NAME	
ROLL NUMBER	
SEMESTER	
COURSE CODE	DCA_3104
COURSE NAME	Python Programming

SET - I

Question 1.A) Explain the features and application areas of python programming in detail.

Answer 1.A) Python, a high-level, interpreted language, is celebrated for its simplicity, readability, and versatility. It boasts a clean and intuitive syntax that enhances code comprehension and maintenance, making it an ideal choice for beginners and experienced programmers alike. Python's key features include a rich standard library, support for various programming paradigms, cross-platform compatibility, and open-source accessibility.

Its extensive standard library accelerates development by offering modules for diverse purposes, from web development with Django and Flask to data science and machine learning via NumPy, Pandas, and Scikit-Learn. Python has carved a niche in AI and scientific computing, with TensorFlow and PyTorch dominating deep learning and SciPy aiding scientific research.

Furthermore, Python serves as a robust tool for automation, scripting, game development (Pygame), and building desktop GUI applications (Tkinter, PyQt). It powers IoT projects, web scraping tasks, financial applications, and educational initiatives due to its adaptability and simplicity.

In essence, Python's adaptability, rich ecosystem, and active community contribute to its prevalence across an array of industries and applications. Whether you're a data scientist, web developer, AI researcher, or educator, Python provides the tools and resources necessary to excel in your domain, making it one of the most popular and versatile programming languages today.

Question 1.B) What are the different types of data types used in python programming?

Answer 1.B) Python supports several data types, categorized as follows:

1.Numeric Types:-These include integers (`int`), floating-point numbers (`float`), and complex numbers (`complex`).

2.Sequence Types:- Python offers strings (`str`), which are sequences of characters; lists (`list`), ordered and mutable collections; and tuples (`tuple`), ordered and immutable collections.

3.Mapping Type:-The dictionary (`dict`) is an unordered collection of key-value pairs.

4.Set Types:- Python has mutable sets (`set`) and immutable frozensets (`frozenset`) for handling collections of unique elements.

5.Boolean Type:- Booleans (`bool`) represent True or False values and are fundamental for logical operations.

6. Binary Types:-Python provides bytes and bytearrays to work with sequences of bytes. Bytes are immutable, while bytearrays are mutable.

7.None Type:-The `None` type signifies the absence of a value, often used to represent null values.

8.Sequence Types for Text:-Although strings were mentioned earlier, they are a fundamental data type for representing text.

9. Type Conversion Types:- Python has built-in functions such as `int()`, `float()`, `str()`, and `bool()` for converting between data types.

10.Custom Data Types:- Programmers can create custom data types using classes, a core feature of Python's object-oriented programming.

These data types are essential for working with diverse data and structures efficiently. Python's flexibility in handling data types, along with its simplicity and readability, contributes to its widespread popularity across various domains, from web development to data science and beyond. Understanding these data types is crucial for effective Python programming and data manipulation.

Question 2.A) Explain membership and identity operators with example.

Answer 2.A) In Python, membership and identity operators are used to test relationships between values and objects. Here's an explanation of each type along with examples:

1. Membership Operators:

Membership operators check whether a value or element is present within a sequence, such as a string, list, tuple, or set. Python has two membership operators:

- in: Returns True if a value is found in the sequence.
- not in: Returns True if a value is not found in the sequence.

```
# Using the 'in' operator
fruits = ['apple', 'banana', 'cherry']
print('banana' in fruits)  # True, because 'banana' is in the list
# Using the 'not in' operator
if 'orange' not in fruits:
    print('No oranges in the list')  # This will be printed
```

2. Identity Operators:

Identity operators compare the memory locations of two objects to check if they are the same object in memory. In Python, there are two identity operators:

- is: Returns True if two variables reference the same object in memory.

  is not: Returns True if two variables reference different objects in memory.

