**Assignment 2 : Data Structure**

**Submitted by : Partha Pratim Borah**

**Q1. Discuss string slicing and provide examples.**

Ans1 : String slicing is a way to retrieve a portion of a string by specifying a range of indices. In Python, strings are indexed, starting from 0 for the first character, -1 for the last character, and so on. With string slicing, we can extract substrings using this format: string[start:stop:step]

### Parameters:

1. **start** (optional): The beginning index of the substring. The character at this index will be included.
2. **stop**: The end index of the substring. The character at this index will not be included.
3. **step** (optional): Determines the increment between each index in the slice.

If any of these parameters are omitted, Python uses default values:

* start defaults to the beginning of the string.
* stop defaults to the end of the string.
* step defaults to 1, meaning each character is taken.

### Examples of String Slicing:

#### Basic Slicing

text = "Hello, World!"

# Suppose we want to slice from index 0 to 5 (not including 5)

print(text[0:5]) # Output: Hello

# Suppose we want to slice from index 7 to the end

print(text[7:]) # Output: World!

# Suppose we want to slice the entire string

print(text[:]) # Output: Hello, World!

#### Negative Indexing

Negative indices allow us to count from the end of the string.

# Suppose we want to get the last 6 characters

print(text[-6:]) # Output: World!

# Suppose we want to get the substring from index -6 to -1

print(text[-6:-1]) # Output: World

#### Using Step

The step parameter can be used to skip characters in a slice.

# Suppose we want to get every second character from index 0 to 11

print(text[0:12:2]) # Output: Hlo ol

# Suppose we want to reverse the string

print(text[::-1]) # Output: !dlroW ,olleH

**Q2. Explan the key features of lists in Python.**

Ans2 : In Python, lists are one of the most commonly used data structures due to their flexibility and versatility. Here are the key features of lists in Python:

### 1. **Ordered**

* Lists maintain the order of elements, meaning items have a fixed position based on the order in which they were added.
* For example:

fruits = ["apple", "banana", "cherry"]

print(fruits[0]) # Output: apple

### 2. **Mutable**

* Lists are mutable, meaning you can change their content after they are created, including adding, removing, or modifying elements.
* Example:

fruits[1] = "blueberry"

print(fruits) # Output: ["apple", "blueberry", "cherry"]

### 3. **Allows Duplicates**

* Lists can contain duplicate values. We can have the same value multiple times in a list.
* Example:

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

### 4. **Supports Heterogeneous Elements**

* Lists can store elements of different data types, including integers, strings, and even other lists.
* Example:

mixed = [1, "apple", 3.5, [2, 4]]

### 5. **Dynamic Size**

* Lists in Python can grow or shrink as needed, meaning we can add or remove elements dynamically.
* Example:

numbers = [1, 2, 3]

numbers.append(4) # Add an element

print(numbers) # Output: [1, 2, 3, 4]

### 6. **Indexed Access**

* Each item in a list has an index, allowing you to access elements by their position.
* Lists also support negative indexing, where -1 refers to the last item.
* Example:

print(fruits[0]) # Output: apple

print(fruits[-1]) # Output: cherry

### 7. **Built-in Functions and Methods**

* Python provides a variety of built-in methods to work with lists, including append(), remove(), pop(), sort(), reverse(), and more.
* Example:

numbers = [3, 1, 2]

numbers.sort()

print(numbers) # Output: [1, 2, 3]

### 8. **Iterable**

* Lists are iterable, which means we can loop through them using a for loop or other iteration techniques.
* Example:

for fruit in fruits:

print(fruit)

### 9. **Supports List Comprehensions**

* Python allows us to create lists using list comprehensions, which provide a concise way to generate lists.
* Example:

squares = [x\*\*2 for x in range(5)]

print(squares) # Output: [0, 1, 4, 9, 16]

Python lists are ordered, mutable, and can contain duplicate and heterogeneous elements. They support indexed access, are dynamic in size, and come with powerful built-in methods for manipulation, making them versatile for a wide range of programming tasks.

**Q3. Describe how to access modify and delete elements in a list with examples.**

Ans3 : In Python, lists are versatile and allow easy access, modification, and deletion of elements. Below are some examples.

