UNIT IV

1. **Describe & differentiate the four primary data structures in Python: tuple, list, set, and dictionary.**

Ans:

1. Tuple

Definition: A tuple is an immutable ordered collection of elements. Once a tuple is created, its elements cannot be modified, added, or removed.

Syntax: Defined by enclosing elements in parentheses ().

Use cases: Suitable for fixed data that shouldn't be modified. Often used for heterogeneous data or when you want to ensure data integrity.

Characteristics:

Ordered: Elements have a defined order, and this order is preserved.

Immutable: Cannot change the tuple's elements after creation.

Can contain duplicates: Tuples can store multiple occurrences of the same element.

Accessed via indexing: You can access individual elements using indices.

Ex: my_tuple = (1, 2, 3, 3)

2. List

Definition: A list is an ordered, mutable collection of items. Lists allow you to add, remove, or change elements.

Syntax: Defined by enclosing elements in square brackets [].

Use cases: Ideal for dynamic collections where the data might need to change over time, such as storing a collection of items in a playlist, shopping cart, etc.

Characteristics:

Ordered: Lists maintain the order of elements.

Mutable: Can add, remove, or modify elements.

Can contain duplicates: Lists can store multiple identical elements.

Accessed via indexing: You can access elements using indices or slice a list.

Ex: my_list = [1, 2, 3, 3]

3. Set

Definition: A set is an unordered, mutable collection of unique elements. Sets automatically eliminate duplicate values.

Syntax: Defined by enclosing elements in curly braces {}.

Use cases: Useful when you need to store a collection of unique items and do not care about order. Great for membership testing or removing duplicates from a sequence.

Characteristics:

Unordered: The elements do not have a defined order, and their order may change.

Mutable: You can add or remove elements.

No duplicates: Sets automatically remove duplicate values.

Accessed via membership testing: You cannot index a set; instead, you check membership (in operator).

Ex: my_set = {1, 2, 3, 3} # {1, 2, 3}

4. Dictionary

Definition: A dictionary is an unordered, mutable collection of key-value pairs. Each element in the dictionary is a pair where a key is mapped to a value.

Syntax: Defined by enclosing key-value pairs in curly braces {}, with keys and values separated by a colon : (e.g., {key1: value1, key2: value2}).

Use cases: Best suited for situations where you need to associate a value with a unique key (e.g., looking up a user by their ID number or counting the frequency of items).

Characteristics:

Unordered: In versions of Python before 3.7, dictionaries are unordered. From Python 3.7 onward, they maintain insertion order.

Mutable: You can change the values associated with keys, add new key-value pairs, or remove them.

No duplicates: Keys must be unique, though values can be duplicated.

Accessed via keys: You retrieve values by accessing them with their corresponding keys.

Ex: my_dict = {'a': 1, 'b': 2, 'c': 3}

2. **Design a Python program that defines a class for a 'Book' with attributes like title and author. Include methods for displaying the book's details. Discuss the importance of classes in programming.**

Ans: Program:

```
class Book:
    # Constructor to initialize the attributes
    def __init__(self, title, author):
        self.title = title
        self.author = author

    # Method to display book details
    def display_details(self):
        print(f"Book Title: {self.title}")
        print(f"Author: {self.author}")

# Create an instance of the Book class
my_book = Book("1984", "George Orwell")

# Display the book's details
my_book.display_details()
```

Output:

```
Book Title: 1984
 Author: George Orwell
```

Importance of Classes in Programming:

Classes are a fundamental concept in Object-Oriented Programming (OOP), and they offer several advantages:

Encapsulation:
Classes allow you to bundle data (attributes) and methods (functions) that operate on the data into a single unit, making the code more organized and modular. This helps in managing and maintaining the codebase more easily.
Reusability:
Once a class is defined, you can create multiple instances (objects) of that class without needing to write the same code again. This promotes code reusability and reduces redundancy.
Abstraction:
Classes provide a way to abstract complex behavior. You can hide implementation details inside methods and expose only the necessary functionality to the outside world, making your code easier to use and understand.
Inheritance:
Classes support inheritance, which allows you to define a new class based on an existing class. This helps you create a hierarchy and reuse code. For example, you could define a Ebook class that inherits from Book and adds additional functionality, such as downloading the book.
Modularity:
By using classes, you can break down complex programs into smaller, manageable components (i.e., objects). Each object can have its own data and functionality, making the program more modular and easier to maintain.
Maintainability and Extensibility:
Classes help in keeping your codebase organized and more easily extendable. You can add new features or modify existing functionality without affecting other parts of the program. For instance, you can modify how books are displayed or add new attributes without rewriting the whole system.