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

y = x  # y references the same list as x

# Using the 'is' operator

print(x is y)  # True, because x and y point to the same list
```

Identity operators are particularly useful when you want to compare whether two variables reference the exact same object in memory, while membership operators are used to check if a value exists within a collection

Question 2.B) Discuss the use of else statement with for and while loop.

Answer 2.B) The `else` statement in Python is a valuable tool for adding conditional behavior after the completion of `for` and `while` loops.

With a `for` loop, the `else` block is executed when the loop iterates through all items in an iterable without encountering a `break` statement. If a `break` statement is executed during the loop, the `else` block is skipped.

In a `while` loop, the `else` block runs when the loop condition becomes `False`, signifying that the loop terminated naturally without any `break` statement intervening.

These `else` clauses provide a clean and readable way to express conditions related to loop completion. For instance, you might use an `else` block to execute a certain action if a specific item is not found in a list (in the case of a `for` loop) or to handle post-loop cleanup tasks (in the case of a `while` loop).

By using `else` with loops, you can write more structured and expressive code, making it easier to convey your intentions and handle scenarios that involve loop completion.

Question 3.A) Write a program to delete duplicate elements from list without using remove () function.

Answer 3.A) ` remove()` function by creating a new list and adding elements to it only if they haven't been added already. Here's a Python program to do that:

```
def remove_duplicates(input_list):
    # Create an empty list to store unique elements
    unique_list = []
 # Iterate through the input list
    for item in input_list:
        # If the item is not already in the unique list, add it
        if item not in unique_list:
            unique_list.append(item)
return unique_list

# Example usage
original_list = [1, 2, 2, 3, 4, 4, 5]
result_list = remove_duplicates(original_list)

print("Original List:", original_list)
print("List with Duplicates Removed:", result_list)
```

In this program:

We define a function remove_duplicates that takes an input list as its parameter.

We create an empty list called unique_list to store the unique elements.

We iterate through the input list, and for each element, we check if it's already in the unique_list. If not, we add it to the unique_list.

Finally, we return the unique_list containing only the unique elements.

When you run this program with the original_list, it will remove the duplicates, and result_list will contain [1, 2, 3, 4, 5]. The original list remains unchanged.

Question 3.B) Explain the differences between list, tuple and set in detail.

Answer 3.B) Lists, tuples, and sets are three fundamental data structures in Python, each with distinct characteristics that make them suitable for different scenarios.

1. List:

- Mutability: Lists are mutable, meaning their contents can be modified after creation.
- Syntax: Defined with square brackets [], and elements are separated by commas.
- Order: Lists are ordered collections, preserving the order of elements based on their insertion sequence.
- Duplicates: Lists allow duplicate elements.
- Access: Elements are accessed using zero-based indexing, e.g., my_list[0] retrieves the first element.
- Use Cases: Lists are ideal for scenarios requiring a dynamic, ordered collection that may change over time. They are well-suited for tasks like maintaining a list of items, records, or data that needs modification.

2. Tuple:

- Immutability: Tuples are immutable, meaning their contents cannot be altered once created.
- Syntax: Defined with parentheses (), and elements are separated by commas.
- Order: Like lists, tuples maintain element order based on insertion.
- Duplicates: Tuples can contain duplicate elements.
- Access: Elements are accessed through indexing, making them suitable for data that shouldn't change.
- Use Cases: Tuples are used when you want to ensure data integrity and immutability. They're suitable for storing data that should remain constant, like coordinates, database records, or function return values.

3. Set:

- Mutability: Sets are mutable for adding and removing elements but not indexable.
- Syntax: Defined with curly braces {} or using the set() constructor, with elements separated by commas.
- Order: Sets are unordered collections; they don't maintain the order of elements.
- Duplicates: Sets automatically eliminate duplicate elements.
- Access: Sets are not indexable, and elements cannot be accessed by index.
- Use Cases: Sets are useful for managing unique collections and performing set operations like union, intersection, or difference. They are efficient for membership testing and handling distinct elements in data.

In choosing between these data structures, consider your specific requirements, including mutability, order, duplicates, and the need for unique elements, to make the best choice for your programming task.

SET - II

Question 1.) "Hi, Python is very popular language". Write a program to count the number of uppercase, lowercase characters, special characters and spaces in the above string

Answer 4.A)

```
# Input string
input_string = "Hi, Python is very popular language"

