a) What are the advantages of Pandas?

Pandas is a powerful Python library used for data manipulation and analysis. The main advantages of Pandas are:

1. **Easy Data Handling**: Pandas provides easy-to-use data structures like Series and DataFrame, which make data manipulation simple.
2. **Handling Missing Data**: Pandas can easily handle missing data using functions like fillna() or dropna(), making it useful for cleaning datasets.
3. **Efficient Data Manipulation**: Pandas is optimized for performance, allowing you to work efficiently with large datasets.
4. **Data Alignment**: It can automatically align data from different sources based on the index, which is especially helpful when merging datasets.
5. **Rich I/O Operations**: It supports reading and writing data from various file formats like CSV, Excel, JSON, SQL, etc.
6. **Data Filtering and Aggregation**: Pandas makes it easy to filter, group, and aggregate data for summary statistics and analytics.
7. **Visualization Support**: It integrates well with libraries like Matplotlib and Seaborn to create visual representations of data.
8. **Flexible Data Transformation**: With functions like apply(), map(), replace(), etc., data transformation becomes very flexible.

---

b) State the uses of TensorFlow

TensorFlow is an open-source machine learning framework developed by Google. Its main uses include:

1. **Deep Learning**: TensorFlow is widely used for building and training deep learning models, including neural networks, CNNs, RNNs, etc.
2. **Machine Learning**: It supports machine learning algorithms like linear regression, classification, and clustering, making it suitable for a wide range of machine learning tasks.
3. **Production-Ready Deployment**: TensorFlow provides a stable environment for deploying machine learning models in production across various platforms, including mobile, web, and cloud.
4. **Natural Language Processing (NLP)**: TensorFlow is used to develop NLP models for tasks like text classification, language translation, and sentiment analysis.
5. **Computer Vision**: TensorFlow is often used in building models for image recognition, object detection, and video analysis.
6. **Time Series Analysis**: TensorFlow can be used for forecasting, time series prediction, and anomaly detection in sequential data.
7. **Reinforcement Learning**: TensorFlow supports reinforcement learning algorithms, helping in applications like robotics, gaming, and AI-based decision-making systems.
8. **Scalability**: It allows developers to distribute the computation across multiple CPUs or GPUs, enhancing performance for large datasets and complex models.

---

c) Write Syntax of Raise Statement

In Python, the raise statement is used to manually raise an exception. The syntax is as follows:

```
raise ExceptionType("Error message")
```

- **ExceptionType**: This refers to the type of exception you want to raise (e.g., ValueError, TypeError, etc.).
- **Error message**: This is an optional custom message that provides more details about the exception.

Example:

```
raise ValueError("This is a custom error message")
```

This raises a ValueError with the message "This is a custom error message."

---

d) List out any 4 Label Options

If you're referring to the Label widget in **Tkinter** (a Python library for building graphical user interfaces), here are four common options (properties) that can be set for the Label widget:

1. **text**: Specifies the text to be displayed on the label.
   - Example: `Label(root, text="Hello World")`
2. **bg (background)**: Sets the background color of the label.
   - Example: `Label(root, bg="yellow")`
3. **fg (foreground)**: Sets the text (foreground) color of the label.
   - Example: `Label(root, fg="blue")`
4. **font**: Defines the font type, size, and style for the label text.
   - Example: `Label(root, font=("Arial", 16, "bold"))`

These are just a few options you can configure for a label in Tkinter. There are many more options available to customize the appearance and behavior of the label widget.

---

e) What are the properties of a Dictionary in Python?

A dictionary in Python is a collection of key-value pairs. Here are the key properties of a dictionary:

1. **Unordered**: Dictionaries are unordered collections, meaning the items (key-value pairs) are not stored in a specific sequence, and their order can change.
2. **Mutable**: Dictionaries are mutable, meaning you can change, add, or remove key-value pairs after the dictionary is created.
3. **Key Uniqueness**: Keys in a dictionary must be unique. If a key is repeated, the last value associated with that key will overwrite the previous value.