3. **Discuss the concept of object-oriented programming. What are its key features, and how does it differ from procedural programming? Describe the process of creating class and object in python with an example.**
Ans:

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around **objects** rather than functions or logic. Objects are instances of **classes**, which can contain data (called **attributes**) and functions (called **methods**) that operate on that data. OOP helps to model real-world entities and interactions, making code easier to manage, understand, and extend.

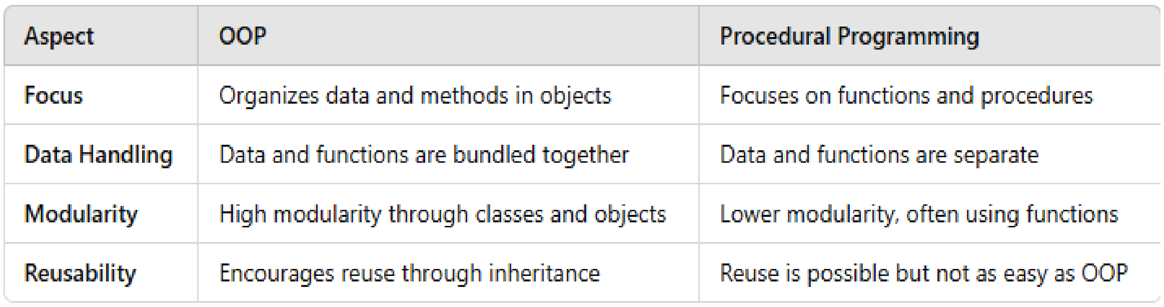
Key Features of OOP:

1. **Encapsulation**:
   - This means bundling data (attributes) and methods (functions) that operate on the data into a single unit, known as a **class**.

- It also helps in hiding the internal state of an object and only exposing a controlled interface (methods) to interact with it. This is known as **data hiding**.

2. **Inheritance**:
   - Allows a class to inherit attributes and methods from another class. This promotes **code reuse** and creates a **hierarchical relationship** between classes.
   - For example, a `Dog` class can inherit from an `Animal` class, meaning the `Dog` will have all the characteristics of `Animal`, plus its own specific features.
3. **Polymorphism**:
   - This means the ability of different objects to respond to the same method in different ways.
   - For example, both a `Dog` and a `Cat` might have a `speak()` method, but they might make different sounds when called (`dog.speak()` could output "Woof", and `cat.speak()` could output "Meow").
4. **Abstraction**:
   - Hiding the complex implementation details and showing only the necessary parts of the object.
   - For example, when you drive a car, you don't need to know how the engine works—only how to steer, brake, and accelerate. Similarly, abstraction hides complex details while exposing essential functionality.

Differences between OOP and Procedural Programming



Aspect	OOP	Procedural Programming
Focus	Organizes data and methods in objects	Focuses on functions and procedures
Data Handling	Data and functions are bundled together	Data and functions are separate
Modularity	High modularity through classes and objects	Lower modularity, often using functions
Reusability	Encourages reuse through inheritance	Reuse is possible but not as easy as OOP

Creating a Class and Object in Python (Example)

In Python, you can define a **class**, which is like a blueprint for creating objects. An **object** is an instance of a class.

Steps:

1. Define a **class** using the `class` keyword.
2. Define the **constructor** method `__init__()` to initialize attributes when an object is created.
3. Create an **object** (an instance of the class) by calling the class.

Example:

```
# Define a class
class Dog:
  # Constructor to initialize the object
  def __init__(self, name, age):
    self.name = name  # Attribute to store dog's name
```

```
        self.age = age    # Attribute to store dog's age

    # Method to make the dog speak
    def speak(self):
        print(f"{self.name} says Woof!")

# Creating an object of the class Dog
my_dog = Dog("Buddy", 5)

# Accessing the object's attributes
print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")

# Calling a method of the object
my_dog.speak()
```

Output:

```
My dog's name is Buddy.
My dog is 5 years old.
Buddy says Woof!
```

4. **Identify the need of list, tuple and dictionary in designing a python program that takes a list of student names and a separate list of their corresponding marks, then creates a dictionary where each student's name is a key, and their mark is the value.**
Ans:

Need for List, Tuple, and Dictionary in the Program

In Python, **lists**, **tuples**, and **dictionaries** each serve specific purposes when designing programs. Let's break down their roles in the context of the program where we want to pair students' names with their marks.

List:

- **Need**: A list is ideal for storing **ordered collections** of items. Since we have a list of student names and a separate list of corresponding marks, lists allow us to store these items in an ordered and mutable way.
- Lists maintain the order of elements and can easily store multiple values. You can modify the lists later (e.g., adding or changing names or marks).

Tuple:

- **Need**: A tuple could be used to store each student's **name and marks as a pair** (if we need them together).
- Tuples are **immutable**, meaning their values can't be changed after they're created. This makes them useful if you want to ensure that each name-mark pair remains unchanged. However, in this specific case, tuples aren't strictly necessary unless you're working with name-mark pairs directly.

