Unit-I

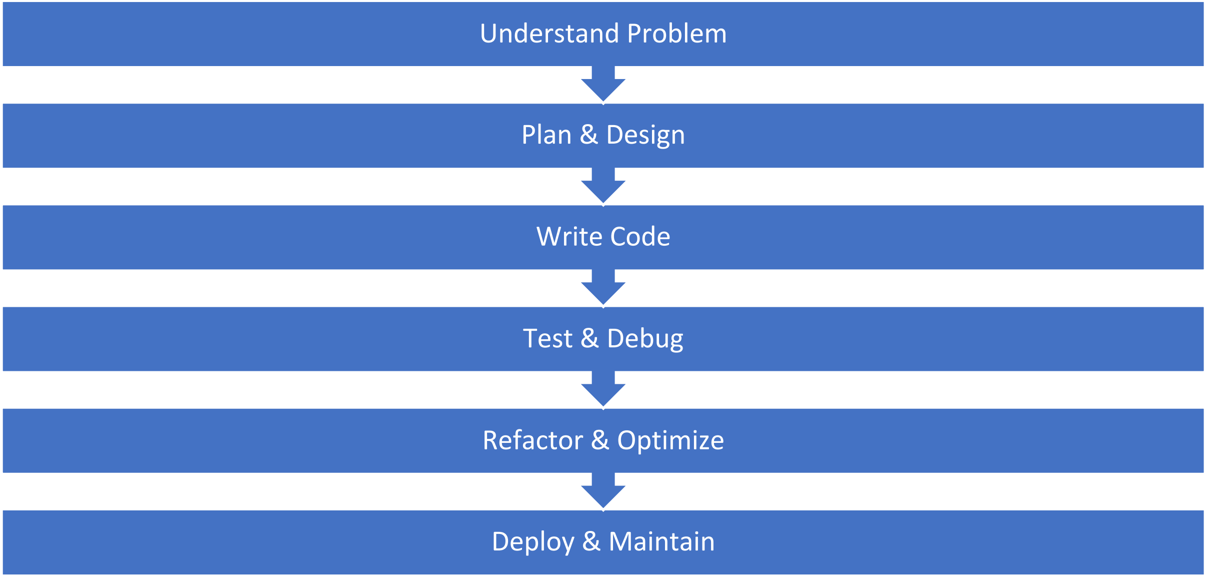
Q1) What is Python Programming? Explain its features. Write the steps of python programming cycle.
Ans: Python programming refers to the process of writing, testing, and executing code using the Python programming language, which is known for its simplicity, readability, and versatility. Python is a high-level, interpreted language that is widely used in various domains, including web development, data analysis, artificial intelligence, scientific computing, and automation.

Features of Python

1. **Easy to Learn and Use**: Python has a straightforward syntax that mimics natural language, making it beginner-friendly and easy to read.
2. **Interpreted Language**: Python code is executed line by line, which allows for easier debugging and testing.
3. **Dynamically Typed**: Variables in Python do not require an explicit declaration of their type, making it flexible and reducing boilerplate code.
4. **Rich Standard Library**: Python comes with a comprehensive standard library that supports many common programming tasks, such as file I/O, system calls, and even Internet protocols.
5. **Cross-Platform Compatibility**: Python can run on various operating systems, including Windows, macOS, and Linux, without modification.
6. **Object-Oriented**: Python supports object-oriented programming (OOP) principles, enabling the creation of reusable and modular code.
7. **Extensive Community Support**: Python has a large and active community that contributes to a wealth of libraries, frameworks, and resources.
8. **Support for Multiple Paradigms**: Besides OOP, Python also supports procedural and functional programming styles.
9. **Integration Capabilities**: Python can easily integrate with other languages like C, C++, and Java, and it can also call C/C++ libraries.
10. **Popular in Data Science and AI**: Python has become the language of choice for data analysis, machine learning, and artificial intelligence, thanks to libraries like NumPy, Pandas, TensorFlow, and scikit-learn.

