

**Name** – Gathram Kesava Murthy

**Registered mail id-** kesavagathram.murthy@outlook.com

**Course Name** – Master Generative AI: Data Science Course

**Assignment Name** - Python Data Structure Assignment

**Submission Date** – 25<sup>th</sup> October 2024

## 1. Discuss string slicing and provide examples

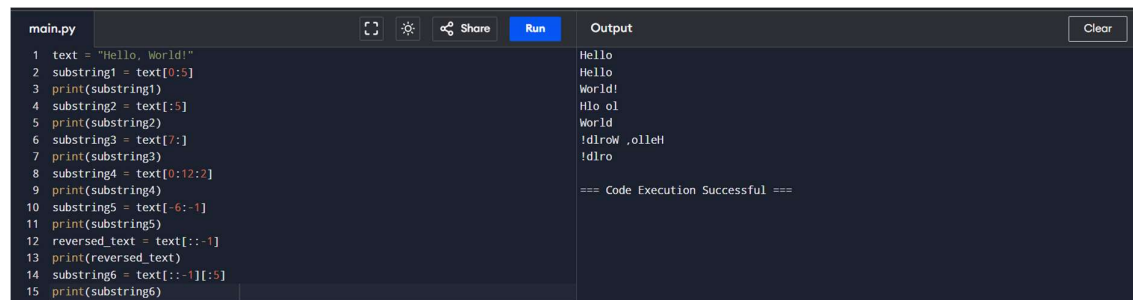
String slicing in Python is a powerful feature that allows you to extract a substring from a string. This is done using a specific syntax that involves indices and can also include steps. Understanding string slicing is essential for manipulating text data effectively.

### Basic Syntax of String Slicing:

`substring = string[start:end:step]`

- **start:** The starting index (inclusive).
- **end:** The ending index (exclusive).
- **step:** The interval between each index (default is 1).
- Python uses zero-based indexing, which means the first character of a string has an index of 0.
- If start is omitted, it defaults to the beginning of the string.
- If end is omitted, it defaults to the end of the string.
- If step is negative, it slices the string in reverse.

### Example of String Slicing:



```
main.py  [Icons]  Run  Output  Clear
1 text = "Hello, World!"
2 substring1 = text[0:5]
3 print(substring1)
4 substring2 = text[:5]
5 print(substring2)
6 substring3 = text[7:]
7 print(substring3)
8 substring4 = text[0:12:2]
9 print(substring4)
10 substring5 = text[-6:-1]
11 print(substring5)
12 reversed_text = text[::-1]
13 print(reversed_text)
14 substring6 = text[::-1][0:5]
15 print(substring6)
```

Output:

```
Hello
Hello
World!
Hlo ol
World
!dlroW ,olleH
!dlro

=== Code Execution Successful ===
```

### Explanation of the example:

**Basic Slicing:** `text[0:5]` extracts characters from index 0 to 4, which gives "Hello".

### Omitting Start and End:

- `text[:5]` starts from the beginning and goes up to index 4, returning "Hello".
- `text[7:]` starts from index 7 and returns the rest of the string, "World!".

**Using a Step:** `text[0:12:2]` takes every second character from index 0 to 11, resulting in "Hlo ol".

**Negative Indices:** `text[-6:-1]` counts from the end of the string, extracting characters from index -6 (which corresponds to W) to index -2 (exclusive), yielding "World".

**Reversing a String:** `text[::-1]` reverses the string by stepping backwards through the entire string, resulting in "!dlroW ,olleH".

**Combining Omitting and Negative Step:** `text[::-1][0:5]` reverses the string and then slices the first five characters, producing "!dlro".

- String slicing is a versatile feature that can be used for various string manipulations, including extraction, modification, and reversal. With a good understanding of indices and steps, you can efficiently handle strings.

## 2. Explain the key features of lists in Python

Lists are one of the most versatile and widely used data structures. They are ordered, mutable (modifiable), and allow duplicate elements. Here are the key features of lists, along with examples for each feature:

## 1. Ordered

- Lists maintain the order of elements, which means that items have a defined order, and that order will not change unless explicitly modified.

