1. What is Garbage Collection in Python and Why is it Important

Garbage collection is a mechanism for automatically freeing up memory by destroying objects that are no longer in use, preventing memory leaks. In Python, garbage collection is part of the memory management process, handled by Python’s automatic reference counting and a cyclic garbage collector.

- Memory Management in Python

  - Reference Counting Each object in Python has an associated reference count, which tracks how many variables point to it. When the count reaches zero, the memory is deallocated.

  - Cyclic Garbage Collection In cases where objects reference each other (circular references), Python's cyclic garbage collector identifies these and frees them.

Importance

- It ensures efficient memory use by freeing up space when objects are no longer needed, helping avoid memory exhaustion.

2. Key Differences Between NumPy Arrays and Python Lists

NumPy Arrays vs. Python Lists

Feature	NumPy Arrays	Python Lists
Data Type	- Homogeneous (same data type)	- Heterogeneous (different data types allowed)
Performance	- Faster due to fixed data types and optimized storage	- Slower as they are more flexible but less efficient
Memory Consumption	- More efficient (less memory overhead)	- Requires more memory
Operations	- Support vectorized operations (element-wise)	- Looping needed for element-wise operations
Multidimensionality	- Supports multidimensional arrays	- Requires lists of lists for multidimensionality

Advantages of Using NumPy Arrays

1. Speed NumPy arrays are implemented in C, making them much faster for numerical computations.

2. Memory Efficiency NumPy arrays use less memory as they are of fixed types.

3. Vectorization With NumPy, you can perform element-wise operations without explicit loops, leading to faster execution of operations.

---

3. How Does List Comprehension Work in Python

List comprehension is a concise way to create lists by applying an expression to each item in a sequence.

Example 1 Generate a list of squared values

```
squares = [x2 for x in range(10)]

print(squares)   # Output [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Example 2 Filter a list to include only even numbers

```
evens = [x for x in range(10) if x % 2 == 0]

print(evens)   # Output [0, 2, 4, 6, 8]
```

4. Shallow vs. Deep Copying in Python

Shallow Copy Creates a new object but references the same objects within it. Changes to nested objects will affect both the original and the copy.

- When to Use When the objects being copied do not contain mutable objects (like lists or dictionaries).

Example

```
import copy
original = [[1, 2, 3], [4, 5, 6]]
shallow_copy = copy.copy(original)

# Modifying nested list affects both original and copy
shallow_copy[0][0] = 99
print(original)   # Output  [[99, 2, 3], [4, 5, 6]]
```

Deep Copy Creates a new object and recursively copies all objects it contains, ensuring complete independence from the original.

- When to Use When you want a completely independent copy, including nested objects.

Example

```
import copy
original = [[1, 2, 3], [4, 5, 6]]
deep_copy = copy.deepcopy(original)

# Modifying deep copy does not affect original
deep_copy[0][0] = 99
print(original)   # Output  [[1, 2, 3], [4, 5, 6]]
```

5. Difference Between Lists and Tuples

Feature	Lists	Tuples
Mutability	Mutable (can change elements)	Immutable (cannot change elements)
Syntax	Defined with `[]`	Defined with `()`
Methods and `index()`	Supports methods like `append()`, `remove()`	Limited methods, supports `count()`
Use Case collection	When you need a dynamic, changeable collection	When you need a static, fixed
Performance	Slightly slower (due to mutability)	Faster due to immutability

Example of Lists

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

my_list.append(4)   # Adds element to the list

print(my_list)   # Output [1, 2, 3, 4]
```

Example of Tuples

```
my_tuple = (1, 2, 3)

# my_tuple[0] = 4  # This will raise an error because tuples are immutable

print(my_tuple)   # Output (1, 2, 3)
```