Programming cycle for Python Programming

Here's a simplified diagram illustrating the basic stages of the Python programming cycle:



- Understand Problem: In this stage, you analyze and comprehend the problem you need to solve. Understand the requirements, and constraints, and expected outcomes.
- Plan and Design: Once you understand the problem, you can plan and design a solution. Break down the problem into smaller, manageable tasks. Decide on the data structures, algorithm, and overall program structure.
- Write Code: This is the implementation stage. Write the actual code based on your plan and design. Convert your algorithmic thinking into a Python program.
- Test and Debug: After writing the code, test it thoroughly to ensure it works as expected. Identity and fix any errors or bugs(debugging) that may arise during testing.
- Refactor and optimize: In this stage, you review your code and make improvements. Optimize the code for better performance, readability, and maintainability. Refactor the code if necessary to enhance its structure and organization.
- Deploy and Maintain: Once your code is thoroughly tested, you can deploy it to desired environment. This could involve integrating it into a larger system or making it available to end-users. Additionally, you may need to maintain the code by applying updates, fixing issues, and addressing user feedback.

Q2) Explain the various operators available in Python with their precedence.
Ans: Python Operator

- The operator is a symbol that performs a certain operational between two operands, according to one definition.
- In a particular programming language, operators serve as the foundation upon which logic is constructed in a program.

The different operators that Python offers are listed here.

1. Arithmetic Operators:
   - Arithmetic operations between two operands are carried out using arithmetic operations.
   - It includes the exponent (**) operator as well as the + (addition), - (subtraction), * (multiplication), / (division), % (remainder), and // (floor division) operators.
2. Comparison Operators:
   - Comparison operator compare the values of the two operands and return a true or false Boolean value in accordance.

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal, then the condition becomes true.
<=	The condition is met if the first operands is smaller than or equal to the second.
>=	The condition is met if the first operand is greater or equal to the second.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

3. Assignment Operators:
   - The right expression's value is assigned to the left operand using the assignment operators.

The following table provides a description of the assignment operators:

Operator	Description
=	Assign the value of the right side of the expression to the left side operand
+=	Add right side operand with left side operand and then assign the result to left operand
-=	Subtract right side operand from left side operand and then assign the result to left operand
*=	Multiply right operand with left operand and then assign the result to the left operand
/=	Divide left operand with right operand and then assign the result to the left operand
%=	Divides the left operand with the right operand and then assign the remainder to the left operand
//=	Divide left operand with right operand and then assign the value(floor) to left operand
**=	Calculate exponent(raise power) value using operands and then assign the result to left operand

4. Bitwise Operators:
   - The two operands value are processed bit by the bitwise operators.

Consider the case below:

Operator	Description
& (AND)	Result bit 1, if both operand bits are 1; otherwise results bit 0.
| (OR)	Result bit 1, if any of the operand bit is 1; otherwise results bit 0.
^ (XOR)	Result bit 1, if any of the operand bit is 1 but not both, otherwise results bit 0.
~ (NOT)	Inverts individual bits.
>> (Right shift)	The left operand's value is moved toward right by the number of bits specified by the right operand.
<< (Left shift)	The left operand's value is moved toward left by the number of bits specified by the right operand.

5. Logical Operators:
   - The assessment of expressions to make decisions typically makes use of the logical operators.

The following logical operators are supported by Python:

Operator	Description
and	This returns True if both values are true
or	This returns True if one value is true
not	Reverses a result, so if something is True , not turns it False

Operator Associativity:

- The term “operator associativity” refers to the order in which operators of the same precedence are evaluated when they appear consecutively in an expression.
- In Python, most operators have left-to-right associativity, which means they are evaluated from left to right.

Left-to-right associativity:

1. Arithmetic Operators: +, -, *, /, %, //, **
2. Assignment Operators: =, +=, -=, *=, /=, %=, //=, **=
3. Bitwise Operators: &, |,^, <<, >>
4. Comparison Operator: ==, !=, >, <, >=, <=
5. Logical Operators: And, Or

Right-to-left associativity:

1. Exponentiation Operator: **
2. Unary Operators: -,+, ~

Q3) Explain:
i) Implicit Type Conversion **ii) Explicit Type Conversion**

i) Implicit Type Conversion: It is when the Python interpreter automatically converts a variable from one data type to another. For example, if you add an integer and a float, the Python interpreter automatically converts the integer to a float before performing the addition.