Dictionary:

- **Need**: A dictionary is ideal for **key-value pairs**, where each student's name (key) is associated with their marks (value).
- Dictionaries allow fast lookups of values based on a unique key (in this case, the student's name). This makes it easy to retrieve a student's marks based on their name.

Example:

```
# List of student names
students = ["Alice", "Bob", "Charlie", "David"]

# List of corresponding marks
marks = [85, 90, 78, 88]

# Create a dictionary with student names as keys and marks as values
student_marks_dict = dict(zip(students, marks))

# Print the resulting dictionary
print(student_marks_dict)
```

Output:

```
{'Alice': 85, 'Bob': 90, 'Charlie': 78, 'David': 88}
```

5. **Define Function. Explain the advantages of function. Design a python program with functions to display the addition, subtraction, multiplication and division to simulate all kinds of arguments(required, keyword, positional, variable length)**
Ans:

A **function** in programming is a **block of code** that performs a specific task. Functions allow you to group code into reusable pieces, which can be called multiple times within a program. They take **inputs**, known as **arguments** or **parameters**, and may return an **output**.

Advantages of Functions

1. **Reusability**:
   - Once a function is defined, it can be reused multiple times in a program, saving you from rewriting the same code.
2. **Modularity**:
   - Functions break down a program into smaller, manageable pieces. Each function performs a specific task, making your code more organized.
3. **Abstraction**:
   - Functions allow you to hide the implementation details and provide a simple interface. The user only needs to know what the function does, not how it does it.
4. **Maintainability**:
   - Functions make it easier to update or fix bugs. Since the logic is encapsulated in functions, you can change it in one place without affecting the rest of the program.
5. **Testing and Debugging**:
   - Functions are easier to test independently, making debugging simpler. You can test individual functions without running the entire program.

Program:

```
# Function to add numbers (variable-length arguments)
def add(*args):
    return sum(args)

# Function to subtract two numbers (required positional arguments)
def subtract(a, b):
    return a - b

# Function to multiply two numbers (keyword arguments with default values)
def multiply(a=1, b=1):
    return a * b

# Function to divide two numbers (required positional arguments)
def divide(a, b):
    if b == 0:
        return "Error: Division by zero!"
    return a / b

# Function to simulate all operations with variable-length arguments
def operations(*args):
    print("Addition:", add(*args))
    print("Subtraction:", subtract(*args[:2]))  # Using first two numbers for subtraction
    print("Multiplication:", multiply(*args[:2])) # Using first two numbers for multiplication
    print("Division:", divide(*args[:2]))  # Using first two numbers for division

# Call the operations function with multiple arguments
operations(10, 20, 30)
```

Output:

```
Addition: 60
Subtraction: -10
Multiplication: 200
Division: 0.5
```

6. **Explain the advantage of lambda functions. Design a lambda function that squares a given number.**

   Ans:

   Lambda functions in Python are small, anonymous functions. That means this is a function without name. In normal functions **def** keyword is used to define a function, but lambda functions defined using the **lambda** keyword. Lambda functions can take any number of arguments but can only have one expression. The result of that expression is returned automatically.

Syntax:

```
lambda arguments: expression
```

Program to print squares:

```
# Lambda function to square a given number
square = lambda x: x ** 2

result = square(5)
print(result)
# Output: 25
```

7. **Define constructor. Explain different types of constructors in python with an example.**
   Ans:

A **constructor** in object-oriented programming (OOP) is a special method that is automatically called when a new object of a class is created. It is typically used to initialize the object's state (i.e., assign values to the object's attributes). In Python, constructors are defined using the __init__() method.

Different Types of Constructors in Python

In Python, there are generally **two types of constructors**:

1. **Default Constructor**
2. **Parameterized Constructor**

1. Default Constructor

A **default constructor** is a constructor that does not accept any arguments other than self. It is automatically called when an object is created without any arguments.

- **Definition**: The __init__() method has no parameters except self.
- **Usage**: It is typically used to set default values for the attributes of the object.

Example of a Default Constructor:

```
class Person:
   def __init__(self):
      self.name = "Unknown"
      self.age = 0

   def display(self):
      print(f"Name: {self.name}, Age: {self.age}")

# Creating an object of Person class without arguments
person1 = Person()
person1.display()  # Output: Name: Unknown, Age: 0
```

2. Parameterized Constructor

A **parameterized constructor** is a constructor that accepts arguments. These arguments allow the object to be initialized with custom values when it is created. This gives more flexibility compared to the default constructor.

- **Definition**: The __init__() method takes additional parameters apart from self.
- **Usage**: It is used when you want to initialize the object with specific values provided at the time of creation.

