

1> How memory is managed in Python?

Memory management in Python is handled automatically, which makes it easier for developers to focus on writing code without worrying about manual memory allocation and deallocation. Here are the key aspects of how Python manages memory:

- ✚ **Private Heap Space:** Python uses a private heap space for memory management. All Python objects and data structures are stored in this heap. The Python interpreter takes care of managing this private heap.
- ✚ **Memory Allocation:** When a new object is created, Python dynamically allocates memory for it. The amount of memory allocated depends on the object's type and size.
- ✚ **Reference Counting:** Python uses a technique called reference counting to keep track of objects in memory. Each object has a reference count that indicates how many references point to it. When an object's reference count drops to zero, it means the object is no longer in use and can be deallocated.
- ✚ **Garbage Collection:** In addition to reference counting, Python has a built-in garbage collector that helps manage memory. The garbage collector periodically checks for objects that are no longer referenced and frees up the memory occupied by them. This helps in reclaiming

memory that might not be immediately freed by reference counting alone.

- ✚ **Memory Fragmentation:** Python manages memory fragmentation by keeping track of free memory blocks and allocating memory from these blocks when needed. This helps in efficiently utilizing memory and reducing fragmentation.

2> What is the purpose continue statement in python?

- ✚ The continue statement in Python is used within loops to skip the rest of the code inside the loop for the current iteration and move to the next iteration immediately. This can be particularly useful when you want to bypass certain conditions or values while iterating over a sequence.

- ✚ Here's a simple example :

```
for i in range(1, 11):  
    if i == 5:  
        continue # Skip the rest of the code inside the  
loop for this iteration  
    print(i)
```

- ✚ In this example, the loop will print numbers from 1 to 10, but it will skip printing the number 5 because of the continue statement.

- ✚ When `i` equals 5, the `continue` statement is executed, and the loop immediately proceeds to the next iteration, skipping the `print(i)` statement for that iteration.

3> What are negative indexes and why are they used?

- ✚ Negative indexes in Python allow you to access elements from the end of a sequence, such as a list or a string, rather than from the beginning.
- ✚ This can be particularly useful when you want to quickly access elements at the end without knowing the exact length of the sequence.
- ✚ Here's how negative indexing works:
 - The index `-1` refers to the last element.
 - The index `-2` refers to the second last element, and so on.

For example,

- ✚ consider the list `my_list = [10, 20, 30, 40, 50]`:
 - `my_list[-1]` returns 50 (the last element).
 - `my_list[-2]` returns 40 (the second last element).

Negative indexing is useful in various scenarios, such as:

1. **Accessing the last elements:** Quickly access elements from the end without calculating the length.
2. **Reversing sequences:** Easily iterate over a sequence in reverse order.
3. **Slicing:** Create slices from the end of a sequence.

Here's an example :

```
my_list = [10, 20, 30, 40, 50]
```


```
# Accessing elements using negative indexing
```

```
print(my_list[-1]) # Output: 50
```

```
print(my_list[-2]) # Output: 40
```

```
# Slicing using negative indexing
```

```
print(my_list[-3:]) # Output: [30, 40, 50]
```

 Negative indexing provides a Convenient and intuitive way to work with sequences from the end.