Example:

```
# Define an integer
num_int = 5

# Define a float
num_float = 2.5

# Perform addition (Python implicitly converts int to float)
result = num_int + num_float

# Output the result and its type
print("Result:", result)         # Output: 7.5
print("Type of result:", type(result))  # Output: <class 'float'>
```

ii) Explicit type conversion: It is when you explicitly convert a variable from one data type to another using a built-in function. The most common type conversion functions in Python are

1. int(): Converts a value to an integer.
2. float(); Converts a value to a float.
3. str(): Converts a value to a string.

Example:

```
# Define a string containing a number
num_str = "100"

# Convert the string to an integer using int()
num_int = int(num_str)

# Perform an arithmetic operation
result = num_int + 50

# Output the result and its type
print("Result:", result)          # Output: 150
print("Type of num_int:", type(num_int))  # Output: <class 'int'>
```

Q4) a) Write a program to generate random number between 0 to 255 and explain some mathematical functions.

b) Discuss arithmetic, assignment, comparison, logical and bitwise operators in detail.

a) **Ans:**

Python Program:

```
import random

# Function to generate a random number between 0 and 255
def generate_random_number():
    return random.randint(0, 255)

# Main function to execute the program
if __name__ == "__main__":
    random_number = generate_random_number()
    print(f"Random number between 0 and 255: {random_number}")
```

b) **Ans:**

1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations.

- **Addition (+)**: Adds two numbers. For example, 5 + 3 results in 8.
- **Subtraction (-)**: Subtracts the second number from the first. For instance, 5 - 3 yields 2.
- **Multiplication (*)**: Multiplies two numbers. For example, 5 * 3 gives 15.
- **Division (/)**: Divides the first number by the second and returns a float. For instance, 5 / 2 results in 2.5.
- **Floor Division (//)**: Divides and returns the largest integer less than or equal to the result. For example, 5 // 2yields 2.
- **Modulus (%)**: Returns the remainder of the division. For instance, 5 % 2 gives 1.
- **Exponentiation (**)**: Raises the first number to the power of the second. For example, 5 ** 3 results in 125.

2. Assignment Operators

Assignment operators are used to assign values to variables, often in conjunction with arithmetic operations.

- **Simple Assignment (=)**: Assigns the value on the right to the variable on the left. For example, x = 5 sets x to 5.
- **Add and Assign (+=)**: Adds the right operand to the left operand and assigns the result to the left operand. For instance, x += 3 is equivalent to x = x + 3.
- **Subtract and Assign (-=)**: Subtracts the right operand from the left operand and assigns the result to the left operand. For example, x -= 3 means x = x - 3.
- **Multiply and Assign (*=)**: Multiplies and assigns. For example, x *= 3 results in x = x * 3.
- **Divide and Assign (/=)**: Divides and assigns. For instance, x /= 2 results in x = x / 2.
- **Floor Divide and Assign (//=)**: Performs floor division and assigns the result. For example, x //= 2.
- **Modulus and Assign (%=)**: Computes the modulus and assigns. For instance, x %= 2 results in x = x % 2.
- **Exponentiate and Assign (**=)**: Raises to a power and assigns. For example, x **= 2 results in x = x ** 2.

3. Comparison Operators

Comparison operators compare two values and return a Boolean result (True or False).

- **Equal to (==)**: Checks if two values are equal. For example, 5 == 5 returns True.
- **Not Equal to (!=)**: Checks if two values are not equal. For instance, 5 != 3 returns True.
- **Greater than (>)**: Checks if the left value is greater than the right. For example, 5 > 3 returns True.
- **Less than (<)**: Checks if the left value is less than the right. For example, 5 < 3 returns False.
- **Greater than or Equal to (>=)**: Checks if the left value is greater than or equal to the right. For example, 5 >= 5returns True.
- **Less than or Equal to (<=)**: Checks if the left value is less than or equal to the right. For example, 5 <= 3 returns False.