**Example:**

```
main.py [ ] [ ] [ ] Share Run Output Clear
1 fruits = ["apple", "banana", "cherry"]
2 print(fruits)
3 print(fruits[0])

['apple', 'banana', 'cherry']
apple

=== Code Execution Successful ===
```

## 2. Mutable

- Lists can be modified after their creation. You can add, remove, or change elements.

**Example:**

```
main.py  [Icons] [Share] [Run] Output [Clear]
1 numbers = [1, 2, 3]
2 numbers[i] = 20
3 print(numbers)
4 numbers.append(4)
5 print(numbers)
6 numbers.remove(20)
7 print(numbers)
```

[1, 20, 3]  
[1, 20, 3, 4]  
[1, 3, 4]

=== Code Execution Successful ===

### 3. Allow Duplicate Elements

- Lists can contain multiple instances of the same value.

**Example:**

```
main.py [ ] [ ] [ ] Share Run Output Clear
1 colors = ["red", "green", "blue", "red"]
2 print(colors)

['red', 'green', 'blue', 'red']

=== Code Execution Successful ===
```

## 4. Heterogeneous

- Lists can store elements of different data types, including other lists.

**Example:**

```
main.py [ ] [ ] [ ] Share Run Output Clear
1 mixed_list = [1, "two", 3.0, [4, 5]]
2 print(mixed_list)

[1, 'two', 3.0, [4, 5]]

=== Code Execution Successful ===
```

## 5. Nested Lists

- Lists can contain other lists as their elements, which allows for multi-dimensional data structures.

**Example:**

```
main.py 6
1 matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
2 print(matrix[1][2])

Output
6
--- Code Execution Successful ---
```

## 6. List Comprehensions

- Python allows for a concise way to create lists using list comprehensions.

Example:

main.py	Output
<pre>1 squares = [x**2 for x in range(5)] 2 print(squares)</pre>	<pre>[0, 1, 4, 9, 16] === Code Execution Successful ===</pre>

## 7. Built-in Methods

- Python provides various built-in methods to manipulate lists, such as `append()`, `remove()`, `pop()`, `sort()`, and `reverse()`.

Example:

main.py	Output
<pre>1 names = ["Alice", "Bob", "Charlie"] 2 names.append("David") 3 print(names) 4 names.sort() 5 print(names) 6 popped_name = names.pop() 7 print(popped_name) 8 print(names)</pre>	<pre>['Alice', 'Bob', 'Charlie', 'David'] ['Alice', 'Bob', 'Charlie', 'David'] David ['Alice', 'Bob', 'Charlie'] === Code Execution Successful ===</pre>

## 8. Indexing and Slicing

- You can access elements of a list using indexing and slicing techniques.

Example:

main.py	Output
<pre>1 letters = ['a', 'b', 'c', 'd', 'e'] 2 print(letters[1:4]) 3 print(letters[-1])</pre>	<pre>['b', 'c', 'd'] e === Code Execution Successful ===</pre>

```
main.py [ ] [ ] [ ] Share Run Output Clear
1 fruits = ["apple", "banana", "cherry", "date"]
2 del fruits[1]
3 print(fruits)

['apple', 'cherry', 'date']
=== Code Execution Successful ===
```

## 2. Using remove() Method

- The `remove()` method removes the first occurrence of a specified value. If the value is not found, it raises a `ValueError`.

**Example:**

```
main.py [ ] [ ] [ ] Share Run Output Clear  
1 fruits = ["apple", "banana", "cherry", "date"]  
2 fruits.remove("kiwi")  
3 print(fruits)  
  
ERROR!  
Traceback (most recent call last):  
  File "<main.py>", line 2, in <module>  
ValueError: list.remove(x): x not in list  
  
=== Code Exited With Errors ===
```

### 3. Using pop() Method

- The `pop()` method removes and returns an element at a specified index. If no index is specified, it removes and returns the last element.

