**Week Two Bootcamp Questions**

**1) Garbage Collection in Python and Memory Management**

Garbage collection in Python is a form of automatic memory management that deals with reclamation of memory occupied by objects that are no longer needed. The main ways Python manages memory include reference counting and a cyclic garbage collector.

Reference Counting: Each Python's entity has a reference count associated with it-a number of references to that best. Once it reaches "zero", the memory is automatically freed.

• Cyclic GC: Cycles are sets of two or more objects referencing each other. In Python, it discovers reference cycles that are no longer reachable and does the garbage collection.

Garbage collection is an essential process, as all manual memory management-like languages like C-is error-prone due to memory leaks and segmentation faults. Python's GC will ensure that Python applications use memory efficiently and prevent crashes and/or excessive memory consumption for long-running applications.

**2) Key Differences Between NumPy Arrays and Python Lists**

|  |  |  |
| --- | --- | --- |
| Feature | NumPy Array | Python List |
| Data Type | Homogeneous (same data type) | Heterogeneous (different data types) |
| Memory Usage | More efficient (compact memory usage) | Less efficient (more memory overhead) |
| Performance | Faster (vectorized operations) | Slower for large data sets |
| Mathematical Operations | Supports element-wise operations directly | Requires loops or list comprehensions |
| Dimensionality | Supports multi-dimensional arrays | Only 1D lists (need nested lists for multi-dimensionality) |

**Advantages of Using NumPy Arrays:**

* **Efficiency**: NumPy arrays consume less memory and perform faster operations due to their homogeneous structure.
* **Vectorized Operations**: Mathematical computations (addition, multiplication, etc.) are optimized for arrays, avoiding loops.
* **Multi-dimensional**: NumPy natively supports n-dimensional arrays, which are crucial for scientific and numerical computations.

Example:

import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr \* 2) # Outputs: [2 4 6 8]

**3) List Comprehension in Python**

List comprehension is a concise way to create lists by iterating over an iterable and optionally filtering elements. It follows the format:

[expression for item in iterable if condition]

**Example 1: Generating squared values**

squares = [x\*\*2 for x in range(10)]

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

**Example 2: Filtering values based on a condition**

even\_numbers = [x for x in range(10) if x % 2 == 0]

print(even\_numbers) # Outputs: [0, 2, 4, 6, 8]

List comprehension simplifies code, making it more readable and compact compared to traditional loops.

**4) Shallow vs. Deep Copying in Python**

* **Shallow Copy**: Creates a new object but inserts references to the original objects in it. Changes to the nested objects reflect in both the original and copied objects.

**Example:**

import copy

original = [[1, 2], [3, 4]]

shallow\_copy = copy.copy(original)

shallow\_copy[0][0] = 10

print(original) # Outputs: [[10, 2], [3, 4]]

print(shallow\_copy) # Outputs: [[10, 2], [3, 4]]

* **Deep Copy**: Creates a new object and recursively copies all objects inside it. The new object is independent of the original one.

**Example:**

deep\_copy = copy.deepcopy(original)

deep\_copy[0][0] = 20

print(original) # Outputs: [[10, 2], [3, 4]]

print(deep\_copy) # Outputs: [[20, 2], [3, 4]]

* **When to Use**:
  + Use **shallow copy** when you need a new object but don't mind if changes to nested objects affect the original.
  + Use **deep copy** when you need complete independence between the original and copied objects, especially when dealing with complex, nested structures.

**5) Difference Between Lists and Tuples**

|  |  |  |
| --- | --- | --- |
| Feature | List | Tuple |
| Mutability | Mutable (can change elements) | Immutable (cannot change elements) |
| Syntax | Defined with square brackets [] | Defined with parentheses () |
| Performance | Slower (due to mutability overhead) | Faster (less overhead as they are immutable) |
| Use Case | Use when you need to modify data | Use when data should remain constant |
| Methods | Supports methods like append(), remove(), etc. | Limited methods (supports only count, index) |

**Example of a List:**

my\_list = [1, 2, 3]

my\_list.append(4) # Now: [1, 2, 3, 4]

**Example of a Tuple:**

my\_tuple = (1, 2, 3)

# Tuples are immutable, so you can't change their elements directly

# my\_tuple[0] = 4 # This will raise an error

* **When to Use**:
  + **Lists** are used when you need a collection that can change over time.
  + **tuples** are usedwhen you need a fixed collection, like coordinates or constant data values, to ensure data integrity.