4. Logical Operators

Logical operators combine conditional statements and return Boolean results.

- **AND (and)**: Returns True if both statements are true. For example, True and False results in False.
- **OR (or)**: Returns True if at least one of the statements is true. For instance, True or False results in True.
- **NOT (not)**: Reverses the logical state of its operand. For example, not True results in False.

5. Bitwise Operators

Bitwise operators work at the binary level and are used to manipulate individual bits of integers.

- **Bitwise AND (&)**: Compares each bit of two numbers and returns 1 if both bits are 1. For example, 5 & 3 (binary 0101 & 0011) results in 1 (binary 0001).
- **Bitwise OR (|)**: Compares each bit and returns 1 if at least one of the bits is 1. For instance, 5 | 3 results in 7(binary 0111).
- **Bitwise XOR (^)**: Compares each bit and returns 1 if the bits are different. For example, 5 ^ 3 results in 6 (binary 0110).
- **Bitwise NOT (~)**: Inverts all bits. For example, ~5 results in -6 (inverts the bits of 5).

- **Left Shift (<<)**: Shifts the bits of a number to the left, filling in with zeros. For instance, 5 << 1 results in 10(binary 1010).
- **Right Shift (>>)**: Shifts the bits of a number to the right. For example, 5 >> 1 results in 2 (binary 0010).

Unit-II

Q 1)

a) **Write a program to convert uppercase letters to lowercase and vice versa.**
b) **Discuss usage of continue, break, and pass keyword in python.**

Ans a)

```
def convert_case(text):
    # Convert uppercase to lowercase and lowercase to uppercase
    converted_text = ''
    for char in text:
        if char.islower():
            converted_text += char.upper()
        elif char.isupper():
            converted_text += char.lower()
        else:
            converted_text += char  # Non-alphabetical characters remain unchanged
    return converted_text

# Get input from the user
user_input = input("Enter a string: ")
result = convert_case(user_input)
print("Converted string:", result)
```

Ans b) In Python, the keywords continue, break, and pass are control flow statements that help manage the flow of loops. Here's a brief overview of each:

1. continue

- **Usage**: The continue statement is used to skip the rest of the code inside a loop for the current iteration and move to the next iteration.
- **When to use**: It's useful when you want to avoid executing certain code under specific conditions.

Example:

```
for i in range(10):
    if i % 2 == 0:
        continue  # Skip even numbers
    print(i)  # This will print only odd numbers
```

Output:

```
1
3
5
7
9
```

2. break

- **Usage**: The break statement is used to exit a loop prematurely. When break is encountered, the loop terminates, and control moves to the statement following the loop.
- **When to use**: It's useful when you need to stop a loop based on a condition that is evaluated during the loop execution.

Example:

```
for i in range(10):
    if i == 5:
        break  # Exit the loop when i equals 5
    print(i)
```

Output:

```
0
1
2
3
4
```

3. pass

- **Usage**: The pass statement is a null operation; it does nothing when executed. It's typically used as a placeholder in situations where syntactically a statement is required but you do not want any action to be taken.
- **When to use**: It's useful during development when you want to outline your code structure but haven't implemented specific parts yet.

Example:

```
for i in range(5):
   if i < 3:
      pass  # Placeholder; no action is taken
   else:
      print(i)
```

Output:

```
3
4
```

Q2) a) How to distinguish variables when global and local both are with same name? Explain command line argument also.

b) Explain while loop and for loop with syntax and example in detail.

Ans a)

In Python, when you have a variable with the same name in both the global and local scope, you can distinguish between them using the global keyword. Here's how it works:

Distinguishing Between Global and Local Variables

1. **Local Scope**: Variables defined inside a function are local to that function.
2. **Global Scope**: Variables defined outside any function are global.

When a local variable has the same name as a global variable, the local variable takes precedence within the function.

