together form **Composite**
- Components are a **part of** Composite
- Bidirectional dependency – Components and Composite are dependent on one another
- Weaker form of association than inheritance
- Classes share component-composite relationship, not parent-child relationship
- Component is contained within the Composite
- Bottom up Approach – Create Component classes, create objects of Component classes in Composite class
- Example

  A Car is made up of Engine, Steering, and 4 Wheels
  Car – Composite

  Create classes Engine, Steering, and Wheel. Create instance variables with these classes instantiated inside Car

8.1.1 Example Notebook – Composition

Composition #1

March 22, 2024

[ ]:
```
# Create a class Car such that it has Engine, Wheels, and Steering
```

[1]:
```
class Engine:
    def __init__(self, size, cylinder):
        self.size, self.cylinder = size, cylinder
    def display(self):
        print(self.size)
        print(self.cylinder)

class Wheel:
    def __init__(self, wheelSize, Type):
        self.wheelSize, self.Type = wheelSize, Type
    def display(self):
        print(self.wheelSize)
        print(self.Type)

class Steering:
    def __init__(self, spk, ctrl):
        self.spk, self.ctrl = spk, ctrl
    def display(self):
        print(self.spk)
        print(self.ctrl)

class Car:
    def __init__(self, make, model, year):
        self.make, self.model, self.year = make, model, year
        # Have an engine inside Car
        self.engine = Engine(input('Engine Size: '), input('Number of Cylinders:
 ↪ '))
        self.wheels = [Wheel(input('Wheel Size: '),
                             input('Wheel Type: ')) for i in range(4)]
        self.steering = Steering(input('Number of Spokes: '),
                                 input('Steering Mount Controls: '))
    def display(self):
        print('Car: ')
        print(self.make)
        print(self.model)
```

```
        print(self.year)
        print('Engine: ')
        self.engine.display()
        print('Wheels: ')
        for i in self.wheels: i.display()
        print('Steering: ')
        self.steering.display()

car = Car('Tata', 'Nexon', 2023)
car.display()
```

```
Engine Size: 1196
Number of Cylinders: 4
Wheel Size: R16
Wheel Type: Alloy
Wheel Size: R16
Wheel Type: Alloy
Wheel Size: R16
Wheel Type: Alloy
Wheel Size: R16
Wheel Type: Alloy
Number of Spokes: 2
Steering Mount Controls: Yes
Car:
Tata
Nexon
2023
Engine:
1196
4
Wheels:
R16
Alloy
R16
Alloy
R16
Alloy
R16
Alloy
Steering:
2
Yes
```

Composition #2

March 22, 2024

[3]:
```
# Laptop has one set of Input, Output, and Processor each
# Input has a Keyboard (No. of Keys) and a Mouse (isWired)
# Output has a Monitor (Size) and a Speaker (Brand)
# Processor has a RAM (Capacity) and a Core (Count of Cores)

class Keyboard:
    def __init__(self, n):
        self.n = n
    def display(self):
        print('Keyboard')
        print('Number of Keys: '+str(self.n))
class Mouse:
    def __init__(self, isWired):
        self.isWired = isWired
    def display(self):
        print('Mouse')
        print('isWired: '+str(self.isWired))
class Monitor:
    def __init__(self, size):
        self.size = size
    def display(self):
        print('Monitor')
        print('Screen Size: '+str(self.size))
class Speaker:
    def __init__(self, br):
        self.br = br
    def display(self):
        print('Speaker')
        print('Brand: '+self.br)
class RAM:
    def __init__(self, capacity):
        self.capacity = capacity
    def display(self):
        print('RAM')
        print('Capacity (GB): '+str(self.capacity))
class Core:
    def __init__(self, countOfCores):
```

```
        self.countOfCores = countOfCores
    def display(self):
        print('Core')
        print('Number of Cores: '+str(self.countOfCores))

class Input:
    def __init__(self):
        self.keyboard = Keyboard(int(input('Number of Keys: ')))
        self.mouse = Mouse(int(input('Wired (1 -- Wired, 0 -- Wireless): ')))
    def display(self):
        print('Input')
        self.keyboard.display()
        self.mouse.display()

class Output:
    def __init__(self):
        self.monitor = Monitor(float(input('Screen Size: ')))
        self.speaker = Speaker(input('Brand of Speaker: '))
    def display(self):
        print('Output')
        self.monitor.display()
        self.speaker.display()

class Processor:
    def __init__(self):
        self.ram = RAM(int(input('Capacity (GB): ')))
        self.core = Core(int(input('Number of Cores: ')))
    def display(self):
        print('Processor')
        self.ram.display()
        self.core.display()

class Laptop:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        self.inputDevice = Input()
        self.output = Output()
        self.processor = Processor()
    def display(self):
        print('Laptop')
        print('Brand: '+str(self.brand))
        print('Model: '+str(self.model))
        self.inputDevice.display()
        self.output.display()
        self.processor.display()
```

```
l = Laptop('Apple', 'MacBook Pro')
l.display()
```

```
Number of Keys: 80
Wired (1 -- Wired, 0 -- Wireless): 0
Screen Size: 15
Brand of Speaker: Apple
Capacity (GB): 16
Number of Cores: 8
Laptop
Brand: Apple
Model: MacBook Pro
Input
Keyboard
Number of Keys: 80
Mouse
isWired: 0
Output
Monitor
Screen Size: 15.0
Speaker
Brand: Apple
Processor
RAM
Capacity (GB): 16
Core
Number of Cores: 8
```