Example of a Parameterized Constructor:

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

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Creating an object of Person class with arguments
person2 = Person("Ram", 30)
person2.display()  # Output: Name: Ram, Age: 30
```

8. **Define self. Explain the need of self in oop with an example program.**
Ans:

In Python, `self` is a **reference variable** that refers to the **current instance** of the class. It is used within an instance method to refer to the attributes and methods of the current object. The `self` parameter is the first argument passed to any instance method in a class. This allows you to access the object's attributes and call other methods within the same class.

Needed in Object-Oriented Programming (OOP)

The use of `self` is crucial in object-oriented programming for the following reasons:

1. **Accessing Instance Variables**: In a class, each object has its own set of instance variables. The `self` keyword is used to refer to those instance variables, allowing you to store and retrieve data specific to each object.
2. **Calling Instance Methods**: `self` is used to call other methods of the same object. Without it, the method would not know which object (instance) to refer to.
3. **Distinguishing between Instance and Local Variables**: By using `self`, Python distinguishes between instance variables (which belong to the object) and local variables (which belong to the method). This is necessary because methods can have their own local variables, and `self` helps avoid confusion between them.
4. **Enabling Object-Oriented Behavior**: `self` makes it possible to create and work with multiple instances of the same class, each having its own state and behavior.

Example:

```
class Car:
   def __init__(self, brand, model):
      self.brand = brand  # Assign brand to instance
      self.model = model  # Assign model to instance

   def display(self):
      print(f"{self.brand} {self.model}")

# Create an object
car1 = Car("Tesla", "Model S")
car1.display()  # Output: Tesla Model S
```

9. **What is access specifier? How many types of access specifier'. Explain each with an example, how to access each type (public, private and protected)**
   Ans:

   An **access specifier** (also known as an access modifier) is a keyword used in object-oriented programming to define the visibility or accessibility of a class's attributes and methods. It controls how the data members (attributes) and methods of a class can be accessed from outside the class. In Python, there are three main types of access specifiers:

   1. **Public**
   2. **Private**
   3. **Protected**

   Public Access Specifier:
   Definition: Public members (attributes and methods) can be accessed from anywhere in the program, both inside and outside the class.
   Usage: The default access level in Python is public unless explicitly specified otherwise.

   Ex:

```
class Car:
   def __init__(self, brand):
      self.brand = brand  # Public attribute

   def display(self):  # Public method
      print(f"Brand: {self.brand}")

# Creating an object of the Car class
car1 = Car("Tesla")
car1.display()  # Accessing public method
print(car1.brand)  # Accessing public attribute
```

   Output:

```
Brand: Tesla
Tesla
```

Private Access Specifier:

Definition: Private members are only accessible within the class. They cannot be accessed or modified directly from outside the class.

Usage: Private members are defined by prefixing the attribute or method name with two underscores (__).

Ex:

```
class Car:
    def __init__(self, brand):
        self.__brand = brand  # Private attribute

    def __display(self):  # Private method
        print(f"Brand: {self.__brand}")

    def show_details(self):  # Public method to access private members
        self.__display()

# Creating an object of the Car class
car1 = Car("Tesla")
car1.show_details()   # Accessing private method indirectly

# Accessing private attribute directly will cause an error
# print(car1.__brand)   # Uncommenting this will raise an AttributeError
```

Output:

```
Brand: Tesla
```

Protected Access Specifier:

Definition: Protected members are intended to be accessible within the class and its subclasses. They are not meant to be accessed directly from outside the class, but they can be inherited by child classes.

Usage: Protected members are defined by prefixing the attribute or method name with one underscore (_).

Ex:

```
class Car:
    def __init__(self, brand):
        self._brand = brand  # Protected attribute

    def _display(self):  # Protected method
        print(f"Brand: {self._brand}")

# Creating an object of the Car class
car1 = Car("Tesla")
car1._display()        # Accessing protected method (not recommended)

print(car1._brand)   # Accessing protected attribute (possible but discouraged)
```

Output:

```
Brand: Tesla
Tesla
```

10. **Explain the difference between class method and instance methods with an example.**
Ans:
Instance Method:
An instance method is a method that operates on an instance of the class (an object). It takes self as its first argument, which is a reference to the instance of the class. This method can access and modify the instance's attributes.

Example of Instance Method:
```
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Instance attribute
        self.model = model  # Instance attribute

    # Instance method
    def display(self):
        print(f"Car Brand: {self.brand}, Model: {self.model}")

# Creating an instance of Car class
car1 = Car("Tesla", "Model S")
car1.display()  # Calling instance method
```
Output:
```
Car Brand: Tesla, Model: Model S
```
Class Method:
A class method is a method that operates on the class itself, rather than on an instance. It takes cls as the first argument, which is a reference to the class. Class methods are defined using the @classmethod decorator. They are often used for factory methods or class-level operations.

Example of Class Method:
```
class Car:
    # Class variable
    wheels = 4

    def __init__(self, brand, model):
        self.brand = brand  # Instance attribute
        self.model = model  # Instance attribute