**Example:**

```
main.py  [Icons] [Share] [Run] Output [Clear]

1 fruits = ["apple", "banana", "cherry", "date"]
2 last_fruit = fruits.pop()
3 print(last_fruit)
4 print(fruits)
5 second_fruit = fruits.pop(1)
6 print(second_fruit)
7 print(fruits)
```

date  
['apple', 'banana', 'cherry']  
banana  
['apple', 'cherry']  
=== Code Execution Successful ===

## 4. Compare and contrast tuples and lists with examples.

Tuples and lists are both data structures that can be used to store collections of items. However, they have several key differences that affect how they are used in programming. Below is a detailed comparison and contrast of tuples and lists, along with examples.

### 1. Mutability

- **Lists:** Lists are mutable, which means you can change their content (add, remove, or modify elements) after they have been created.
- **Tuples:** Tuples are immutable, meaning once they are created, their content cannot be changed.

Example:

<pre>main.py 1 my_list = [1, 2, 3] 2 my_list[1] = 5 3 my_list.append(4) 4 print(my_list) 5 my_tuple = (1, 2, 3) 6 print(my_tuple)</pre>	<pre>Output [1, 5, 3, 4] (1, 2, 3) === Code Execution Successful ===</pre>
---	--

### 2. Syntax

- **Lists:** Lists are defined using square brackets [].
- **Tuples:** Tuples are defined using parentheses ().

Example:

<pre>main.py 1 my_list = [1, 2, 3] 2 my_tuple = (1, 2, 3)</pre>	<pre>Output === Code Execution Successful ===</pre>
---	---

### 3. Performance

- **Lists:** Because lists are mutable, they incur some overhead in terms of memory and performance. Operations that modify the list can be slower than equivalent operations on tuples.
- **Tuples:** Tuples are generally faster than lists for iteration and can be used as dictionary keys because they are hashable, unlike lists.

Example:

<pre>main.py 1 import time 2 list_test = list(range(10000)) 3 tuple_test = tuple(range(10000)) 4 start_time = time.time() 5 for item in list_test: 6     pass 7 print("List iteration time:", time.time() - start_time) 8 start_time = time.time() 9 for item in tuple_test: 10     pass 11 print("Tuple iteration time:", time.time() - start_time)</pre>	<pre>Output List iteration time: 0.0002377033233642578 Tuple iteration time: 0.0002002716064453125 === Code Execution Successful ===</pre>
--	--

### 4. Use Cases

- **Lists:** Use lists when you need a collection of items that may need to be modified, such as adding or removing elements. They are suitable for dynamic data.
- **Tuples:** Use tuples when you want to store a collection of items that should not change. They are ideal for fixed collections of items and can be used as keys in dictionaries.

### Example:

main.py	Output
<pre>1 shopping_list = ["eggs", "milk", "bread"] 2 shopping_list.append("butter") 3 print(shopping_list) 4 coordinates = (10.0, 20.0) 5 print(coordinates)</pre>	<pre>['eggs', 'milk', 'bread', 'butter'] (10.0, 20.0) === Code Execution Successful ===</pre>

## 5. Methods

- **Lists:** Lists have many built-in methods for modification, such as `append()`, `remove()`, `pop()`, `sort()`, etc.
- **Tuples:** Tuples have very few built-in methods, primarily `count()` and `index()`, because they do not support modification.

### Example:

main.py	Output
<pre>1 my_list = [1, 2, 3] 2 my_list.append(4) 3 my_list.remove(2) 4 print(my_list) 5 my_tuple = (1, 2, 3, 2) 6 print(my_tuple.count(2)) 7 print(my_tuple.index(3))</pre>	<pre>[1, 3, 4] 2 2 === Code Execution Successful ===</pre>



## 5. Describe the key features of sets and provide examples of their use.

Sets are a built-in data structure in Python that represent an unordered collection of unique elements. They are particularly useful for operations involving membership testing, deduplication, and mathematical set operations like union and intersection. Below are the key features of sets, along with examples of their use.

### Key Features of Sets

#### 1. Unordered Collection:

- Sets do not maintain the order of elements. This means that the elements in a set are not indexed, and their position may change.

