Q1. Is an assignment operator like += only for show? Is it possible that it would lead to faster results at the runtime?

Sol:-

The += operator is an example of an in-place operation, where the result of the operation is stored back into the original variable. Instead of creating a new object, the existing object is modified. This can be more efficient in terms of memory and execution time compared to creating a new object each time.

In the case of mutable objects like lists or arrays, using += can be faster than creating a new list and assigning it back to the variable. This is because appending elements to an existing list in-place requires fewer memory allocations and copying operations.

# Using += operator

my\_list = [1, 2, 3]

my\_list += [4, 5, 6]

print(my\_list) # Output: [1, 2, 3, 4, 5, 6]

# Creating a new list and assigning it back

my\_list = [1, 2, 3]

my\_list = my\_list + [4, 5, 6]

print(my\_list) # Output: [1, 2, 3, 4, 5, 6]

Q2. What is the smallest number of statements you'd have to write in most programming languages to replace the Python expression a, b = a + b, a?

Sol:-

temp = a + b;

b = a;

a = temp;

Q3. In Python, what is the most effective way to set a list of 100 integers to 0?

Sol:-

Using a list comprehension:

my\_list = [0 for \_ in range(100)]

Using the \* operator with the range() function:

my\_list = [0] \* 100

Q4. What is the most effective way to initialise a list of 99 integers that repeats the sequence 1, 2, 3? S If necessary, show step-by-step instructions on how to accomplish this.

Sol:-

Create a list comprehension that iterates over a range of 99 elements.

Use the modulo operator % to calculate the index position in the sequence 1, 2, 3.

Assign the corresponding value from the sequence 1, 2, 3 to each element in the list comprehension.

my\_list = [(i % 3) + 1 for i in range(99)]

Q5. If you're using IDLE to run a Python application, explain how to print a multidimensional list as efficiently?

Sol:-

When using IDLE to run a Python application, you can print a multidimensional list efficiently by utilizing the pprint module, which provides a pprint function for pretty-printing complex data structures.

import pprint

my\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

pprint.pprint(my\_list)

[[1, 2, 3],

[4, 5, 6],

[7, 8, 9]]

Q6. Is it possible to use list comprehension with a string? If so, how can you go about doing it?

Sol:-

Yes, it is possible to use list comprehension with a string in Python. You can iterate over each character in the string and perform operations or filters to create a new list based on the characters of the string.

my\_string = "Hello, World!"

# Example 1: Create a list of individual characters from the string

char\_list = [char for char in my\_string]

print(char\_list)

# Output: ['H', 'e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd', '!']

# Example 2: Create a list of uppercase characters from the string

uppercase\_list = [char.upper() for char in my\_string if char.isalpha()]

print(uppercase\_list)

# Output: ['H', 'E', 'L', 'L', 'O', 'W', 'O', 'R', 'L', 'D']

Q7. From the command line, how do you get support with a user-written Python programme? Is this possible from inside IDLE?

Sol:-

Use the --help or -h flag: Many Python programs provide a command-line interface that includes a help option. You can typically access this help by running the program with the --help or -h flag.

python my\_program.py --help

Although IDLE does not provide direct support for user-written Python programs beyond its standard features.

Q8. Functions are said to be “first-class objects” in Python but not in most other languages, such as C++ or Java. What can you do in Python with a function (callable object) that you can't do in C or C++?

Sol:-

In Python, functions are considered "first-class objects," which means they have the following characteristics:

Functions can be assigned to variables: In Python, you can assign a function to a variable, just like any other value. This allows you to treat functions as objects and pass them around, store them in data structures, and assign them dynamically.

Functions can be passed as arguments to other functions: Python allows you to pass functions as arguments to other functions. This enables you to implement higher-order functions that can accept functions as parameters and operate on them.

Functions can be returned from other functions: Functions in Python can also be returned as values from other functions. This enables you to create functions that generate and return new functions based on certain conditions or inputs.

Functions can be defined inside other functions: Python supports nested functions, which means you can define a function within another function. The inner function can access variables from the outer function's scope, allowing for more flexible and modular code organization.

Functions can be stored in data structures: Since functions are first-class objects, you can store them in data structures like lists, dictionaries, or sets. This allows for dynamic manipulation and selection of functions at runtime.

Q9. How do you distinguish between a wrapper, a wrapped feature, and a decorator?

Sol:-

Wrapper:

A wrapper is a function or class that wraps around another function or object, providing additional functionality or modifying its behavior. It acts as an intermediary between the caller and the wrapped feature.

def wrapper(func):

def inner(\*args, \*\*kwargs):

# Do something before invoking the wrapped function

print("Wrapper: Before function invocation")

result = func(\*args, \*\*kwargs)

# Do something after invoking the wrapped function

print("Wrapper: After function invocation")

return result

return inner

def greet():

print("Hello, world!")

wrapped\_greet = wrapper(greet)

wrapped\_greet()

Wrapped feature:

The wrapped feature refers to the original function or object that is being wrapped or modified by a wrapper. In the previous example, the wrapped feature is the greet function.

Decorator:

A decorator is a specific type of wrapper that is used to modify the behavior of functions or methods in a concise and reusable way. Decorators in Python use a special syntax using the "@" symbol.

def decorator(func):

def inner(\*args, \*\*kwargs):

# Do something before invoking the wrapped function

print("Decorator: Before function invocation")

result = func(\*args, \*\*kwargs)

# Do something after invoking the wrapped function

print("Decorator: After function invocation")

return result

return inner

@decorator

def greet():

print("Hello, world!")

greet()

Q10. If a function is a generator function, what does it return?

Sol:-

A generator function in Python does not return a single value like a regular function. Instead, it returns a generator object.

A generator object is an iterator that generates a sequence of values on-the-fly, as opposed to computing and returning all the values at once. The values are generated using the yield keyword within the generator function. Each time the yield statement is encountered, the current value is returned, and the function's state is suspended. The next time the generator is iterated, the function resumes execution from where it left off and continues generating the next value.

def number\_generator(n):

for i in range(n):

yield i

# Creating a generator object

generator = number\_generator(5)

# Iterating over the generator

for num in generator:

print(num)

Q11. What is the one improvement that must be made to a function in order for it to become a generator function in the Python language?

Sol:-

To turn a regular function into a generator function in Python, you need to introduce the yield keyword in the function body. The presence of the yield keyword is what distinguishes a generator function from a regular function.

def regular\_function():

print("Executing regular function.")

return 42

def generator\_function():

print("Executing generator function.")

yield 42

# Calling regular function

result\_regular = regular\_function()

print("Regular function result:", result\_regular)

# Calling generator function

generator = generator\_function()

print("Generator function result:", generator)

Q12. Identify at least one benefit of generators.

One benefit of using generators in Python is their memory efficiency. Generators generate values on-the-fly and only keep track of the current state, rather than generating all values upfront and storing them in memory like a list. This makes generators particularly useful when dealing with large or infinite sequences of data.

By using generators, you can generate and process values one at a time, without the need to store the entire sequence in memory. This can significantly reduce memory consumption, especially when working with large datasets or when generating values dynamically.

Additionally, generators support lazy evaluation, meaning that the values are computed only when needed. This can be advantageous when working with computations or operations that are time-consuming or resource-intensive. The values are generated on-demand, allowing you to iterate over them gradually, potentially saving time and resources.