Using the global Keyword

If you want to modify a global variable inside a function, you need to declare it as global using the global keyword. This tells Python to use the global variable instead of creating a local one.

Example:

```
x = 10  # Global variable

def example():
   global x  # Declare x as global
   x = 20    # Modify the global variable
   print("Inside function:", x)

example()
print("Outside function:", x)
```

Output:

```
Inside function: 20
Outside function: 20
```

Command Line Arguments

Command line arguments allow you to pass parameters to a Python script from the command line when you run it. You can access these arguments using the sys module.

Using sys.argv

- sys.argv is a list in Python that contains the command line arguments passed to the script. The first element (sys.argv[0]) is the name of the script itself.

Example:

```
import sys
```

```
if __name__ == "__main__":
    print("Script name:", sys.argv[0])
    print("Number of arguments:", len(sys.argv) - 1)
    print("Arguments:", sys.argv[1:])  # Exclude the script name
```

Running the Script

You can run the script from the command line and pass arguments to it. For example:

```
python script.py arg1 arg2 arg3
```

Output:

```
Script name: script.py
Number of arguments: 3
Arguments: ['arg1', 'arg2', 'arg3']
```

Q3) What is dictionary in python? Explain the various methods in dictionary.

Ans) A **dictionary** in Python is an unordered, mutable collection of items. It stores data in key-value pairs, where each key is unique. Dictionaries are commonly used when there is a need to store and retrieve data based on specific keys.

Properties of Dictionaries

1. **Unordered**: Before Python 3.7, dictionaries were unordered collections. However, starting from Python 3.7, dictionaries maintain insertion order.
2. **Mutable**: The contents of a dictionary (key-value pairs) can be changed (add, update, or delete).
3. **Keys must be unique**: No two keys can have the same value. If a key is duplicated, the last assignment will prevail.
4. **Keys must be immutable**: Keys can be of any data type that is immutable (e.g., strings, numbers, or tuples), but not lists or other dictionaries.
5. **Values can be any data type**: Values can be of any type, and they can even be lists or other dictionaries.

Creating and Accessing a Dictionary

You can create a dictionary using curly braces {} or the dict() constructor.

e.g.

```
# Creating a dictionary
my_dict = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Accessing dictionary values using keys
print(my_dict["name"])  # Output: Alice
print(my_dict.get("age"))  # Output: 25
```

Dictionary Operations

1. **Adding or Updating Items**: You can add new key-value pairs or update existing ones.

   e.g.

   ```
   # Adding a new key-value pair
   my_dict["profession"] = "Engineer"

   # Updating an existing key-value pair
   my_dict["age"] = 26
   ```
2. **Removing Items**: You can remove items from a dictionary using del, pop(), or popitem().

   e.g.

   ```
   # Using del to remove a key-value pair
   del my_dict["city"]
   ```

```
# Using pop() to remove and return a value
age = my_dict.pop("age")
print(age)  # Output: 26

# Using popitem() to remove and return the last inserted pair (from Python 3.7 onwards)
last_item = my_dict.popitem()
print(last_item)  # Output: ('profession', 'Engineer')
```

3. **Checking if a Key Exists**: Use the in keyword to check if a key exists in a dictionary.

e.g.

```
if "name" in my_dict:
   print("Key 'name' exists in the dictionary")
```

4. **Dictionary Length**: Use the len() function to find the number of key-value pairs in a dictionary.

e.g.

```
print(len(my_dict))  # Output: 1 (after the deletions above)
```

5. **Iterating Through a Dictionary**: You can loop through keys, values, or key-value pairs.

e.g.

```
# Iterating through keys
for key in my_dict:
   print(key)

# Iterating through values
for value in my_dict.values():
   print(value)

# Iterating through key-value pairs
for key, value in my_dict.items():
   print(f"{key}: {value}")
```

Dictionary Methods

1. **clear()**: Removes all items from the dictionary.

e.g.

```
my_dict.clear()
print(my_dict)  # Output: {}
```

2. **copy()**: Returns a shallow copy of the dictionary.