4. **Hashable Keys**: Keys in a dictionary must be hashable, meaning they must be of immutable types like strings, numbers, or tuples (with immutable elements).
5. **Efficient Lookup**: Dictionary operations like insertion, deletion, and lookup have an average time complexity of O(1), making them very efficient.
6. **Dynamic Size**: The size of a dictionary can grow and shrink as needed. There is no fixed limit on the number of items that can be stored.
7. **Supports Nesting**: You can nest dictionaries within other dictionaries, allowing complex data structures.

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
```

---

f) List out geometry management methods in Tkinter

In Tkinter (the Python library for creating GUIs), geometry management methods are used to arrange widgets in a window. The main methods are:

1. **pack()**: Arranges widgets by packing them in blocks, which can be placed on the top, bottom, left, or right of the parent widget.
   - Example: `widget.pack(side="left")`
2. **grid()**: Organizes widgets in a table-like structure using rows and columns.
   - Example: `widget.grid(row=0, column=1)`
3. **place()**: Allows you to place widgets at an absolute position using x and y coordinates.
   - Example: `widget.place(x=50, y=100)`
4. **pack_forget(), grid_forget(), and place_forget()**: These methods hide widgets that have been packed, gridded, or placed.

These are the primary geometry management methods used in Tkinter to control widget layout.

---

g) Difference between Python List and NumPy Array

Feature	Python List	NumPy Array
Data Type	Can hold elements of different data types.	Elements must be of the same data type.
Performance	Slower, especially for large datasets.	Faster for numerical computations, as it is optimized for performance.
Memory Efficiency	Consumes more memory as each element is an independent Python object.	Consumes less memory, as elements are stored in contiguous blocks.
Operations	Lacks support for vectorized operations (i.e., element-wise operations need to be done using loops).	Supports vectorized operations, making mathematical operations faster and more concise.
Functionality	General-purpose, not optimized for numerical computing.	Specifically designed for numerical computation with a wide range of mathematical functions available.
Multidimensional Support	Lists are typically one-dimensional but can contain other lists (nested lists) for multidimensional use.	Natively supports multidimensional arrays (e.g., 2D, 3D arrays) without nesting.

Example:

- **Python List**:

```
my_list = [1, 2, 3, 4]
```

- **NumPy Array**:

```
import numpy as np

my_array = np.array([1, 2, 3, 4])
```

---

h) What is the use of random() in the random module?

The random() function in Python's random module is used to generate a random floating-point number between 0.0 and 1.0, including 0.0 but excluding 1.0. It is commonly used for generating random values in various probability-related applications.

Key Features of random():

- Returns a floating-point number between **0.0** (inclusive) and **1.0** (exclusive).
- It can be useful when you need a random decimal value, such as in simulations, testing, or probability models.

Example:

```
import random

print(random.random())  # Output: A random float between 0.0 and 1.0
```

If you need random integers or numbers within a specific range, there are other functions in the random module like randint(), uniform(), etc.

---

i) Explain any two tuple operations with an example

Tuples are immutable sequences in Python. Here are two common tuple operations:

1. **Indexing**: You can access elements in a tuple by using indexing. The index starts from 0 for the first element.
   Example:
   ```
   my_tuple = (10, 20, 30, 40)

   print(my_tuple[1])  # Output: 20
   ```

2. **Slicing**: You can extract a portion of the tuple using slicing. It allows you to specify a range of indices to get a new tuple.
   Example:
   ```
   my_tuple = (1, 2, 3, 4, 5, 6)