# Initialize count variables
uppercase_count = 0
lowercase_count = 0
special_char_count = 0
space_count = 0

# Iterate through each character in the string
for char in input_string:
    if char.isupper():
        uppercase_count += 1
    elif char.islower():
        lowercase_count += 1
    elif char.isspace():
        space_count += 1
    else:
        special_char_count += 1
# Print the results
print("Uppercase characters:", uppercase_count)
print("Lowercase characters:", lowercase_count)
print("Special characters:", special_char_count)
print("Spaces:", space_count)
```

Program Output look like

Uppercase characters: 2
Lowercase characters: 27
Special characters: 1
Spaces : 5

Question 4.B) Explain the ways of deleting an element from dictionaries.

Answer 4.B)

1. Using the `del` Statement : -The ` del ` statement is a straightforward and commonly used way to remove a specific key-value pair from a dictionary. You can use it by specifying the key you want to delete . For example , `del my_dict['key']` will remove the key `' key '` along with its associated value from the dictionary `my_dict`. You can also use ` del ` to delete the entire dictionary by using `del my_dict`.

2. Using the `pop(key)` Method : - The ` pop( key ) ` method is another method for deleting elements from a dictionary . By providing the key as an argument, this method removes the corresponding key - value pair and returns the value associated with the key . It ' s useful when you want to both remove an item and retrieve its value in one step. For example, `value = my_dict.pop('key')` removes the key ` ' key ' ` from ` my_dict ` and assigns its associated value to the variable `value`. If the key is not found in the dictionary, a KeyError will be raised, so it's advisable to provide a default value as a second argument, like `my_dict.pop('key', default_value)`.

3. Using the `popitem()` Method : - The `popitem()` method is a unique way to delete elements from a dictionary. It removes and returns the last key-value pair from the dictionary as a tuple. In Python 3.7 and later versions, dictionaries are guaranteed to maintain insertion order, which means that the " last added " item will be removed. While this method might not be suitable for situations where you need to specify a specific key to delete , it can be handy when you want to remove and process items in a sequential manner.

4. Using the `clear()` Method : -The `clear()` method provides a way to delete all key - value pairs from a dictionary , effectively making it an empty dictionary . This method is useful when you need to reset or clear the entire dictionary while retaining its structure .

In summary, Python offers several methods to delete elements from dictionaries, allowing you to manage the contents of dictionaries efficiently based on your specific needs. Whether you want to delete a specific key - value pair , remove the last item added , or clear the entire dictionary, these methods provide flexibility and control over dictionary operations.

Question 5.A) Explain the difference between keyword arguments and variable length function with example.

Answer 5.A) Keyword arguments and variable-length functions are two important concepts in Python that enhance the flexibility of function parameter passing, making your code more readable and versatile. Here, we'll discuss the differences between them without providing specific code examples.

Keyword Arguments:

Keyword arguments are a way to pass arguments to a function by explicitly specifying the parameter names in the function call. This approach allows you to provide values for specific parameters in any order, making your code more self-explanatory and less error-prone.

With keyword arguments, you can:

- Specify which parameter receives which value, enhancing code readability.
- Skip optional parameters by providing values only for the parameters you want to set.
- Use default values for parameters that are not provided in the function call.

Keyword arguments are particularly useful when dealing with functions that have a large number of parameters or when you want to make your code more explicit and easy to understand.

Variable-Length Functions:

Variable-length functions, often implemented using *args for positional arguments and **kwargs for keyword arguments, allow you to pass a variable number of arguments to a function without explicitly specifying their names.

With variable-length functions, you can:

- Handle an arbitrary number of arguments, which can be beneficial for functions with varying input requirements.
- Create more flexible and generic functions that can accept different numbers of arguments.
- Process arguments in a way that doesn't depend on the parameter names.

Variable-length functions are commonly used in situations where you want to build functions that can adapt to different use cases, such as mathematical operations, data processing, or creating custom utilities.

In summary, keyword arguments and variable-length functions offer different advantages and use cases in Python. Keyword arguments provide clarity and readability in function calls by specifying parameter names explicitly, whereas variable-length functions enable you to work with a variable number of arguments, offering flexibility and adaptability in your code. The choice between them depends on your specific requirements and coding style, but understanding both concepts is essential for writing effective and maintainable Python code.

Question 5.B) Write a program to create user defined exception and raise it.

Answer 5.B) In Python, you can create custom exceptions by defining a new class that inherits from the built-in Exception class or one of its subclasses. Here's an example of how to create a user-defined exception and raise it in a program:

1. In this program: -We define a custom exception class called MyCustomException, which inherits from the built-in Exception class. You can provide a custom error message when initializing an instance of this class, or it will default to "This is a custom exception."
2. We create a function divide_numbers(a, b) that attempts to divide two numbers a and b. If b is equal to zero, it raises our custom exception MyCustomException.
3. In the try block, we call the divide_numbers function with a numerator of 10 and a denominator of 0, which triggers the custom exception.
4. We catch the custom exception using an except block for MyCustomException and print the custom error message.
5. Additionally, we have a separate except block to catch the built-in ZeroDivisionError that might be raised if the denominator is zero. This demonstrates that our custom exception is distinct from the built-in exceptions.

