Don Bosco Institute of Technology

Department of Electronics and Telecommunication Engg.
SUB: Skill Lab (ECL404)

Classes and Objects

Expt No :	**04**
Aim:	To create Classes and Objects.
Tool used:	Anaconda Navigator+ Jupiter Notebook
Theory:	Answer in brief (not more than 6 to7 sentences.) Resource: https://www.w3schools.com/python/

1. What is an Object-Oriented Programming language? Enlist at least 4 different software languages which are object oriented.

A: Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields or attributes, and code, in the form of procedures or methods. OOP languages enable the organization of software as a collection of objects that interact with each other.

Here are four different object-oriented programming languages:

Java: Developed by Sun Microsystems (now owned by Oracle), Java is a widely used object-oriented programming language known for its platform independence and robustness. It is commonly used for building enterprise-level applications, web applications, mobile apps (Android), and more.

C++: An extension of the C programming language, C++ is a powerful and versatile object-oriented programming language. It provides features such as classes, inheritance, polymorphism, and encapsulation. C++ is commonly used in system/application software, game development, and performance-critical applications.

Python: Python is a high-level, interpreted programming language that emphasizes simplicity and readability. It supports multiple programming paradigms, including object-oriented, imperative, functional, and procedural programming. Python is widely used in web development, data analysis, artificial intelligence, scientific computing, and more.

C: Developed by Microsoft, C is an object-oriented programming language designed for building applications on the Microsoft .NET framework. It shares syntax similarities with C++ and Java but offers features specific to the .NET platform. C# is commonly used for developing Windows desktop applications, web applications, and games using the Unity game engine.

2. How is an Object-oriented language different from a normal programming language.

A: An object-oriented programming (OOP) language differs from a procedural or functional programming language in several key ways:

Object-Oriented Paradigm:

In an object-oriented language, the primary paradigm is object-oriented programming, where concepts such as encapsulation, inheritance,

polymorphism, and abstraction are central.
In contrast, procedural or functional languages focus on procedures or functions as the primary building blocks of programs, with less emphasis on organizing code around objects and their interactions.

Objects and Classes:

Object-oriented languages support the concept of objects and classes. Objects are instances of classes, which encapsulate data (attributes) and behaviors (methods) related to a specific entity or concept.
Procedural languages typically lack built-in support for objects and classes, although some may have rudimentary mechanisms for implementing similar concepts through structs or records.

Inheritance:

Object-oriented languages often support inheritance, allowing new classes (subclasses) to inherit attributes and methods from existing classes (superclasses). This promotes code reuse and facilitates hierarchical relationships between classes. Procedural languages may lack native support for inheritance, requiring developers to implement similar functionality through other means such as code duplication or composition.

Polymorphism:

Object-oriented languages support polymorphism, which allows objects of different classes to be treated uniformly through a common interface. This enhances flexibility and extensibility in code design. Procedural languages may support polymorphism to some extent through techniques such as function overloading or generic programming, but they often lack the rich polymorphic capabilities provided by object-oriented languages.

Encapsulation:

Object-oriented languages emphasize encapsulation, which involves bundling data (attributes) and methods (behaviors) that operate on the data together within a single unit (class). This promotes modularity, abstraction, and information hiding.
Procedural languages may also support encapsulation to some extent through mechanisms such as modules or namespaces, but they may not enforce it as rigorously as object-oriented languages.

Real-world Modeling:

Object-oriented languages are well-suited for modeling real-world entities and concepts, as they provide natural mechanisms for representing objects, their properties, and their interactions.
Procedural languages may require more effort to model real-world entities in a structured and maintainable way, as they lack built-in support for objects and classes.

3. What are Classes? Where and how is it used?

A: Classes are a fundamental concept in object-oriented programming (OOP). A class is a blueprint for creating objects (instances) that share common attributes and behaviors. It defines the structure and behavior of objects.

Here's a breakdown of key points about classes:

Blueprint for Objects: A class serves as a blueprint or template for creating objects. It defines the attributes (data) and methods (functions) that all objects of that class will have.