### 1. **Accessing Elements**

We can access elements in a list using **indexing** and **slicing**.

my\_list = [10, 20, 30, 40, 50]

# Accessing a single element by index

print(my\_list[0]) # Output: 10

# Accessing the last element with negative indexing

print(my\_list[-1]) # Output: 50

# Accessing a range of elements using slicing

print(my\_list[1:4]) # Output: [20, 30, 40]

### 2. **Modifying Elements**

Since lists are mutable, we can change individual elements or multiple elements using indexing and slicing.

my\_list = [10, 20, 30, 40, 50]

# Modifying a single element

my\_list[1] = 25

print(my\_list) # Output: [10, 25, 30, 40, 50]

# Modifying a range of elements

my\_list[2:4] = [35, 45]

print(my\_list) # Output: [10, 25, 35, 45, 50]

# Using append() to add an element at the end

my\_list.append(60)

print(my\_list) # Output: [10, 25, 35, 45, 50, 60]

### 3. **Deleting Elements**

There are several ways to remove elements from a list such as del statement, remove(), and pop().

* **Using** del **statement**:

my\_list = [10, 20, 30, 40, 50]

# Deleting an element by index

del my\_list[1]

print(my\_list) # Output: [10, 30, 40, 50]

# Deleting a range of elements

del my\_list[1:3]

print(my\_list) # Output: [10, 50]

* **Using** remove(): remove() deletes the first occurrence of a specific value.

my\_list = [10, 20, 30, 20, 50]

my\_list.remove(20)

print(my\_list) # Output: [10, 30, 20, 50]

* **Using** pop(): pop() removes and returns the element at a specific index. If no index is provided, it removes the last element.

my\_list = [10, 20, 30, 40, 50]

# Popping an element by index

popped\_element = my\_list.pop(2)

print(popped\_element) # Output: 30

print(my\_list) # Output: [10, 20, 40, 50]

# Popping the last element

my\_list.pop()

print(my\_list) # Output: [10, 20, 40]

**Q4. Compare and contrast tuples and lists with examples.**

Ans4 : Tuples and lists are both commonly used data structures in Python, but they have distinct differences in terms of mutability, syntax, and performance. Below are their similarities and differences.

### 1. **Mutability**

* **Lists** are mutable, meaning their contents can be changed after creation. We can add, remove, or modify elements in a list.
* **Tuples** are immutable, meaning their contents cannot be changed once they are created. This immutability makes tuples useful for fixed collections of items that should not change.

# Example of list mutability

my\_list = [1, 2, 3]

my\_list[0] = 10 # Modifying the first element

print(my\_list) # Output: [10, 2, 3]

# Example of tuple immutability

my\_tuple = (1, 2, 3)

# my\_tuple[0] = 10 # This will raise a TypeError

### 2. **Syntax**

* **Lists** are defined using square brackets [].
* **Tuples** are defined using parentheses () or simply by separating items with commas.

# Creating a list

my\_list = [1, 2, 3]

# Creating a tuple

my\_tuple = (1, 2, 3)

another\_tuple = 1, 2, 3 # Parentheses are optional

### 3. **Usage and Purpose**

* **Lists** are used when we need a collection of items that may change during the program’s runtime, like a collection of user inputs.
* **Tuples** are used for fixed collections of items, such as representing a coordinate (x, y) or an RGB color.

# List for a collection of modifiable items

fruits = ["apple", "banana", "cherry"]

fruits.append("date") # Lists can be modified

# Tuple for fixed items like coordinates

coordinates = (10, 20) # Tuples are typically used for fixed values

### 4. **Performance**

* **Tuples** are generally faster than lists for accessing elements, which can be advantageous when working with large datasets that don’t need modification.
* This performance difference is due to tuples’ immutability, which allows Python to optimize their storage.

### 5. **Functions and Methods**

* **Lists** come with a larger set of methods like append(), remove(), pop(), clear(), sort(), etc., since they are designed to be modified.
* **Tuples** have limited methods, mainly count() and index(), because they are immutable and lack modification methods.