```
# Define a custom exception class by inheriting from Exception
class MyCustomException(Exception):
    def __init__(self, message="This is a custom exception"):
        self.message = message
        super().__init__(self.message)
def divide_numbers(a, b):
    if b == 0:
        # Raise the custom exception when attempting to divide by zero
        raise MyCustomException("Division by zero is not allowed")
    return a / b
try:
    numerator = 10
    denominator = 0
    result = divide_numbers(numerator, denominator)
    print(f"Result: {result}")
except MyCustomException as e:
    print(f"Custom Exception: {e}")
except ZeroDivisionError as e:
    print(f"Zero Division Error: {e}")
```

Program Output look like

Custom Exception: Division by zero is not allowed

Question 6.A) Write a program to implement any( ), all( ), sum( ), min( ), max( ), bool( ), bytes( ) functions. Explain their uses also.

Answer 5.A)

1. The any() function checks if at least one element in the numbers list is True (non-zero). It returns True because there are non-zero elements.
2. The all() function checks if all elements in the numbers list are True. It returns False because there's a "falsy" value (0) in the list.
3. The sum() function calculates the sum of all elements in the numbers list, resulting in a sum of 15.
4. The min() function finds the minimum element in the numbers list, which is 0.
5. The max() function finds the maximum element in the numbers list, which is 5.
6. The bool() function converts the numbers list to a Boolean value. Since the list is not empty, it returns True.
7. The bytes() function converts the byte_values list, containing ASCII values, into a bytes object, resulting in b'Hello'.

These functions provide valuable tools for working with iterable data and performing common operations like checking for truthiness, calculating sums, and finding minimum and maximum values. The bytes() function is especially useful for handling binary data.

```
 # Define a list of numbers
numbers = [1, 2, 3, 4, 5, 0]
# 1. any() function: Checks if at least one element in the iterable is True.
any_result = any(numbers)
print(f"Any result: {any_result}")  # Output: True (because there are non-zero elements)
# 2. all() function: Checks if all elements in the iterable are True.
all_result = all(numbers)
print(f"All result: {all_result}")  # Output: False (because 0 is a "falsy" value)
# 3. sum() function: Calculates the sum of all elements in the iterable.
sum_result = sum(numbers)
print(f"Sum result: {sum_result}")  # Output: 15 (sum of all elements)
# 4. min() function: Finds the minimum element in the iterable.
min_result = min(numbers)
print(f"Min result: {min_result}")  # Output: 0 (the minimum element)
# 5. max() function: Finds the maximum element in the iterable.
max_result = max(numbers)
print(f"Max result: {max_result}")  # Output: 5 (the maximum element)
# 6. bool() function: Converts a value to a Boolean (True or False).
bool_result = bool(numbers)
print(f"Bool result: {bool_result}")  # Output: True (because the list is not empty)
# 7. bytes() function: Converts an iterable of integers (0-255) to a bytes object.
byte_values = [72, 101, 108, 108, 111]  # ASCII values for "Hello"
bytes_result = bytes(byte_values)
print(f"Bytes result: {bytes_result}")  # Output: b'Hello' (a bytes object)
# Note: The bytes() function is often used for encoding text data in binary format.
```

Program Output look like

```
Min result: 0
Max result: 5
Bool result: True
Bytes result: b’Hello’
```

Question 6.B) Explain the use of pandas, numpy and matplotlib libraries with example.

Answer 5.A)

```
import pandas as pd

# Create a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}

df = pd.DataFrame(data)

# Display the DataFrame
print(df)
```

```
 import numpy as np
# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])
# Perform array operations
mean = np.mean(arr)
std_dev = np.std(arr)
# Display results
print(f"Array: {arr}")
print(f"Mean: {mean}")
print(f"Standard Deviation: {std_dev}")
```

```
 import matplotlib.pyplot as plt
# Create data for a simple line plot
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
# Create a line plot
plt.plot(x, y)
# Add labels and a title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')
# Display the plot
plt.show()
```

These examples showcase the basic functionality of each library: creating DataFrames in Pandas, performing numerical operations with NumPy, and creating a simple line plot using Matplotlib. These libraries are versatile and can be used for much more complex tasks in their respective domains.