e.g.

```
new_dict = my_dict.copy()
```

3. **fromkeys()**: Creates a new dictionary with keys from an iterable and values set to a specified value (default is None).

e.g.

```
keys = ["name", "age", "city"]
new_dict = dict.fromkeys(keys, "Unknown")
print(new_dict)  # Output: {'name': 'Unknown', 'age': 'Unknown', 'city': 'Unknown'}
```

4. **get()**: Returns the value for a specified key. If the key is not found, it returns a default value (default is None).

e.g.

```
print(my_dict.get("name"))
# Output: None (as the dictionary is empty after the clear)
```

5. **items()**: Returns a view object that displays a list of the dictionary's key-value pairs.

e.g.

```
person_dict = {"name": "Eve", "age": 28}
print(person_dict.items())
# Output: dict_items([('name', 'Eve'), ('age', 28)])
```

6. **keys()**: Returns a view object that displays a list of all the keys.

e.g.

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

7. **values()**: Returns a view object that displays a list of all the values.

e.g.

```
print(person_dict.values())
# Output: dict_values(['Eve', 28])
```

8. **setdefault()**: Returns the value for a specified key if it is in the dictionary. If not, it inserts the key with a default value.
   e.g.
   ```
   age = person_dict.setdefault("age", 25)
   # Output: 28 (already exists)
   profession = person_dict.setdefault("profession", "Artist")
   print(person_dict)
   # Output: {'name': 'Eve', 'age': 28, 'profession': 'Artist'}
   ```

9. **update()**: Updates the dictionary with elements from another dictionary or an iterable of key-value pairs.
   e.g.
   ```
   extra_info = {"city": "Chicago", "age": 29}  # age will be updated
   person_dict.update(extra_info)
   print(person_dict)
   # Output: {'name': 'Eve', 'age': 29, 'profession': 'Artist', 'city': 'Chicago'}
   ```

Q4) What is list? How to define and access the elements of list? Also discuss the operations.
Ans) A **list** in Python is a collection of items (elements) that are ordered, changeable (mutable), and allow duplicate elements. Lists are one of the most commonly used data types in Python.

Features of a list:

- **Ordered**: Items have a defined order, and that order will not change unless explicitly modified.
- **Mutable**: You can change, add, and remove elements after the list has been created.
- **Allow duplicates**: Lists can contain the same value multiple times.

Creating a List:

Lists are created using square brackets [], and elements are separated by commas.
e.g.

```
my_list = [1, 2, 3, "Hello", True]
print(my_list)
Output:[1, 2, 3, 'Hello', True]
```

Accessing List Items:

You can access items in a list using their **index**. The first element has an index of 0.
e.g.

```
my_list = [10, 20, 30, 40]
print(my_list[0])  # Output: 10
print(my_list[2])  # Output: 30
```

Negative Indexing:
Negative indices can be used to access elements from the end of the list.
e.g.

```
my_list = [10, 20, 30, 40]
print(my_list[-1])  # Output: 40
print(my_list[-2])  # Output: 30
```

List Operations

Here are some basic list operations:

- **Adding elements**:
  - Use append() to add a single element to the end of the list.
  - Use extend() to add multiple elements.

  e.g.
  ```
  my_list = [1, 2, 3]
  my_list.append(4)  # Adds 4 at the end
  my_list.extend([5, 6])  # Adds multiple elements
  print(my_list)  # Output: [1, 2, 3, 4, 5, 6]
  ```

- **Modifying elements**: You can modify an element by assigning a new value at a specific index.
  e.g.
  ```
  my_list = [1, 2, 3]
  my_list[1] = 10
  print(my_list)  # Output: [1, 10, 3]
  ```
- **Removing elements**:
  - Use remove() to remove a specific element.
  - Use pop() to remove an element by index.

e.g.

```
my_list = [1, 2, 3, 4]
my_list.remove(2)  # Removes 2
my_list.pop(1)  # Removes the element at index 1
print(my_list)  # Output: [1, 4]
```

- **Slicing**: You can extract a sublist using slicing.

  e.g.

  ```
  my_list = [1, 2, 3, 4, 5]
  print(my_list[1:4])  # Output: [2, 3, 4]
  ```

List Functions

Python provides several built-in functions for working with lists:

- len(list): Returns the number of elements in a list.
- sorted(list): Returns a sorted version of the list.
- sum(list): Returns the sum of all numeric elements.

e.g.

```
numbers = [10, 20, 30]
print(len(numbers))  # Output: 3
print(sum(numbers))  # Output: 60
```

List Operations (Functions)

These are general **functions** that can be used with lists and other iterable objects.