# List methods

my\_list = [1, 2, 3]

my\_list.append(4) # Adds an element

print(my\_list) # Output: [1, 2, 3, 4]

# Tuple methods

my\_tuple = (1, 2, 3, 1)

print(my\_tuple.count(1)) # Output: 2 (counts occurrences of 1)

### 6. **Unpacking**

* Both **lists** and **tuples** support unpacking, allowing us to assign elements to variables in a single step.

# Unpacking a list

my\_list = [1, 2, 3]

a, b, c = my\_list

print(a, b, c) # Output: 1 2 3

# Unpacking a tuple

my\_tuple = (4, 5, 6)

x, y, z = my\_tuple

print(x, y, z) # Output: 4 5 6

### 7. **Nested Structures**

* Both lists and tuples can contain other lists or tuples, allowing for nested structures.

# Nested list

nested\_list = [[1, 2], [3, 4]]

# Nested tuple

nested\_tuple = ((1, 2), (3, 4))

**Q5. Describe the key featres of sets and provide examples of their use.**

Ans5 : In Python, a **set** is a collection of unique, unordered elements. Sets are particularly useful when we need to store distinct values and perform common mathematical operations, such as unions, intersections, and differences. Let’s explore the key features of sets and their uses.

### Key Features of Sets

1. **Unordered Collection**
   * Sets are unordered, meaning the elements are not stored in any specific sequence. We cannot access elements using an index.
   * For example:

my\_set = {3, 1, 4}

print(my\_set) # Output may vary in order, e.g., {1, 3, 4}

1. **Unique Elements**
   * Sets only store unique elements. If we try to add a duplicate, it will be ignored.
   * Example:

my\_set = {1, 2, 2, 3}

print(my\_set) # Output: {1, 2, 3} (only unique elements are stored)

1. **Mutable**
   * Sets are mutable, so we can add or remove elements after creation.
   * Example:

my\_set = {1, 2, 3}

my\_set.add(4)

print(my\_set) # Output: {1, 2, 3, 4}

1. **No Indexing or Slicing**
   * Since sets are unordered, we cannot access elements by their position (like lists or tuples).
   * We also cannot slice sets.
   * Example:

my\_set = {1, 2, 3}

# my\_set[0] # This will raise a TypeError

1. **Efficient for Membership Testing**
   * Sets provide fast membership testing, which makes them useful for checking if an item exists within a set.
   * Example:

my\_set = {1, 2, 3}

print(2 in my\_set) # Output: True

print(4 in my\_set) # Output: False

1. **Support for Mathematical Set Operations**
   * Sets support operations like **union**, **intersection**, **difference**, and **symmetric difference**.
   * Example:

set\_a = {1, 2, 3}

set\_b = {3, 4, 5}

# Union

print(set\_a | set\_b) # Output: {1, 2, 3, 4, 5}

# Intersection

print(set\_a & set\_b) # Output: {3}

# Difference

print(set\_a - set\_b) # Output: {1, 2}

# Symmetric Difference

print(set\_a ^ set\_b) # Output: {1, 2, 4, 5}

### Examples of Set Usage

1. **Removing Duplicates from a List**
   * Sets are a simple way to remove duplicates from a list since they only allow unique elements.
   * Example:

python

Copy code

my\_list = [1, 2, 2, 3, 4, 4, 5]

unique\_elements = set(my\_list)

print(unique\_elements) # Output: {1, 2, 3, 4, 5}

1. **Membership Testing**
   * Sets allow fast membership testing, making them ideal for situations where you need to check if an item exists in a large collection.
   * Example:

allowed\_values = {"yes", "no", "maybe"}

answer = "yes"

if answer in allowed\_values:

print("Answer is valid.") # Output: Answer is valid.