    # Class method
    @classmethod
    def get_wheels(cls):
        return cls.wheels  # Accesses class variable

# Creating an instance of Car class
car1 = Car("Tesla", "Model S")
```

```
# Calling class method using the class name and instance
print(Car.get_wheels())  # Output: 4
print(car1.get_wheels())  # Output: 4
```

Output:

```
4
4
```

11. Explain module with an example.

Ans:

A module is simply a file that contains Python standard functionalities and statements. For using those functionalities, it is essential to import the corresponding module first.

- Basically modules in python are .py files in which set of functions are written.
- Modules are imported using import function.
- When python file is executed directly it is considered as main module.
- The main module is recognized as __main__ and provide the basis for a complete python program.
- The Main module can import any number of other modules. But main module cannot be imported into some other module.

Creating a Module:

A module is created by saving a Python file with a .py extension. For example, if a file named mymodule.py, that file becomes a module named mymodule.

Example: Creating a Module

```
# mymodule.py
def greet(name):
   return f"Hello, {name}!"
def add(a, b):
   return a + b
```

Importing Modules:

We can import a module into another Python script using the import statement.

Example: Importing a Module

```
# main.py
import mymodule
print(mymodule.greet("Ram"))  # Output: Hello, Ram!
print(mymodule.add(5, 3))        # Output: 8
```

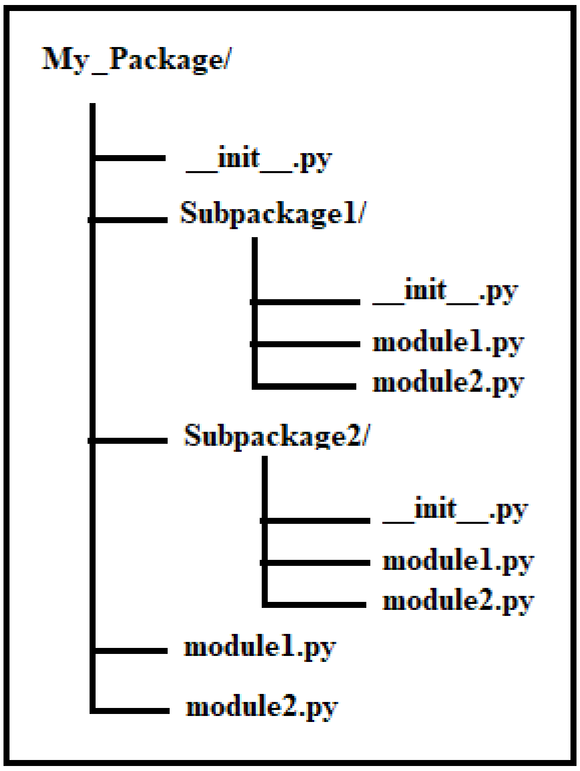
12. How to create a package in python. Explain clearly with an example.

Ans:

A package is essentially a directory that contains a special file called __init__.py and one or more module files (Python scripts). The presence of __init__.py indicates to Python that this directory should be treated as a package.

- Along with packages and modules the package contains a file named __init__.py. In fact to be a package, there must be a file called __init__.py in the folder.

• Packages can be nested to any depth, provided that the corresponding directories contain their own __init__.py file.



For Example:

Let us consider an example: Create a package to perform arithmetic operations addition, subtraction, multiplication and division operations

Step1: Create a folder named MyMaths in your working drive. (Let D Drive or any)

Step2: Inside MyMaths directory create __init__.py file. Just create an empty file

Step3: Create a folder named Add inside MyMaths. Inside Add folder create file named addition.py as follows.

```
addition.py
def add_num(a,b):
        return a+b
```

Step4: Similarly create a folder named Sub inside MyMaths. Inside Sub folder create file named subtraction.py as follows.

```
subtraction.py
def sub_num(a,b):
        return a-b
```

Step5: Similarly create a folder named Mul inside MyMaths. Inside Mul folder create file named multiplication.py as follows.

```
multiplication.py
def mul_num(a,b):
        return a*b
```

Step6: Similarly create a folder named Div inside MyMaths. Inside Div folder create file named division.py as follows.