##### Example:

main.py	Run	Output
<pre>1 my_set = {1, 2, 3, 4} 2 print(my_set)</pre>		<pre>{1, 2, 3, 4} === Code Execution Successful ===</pre>

#### 2. Unique Elements:

- Sets automatically eliminate duplicate entries. If you try to add duplicate elements, they will not be included in the set.

##### Example:

main.py	Run	Output
<pre>1 my_set = {1, 2, 2, 3, 4, 4} 2 print(my_set)</pre>		<pre>{1, 2, 3, 4} === Code Execution Successful ===</pre>

#### 3. Mutable:

- Sets are mutable, meaning you can add or remove elements after the set has been created.

##### Example:

main.py	Run	Output
<pre>1 my_set = {1, 2, 3} 2 my_set.add(4) 3 print(my_set) 4 my_set.remove(2) 5 print(my_set)</pre>		<pre>{1, 2, 3, 4} {1, 3, 4} === Code Execution Successful ===</pre>

#### 4. No Indexing:

- Since sets are unordered, you cannot access elements by their position or index. Instead, you can only check for the existence of an element.

##### Example:

main.py	Run	Output
<pre>1 my_set = {1, 2, 3} 2 print(1 in my_set) 3 print(4 in my_set) 4</pre>		<pre>True False === Code Execution Successful ===</pre>

### 5. Set Operations

- Sets support mathematical set operations, such as union, intersection, difference, and symmetric difference.

## Example:

main.py	Output
<pre>1 set_a = {1, 2, 3} 2 set_b = {3, 4, 5} 3 union_set = set_a   set_b 4 print(union_set) 5 intersection_set = set_a &amp; set_b 6 print(intersection_set) 7 difference_set = set_a - set_b 8 print(difference_set) 9 symmetric_difference_set = set_a ^ set_b 10 print(symmetric_difference_set)</pre>	<pre>{1, 2, 3, 4, 5} {3} {1, 2} {1, 2, 4, 5}  === Code Execution Successful ===</pre>

## 6. Comprehension:

- Like lists and dictionaries, sets can also be created using set comprehensions.

## Example:

main.py	Output
<pre>1 squares = {x**2 for x in range(5)} 2 print(squares)</pre>	<pre>{0, 1, 4, 9, 16}  === Code Execution Successful ===</pre>

## 7. Built-in Methods:

- Sets come with various built-in methods for common tasks, such as `add()`, `remove()`, `discard()`, `clear()`, `pop()`, and more.

## Example:

main.py	Output
<pre>1 my_set = {1, 2, 3} 2 my_set.add(4) 3 print(my_set) 4 my_set.discard(2) 5 print(my_set) 6 popped_element = my_set.pop() 7 print(popped_element) 8 print(my_set)</pre>	<pre>{1, 2, 3, 4} {1, 3, 4} 1 {3, 4}  === Code Execution Successful ===</pre>

## 6. Discuss the use cases of tuples and sets in Python programming.

Tuples and sets are versatile data structures, each with unique characteristics that make them suitable for different use cases. Here is a detailed discussion of the use cases for both tuples and sets in Python programming.

### Use Cases of Tuples:

#### 1. Fixed Collections of Items:

Tuples are ideal for storing a fixed collection of items that should not change. This is particularly useful when you want to ensure that the data remains constant throughout the program.

##### Example:

```
main.py  [Icons]  Run  Output  Clear
1 color_rgb = (255, 0, 0)

=== Code Execution Successful ===
```

#### 2. Returning Multiple Values from Functions:

Tuples can be used to return multiple values from functions. This allows for a clean and efficient way to bundle several outputs together.

##### Example:

```
main.py  [Icons]  Run  Output  Clear
1 def get_coordinates():
2     return (10, 20)
3 x, y = get_coordinates()
4 print(x, y)

10 20
=== Code Execution Successful ===
```

#### 3. Using as Dictionary Keys:

Because tuples are immutable, they can be used as keys in dictionaries. This is not possible with lists, which are mutable.

##### Example:

```
main.py  [Icons]  Run  Output  Clear
1 location = {}
2 location[(10, 20)] = "Home"
3 location[(30, 40)] = "Office"
4 print(location)

{(10, 20): 'Home', (30, 40): 'Office'}
=== Code Execution Successful ===
```

#### 4. Data Integrity:

Tuples provide a way to ensure data integrity, as their immutable nature prevents accidental modification of data.