1. **len()**: Returns the number of items in a list.

   e.g.

   ```
   my_list = [1, 2, 3, 4]
   print(len(my_list))  # Output: 4
   ```
2. **max()**:Returns the largest item in the list.

   e.g.

   ```
   my_list = [1, 5, 3, 9]
   print(max(my_list))  # Output: 9
   ```
3. **min()**:Returns the smallest item in the list.

   e.g.

   ```
   my_list = [1, 5, 3, 9]
   print(min(my_list))  # Output: 1
   ```
4. **sum()**:Returns the sum of all elements in the list (only works with numeric data).

   e.g.

   ```
   my_list = [1, 2, 3, 4]
   print(sum(my_list))  # Output: 10
   ```
5. **sorted()**:Returns a new list that is a sorted version of the original list.

   e.g.

   ```
   my_list = [3, 1, 4, 2]
   print(sorted(my_list))  # Output: [1, 2, 3, 4]
   ```
6. **list()**:Converts an iterable (like a string or tuple) into a list.

   e.g.

   ```
   my_string = "hello"
   print(list(my_string))  # Output: ['h', 'e', 'l', 'l', 'o']
   ```
7. **any()**:Returns True if any element in the list is True.

   e.g.

   ```
   my_list = [0, False, True]
   print(any(my_list))  # Output: True
   ```
8. **all()**:Returns True if all elements in the list are True.

   e.g.

   ```
   my_list = [1, 2, 3]
   print(all(my_list))  # Output: True
   ```

List Methods

These are **methods** that are specifically designed to work with lists. Methods are called on the list object itself.

1. **append()**:Adds a single item to the end of the list.

   e.g.

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

```
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

2. **extend()**:Adds all items from another iterable (like another list) to the end of the list.
   e.g.
   ```
   my_list = [1, 2, 3]
   my_list.extend([4, 5])
   print(my_list)  # Output: [1, 2, 3, 4, 5]
   ```
3. **insert()**:Inserts an item at a specific index.
   e.g.
   ```
   my_list = [1, 2, 4]
   my_list.insert(2, 3)  # Insert 3 at index 2
   print(my_list)  # Output: [1, 2, 3, 4]
   ```
4. **remove()**:Removes the first occurrence of a specific item.
   e.g.
   ```
   my_list = [1, 2, 3, 2]
   my_list.remove(2)
   print(my_list)  # Output: [1, 3, 2]
   ```
5. **pop()**:Removes and returns the item at the given index (or the last item if no index is provided).
   e.g.
   ```
   my_list = [1, 2, 3]
   my_list.pop(1)  # Removes and returns the item at index 1
   print(my_list)  # Output: [1, 3]
   ```
6. **clear()**:Removes all elements from the list.
   e.g.
   ```
   my_list = [1, 2, 3]
   my_list.clear()
   print(my_list)  # Output: []
   ```
7. **index()**:Returns the index of the first occurrence of a specific item.
   e.g.
   ```
   my_list = [1, 2, 3, 2]
   print(my_list.index(2))  # Output: 1
   ```
8. **count()**:Returns the number of occurrences of a specific item in the list.
   e.g
   ```
   my_list = [1, 2, 2, 3]
   print(my_list.count(2))  # Output: 2
   ```
9. **reverse()**:Reverses the order of the elements in the list in place.
   e.g.
   ```
   my_list = [1, 2, 3]
   my_list.reverse()
   print(my_list)  # Output: [3, 2, 1]
   ```
10. **sort()**:Sorts the list in ascending order by default. You can use the reverse=True argument to sort in descending order.
    e.g.
    ```
    my_list = [3, 1, 2]
    my_list.sort()
    print(my_list)  # Output: [1, 2, 3]

    my_list.sort(reverse=True)
    print(my_list)  # Output: [3, 2, 1]
    ```
11. **copy()**:Returns a shallow copy of the list.
    e.g.
    ```
    my_list = [1, 2, 3]
    new_list = my_list.copy()
    print(new_list)  # Output: [1, 2, 3]
    ```

Q5) What is tuple define its properDes and working with funcDons.

Ans) A **tuple** is an ordered, immutable collection of elements. Tuples are similar to lists but with a key difference—once a tuple is created, its elements cannot be modified, added, or removed. Tuples are commonly used when a sequence of elements should remain unchanged throughout the program.

Properties of Tuples

- **Ordered**: The elements in a tuple are ordered and indexed. Each element has a specific position within the tuple.

 Immutable: Once a tuple is created, its elements cannot be changed. You cannot add, remove, or modify elements.
 Heterogeneous: Tuples can contain elements of different data types (e.g., integers, strings, floats).
 Allow duplicates: Tuples can have duplicate elements.
 Fixed size: Once created, the size of a tuple cannot change.

Creating and Accessing a Tuple

You can create a tuple by placing elements inside parentheses () separated by commas.
e.g.