1. **Mathematical Set Operations**
   * Sets make it easy to perform operations like finding common items or items unique to a list.
   * Example:

students\_math = {"Alice", "Bob", "Charlie"}

students\_science = {"Bob", "David"}

# Students taking both subjects

both\_subjects = students\_math & students\_science

print(both\_subjects) # Output: {'Bob'}

1. **Filtering Data**
   * Sets can be useful when we need to filter out unwanted values from a dataset.
   * Example:

all\_words = {"apple", "banana", "grape", "orange", "lemon"}

forbidden\_words = {"grape", "orange"}

allowed\_words = all\_words - forbidden\_words

print(allowed\_words) # Output: {'apple', 'banana', 'lemon'}

**Q6. Discuss the use cases of tuples and sets in Python programming.**

Ans6 : Tuples and sets each have unique properties in Python, making them ideal for different types of programming tasks. Here are some common use cases for both:

### Use Cases for Tuples

1. **Immutable Data Collection**
   * When we need a collection of items that should remain constant throughout the program, tuples are ideal because they are immutable.
   * Use Case: Storing fixed configuration values or constants.
   * Example:

server\_config = ("192.168.1.1", 8080, "https")

1. **Data Integrity**
   * Tuples help ensure data integrity because they cannot be modified accidentally after creation.
   * Use Case: Storing fixed sets of coordinates, dates, or key-value pairs that should not be changed.
   * Example:

coordinate = (10, 20) # Immutable representation of a point

1. **Using Tuples as Dictionary Keys**
   * Because tuples are hashable (due to their immutability), they can be used as keys in dictionaries.
   * Use Case: Using coordinates or other multi-part identifiers as dictionary keys.
   * Example:

distances = {(0, 0): 5, (1, 2): 10}

print(distances[(0, 0)]) # Output: 5

1. **Packing and Unpacking**
   * Tuples are commonly used for packing and unpacking multiple values in Python.
   * Use Case: Functions that return multiple values often use tuples.
   * Example:

def get\_min\_max(numbers):

return min(numbers), max(numbers)

min\_val, max\_val = get\_min\_max([1, 2, 3, 4])

print(min\_val, max\_val) # Output: 1 4

1. **Returning Multiple Values from Functions**
   * Tuples provide a natural way to return multiple values from a function, which can be unpacked easily by the caller.
   * Example:

def divide(a, b):

quotient = a // b

remainder = a % b

return quotient, remainder

q, r = divide(10, 3)

print(q, r) # Output: 3 1

1. **Grouping Related Data**
   * Tuples work well for grouping related data, especially for short, fixed collections where field names are unnecessary.
   * Use Case: Representing a person’s name, birthdate, or other small data groups.
   * Example:

person = ("Alice", "Doe", 1985)

### Use Cases for Sets

1. **Removing Duplicates**
   * Sets automatically eliminate duplicates, making them ideal for removing duplicate items from a list.
   * Use Case: Cleaning a dataset with redundant entries.
   * Example:

numbers = [1, 2, 2, 3, 4, 4, 5]

unique\_numbers = list(set(numbers))

print(unique\_numbers) # Output: [1, 2, 3, 4, 5]

1. **Membership Testing**
   * Sets are optimized for fast membership testing, which is efficient when checking if a value exists in a collection.
   * Use Case: Checking for invalid entries in a dataset against a list of allowed values.
   * Example:

allowed\_colors = {"red", "blue", "green"}

color = "yellow"

if color not in allowed\_colors:

print("Color not allowed.") # Output: Color not allowed.

1. **Mathematical Set Operations**
   * Sets support operations like union, intersection, difference, and symmetric difference, which can be useful in data analysis or filtering tasks.
   * Use Case: Finding common or unique items between two datasets.
   * Example:

students\_math = {"Alice", "Bob", "Charlie"}

students\_science = {"Bob", "David"}

# Students in both classes

both\_classes = students\_math & students\_science

print(both\_classes) # Output: {'Bob'}

1. **Filtering Data**
   * Sets are useful when filtering out unwanted elements from a collection.
   * Use Case: Removing forbidden words from a list of words.
   * Example:

words = {"apple", "banana", "grape", "orange"}

forbidden = {"grape", "orange"}

allowed\_words = words - forbidden

print(allowed\_words) # Output: {'apple', 'banana'}

1. **Maintaining Unique Collections**
   * Sets are excellent for managing collections where each element should be unique, such as tracking items in a shopping cart or unique user IDs.
   * Use Case: Storing unique tags, IDs, or categories.
   * Example:

user\_ids = {101, 102, 103}

user\_ids.add(104)

print(user\_ids) # Output: {101, 102, 103, 104}

1. **Comparing Large Datasets**
   * Sets allow efficient comparisons between large collections using operations like union and difference.
   * Use Case: Identifying shared or unique items between large datasets.
   * Example:

dataset\_a = set(range(1000))

dataset\_b = set(range(500, 1500))

