**Assignment- 1**

**DATATPES AND OPERATORS**

1. **How do lists and tuples differ in terms of mutability and performance? When would you choose one over the other?**

Ans-

**Mutability-** Lists are changeable so this is mutable where as tuples are immutable and can’t be changes once initialized

**Performance-** lists are slower due to more memory overhead where lists are faster.

**Selection-** We can select a list when we need to perform operation like add, remove or modify operations where tuples can be selected when we need a fixed collection of values.

1. **Explain how Python handles type conversion between different data types, such as between integers and floats or between strings and lists.**

**Ans-** Python supports 2 types of type conversions-

* Implicit
* Explicit

**Implicit-** automatically done by Python interpreter itself.

Ex-

x = 10 # int

y = 2.5 # float

result = x + y # int is converted to float → result = 12.5

**Explicit-** Explicit type conversion is a type of type conversion which is done by the developer through the type casting.

Ex-

int("123") # Converts string to int

float("3.14") # Converts string to float

list("abc") # ['a', 'b', 'c']

str([1,2,3]) # '[1, 2, 3]'

1. **What are the key differences between Python’s `list`, `set`, and `dictionary` data types? Provide examples of scenarios where each would be the most appropriate choice.**

**Ans-**

**List vs Set vs Dictionary-**

| **Feature** | **List** | **Set** | **Dictionary** |
| --- | --- | --- | --- |
| **Syntax** | [1, 2, 3] | {1, 2, 3} | {"name": "Alice", "age": 25} |
| **Duplicates** | Allowed | Not allowed | Keys must be unique |
| **Order** | Ordered (as of 3.7+) | Unordered (as of <3.7) | Ordered (as of 3.7+) |
| **Use case** | Store ordered items | Membership tests, unique data | Key-value mapping |

**Examples:**

* **List**: Keep student scores in order → [85, 90, 78]
* **Set**: Remove duplicates from a list → set([1,1,2,3]) → {1,2,3}
* **Dictionary**: Store student info → {"name": "Ankit", "roll": 21}

1. **Discuss the role of the `\_\_repr\_\_` and `\_\_str\_\_` methods in custom data types. How do they differ, and when should you implement them?**

**Ans-**

In Python, \_\_repr\_\_ and \_\_str\_\_ are special methods used to define how objects of a class should be represented as strings. They are mainly used when we want to print or inspect objects.

\_\_str\_\_ is meant for the end user. It returns a readable and nicely formatted string. It is called when we use print(object) or str(object).

\_\_repr\_\_ is meant for developers and debugging. It should return an unambiguous string that can ideally be used to recreate the object. It is called when we use repr(object) or simply write the object in the Python shell.

| Feature | \_\_str\_\_ | \_\_repr\_\_ |
| --- | --- | --- |
| Purpose | User-friendly output | Developer/debugging output |
| Called by | print() or str() | repr() or interactive console |
| Format | Informal and readable | Formal and detailed |
| Fallback | If missing, uses \_\_repr\_\_ | If missing, shows default object format |

1. **How does Python handle large integers? Explain the difference between `int` in Python 2 and Python 3.**

**Ans-**

Python handles large integers very smoothly. If we give it a really big number, it can still store and calculate with it without any overflow or error. This is one of the good things about Python compared to other languages like C or Java, where integers have a fixed size.

In Python 2, there were two different types for integers: int and long.

* int was used for normal-sized numbers, and
* long was used when the number was too big for int.

But in Python 3, things got simpler. There is only one type now, called int, and it automatically becomes as big as needed. So even if the number is very large, Python 3 will still store it properly without us doing anything special.

1. **What is the difference between the `+=` operator and the `+` operator in Python when used with mutable and immutable types?**

**Ans-**

**Using + Operator:**

The + operator always creates a new object. It doesn’t change the original one. This is true for both mutable and immutable types.

Ex-

s = "Hello"

s + " World"

print(s) # Output: Hello (original string didn’t change)

**Using += Operator:**

The += operator is like a shortcut for “add and assign”, but how it behaves depends on the data type:

If the object is immutable (like int, str, tuple), then += creates a new object, just like +.

Ex-

x = 5

x += 1

# A new integer is created and assigned to x

**Difference:**

+: never changes the original object.

+=: Creates a new object for immutable types.

Updates the same object for mutable types.

1. **Explain the purpose and use of the `in` operator in Python. How does it behave differently when used with different data types, such as strings, lists, and dictionaries?**

**Ans-**

The in operator is used to check whether a value exists inside another object like a string, list, or dictionary. It returns True if the value is found, otherwise it returns False.

**In strings:-** it checks whether a small part (substring) exists inside the main string.

Example:

"py" in "python" → True

"z" in "python" → False

**In lists:-** it checks whether the element is present in the list.

Example:

2 in [1, 2, 3] → True

5 in [1, 2, 3] → False

**In dictionaries:-** it only checks for keys, not values.

Example:

"name" in {"name": "Pushpendra", "age": 21} → True

"Pushpendra" in {"name": "Pushpendra", "age": 21} → False

To check values, we use .values() with the dictionary.

1. **How do the bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`) work in Python? Provide examples of their usage.**

**Ans-**

**& (AND)** → Returns 1 if both bits are 1  
Example: 5 & 3 → 1 (binary: 101 & 011 = 001)

**| (OR)** → Returns 1 if at least one bit is 1  
Example: 5 | 3 → 7 (binary: 101 | 011 = 111)

**^ (XOR)** → Returns 1 if the bits are different  
Example: 5 ^ 3 → 6 (binary: 101 ^ 011 = 110)

**~ (NOT)** → Flips all the bits (changes 0 to 1 and 1 to 0)  
Example: ~5 → -6

**<< (Left Shift)** → Shifts bits to the left and adds 0s on the right  
Example: 5 << 1 → 10 (binary: 101 becomes 1010)

**>> (Right Shift)** → Shifts bits to the right and removes bits from the right  
Example: 5 >> 1 → 2 (binary: 101 becomes 10)

These operators are helpful when working with low-level data or optimizing performance in specific tasks.

1. **What are augmented assignment operators, and how do they work in Python? Give examples with `+=`, `-=`, and `\*=`.**

**Ans-**

Augmented assignment operators are used to update the value of a variable in a shorter way. Instead of writing the full expression again, we can use these operators to modify and assign the value at once.

**+= (Addition and assignment)**  
It adds the right value to the left and updates the variable.  
Example:  
x = 5  
x += 3 → Now x becomes 8

**-= (Subtraction and assignment)**  
It subtracts the right value from the left and updates the variable.  
Example:  
x = 10  
x -= 4 → Now x becomes 6

**\*= (Multiplication and assignment)**  
It multiplies the variable by the right value and updates it.  
Example:  
x = 6  
x \*= 2 → Now x becomes 12

These operators help in writing cleaner and shorter code.

1. **How does Python’s `is` operator differ from `==`, especially in terms of comparing lists, strings, and other complex data types?**

**Ans-**

The == operator checks if two values are **equal in content**, while the is operator checks if two variables **point to the same memory location** (same object).

**With strings** (which are immutable), is sometimes works like == because Python stores small strings in the same memory.

Example:  
a = "hello"  
b = "hello"  
a == b → True  
a is b → True

**With lists** (which are mutable), even if contents are same, they are stored in different places.

Example:  
x = [1, 2, 3]  
y = [1, 2, 3]  
x == y → True (same content)  
x is y → False (different objects)

So use == when checking values, and is only when you want to check if two variables are exactly the same object.