##### Example:

```
main.py  [Icons]  Run  Output  Clear
1 DATABASE_CONFIG = ("localhost", "user", "password")

=== Code Execution Successful ===
```

#### 5. Packing and Unpacking:

Tuples facilitate packing and unpacking of data. This feature is often used in loop iterations, such as with enumerations.

##### Example:

```
main.py [ ] [ ] [ ] Share Run Output Clear
1 points = [(1, 2), (3, 4), (5, 6)]
2 for x, y in points:
3     print(f"Point: {x}, {y}")

Point: 1, 2
Point: 3, 4
Point: 5, 6

=== Code Execution Successful ===
```

## Use Cases of Sets

### 1. Removing Duplicates:

Sets are excellent for removing duplicate entries from collections, such as lists.

**Example:**

```
main.py 1 items = [1, 2, 2, 3, 4, 4] 2 unique_items = set(items) == Code Execution Successful ==
```

## 2. Membership Testing:

Sets provide  $O(1)$  average-time complexity for membership testing, making them faster than lists for checking if an item exists.

**Example:**

```
main.py [ ] [ ] [ ] Share Run Output Clear
1 my_set = {1, 2, 3, 4}
2 print(3 in my_set)
True
=== Code Execution Successful ===
```

### 3. Membership Testing:

Sets provide  $O(1)$  average-time complexity for membership testing, making them faster than lists for checking if an item exists.

**Example:**

```
main.py  Run  Output  Clear
1 set_a = {1, 2, 3}
2 set_b = {3, 4, 5}
3 print(set_a | set_b)
4 print(set_a & set_b)
5 print(set_a - set_b)

{1, 2, 3, 4, 5}
{3}
{1, 2}

=== Code Execution Successful ===
```

#### 4. Data Analysis:

In data analysis, sets can be useful for tasks such as filtering unique items or performing set-based calculations.

**Example:**

```
main.py [ ] [ ] [ ] Share Run Output Clear
1 orders = ["apple", "banana", "apple", "orange", "banana"]
2 unique_orders = set(orders)

=== Code Execution Successful ===
```

## 5. Membership Testing:

Sets can be used to implement logic that requires understanding of relationships between groups, such as finding common elements or exclusive elements between datasets.

**Example:**

```
main.py  [ ]  [ ]  [ ]  Share  Run  Output  Clear

1 enrolled_students = {"Alice", "Bob", "Charlie"}
2 passed_students = {"Bob", "David"}
3 passing_enrolled = enrolled_students & passed_students

=== Code Execution Successful ===
```

## 7. Describe how to add, modify, and delete items in a dictionary with examples.

Dictionaries in Python are versatile data structures that allow you to store key-value pairs. They are mutable, meaning you can add, modify, and delete items after the dictionary has been created.

### 1. Adding Items to a Dictionary

You can add a new key-value pair to a dictionary by using either the assignment operator or the `update()` method.

#### Example Using the Assignment Operator:

<pre>main.py 1 my_dict = {} 2 my_dict['name'] = 'Alice' 3 my_dict['age'] = 30 4 print(my_dict)</pre>	<pre>Output {'name': 'Alice', 'age': 30} === Code Execution Successful ===</pre>
--	--

#### Using the `update()` Method

The `update()` method can be used to add multiple key-value pairs at once.

<pre>main.py 1 my_dict = {} 2 my_dict.update({'city': 'New York', 'country': 'USA'}) 3 print(my_dict)</pre>	<pre>Output {'city': 'New York', 'country': 'USA'} === Code Execution Successful ===</pre>
---	--

### 2. Modifying Items in a Dictionary

You can modify the value of an existing key by assigning a new value to that key.