   print(my_tuple[1:4])  # Output: (2, 3, 4)
   ```

---

j) List out any 5 button options in Python

In **Tkinter**, a popular GUI library in Python, buttons have several options to customize their appearance and behavior. Here are five common options:

1. **text**: Specifies the label or text displayed on the button.
   - Example: Button(root, text="Click Me")
2. **command**: Specifies the function to be called when the button is clicked.
   - Example: Button(root, command=my_function)
3. **bg** **(background)**: Sets the background color of the button.

    - Example: Button(root, bg="lightblue")
4. **fg (foreground)**: Sets the color of the text displayed on the button.
    - Example: Button(root, fg="white")
5. **font**: Specifies the font type, size, and style of the button text.
    - Example: Button(root, font=("Helvetica", 14, "bold"))

Q2 a) Explain Frame Widget in Tkinter with Example

A **Frame** in Tkinter is a container widget used to group and organize other widgets, such as buttons, labels, and other frames. It acts like a box or section that can contain and organize multiple child widgets, which can be laid out using geometry management methods like pack(), grid(), or place().

The Frame is commonly used to break down a complex layout into smaller, manageable sections and is especially useful for creating more organized and readable GUI designs.

Key Features:

- It's used as a **container** for other widgets.
- It allows for organizing the layout by grouping widgets together.
- It supports various options like background color, borders, and padding.

Example:

```
import tkinter as tk

root = tk.Tk()
root.title("Frame Example")

# Creating a Frame
frame = tk.Frame(root, bg="lightgray", bd=5)
frame.pack(pady=10)

# Adding Widgets to the Frame
label = tk.Label(frame, text="This is a label inside the frame")
label.pack()

button = tk.Button(frame, text="Click Me")
button.pack()

root.mainloop()
```

In this example:

- We created a **Frame** and added a label and a button inside it.
- The frame.pack() method is used to display the frame in the window.
- The widgets inside the frame are packed as well.

b) Explain Function Arguments in Detail

In Python, function arguments are the values passed to a function when it is called. These arguments can be of various types, and Python provides flexibility in how you define and pass them. There are several types of function arguments:

1. **Positional Arguments**: These are the most common way of passing arguments to a function. The arguments are assigned to the parameters in the order in which they are passed.
   Example:
   ```
   def greet(name, age):

    print(f"Hello {name}, you are {age} years old.")

   greet("John", 25)  # Output: Hello John, you are 25 years old.
   ```

2. **Keyword Arguments**: You can specify arguments by the parameter name, allowing you to pass arguments out of order.
   Example:
   ```
   greet(age=30, name="Alice")  # Output: Hello Alice, you are 30 years old.
   ```

3. **Default Arguments**: A function can have default parameter values. If no argument is passed, the default value is used.
   Example:
   ```
   def greet(name, age=18):

    print(f"Hello {name}, you are {age} years old.")


   greet("Tom")  # Output: Hello Tom, you are 18 years old.
   ```

4. **Variable-Length Arguments (Using *args)**: If you don't know in advance how many arguments will be passed to a function, you can use *args. This allows a function to accept any number of positional arguments.
   Example:
   ```
   def sum_all(*args):

       total = 0

       for num in args:

           total += num
   ```

```
    print(f"Sum: {total}")


sum_all(1, 2, 3, 4)  # Output: Sum: 10
```

Summary of Function Argument Types:

- **Positional Arguments**: Passed in order.
- **Keyword Arguments**: Passed with the parameter name.
- **Default Arguments**: Has a predefined value if no argument is provided.
- **Variable-Length Arguments (*args)**: Allows multiple positional arguments.
- **Keyword Variable-Length Arguments (**kwargs)**: Allows multiple keyword arguments.

---

c) Explain any three built-in dictionary functions with an example

1. **keys()**: This method returns a view object that displays a list of all the keys in the dictionary.
   Example:
   ```
   my_dict = {"name": "Alice", "age": 25, "city": "New York"}

   print(my_dict.keys())  # Output: dict_keys(['name', 'age', 'city'])
   ```

2. **values()**: This method returns a view object that displays a list of all the values in the dictionary.
   Example:
   ```
   my_dict = {"name": "Alice", "age": 25, "city": "New York"}

   print(my_dict.values())  # Output: dict_values(['Alice', 25, 'New York'])
   ```

3. **items()**: This method returns a view object containing the dictionary's key-value pairs as tuples.
   Example:
   ```
   my_dict = {"name": "Alice", "age": 25, "city": "New York"}

   print(my_dict.items())  # Output: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])
   ```

---

d) Write a Python program to display the current date and time

You can use the datetime module in Python to get and display the current date and time.

Example:

```
from datetime import datetime

# Get the current date and time

current_time = datetime.now()


# Display the current date and time

print("Current Date and Time:", current_time)
```

Output:

```
Current Date and Time: 2024-10-01 10:35:27.123456
```

In this example:

- The datetime.now() function gets the current date and time.
- The program prints the result in a readable format.

---

e. Write an anonymous function to find area of rectangle. Please write in English language.

An **anonymous function** in Python is created using the lambda keyword. To find the area of a rectangle using an anonymous function, you can define the lambda function with two parameters, length and width, and calculate the area by multiplying them.

Anonymous Function to Find the Area of a Rectangle:

```
# Lambda function to calculate the area of a rectangle

area_of_rectangle = lambda length, width: length * width

# Example usage
length = 5
width = 3
print("Area of the rectangle:", area_of_rectangle(length, width))
```

Explanation:

- lambda length, width: length * width is the anonymous function.
- It takes two arguments (length and width) and returns their product, which is the area of the rectangle.
- In the example, length = 5 and width = 3, so the output will be 15.

---

a) What are lists and tuples? What is the key difference between these two?

- **Lists**:
  - A **list** is a mutable (changeable) collection of items in Python.
  - Items in a list can be of different data types (e.g., integers, strings, etc.).
  - Lists are defined using square brackets [].
  - You can modify a list by adding, removing, or updating elements.

    Example:

    ```
    my_list = [1, 2, 3, "apple", 4.5]
    my_list[1] = "banana"  # Modifying an element
    print(my_list)  # Output: [1, 'banana', 3, 'apple', 4.5]
    ```

- **Tuples**:
  - A **tuple** is an immutable (unchangeable) collection of items.
  - Like lists, a tuple can hold items of different data types.
  - Tuples are defined using parentheses ().
  - Once created, you cannot modify, add, or remove elements in a tuple.

    Example:

    ```
    my_tuple = (1, 2, 3, "apple", 4.5)
    print(my_tuple)  # Output: (1, 2, 3, 'apple', 4.5)
    ```

Key Differences:

Feature	List	Tuple
Mutability	Mutable (can be changed)	Immutable (cannot be changed)
Syntax	Defined using [ ]	Defined using ()
Performance	Slower due to dynamic nature	Faster due to immutability
Use Case	Suitable for collections that may change	Suitable for fixed collections

---

b) Explain the concept of inheritance with an example

Inheritance is a key concept in Object-Oriented Programming (OOP) that allows a class to **inherit** properties and behaviors (methods) from another class. This promotes code reusability and hierarchical classification.

- The class that is being inherited from is called the **parent class** or **base class**.
- The class that inherits from the parent class is called the **child class** or **derived class**.

Types of Inheritance:

1. **Single Inheritance**: One class inherits from another.
2. **Multiple Inheritance**: A class can inherit from multiple classes.
3. **Multilevel Inheritance**: A class can inherit from a class that is also inherited from another class.

Example of Single Inheritance:

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def sound(self):
        print(f"{self.name} makes a sound")

# Child class inheriting from Animal
class Dog(Animal):
    def sound(self):
        print(f"{self.name} barks")

# Creating an object of the Dog class
dog = Dog("Buddy")
dog.sound()  # Output: Buddy barks
```

Explanation:

- The Dog class inherits the properties (like name) and methods from the Animal class.
- The Dog class also overrides the sound() method from the parent class to provide its own behavior.

