Module-2

Q. How memory is managed in Python?

Answer :- According to the Python memory management documentation, Python has a private heap that stores our program's objects and data structures. Python memory manager takes care of the bulk of the memory management work and allows us to concentrate on our code.

Types of memory allocation

There are two types of memory allocation in Python, static and dynamic.

1. Static memory

The stack data structure provides **static memory allocation**, meaning the variables are in the stack memory. Statically assigned variables, as the name implies, are permanent; this means that they must be allocated in advance and persist for the duration of the program. Another point to remember is that we cannot reuse the memory allocated in the stack memory. Therefore, memory reusability is not possible.

Ex. X=10

2. Dynamic memory

The **dynamic memory allocation** uses heap data structures in its implementation, implying that variables are in the heap memory. As the name suggests, dynamically allocated variables are not permanent and can be changed while a program is running. Additionally, allotted memory can be relinquished and reused. However, it takes longer to complete because dynamic memory allocation occurs during program execution. Furthermore, after utilizing the allocated memory, we must release it. Otherwise, problems such as memory leaks might arise.

Ex. X = [0]*5 #This memory for 5 integers is allocated on heap.

Python garbage collection

The Python garbage collector handles memory allocation and deallocation automatically in Python. Python developers have designed it to eliminate the need for manual garbage collection. Garbage collection in Python refers to the interpreter's memory management process of freeing up unneeded and undesired memory for our applications.

The garbage collector (GC) operates in the background and is triggered when the reference count reaches zero.

The reference count rises when the following occur:

- An object is given a new name
- An object is placed in a container, such as a tuple or a dictionary

The reference count lowers when the following occurs:

- An object's reference is reassigned

- An object’s reference moves out of scope
- An object is removed

The memory is a heap that stores the program’s objects and other data structures. The Python memory manager uses API methods to handle the allocation and deallocation of this heap space.

Q. What is the purpose continue statement in python?

Answer :- The continue statement in Python is used to skip the rest of the code inside a loop for the current iteration and immediately move on to the next iteration. It is typically used when you want to bypass certain parts of the loop's code based on a condition.

Here's an **example:**

```
for i in range(5):

    if i == 2:
        continue          # Skip the rest of the loop when i is 2
    print(i)
```

Output:-

0

1

2

3

Q. What are negative indexes and why are they used?

Answer :- Negative indexing allows you to access elements of a sequence from the end, using negative numbers as indexes. This can be useful for getting the last few elements of a sequence, reversing a sequence, or performing other operations that require accessing elements from the end.

Negative indexes start from index number -1, while positive indexes start from index number 0.

Why Use Negative Indexing?

1. **Convenience**: When you want to access elements from the end of a sequence, negative indexing is more intuitive and saves you from having to calculate the index relative to the length of the list.
2. **Code Simplicity**: It makes code shorter and more readable. Without negative indexing, you would have to use expressions like len(my_list) - **1** to get the last element, which is more cumbersome.
3. **Flexibility**: It allows you to easily perform operations that depend on the tail of a list or string without needing to know its exact length. This is especially useful in contexts where the length of the sequence can change.

Negative indexing is a common, useful tool that simplifies accessing elements from the end of sequences in languages that support it.

Example:-

```
mylist = [10,20,30,40,50,60,70]

print(mylist[-1])
```

output :- 70