```
division.py
```

```
                def div_num(a,b):
                        return a/b
```
Usage of package:

```
import MyMaths.Add.addition
import MyMaths.Sub.subtraction
import MyMaths.Mul.multiplication
import MyMaths.Div.division
print(“The addition of 20 and 10 is: ”, MyMaths.Add.addition.add_num(20,10)
print(“The subtraction of 20 and 10 is: ”, MyMaths.Sub.subtraction.sub_num(20,10)
print(“The multiplication of 20 and 10 is: ”, MyMaths.Mul.multiplication.mul_num(20,10)
print(“The division of 20 by 10 is: ”, MyMaths.Div.division.div_num(20,10)
```
Output:

```
          The addition of 20 and 10 is: 30
          The subtraction of 20 and 10 is: 10
          The multiplication of 20 and 10 is: 200
          The division of 20 by 10 is: 2.0
```

Unit V

Q1. Develop a recursive function to compute the Fibonacci series. Discuss the advantages and disadvantages of using recursion for this problem

Ans:

Recursion: Recursion is a programming technique where a function calls itself in order to solve a problem.

Fibonacci Series

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1.

Program:

```
 def fibonacci_recursive(n):
     if (n<=1):
             return n
     else:
             return(fibonacci_recursive(n-1)+ fibonacci_recursive(n-2))
n=int(input(“ Enter the number of terms))
for i in range(n):
         print(fibonacci_recursive(i),end= ” “)
```

Output:

```
   Enter the number of terms:   10
    0 1 1 2 3 5 8 13 21 34
```

Advantages and Disadvantages of Recursion:

Advantages:

Simplicity: Recursive solutions can be more straightforward and easier to understand than their iterative counterparts.
Elegance: Recursive solutions can be more elegant and concise for problems like permutations, combinations, and factorials.
Disadvantages:
Performance: Recursive solutions can be less efficient due to overhead from function calls and potential for stack overflow with deep recursion.
Memory Usage: Each recursive call adds a new layer to the call stack, which can lead to increased memory usage.

Q2. Illustrate the concept of inheritance in object-oriented programming. Provide an example, how inheritance can be implemented in Python.

Ans:

Inheritance is one of the core features of Object-Oriented Programming (OOP) that allows a class (called the **subclass** or **child class**) to inherit properties and behaviors (attributes and methods) from another class (called the **superclass** or **parent class**). This helps in code reuse, as the subclass can use the functionality of the parent class without needing to rewrite it.

In inheritance, a subclass can:

- **Inherit** attributes and methods from the superclass.
- **Override** methods to provide specific behavior for the subclass.
- **Add new attributes and methods** specific to the subclass.

Advantages of Inheritance:

1. **Code Reusability**: You can reuse the code of the parent class without modifying it.
2. **Extensibility**: New functionality can be added by creating new subclasses based on existing classes.
3. **Maintainability**: Changes in the parent class are automatically reflected in the subclasses.

Example:

```
class Animal:
    def speak(self):
        return "Animal makes a sound"

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        return "Woof!"  # Override speak method for Dog

class Cat(Animal):  # Cat inherits from Animal
    def speak(self):
        return "Meow!"  # Override speak method for Cat

# Create instances of Dog and Cat
```

```
dog = Dog()
cat = Cat()

# Call the speak method on both
print(dog.speak())        # Output: Woof!
print(cat.speak())        # Output: Meow!
```

Inheritance: Both Dog and Cat classes inherit from the Animal class, meaning they can access its methods and attributes

Q3. Design an inheritance hierarchy for a 'Vehicle' class that includes subclasses for 'Car' and 'Bike'. Discuss how inheritance promotes code reusability.

Ans:

```
class Vehicle:
    def __init__(self, brand, model, color):
        self.brand = brand
        self.model = model
        self.color = color

    def start(self):
        print(f"{self.brand} {self.model} is starting.")

class Car(Vehicle):
    def __init__(self, brand, model, color, fuel_type):
        super().__init__(brand, model, color)
        self.fuel_type = fuel_type

    def honk(self):
        print("Car honking!")

class Bike(Vehicle):
    def __init__(self, brand, model, color, has_gear):
        super().__init__(brand, model, color)
        self.has_gear = has_gear

    def ring_bell(self):
        print("Bike bell ringing!")

