**Assignment 1:-**

Q1. **1. Mutability**

* + You can modify (add, remove, change) elements after the list is created.
  + Example:

my\_list = [1, 2, 3]

my\_list[0] = 10 # OK

my\_list.append(4) # OK

* **Tuple**: **Immutable**
  + You cannot modify the contents once the tuple is created.
  + Example

my\_tuple = (1, 2, 3)

my\_tuple[0] = 10 # TypeError

**2. Performance**

* **Tuple** is generally **faster** than a list for iteration and fixed-size data due to its immutability.
  + It has **less overhead** than a list.
  + Good for **read-only** operations.
* **List** is **slower** comparatively because it needs to maintain extra machinery to allow mutation.

**3. When to Use What**

| **Situation** | **Use List** | **Use Tuple** |
| --- | --- | --- |
| You need to modify elements | ✅ | ❌ |
| Fixed set of values (e.g., coordinates) | ❌ | ✅ |
| You care about performance for read-only data | ❌ | ✅ |
| You want to use as a **dictionary key** or in a **set** | ❌ | ✅ (only if elements inside are also immutable) |
| Semantics: data as a collection vs. record | ✅ | ✅ (use tuple for a record-like data structure) |
| Q2.  **1. Implicit Type Conversion (Type Coercion)**  Python **automatically converts** one data type to another **when needed**, especially in expressions involving mixed types.  **Example: Integer to Float**  result = 5 + 3.2  print(result) # 8.2 (int + float → float)   * Here, Python **implicitly** converts 5 (int) to 5.0 (float) to perform the operation.   **Rules:**   * Lower precision → higher precision (int → float → complex) * **Safe conversions only** (no data loss)   **🛠️ 2. Explicit Type Conversion (Type Casting)**  You manually convert a value from one type to another using **built-in functions**:   | **To Type** | **Function** | **Example** | | --- | --- | --- | | int | int(x) | int(4.7) → 4 | | float | float(x) | float('3') → 3.0 | | str | str(x) | str(100) → '100' | | list | list(x) | list('abc') → ['a','b','c'] | | tuple | tuple(x) | tuple([1, 2]) → (1, 2) |   **❗ Example: String to List**  python  CopyEdit  s = "hello"  print(list(s)) # ['h', 'e', 'l', 'l', 'o']  **Example: Float to Int (truncates!)**  x = int(5.99)  print(x) # 5  Q3.  **1. List ([])**   * **Ordered** (since Python 3.7+) * **Allows duplicates** * **Indexable** * **Mutable**   **Example:**  names = ['Alice', 'Bob', 'Alice']  print(names[0]) # Alice  **Set (set())**   * **Unordered** * **No duplicates** * **Mutable** * **Not indexable**   **Example:**  unique\_names = {'Alice', 'Bob', 'Alice'}  print(unique\_names) # {'Alice', 'Bob'}  **3. Dictionary ({})**   * **Key-value pairs** * **Keys are unique** * **Order-preserving** (since Python 3.7+) * **Mutable**   **Example:**  person = {'name': 'Alice', 'age': 30}  print(person['name']) # Alice  Q4.  **\_\_repr\_\_: Developer-Friendly Representation**   * Called by:   + The **interpreter** (>>> obj)   + repr(obj) * Purpose:   + Should return a **string that could be used to recreate the object**.   + Used for **debugging** and **logging**. * **Fallback** for \_\_str\_\_ if it's not defined.   **Example:**  class Book:  def \_\_init\_\_(self, title):  self.title = title  def \_\_repr\_\_(self):  return f"Book(title={self.title!r})"  book = Book("Python 101")  print(repr(book)) # Book(title='Python 101')  **\_\_str\_\_: User-Friendly Representation**   * Called by:   + print(obj)   + str(obj) * Purpose:   + Should return a **readable** or **nicely formatted** string for **end users**.   **Example:**  class Book:  def \_\_init\_\_(self, title):  self.title = title  def \_\_str\_\_(self):  return f"'{self.title}'"  book = Book("Python 101")  print(book) # 'Python 101'  Q5.  **Python 3: Unified int**   * There is **only one integer type**: int * It can represent **any size** of integer, automatically growing in precision. * No need for a separate long type. * Example:   x = 10\*\*100 # Valid in Python 3  print(x)  **🟡 Python 2: Two Integer Types**   * int: Fixed-size (like C long), usually 32 or 64 bits depending on the system.   + Overflow raises an error or silently converts to long. * long: Arbitrary-precision type, indicated by an L suffix.   x = 12345678901234567890 # Becomes long automatically  print(type(x)) # <type 'long'>  **How Python 3 Handles Large Integers**   * Python 3 int uses **arbitrary-precision math** via a **dynamic internal representation** (similar to BigInteger in Java). * It dynamically allocates more memory as needed. * You don’t need to worry about overflow.   **Example:**  a = 999999999999999999999999999999  print(a \*\* 10)  Q6.  **With Immutable Types (e.g., int, str, tuple)**   * These **cannot be changed in place**. * Both + and += create **new objects**.   **Example: Immutable (int)**  a = 10  b = a  a += 5 # Same as: a = a + 5  print(a) # 15  print(b) # 10 → original not affected  **Example: Immutable (str)**  s = "Hi"  t = s  s += " there"  print(s) # "Hi there"  print(t) # "Hi" → not modified  **With Mutable Types (e.g., list, set, dict)**   * + creates a **new object** * += **modifies the object in place** (if supported)   **Example: Mutable (list)**  x = [1, 2]  y = x  x += [3, 4] # Modifies x in place  print(x) # [1, 2, 3, 4]  print(y) # [1, 2, 3, 4] → also changed  Q7.  **Behavior by Data Type**  **📜 1. String**   * Checks if a **substring** exists in a string.   "py" in "python" # True  "x" in "python" # False  **📋 2. List / Tuple**   * Checks if an **element** is present.   3 in [1, 2, 3, 4] # True  "cat" in ("dog", "cat") # True  **🧱 3. Set**   * Checks for **element membership** (fast lookup).   'a' in {'a', 'b', 'c'} # True  'z' in {'a', 'b', 'c'} # False  **📖 4. Dictionary**   * Checks if a **key** exists (not the value!).   d = {'name': 'Alice', 'age': 25}  'name' in d # True  'Alice' in d # False  25 in d # False  Q8.  **Bitwise Operators in Python**   | **Operator** | **Name** | **Description** | | --- | --- | --- | | & | AND | Bits that are **1 in both** | | ` | ` | OR | | ^ | XOR | Bits that are **1 in only one** | | ~ | NOT (Complement) | **Flips all bits** | | << | Left Shift | Shifts bits **left**, adds zeros on right | | >> | Right Shift | Shifts bits **right**, drops rightmost bits |   **1. & (Bitwise AND)**  a & b # 0b0101 & 0b0011 = 0b0001 = 1  ✅ Only 1 where **both bits are 1**.  **🔹 2. | (Bitwise OR)**  a | b # 0b0101 | 0b0011 = 0b0111 = 7  ✅ 1 where **either bit is 1**.  **🔹 3. ^ (Bitwise XOR)**  a ^ b # 0b0101 ^ 0b0011 = 0b0110 = 6  ✅ 1 where **only one bit is 1** (exclusive OR).  **🔹 4. ~ (Bitwise NOT)**  ~a # ~0b0101 = -(a + 1) = -6  ✅ **Inverts all bits**. For positive integers, it becomes -(n + 1).  **🔹 5. << (Left Shift)**  a << 1 # 0b0101 << 1 = 0b1010 = 10  a << 2 # 0b0101 << 2 = 0b10100 = 20  ✅ Shifts all bits **left**, **multiplies by powers of 2**.  **🔹 6. >> (Right Shift)**  a >> 1 # 0b0101 >> 1 = 0b0010 = 2  a >> 2 # 0b0101 >> 2 = 0b0001 = 1  Q9.  **Syntax and Meaning**   | **Operator** | **Equivalent To** | **Description** | | --- | --- | --- | | += | x = x + y | Addition | | -= | x = x - y | Subtraction | | \*= | x = x \* y | Multiplication | | /= | x = x / y | Division (float) | | //= | x = x // y | Floor division | | %= | x = x % y | Modulus | | \*\*= | x = x \*\* y | Exponentiation | | &=, ` | =, ^=, <<=, >>=` | Bitwise operations |   **Examples**  **+= (Addition)**  x = 10  x += 5 # same as x = x + 5  print(x) # 15  **-= (Subtraction)**  x = 10  x -= 3 # same as x = x - 3  print(x) # 7  **\*= (Multiplication)**  x = 4  x \*= 3 # same as x = x \* 3  print(x) # 12  Q10.  **Summary: == vs is**   | **Operator** | **Compares...** | **Checks for...** | | --- | --- | --- | | == | **Values** | Are the contents the same? | | is | **Identities (memory)** | Are they the **same object**? |   **== → Value Equality**  Checks if **two objects have the same value**, even if they're stored in different locations.  a = [1, 2, 3]  b = [1, 2, 3]  print(a == b) # True → values are the same  **is → Identity Equality**  Checks if **two variables point to the exact same object** in memory.  a = [1, 2, 3]  b = [1, 2, 3]  print(a is b) # False → different objects in memory |  |  |