**Task 1: Python Programming Basics**

1 .) Describe the difference between mutable and immutable types in Python. Provide examples.

In Python, Every variable in Python holds an instance of an object. There are two types of objects in Python i.e. **Mutable**and **Immutable objects**. Whenever an object is instantiated, it is assigned a unique object id. The type of the object is defined at the runtime and it can’t be changed afterward. However, its state can be changed if it is a mutable object.

**Difference Between Mutable and Immutable Objects**

|  |  |  |
| --- | --- | --- |
| **Features** | **Mutable Objects** | **Immutable Objects** |
| **Definition** | Can be modified after they are created | Cannot be modified after they are created |
| **Modification Behaviour** | Changes affect the original object directly | Any change creates a new object with the modified value |
| **Variable References** | Multiple variables can refer to the same mutable object, leading to unexpected changes | Immutable objects are thread-safe and can be shared between threads without the risk of unexpected changes |
| **Use Cases** | Ideal for dynamic data structures needing frequent updates, like lists or dictionaries | Suitable as dictionary keys or set elements since their values remain constant and always have the same hash value |
| **Performance** | Modifying a mutable object can be quicker and use less memory than creating a new object, especially for large data structures | Often used for constants or values that should not change, like numeric values or strings used for formatting messages |

 **Mutable types** can be modified after creation (e.g., list, dict, set).

# List is mutable

my\_list = [1, 2, 3]

my\_list[0] = 10

print(my\_list) # Output: [10, 2, 3]

# Dictionary is mutable

my\_dict = {'a': 1, 'b': 2}

my\_dict['a'] = 100

print(my\_dict) # Output: {'a': 100, 'b': 2}

 **Immutable types** cannot be modified after creation (e.g., int, str, tuple).

# Creating a number

first = 5                   # Changing the number creates a new object

first = first + 1

print(first)

**2.) Write a Python program to check if a number is a palindrome.**

2.) Write a Python program to check if a number is a palindrome.

def is\_palindrome(number):

    num\_str = str(number)

    return num\_str == num\_str[::-1]

number = int(input("Enter a number: "))

if is\_palindrome(number):

    print(f"{number} is a palindrome.")

else:

    print(f"{number} is not a palindrome.")

**output**:

PS C:\Users\VAIDIK PATEL\demo\task> python -u "c:\Users\VAIDIK PATEL\demo\task\task2.py"

Enter a number: 123321

123321 is a palindrome.

**3. What are Python decorators? Provide an example of how to use one. Answer: A decorator is a function that takes another function as input and extends or modifies its behavior without changing the original function.**

def my\_decorator(func):

    def wrapper():

        print("Something is happening before the function is called.")

        func()

        print("Something is happening after the function is called.")

    return wrapper

@my\_decorator

def say\_hello():

    print("Hello!")

say\_hello()

**output:**

Something is happening before the function is called.

Hello!

Something is happening after the function is called.

**4. Explain the purpose of \_\_init\_\_ in Python classes.**

The \_\_init\_\_ method in Python is a special method that is automatically called when a new instance of a class is created. It serves as the constructor for the class, allowing you to initialize the object's attributes or perform any setup actions required when the object is instantiated.

### Purpose of \_\_init\_\_:

* **Initialization**: The \_\_init\_\_ method is used to initialize the attributes of an object when it is created.
* **Customization**: You can pass arguments to \_\_init\_\_ to customize how the object's attributes are set when it is created.
* **Automatic Invocation**: The \_\_init\_\_ method is automatically called when a new instance of the class is created.

class Person:

    def \_\_init\_\_(self, name, age):

        self.name = name

        self.age = age

person1 = Person("Alice", 30)

print(person1.name)

print(person1.age)

**output:**

PS C:\Users\VAIDIK PATEL\demo\task> python -u "c:\Users\VAIDIK PATEL\demo\task\task1\_4.py"

Alice

30

**5. What is the difference between a list and a tuple? When would you use each?**

Lists and tuples are both sequence data types that can store a collection of items. However, there are some key differences between them:

Lists:

- Mutable: You can modify, add, or remove elements after the list is created.

- Syntax: Lists are defined using square brackets `[]`.

- Performance: Lists have a slight overhead due to their dynamic nature, which can make them slower than tuples for certain operations.

- Use Case: Use lists when you need a collection of items that may change over time, such as a list of tasks or a collection of user inputs.

Example:

```python

my\_list = [1, 2, 3]

my\_list.append(4)  # Modifies the list in place

print(my\_list)  # Output: [1, 2, 3, 4]

```

Tuples:

- Immutable: Once a tuple is created, you cannot modify, add, or remove elements.

- Syntax: Tuples are defined using parentheses `()`.

- Performance: Tuples are generally faster than lists because of their immutability.

- Use Case: Use tuples when you need a collection of items that should not change, such as coordinates, fixed data records, or as keys in a dictionary.

Example:

```python

my\_tuple = (1, 2, 3)

new\_tuple = my\_tuple + (4,)  # Creates a new tuple

print(my\_tuple)  # Output: (1, 2, 3)

print(new\_tuple)  # Output: (1, 2, 3, 4)

```