**UDAY RAJPUT**

**PYTHON FUNDAMENTALS**

**Q. 4**  **How memory is managed in Python?**

**Ans :**

Memory management in Python is primarily handled by Python's memory manager, which is responsible for allocating and deallocating memory as needed. Here's an overview of how memory is managed in Python:

1. \*\*Garbage Collection\*\*: Python uses automatic memory management through a mechanism known as garbage collection. It automatically deallocates memory occupied by objects that are no longer referenced or reachable. Python's garbage collector periodically sweeps through memory to identify and reclaim unused memory.

2. \*\*Reference Counting\*\*: Python uses reference counting as one of its memory management strategies. Each object in Python contains a reference count, which is incremented when a new reference to the object is created and decremented when a reference to the object is deleted or goes out of scope. When the reference count reaches zero, indicating that there are no more references to the object, the memory occupied by the object is deallocated.

3. \*\*Memory Pools\*\*: Python uses memory pools to efficiently manage memory allocation for small objects. Memory pools are pre-allocated blocks of memory maintained by Python's memory manager. When small objects (such as integers or small strings) are created, they are allocated from these memory pools, reducing the overhead associated with frequent memory allocation and deallocation.

4. \*\*Dynamic Memory Allocation\*\*: For larger objects or data structures, Python uses dynamic memory allocation provided by the underlying system (typically the C runtime library). Python's memory manager interacts with the system's memory allocator to request and release memory as needed.

5. \*\*Memory Optimization Techniques\*\*: Python employs various memory optimization techniques to reduce memory overhead and improve performance. These techniques include object reuse, memory sharing, and memory pooling.

**Q 5 What is the purpose continue statement in python?**

**Ans :**

In Python, the `continue` statement is used inside loops (such as `for` loops and `while` loops) to skip the rest of the code inside the loop for the current iteration and proceed to the next iteration. Its primary purpose is to selectively execute code based on certain conditions, without exiting the loop entirely.

The `continue` statement is typically used in situations where you want to skip certain iterations of a loop based on specific conditions, but you still want to continue looping through the remaining elements or iterations.

Here's an example to illustrate the use of the `continue` statement:

```python

# Print even numbers from 1 to 10

for i in range(1, 11):

if i % 2 != 0:

# Skip odd numbers

continue

print(i)

```

In this example:

- The loop iterates through the numbers from 1 to 10.

- The `if` condition checks if the current number `i` is odd (`i % 2 != 0`).

- If the condition is `True`, indicating that the number is odd, the `continue` statement is executed. This skips the remaining code inside the loop for the current iteration and proceeds to the next iteration.

- If the condition is `False`, indicating that the number is even, the `print(i)` statement is executed, and the even number is printed.

As a result, only even numbers from 1 to 10 are printed, and odd numbers are skipped using the `continue` statement.

**Q 14 What are negative indexes and why are they used?**

**Ans :**

Negative indexes in Python are used to access elements from the end of a sequence, such as a string, list, or tuple. Instead of counting from the beginning of the sequence (where the first element has index 0), negative indexes count backward from the end of the sequence, starting with -1 for the last element, -2 for the second-to-last element, and so on.

Negative indexes are useful in situations where you want to access elements relative to the end of a sequence without needing to know its length. They provide a convenient way to access elements from the end of the sequence without explicitly calculating the index based on its length.

Here's an example to illustrate the use of negative indexes:

```python

# Accessing elements using negative indexes

my\_list = [1, 2, 3, 4, 5]

# Accessing the last element using negative index

print("Last element:", my\_list[-1])

# Accessing the second-to-last element using negative index

print("Second-to-last element:", my\_list[-2])

```

In this example:

- `my\_list[-1]` accesses the last element of the list `my\_list`.

- `my\_list[-2]` accesses the second-to-last element of the list `my\_list`.

Negative indexes provide a concise and intuitive way to access elements from the end of a sequence, especially when the length of the sequence is unknown or irrelevant. They are commonly used in Python programming for tasks such as accessing the last element of a list or string, accessing elements relative to the end of a sequence, and implementing algorithms that involve backward traversal of sequences.