# Find items only in dataset\_a

only\_in\_a = dataset\_a – dataset\_b

**Q7. Describe how to add, moidy and delete items in a dictionary with examples.**

Ans7 : In Python, dictionaries are mutable collections that store data in key-value pairs. Here’s how we can add, modify, and delete items in a dictionary with examples:

### 1. **Adding Items to a Dictionary**

We can add new key-value pairs to a dictionary by simply assigning a value to a new key or by using the update() method.

# Using assignment to add a new key-value pair

my\_dict = {"name": "Alice", "age": 25}

my\_dict["city"] = "New York"

print(my\_dict)

# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

# Using the update() method to add multiple items at once

my\_dict.update({"country": "USA", "occupation": "Engineer"})

print(my\_dict)

# Output: {'name': 'Alice', 'age': 25, 'city': 'New York', 'country': 'USA', 'occupation': 'Engineer'}

### 2. **Modifying Items in a Dictionary**

Modifying an item in a dictionary is as simple as reassigning a value to an existing key. We can also modify multiple items using update().

my\_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Modify a single item by assigning a new value to an existing key

my\_dict["age"] = 30

print(my\_dict)

# Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Modify multiple items with update()

my\_dict.update({"name": "Alicia", "city": "Los Angeles"})

print(my\_dict)

# Output: {'name': 'Alicia', 'age': 30, 'city': 'Los Angeles'}

### 3. **Deleting Items from a Dictionary**

There are several ways to delete items in a dictionary:

* **Using** del **statement**: Deletes a specific key-value pair by key.
* **Using** pop(): Removes a key-value pair by key and returns the value.
* **Using** popitem(): Removes the last inserted key-value pair and returns it as a tuple (useful for dictionaries where the insertion order matters).
* **Using** clear(): Removes all items from the dictionary, leaving it empty.

my\_dict = {"name": "Alice", "age": 25, "city": "New York", "occupation": "Engineer"}

# Deleting a specific item by key using del

del my\_dict["city"]

print(my\_dict)

# Output: {'name': 'Alice', 'age': 25, 'occupation': 'Engineer'}

# Deleting an item by key using pop() and capturing the removed value

removed\_value = my\_dict.pop("age")

print(removed\_value) # Output: 25

print(my\_dict)

# Output: {'name': 'Alice', 'occupation': 'Engineer'}

# Removing the last inserted item using popitem()

last\_item = my\_dict.popitem()

print(last\_item) # Output: ('occupation', 'Engineer')

print(my\_dict)

# Output: {'name': 'Alice'}

# Clearing all items in the dictionary

my\_dict.clear()

print(my\_dict) # Output: {}

**Q8. Discuss the importance of dictionary keys being immtable and provide examples.**

Ans8 : In Python, dictionary keys must be **immutable** (e.g., numbers, strings, tuples) because:

1. **Hashing Requirement**: Keys are hashed for fast lookup. If a key is mutable (e.g., a list), its hash could change, leading to errors in retrieving values.
2. **Data Integrity**: Immutability ensures the key remains constant, maintaining consistent mapping between keys and values.

### Example of Immutable Keys:

# Using a string as a key

my\_dict = {"name": "Alice"}

print(my\_dict["name"]) # Output: Alice

# Using a tuple as a key

my\_dict = {(1, 2): "point1"}

print(my\_dict[(1, 2)]) # Output: point1

### Example of Invalid Mutable Keys:

# Using a list as a key (raises an error)

my\_dict = {[1, 2]: "value"} # TypeError: unhashable type: 'list'

In summary, immutable keys ensure reliable hashing, quick lookups, and data consistency in dictionaries.