Inheritance helps to build a relationship between classes, avoid redundancy, and enhance code modularity.

c) Explain the features of NumPy

NumPy (Numerical Python) is a powerful Python library used for numerical and scientific computing. It supports large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

Key Features of NumPy:

1. **Efficient Multidimensional Array**: NumPy provides the ndarray object, which is an efficient and optimized way to handle multidimensional arrays.
2. **Element-wise Operations**: It supports vectorized operations, which allow for performing element-wise operations on arrays without using explicit loops, making operations faster and more concise.
3. **Broadcasting**: It supports broadcasting, a mechanism that allows operations on arrays of different shapes in a way that avoids making redundant copies of data.
4. **Mathematical Functions**: NumPy has built-in support for many mathematical operations like linear algebra, Fourier transforms, random number generation, and statistical operations.
5. **Memory Efficient**: NumPy arrays are more memory-efficient than traditional Python lists as they store elements in contiguous blocks of memory and require less overhead.
6. **Integration with Other Libraries**: NumPy is the foundation for other libraries like **Pandas**, **SciPy**, and **Matplotlib**. It can be easily integrated with other frameworks for data science and machine learning.
7. **N-dimensional Arrays**: It supports the creation and manipulation of multi-dimensional arrays like 1D (vectors), 2D (matrices), and beyond.

---

d) Write a Python GUI program to create a background with changing colors

You can use **Tkinter** to create a GUI program where the background color changes dynamically.

Example:

```
import tkinter as tk
import random

# Function to change the background color randomly
def change_color():
    # Generate a random color in hexadecimal
    color = "#{:06x}".format(random.randint(0, 0xFFFFFF))
    root.config(bg=color)
    # Call the function after 1000 milliseconds (1 second)
    root.after(1000, change_color)

# Create the main window
root = tk.Tk()
```

```
root.title("Changing Background Color")

# Set window size
root.geometry("400x400")

# Call the color changing function
change_color()

# Start the main event loop
root.mainloop()
```

Explanation:

- The change_color() function generates a random color and changes the background of the main window (root).
- The after() method is used to repeatedly call the function every 1000 milliseconds (1 second).
- The GUI will continuously change the background color.

---

e) Write a Python program to implement the concept of queue and list

Here's how you can implement a **queue** using a **list** in Python. We will use **append()** to enqueue and **pop(0)** to dequeue, simulating a queue (FIFO: First In, First Out).

Example:

```
# Queue implementation using list
class Queue:
    def __init__(self):
        self.queue = []

    # Enqueue method
    def enqueue(self, item):
        self.queue.append(item)
        print(f"Enqueued: {item}")

    # Dequeue method
    def dequeue(self):
        if len(self.queue) < 1:
            return "Queue is empty"
        return self.queue.pop(0)

    # Display the queue
    def display(self):
        print("Queue:", self.queue)
```

```
# Example usage
q = Queue()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
q.display()  # Output: Queue: [10, 20, 30]

print(f"Dequeued: {q.dequeue()}")  # Output: Dequeued: 10
q.display()  # Output: Queue: [20, 30]
```

Explanation:

- **enqueue()** adds an item to the end of the list (queue).
- **dequeue()** removes an item from the front of the list.
- The queue behaves as FIFO, ensuring that the first element added is the first one to be removed.

Q4 a) Explain the features of Pandas in Python

Pandas is a powerful and flexible Python library used for data manipulation and analysis. It provides two main data structures: **Series** (1D data) and **DataFrame** (2D data), which allow for handling and processing structured data efficiently.

Key Features of Pandas:

1. **DataFrame and Series**:
   - **Series**: A one-dimensional labeled array that can hold any data type (similar to a column in a table).
   - **DataFrame**: A two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled axes (rows and columns).
2. **Handling Missing Data**:
   - Pandas provides robust support for detecting and handling missing data using functions like isnull(), fillna(), and dropna().
3. **Data Alignment**:
   - Pandas automatically aligns data for arithmetic operations based on the labels (index) of rows and columns, making operations on data more intuitive.
4. **Data Filtering**:
   - Pandas allows for easy filtering and subsetting of data based on conditions. For example, you can filter rows by conditions on column values.
5. **Merging and Joining**:
   - Pandas supports merging and joining different DataFrames based on a key, similar to SQL operations (merge(), join()).
6. **Group By Operations**:

    - The groupby() function lets you group data based on some criteria and then apply aggregation functions like sum, mean, or count.
7. **Data Cleaning and Transformation**:
    - It offers methods for cleaning and transforming data, such as removing duplicates (drop_duplicates()), renaming columns (rename()), and reshaping data (pivot()).
8. **Powerful I/O Tools**:
    - Pandas supports reading and writing data from/to various formats like CSV, Excel, SQL databases, JSON, and more (read_csv(), to_csv()).