Encapsulation: Classes enable encapsulation, which means bundling data (attributes) and methods that operate on the data together within a single unit. This helps in organizing and managing code.
Abstraction: Classes provide abstraction by hiding the internal implementation details of objects and exposing only the necessary features through methods. Users of the class interact with objects using methods without needing to know the internal workings.
Inheritance: Classes support inheritance, allowing new classes (derived or subclass) to inherit attributes and methods from existing classes (base or superclass). This promotes code reusability and facilitates the creation of hierarchical relationships between classes.
Polymorphism: Classes support polymorphism, which allows objects of different classes to be treated uniformly through a common interface. Polymorphism enables code to work with objects of different classes in a consistent manner.
Instantiation: Objects are instances of classes. When a class is instantiated, a new object is created based on the class definition. Each object has its own unique set of attributes, but shares the same methods defined in the class.
Classes are used in various scenarios across software development, including:
Modularization: Classes help break down complex systems into manageable components, improving code organization and maintainability.
Code Reusability: By defining classes with reusable attributes and methods, developers can avoid redundant code and promote code reusability.
Modeling Real-world Entities: Classes are often used to model real-world entities such as people, animals, vehicles, and more. Each class encapsulates the characteristics and behaviors of its corresponding real-world entity.
Graphical User Interface (GUI) Development: In GUI development, classes are used to represent UI elements such as windows, buttons, and text fields, encapsulating their properties and behaviors.
Database Interaction: Classes are used to represent database tables and entities in object-relational mapping (ORM) frameworks, enabling seamless interaction between application code and databases.
Overall, classes provide a powerful mechanism for organizing code, modeling real-world entities, and promoting code reuse and maintainability in object-oriented programming.

4. What are Objects. Summarize its utility in Python.
A: Objects are instances of classes in object-oriented programming (OOP). When a class is instantiated, it creates an object that inherits attributes and methods from the class definition. Objects represent individual entities or instances of a particular class, each with its own unique state and behavior.
Here's a summary of the utility of objects in Python:
Encapsulation: Objects encapsulate data (attributes) and behaviors (methods) within a single unit. This promotes code organization, modularization, and abstraction, allowing developers to manage complexity more effectively.
Abstraction: Objects provide abstraction by hiding the internal

implementation details of data and exposing only the necessary features through methods. Users interact with objects through well-defined interfaces, without needing to know the underlying implementation.

Code Reusability: Objects facilitate code reusability by allowing classes to define reusable attributes and methods. Objects of the same class share the same methods but can have different values for attributes, enabling efficient code reuse across multiple instances.

Inheritance: Objects support inheritance, allowing new classes (subclasses) to inherit attributes and methods from existing classes (superclasses). This promotes code reuse and facilitates the creation of hierarchical relationships between classes, leading to more modular and maintainable code.

Polymorphism: Objects support polymorphism, which allows objects of different classes to be treated uniformly through a common interface. Polymorphism enables code to work with objects of different classes in a consistent manner, enhancing flexibility and extensibility.

Modeling Real-world Entities: Objects are used to model real-world entities such as people, animals, vehicles, and more. Each object represents a specific instance of its corresponding class, encapsulating its characteristics and behaviors.

Dynamic Typing: Python is dynamically typed, meaning the type of an object is determined at runtime. This flexibility allows objects to be created and manipulated dynamically, facilitating rapid development and prototyping.'

Tasks and outputs:

1. class MyClass:
   x = 5
   Create an object named p1, and print the value of x:

```
class MyClass:
    x = 5
P_01 = MyClass()
print(P_01.x)
```

```
5
```

2. Create a class named Person, use the __init__() function to assign values for name and age. Print these values.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return f"Name: {self.name}, Age: {self.age}"
person = Person("DBIT", 100)
print(person)
```

```
Name: DBIT, Age: 100
```

3. Create a Class named Line which calculates the distance and slope between any two points lying on the line. Pass the co-ordinates of the two points as tuples.