#### Example:

<pre>main.py 1 my_dict = {} 2 my_dict['name'] = 'Alice' 3 my_dict['age'] = 30 4 my_dict.update({'city': 'New York', 'country': 'USA'}) 5 print(my_dict) 6 my_dict['age'] = 31 7 print(my_dict)</pre>	<pre>Output {'name': 'Alice', 'age': 30, 'city': 'New York', 'country': 'USA'} {'name': 'Alice', 'age': 31, 'city': 'New York', 'country': 'USA'} === Code Execution Successful ===</pre>
--	---

### 3. Deleting Items from a Dictionary

You can delete items from a dictionary using the `del` statement, the `pop()` method, or the `popitem()` method.

#### Using the `del` Statement

The `del` statement removes a specified key-value pair from the dictionary



#### Example:

<pre>main.py 1 my_dict = {} 2 my_dict['name'] = 'Alice' 3 my_dict['age'] = 30 4 my_dict.update({'city': 'New York', 'country': 'USA'}) 5 print(my_dict) 6 del my_dict['country'] 7 print(my_dict)</pre>	<pre>Output {'name': 'Alice', 'age': 30, 'city': 'New York', 'country': 'USA'} {'name': 'Alice', 'age': 30, 'city': 'New York'} === Code Execution Successful ===</pre>
---	---

#### Using the `pop()` Method

The `pop()` method removes the specified key and returns its value. If the key does not exist, it raises a `KeyError` unless a default value is provided.

## Example:

main.py	   Share	Run	Output	 Clear
<pre>1 my_dict = {} 2 my_dict['name'] = 'Alice' 3 my_dict['age'] = 30 4 my_dict.update({'city': 'New York', 'country': 'USA'}) 5 print(my_dict) 6 age = my_dict.pop('age') 7 print(age) 8 print(my_dict)</pre>			<pre>{'name': 'Alice', 'age': 30, 'city': 'New York', 'country': 'USA'} 30 {'name': 'Alice', 'city': 'New York', 'country': 'USA'}  === Code Execution Successful ===</pre>	

### 8. Discuss the importance of dictionary keys being immutable and provide example

In Python, dictionary keys must be immutable data types. This requirement is crucial for several reasons, primarily concerning the behavior and functionality of dictionaries.

## Importance of Immutable Keys in Dictionaries

## 1. Uniqueness and Hashing:

- Dictionary keys must be unique, and Python uses hashing to ensure that keys can be quickly accessed. An immutable object can generate a consistent hash value, which is essential for the dictionary's internal structure. If keys could change, their hash value might also change, leading to inconsistencies and unpredictable behavior.

**Example:**

```
main.py [ ] [ ] [ ] Share Run Output Clear  
1 my_dict = {}  
2 my_dict[[1, 2]] = "Point A"  
3 print(my_dict)  
4 - try:  
5     my_dict[[1, 2]] = "Point B"  
6 except TypeError as e:  
7     print(e)  
  
{(1, 2): 'Point A'}  
unhashable type: 'list'  
  
=== Code Execution Successful ===
```

## 2. Consistency:

- Since dictionary keys must be immutable, it ensures that once a key is set, it cannot be modified. This guarantees that lookups will always return the expected value, as the key will remain constant throughout the lifetime of the dictionary.

**Example:**

```
main.py  [Run] [Share] [Output] [Clear]
1 my_dict = {'1': "One", 2: "Two"}
2 print(my_dict[1])

One

=== Code Execution Successful ===
```

### 3. Performance:

- Immutable keys contribute to the performance of dictionary operations. The hashing mechanism relies on immutable types, allowing for faster lookups, insertions, and deletions.

**Example:**

```
main.py  [ ] [ ] [ ] Share Run Output Clear
1 keys = (1, 2, 3)
2 my_dict = {keys: "Tuple as key"}
3 print(my_dict)

{(1, 2, 3): 'Tuple as key'}

=== Code Execution Successful ===
```

- If you attempt to use a mutable type, like a list or a dictionary, as a key, Python raises a `TypeError`. This behavior prevents any potential issues that could arise from changing the keys while they are being used to access values in the dictionary.

**Example:**

```
main.py  [Icons]  Run  Output  Clear
1: try:
2:     mutable_key_dict = {[1, 2]: "This won't work"}
3: except TypeError as e:
4:     print(e)
```

unhashable type: 'list'

=== Code Execution Successful ===