---

b) Explain Exception Handling in Python with Example

Exception handling in Python is a mechanism that allows the program to deal with unexpected errors or exceptions that may arise during execution. Instead of the program crashing, you can catch and handle exceptions gracefully.

Key Terms:

- **try block**: Code that might raise an exception is placed inside the try block.
- **except block**: Code to handle the exception is written inside the except block.
- **finally block**: This block contains code that will run no matter what, whether an exception occurs or not.
- **else block**: Code inside this block runs if no exceptions occur in the try block.

Example of Exception Handling:

```
# Example of exception handling
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter numbers only.")
finally:
    print("This block will always execute.")

print("Program continues after exception handling.")
```

Explanation:

- The code inside the try block may raise a ZeroDivisionError (if the user enters 0 as the second number) or a ValueError (if the user enters non-numeric data).

- The except blocks catch specific exceptions and handle them, preventing the program from crashing.
- The finally block is always executed, regardless of whether an exception occurred or not.
- The program will continue to run even after handling the exceptions.

---

c) Explain methods for geometry management in Tkinter with an example

In **Tkinter**, geometry management refers to controlling the size, position, and layout of widgets in a window. Tkinter provides three main geometry management methods:

1. **pack()**:
    - This method organizes widgets in blocks before placing them in the parent widget.
    - It arranges the widgets either vertically or horizontally.
    - You can specify the side parameter (top, bottom, left, right) to determine the positioning.

    Example:
    ```
    import tkinter as tk


    root = tk.Tk()

    # Creating buttons
    button1 = tk.Button(root, text="Button 1")
    button1.pack(side="top")  # Positioned at the top

    button2 = tk.Button(root, text="Button 2")
    button2.pack(side="left")  # Positioned on the left

    button3 = tk.Button(root, text="Button 3")
    button3.pack(side="right")  # Positioned on the right

    root.mainloop()
    ```

2. **grid()**:
    - This method organizes widgets in a table-like structure of rows and columns.
    - You can specify the row and column parameters to place widgets in specific grid cells.

    Example:
    ```
    import tkinter as tk

    root = tk.Tk()

    # Creating and positioning labels using grid method
    label1 = tk.Label(root, text="Name")
    ```

```
label1.grid(row=0, column=0)

entry1 = tk.Entry(root)
entry1.grid(row=0, column=1)

label2 = tk.Label(root, text="Age")
label2.grid(row=1, column=0)

entry2 = tk.Entry(root)
entry2.grid(row=1, column=1)

root.mainloop()
```

3. **place()**:
   - This method places widgets at an absolute position based on pixel coordinates using the x and y parameters.
   - It allows more precise control over widget positioning but is less flexible than pack() and grid().

     Example:
     python
     Copy code

```
import tkinter as tk

root = tk.Tk()

# Creating and placing a label and button at specific coordinates
label = tk.Label(root, text="This is a label")
label.place(x=50, y=50)

button = tk.Button(root, text="Click Me")
button.place(x=100, y=100)

root.mainloop()
```

Summary of Geometry Managers:

- **pack()**: Simple layout, stacks widgets.
- **grid()**: Organized in rows and columns.
- **place()**: Places widgets at specific coordinates.

---

d) Write a Python program to swap the values of two variables

You can swap the values of two variables in Python using multiple techniques. Here is an easy method using Python's tuple assignment.