```
class Line:
    def __init__(self, point1, point2):
        self.point1 = point1
        self.point2 = point2
    def distance(self):
        x1, y1 = self.point1
        x2, y2 = self.point2
        return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
    def slope(self):
        x1, y1 = self.point1
        x2, y2 = self.point2
        if x2 - x1 == 0:
            return "Undefined"
        else:
            return (y2 - y1) / (x2 - x1)
line = Line((1, 2), (4, 6))
print("Distance between the two points:", line.distance())
print("Slope of the line:", line.slope())
```

```
Distance between the two points: 5.0
Slope of the line: 1.3333333333333333
```

4. Create a Cylinder Class with two methods to calculate the volume and Total surface area of the Cylinder.

```
import math
class Cylinder:
    def __init__(self, radius, height):
        self.radius = radius
        self.height = height
    def volume(self):
        return math.pi * self.radius ** 2 * self.height
    def surface_area(self):
        return 2 * math.pi * self.radius * (self.radius + self.height)
cylinder = Cylinder(radius = 10, height = 5)
print("Volume of the Cylinder:", cylinder.volume())
print("Total Surface Area of the Cylinder:", cylinder.surface_area())
```

```
Volume of the Cylinder: 1570.7963267948967
Total Surface Area of the Cylinder: 942.4777960769379
```

5. **Create a Bank account class**.

   The Class has two attributes:

   1. Owner
   2. Balance:

   And two methods:

1. Deposit: The Deposit method takes a parameter dep_amt . It adds the dep_amt to the balance and displays (prints) the amount deposited (added to balance).

2. Withdraw: The Withdrawal method takes a parameter wd_amt. Withdrawals cannot exceed the available balance. If balance >= wd_amt, wd_amt can be withdrawn from balance and "Withdrawal Accepted." message can be printed. If balance < wd_amt, print "INSUFFICIENT FUNDS"

3. Use the special __str__() which can display Owner and Balance before and after a transaction .

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance
    def deposit(self, dep_amt):
        self.balance += dep_amt
        print(f"Amount deposited: {dep_amt}")
        print(self)
    def withdraw(self, wd_amt):
        if self.balance >= wd_amt:
            self.balance -= wd_amt
            print("Withdrawal Accepted")
        else:
            print("Insufficient Funds!")
        print(self)
    def __str__(self):
        return f"Owner: {self.owner}\nBalance: {self.balance}"
account = BankAccount("DBIT", 1000)
print(account)
account.deposit(500)
account.withdraw(2000)
account.withdraw(1000)
```

```
Owner: DBIT
Balance: 1000
Amount deposited: 500
Owner: DBIT
Balance: 1500
Insufficient Funds!
Owner: DBIT
Balance: 1500
Withdrawal Accepted
Owner: DBIT
Balance: 500
```

Observation and Conclusion:	Were there any other programs not mentioned above that the batch could execute during the practical session. Write about those tasks and your algorithm used.

Program:

```
class Student:
    def __init__(self, name, roll_number, marks):
        self.name = name
        self.roll_number = roll_number
        self.marks = marks
    def calculate_total_marks(self):
        return sum(self.marks)
    def calculate_average_marks(self):
        return sum(self.marks) / len(self.marks)
    def calculate_grade(self):
        average_marks = self.calculate_average_marks()
        if average_marks >= 90:
            return 'A+'
        elif average_marks >= 80:
            return 'A'
        elif average_marks >= 70:
            return 'B'
        elif average_marks >= 60:
            return 'C'
        elif average_marks >= 50:
            return 'D'
        else:
            return 'F'
student = Student("DBIT", "100000", [85, 90, 95, 88, 92])
print("Total Marks:", student.calculate_total_marks())
print("Average Marks:", student.calculate_average_marks())
print("Grade:", student.calculate_grade())
```

```
Total Marks: 450
Average Marks: 90.0
Grade: A+
```

Algorithm:

1. This code defines a Student class with attributes name, roll_number, and marks.
2. The calculate_total_marks() method sums up the marks of the student, calculate_average_marks() calculates the average marks, and calculate_grade() determines the grade based on the average marks.
3. Finally, an instance of the Student class is created, and the total marks, average marks, and grade are printed.