# Usage
car = Car("Toyota", "Corolla", "Red", "Petrol")
car.start()
car.honk()

bike = Bike("Giant", "Escape 3", "Blue", True)
bike.start()
bike.ring_bell()
```

Inheritance allows a subclass to inherit attributes and methods from a parent class, promoting code reuse.

Q 4. Design an algorithm and explain the importance of Divide and Conquer and Illustrate the process of Binary search with the following example.

Elements: 22,32,11,34,65,3,5,63 | element to be found: 12

Ans:

Binary Search is a highly efficient algorithm for finding the position of a target value within a **sorted array**. It works by repeatedly dividing the search interval in half. If the value of the target is less than the value at the midpoint of the interval, it narrows the search to the lower half; otherwise, it narrows the search to the upper half. This process is repeated until the target value is found or the search interval is empty.

Binary Search Algorithm (Step-by-Step):

1. **Input**: A sorted list and an element to search for.
2. **Output**: The index of the element if found, otherwise return `-1`.

Steps:

1. Sort the list (if not already sorted).
2. Set the **left** pointer to the start and the **right** pointer to the end of the list.
3. Find the middle element using the formula:
   `mid = (left + right) // 2`
4. Compare the middle element with the target:
   - If equal, return the index of the middle element.
   - If the target is smaller, search the left half (`right = mid - 1`).
   - If the target is larger, search the right half (`left = mid + 1`).
5. Repeat until the target is found or the left pointer exceeds the right pointer.

Binary Search Example

Given:

- **List**: `[22, 32, 11, 34, 65, 3, 5, 63]`
- **Element to Find**: `12`

Step-by-Step Process:

1. **Sort the List**: First, sort the list in ascending order.
   - Sorted list: `[3, 5, 11, 22, 32, 34, 63, 65]`
2. **Set Initial Pointers**:
   - `left = 0`
   - `right = 7` (last index)
3. **First Iteration**:
   - Find middle index: `mid = (0 + 7) // 2 = 3`

- The middle element is `22`.
- Compare `22` with the target `12`.
- Since `12` is smaller than `22`, we search the left half (from index 0 to 2).

4. **Second Iteration**:
   - Set new `right = 2` and `left = 0`.
   - Find middle index: `mid = (0 + 2) // 2 = 1`.
   - The middle element is `5`.
   - Compare `5` with `12`.
   - Since `12` is larger than `5`, we search the right half (from index 2 to 2).
5. **Third Iteration**:
   - Set new `left = 2` and `right = 2`.
   - Find middle index: `mid = (2 + 2) // 2 = 2`.
   - The middle element is `11`.
   - Compare `11` with `12`.
   - Since `12` is larger than `11`, we search the right half (from index 3 to 2).
6. **Conclusion**:
   - At this point, `left = 3` and `right = 2`, meaning the search range is invalid. The element `12` is **not found**.

Q5. Explain various types of inheritance using examples for each.

Ans: Inheritance allows a class to inherit attributes and methods from another class. Here are the main types of inheritance:

1. Single Inheritance

A class inherits from only one parent class.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

dog = Dog()
dog.speak()  # Inherited from Animal
dog.bark()   # Defined in Dog
```

2. Multilevel Inheritance

A class inherits from a class that is already a subclass of another class.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

class Puppy(Dog):
    def whine(self):
        print("Puppy whines")
```

```
puppy = Puppy()
puppy.speak()  # Inherited from Animal
puppy.bark()   # Inherited from Dog
puppy.whine()  # Defined in Puppy
```

3. Multiple Inheritance

A class inherits from two or more parent classes.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Bird:
    def fly(self):
        print("Bird flies")

class Bat(Animal, Bird):
    def hang(self):
        print("Bat hangs upside down")

bat = Bat()
bat.speak()   # Inherited from Animal
bat.fly()     # Inherited from Bird
bat.hang()    # Defined in Bat
```

4. Hierarchical Inheritance

Multiple classes inherit from a single parent class.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

class Cat(Animal):
    def meow(self):
        print("Cat meows")

dog = Dog()
cat = Cat()

dog.speak()  # Inherited from Animal
cat.speak()  # Inherited from Animal
```

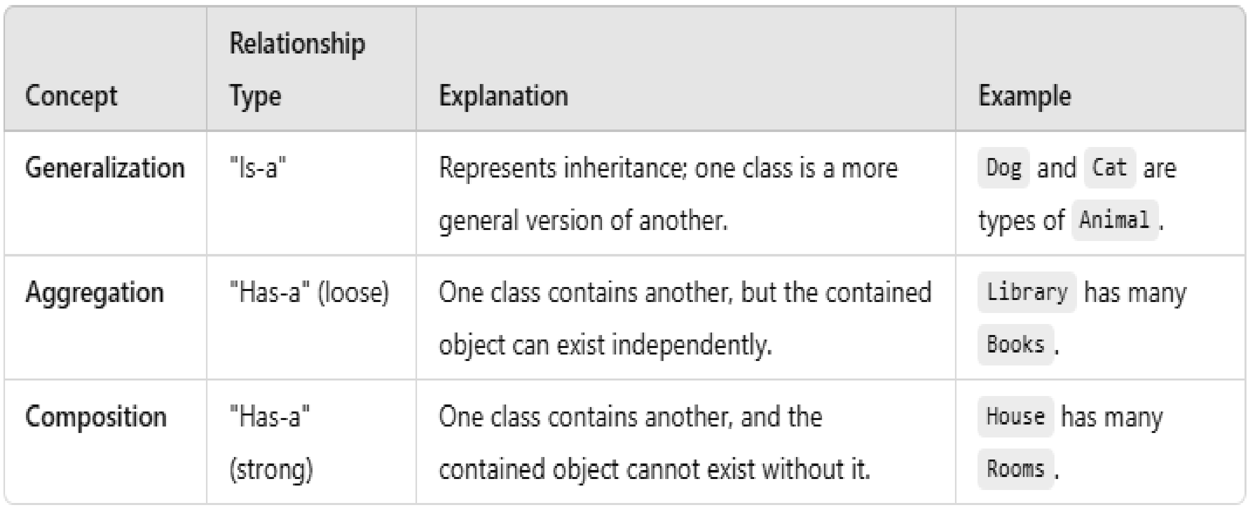
5. Hybrid Inheritance

A combination of multiple types of inheritance (e.g., multiple + multilevel).

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Mammal(Animal):
    def nurse(self):
        print("Mammal nurses its young")

class Bird(Animal):
    def lay_eggs(self):
        print("Bird lays eggs")

class Bat(Mammal, Bird):
    def fly(self):
        print("Bat flies")

bat = Bat()
bat.speak()    # Inherited from Animal
bat.nurse()    # Inherited from Mammal
bat.lay_eggs() # Inherited from Bird
bat.fly()      # Defined in Bat
```