Example:

```
# Swapping using tuple assignment
a = 5
b = 10

print("Before swap: a =", a, ", b =", b)

# Swapping the values of a and b
a, b = b, a

print("After swap: a =", a, ", b =", b)
```

Output:

```
Before swap: a = 5 , b = 10
After swap: a = 10 , b = 5
```

Explanation:

- The values of a and b are swapped using tuple unpacking in the line a, b = b, a. No temporary variable is needed.
- The values of the variables are swapped in one line, making it a concise solution.

---

e) Difference between Local and Global Variables:

In Python, variables can either be **local** or **global**, depending on where they are declared and how they are used.

1. **Global Variable**:
    - A global variable is declared **outside of all functions** and is accessible from any function in the program.
    - Global variables are accessible throughout the entire program, meaning they can be used inside any function or block of code.
    - To modify a global variable inside a function, the global keyword must be used.

2. **Local Variable**:
    - A local variable is declared **inside a function** and is only accessible within that function.
    - Local variables are created when the function is called and are destroyed when the function exits.
    - They have a limited scope and cannot be accessed outside the function where they are defined.

Key Differences:

Feature	Global Variable	Local Variable
Scope	Accessible throughout the program	Accessible only within the function
Declared	Outside of all functions	Inside a function
Lifetime	Exists as long as the program runs	Created and destroyed within function execution
Modification	Can be modified inside a function using global	Cannot be modified outside its function

---

Q5 Short Notes:

a) Raise Statement

In Python, the raise statement is used to explicitly raise an exception. It allows you to trigger an exception at any point in the program, typically to signal that something unexpected or erroneous has occurred.

- You can use raise to raise built-in exceptions (like ValueError, TypeError) or custom exceptions.
- When the raise statement is executed, the program will stop executing and move to the nearest except block that can handle the raised exception, if available.

Syntax:

```
raise ExceptionType("Error message")
```

Example:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(e)
```

Explanation:

- In the above example, if b is zero, a ValueError exception is raised with the message "Cannot divide by zero".
- The program will catch the exception in the except block and handle it gracefully.

---

b) Package

In Python, a **package** is a way of organizing related modules into a directory hierarchy. A package is essentially a collection of modules, and it allows developers to organize their code logically, making it easier to maintain and reuse.

- A package is simply a directory that contains a special __init__.py file, which indicates that the directory should be treated as a package.
- Packages can contain sub-packages and modules, allowing for hierarchical structuring of complex projects.

Key Features:

1. **Modular Organization**: Helps in organizing a large collection of modules into smaller, manageable sub-packages.
2. **Namespace Management**: Each package has its own namespace, preventing naming conflicts between modules.

Example:

Folder structure:

```
my_package/
    __init__.py
    module1.py
    module2.py
```

Usage:

```
# Importing a module from a package
from my_package import module1

module1.some_function()
```

Explanation:

- my_package is a package containing two modules: module1.py and module2.py.
- The __init__.py file can be empty, or it can include initialization code for the package.

---

c) HAS-A Relationship

In object-oriented programming (OOP), the **HAS-A relationship** refers to **composition** or **aggregation**, where one class contains a reference to one or more objects of another class. This relationship implies that one object "has" another object as a part of its functionality, rather than inheriting its behavior (which would be an IS-A relationship).

Characteristics of HAS-A Relationship:

1. **Composition**:
   - The containing object (whole) "owns" the contained object (part).
   - The lifecycle of the contained object depends on the lifecycle of the containing object. If the containing object is destroyed, the contained object is also destroyed.
   - This is a stronger form of HAS-A relationship.
2. **Aggregation**:
   - In aggregation, the contained object can exist independently of the containing object.
   - The relationship is more loosely coupled than in composition.

Example (Composition):

```
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

class Car:
    def __init__(self, make, engine):
        self.make = make
        self.engine = engine  # Car HAS-A Engine

    def display_info(self):
        print(f"{self.make} car has an engine with {self.engine.horsepower}
horsepower.")

# Creating an Engine object
engine = Engine(200)

# Creating a Car object that HAS-A Engine
car = Car("Toyota", engine)
car.display_info()
```

Output: Toyota car has an engine with 200 horsepower.