```
# Creating a tuple
my_tuple = (1, "hello", 3.14)

# Accessing elements using indexing
print(my_tuple[0])  # Output: 1
print(my_tuple[1])  # Output: "hello"

# Accessing elements using negative indexing
print(my_tuple[-1])  # Output: 3.14
```

Tuple Operations

1.Concatenation: You can concatenate two or more tuples using the + operator.
e.g.

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
tuple3 = tuple1 + tuple2
print(tuple3)  # Output: (1, 2, 3, 4, 5, 6)
```

2. Repetition: You can repeat a tuple using the * operator.
e.g.

```
tuple1 = (1, 2)
tuple2 = tuple1 * 3
print(tuple2)  # Output: (1, 2, 1, 2, 1, 2)
```

3. Slicing: You can slice a tuple to get a portion of its elements.
e.g.

```
my_tuple = (1, 2, 3, 4, 5)
sliced_tuple = my_tuple[1:4]
print(sliced_tuple)  # Output: (2, 3, 4)
```

4. **Length**: Use the len() function to get the number of elements in a tuple.
e.g.

```
my_tuple = (1, 2, 3)
print(len(my_tuple))  # Output: 3
```

5. **Membership Test**: Use the in and not in operators to check if an element is present in a tuple.
e.g.

```
my_tuple = (1, 2, 3)
print(2 in my_tuple)  # Output: True
print(4 not in my_tuple)  # Output: True
```

6. **Tuple Unpacking**: You can unpack a tuple into individual variables.
e.g.

```
my_tuple = (1, 2, 3)
a, b, c = my_tuple
print(a, b, c)  # Output: 1 2 3
```

Working with Tuple Functions and Methods

1.count(): Returns the number of times an element appears in a tuple.
e.g.

```
my_tuple = (1, 2, 2, 3, 2)
print(my_tuple.count(2))  # Output: 3
```

2. **index()**: Returns the index of the first occurrence of an element.
e.g.

```
my_tuple = (1, 2, 3, 2)
```

```
print(my_tuple.index(2))  # Output: 1
```

Working with Functions and Tuples

1. **Returning Multiple Values**: Functions can return multiple values as a tuple.
e.g.

```
def get_point():
   return (2, 3)

x, y = get_point()
print(x, y)  # Output: 2 3
```

2. **Passing a Tuple to a Function**: Tuples can be passed as arguments to functions.
e.g.

```
def print_tuple(t):
   for item in t:
      print(item)

my_tuple = (1, 2, 3)
print_tuple(my_tuple)
# Output:
# 1
# 2
# 3
```