8.2 Aggregation

- Another form of association between classes
- Classes share a **has-a** relationship
- Parent object has reference to a child object
- Weaker form of association than composition
- Child and parent objects can exist independently
- Unidirectional dependency
- More flexible than composition
- Top down Approach – Create references of child objects in parent class
- Example

  A university has departments, departments have professors, professors teach courses

- Example

  A paragraph has sentences, a sentence has words, a word has letters

8.2.1 Example Notebook – Aggregation

Aggregation #1

March 22, 2024

[1]:
```
# A University has 2 Departments, Each Department has 2 Professors
# Each Professor handles 2 Courses
```

[10]:
```
class Course:
    def __init__(s, n):
        s.cn = n
class Prof:
    def __init__(s, n):
        s.pn = n
        s.cl = []
    def add_course(s, c):
        s.cl.append(c)
class Dept:
    def __init__(s, n):
        s.dn = n
        s.pl = []
    def add_prof(s, p):
        s.pl.append(p)
class Uni:
    def __init__(s, n):
        s.un = n
        s.dl = []
    def add_dept(s, d):
        s.dl.append(d)
u = Uni('PES University')
# Create Departments, Add to the Uni
d1 = Dept('CA')
d2 = Dept('CSE')
u.add_dept(d1)
u.add_dept(d2)
# Create Professors, Add to the Depts.
p1 = Prof('P1')
p2 = Prof('P2')
p3 = Prof('P3')
p4 = Prof('P4')
d1.add_prof(p1)
d1.add_prof(p2)
d2.add_prof(p3)
```

```
d2.add_prof(p4)
# Create Courses, Add to the Profs.
c1 = Course('C1')
c2 = Course('C2')
c3 = Course('C3')
c4 = Course('C4')
c5 = Course('C5')
c6 = Course('C6')
c7 = Course('C7')
c8 = Course('C8')
p1.add_course(c1)
p1.add_course(c2)
p2.add_course(c3)
p2.add_course(c4)
p3.add_course(c5)
p3.add_course(c6)
p4.add_course(c7)
p4.add_course(c8)

# Display Uni name
print('University: '+u.un)
# Display Departments, Professors, Courses

for dept in u.dl:
    print('Department: ')
    print(dept.dn)

    for prof in dept.pl:
        print('Professor: ')
        print(prof.pn)
        print('Courses')
        for course in prof.cl:
            print(course.cn)
        print()
    print()
```

```
University: PES University
Department:
CA
Professor:
P1
Courses
C1
C2

Professor:
```

P2
Courses
C3
C4

Department:
CSE
Professor:
P3
Courses
C5
C6

Professor:
P4
Courses
C7
C8

8.3 Composition vs. Aggregation

Benchmark	Composition	Aggregation
Relationship	Part-of	Has-a
Strength of association	Weaker than inheritance	Weaker than composition
Dependency	Bidirectional – Composite and Component and co-dependent	Unidirectional
Implementation	Component object is inside Composite object	Reference of child object inside parent object

Table 8.1: Composition vs. Aggregation