Q6. What is the difference between Generalization, Aggregation and Composition

Ans:



Concept	Relationship Type	Explanation	Example
Generalization	"Is-a"	Represents inheritance; one class is a more general version of another.	`Dog` and `Cat` are types of `Animal`.
Aggregation	"Has-a" (loose)	One class contains another, but the contained object can exist independently.	`Library` has many `Books`.
Composition	"Has-a" (strong)	One class contains another, and the contained object cannot exist without it.	`House` has many `Rooms`.

Q7.Design a program to find the factorial of a given number using recursion.

Ans: Recursion: Recursion is a programming technique where a function calls itself in order to solve a problem. It's particularly useful for problems that can be broken down into smaller, similar sub problems.

The factorial function is an example of recursion. The recursive formula is:

- Base Case: $n!=1$ if n=0
- Recursive Case: $n!=n\times(n-1)!$

Program:

```
def factorial(n):
  if n == 0:        # Base case
      return 1
  else:              # Recursive case
        return n * factorial(n - 1)
n=int(input("Enter a number: "))
print(f'Factorial of {n} is: ",factorial(n))
```

# Output:

```
 Enter a number:  5
 Factorial of 5 is:  120
```

Q8. Explain brute force in problem solving. Explain it with an example.

Ans: In computer science, **Brute Force** is a straightforward approach for solving problems by trying all possible solutions. It involves systematically checking every potential solution until the correct one is found.
One common application of the brute force method is Naive String Matching, which is used to find a pattern (or substring) in a larger text string. This method compares the pattern to the text at every possible position, checking character by character to see if there is a match.

Naive String Matching Algorithm:

The Naive String Matching algorithm is one of the simplest algorithms for string matching. The basic idea is to check whether the pattern occurs at every possible position in the text. For each position, we compare the characters of the pattern with the corresponding characters of the text. If all characters match, the pattern is found at that position.

Algorithm Steps:

1. Let the pattern be P of length m and the text be T of length n.
2. Slide the pattern P over the text T from position i = 0 to i = n - m (i.e., all possible starting positions in the text).
3. At each position i, compare the substring of T starting at i with the entire pattern P. This means comparing T[i] with P[0], T[i+1] with P[1], and so on.
4. If a mismatch occurs at any position, move the pattern one step forward and repeat the comparison at the next position.
5. If all characters match at a particular position, the pattern is found at that position.
   Example:
   Let the text T = "ABABDABACDABABCABAB" and the pattern P = "ABAB". The algorithm will check every position in the text, starting at i = 0, and compare the substring of T with P. If it finds a match, it prints the starting index of the match.

1. At index i = 0, compare "ABAB" (substring of T) with "ABAB" (pattern). match.
2. At index i = 1, compare "BABD" (substring of T) with "ABAB" (pattern). No match.
3. Continue sliding the pattern until a match is found, say at index i = 10,15.

Program:

```
def naive_string_matching(text, pattern):
     n = len(text)   # Length of the text
    m = len(pattern)  # Length of the pattern
    # Loop through each position in the text where the pattern can start
     for i in range(n - m + 1):
          match = True                    # Flag to check if all characters match

      # Check each character of the pattern with the corresponding character in the text
          for j in range(m):
               if text[i + j] != pattern[j]:
                    match = False         # If characters do not match, set flag to False
                    break

      # If all characters match, print the index where the pattern is found
         if match:
               print(f"Pattern found at index {i}")

# Test the function
text = "ABABDABACDABABCABAB"
pattern = "ABAB"
naive_string_matching(text, pattern)
```

Output:

```
          Pattern found at index 0
          Pattern found at index 10
          Pattern found at index 15
```