Day 11- Trainer FG
(Tuples)

Index

For Trainer(1.50 hrs) ILT

I. Recap

In our last session we learned:

- **What is a List?:**In Python, a list is a built-in data structure that represents an ordered collection of items. Lists are one of the most commonly used data structures in Python and are very versatile.
- **Traversing a list:**Traversing a list in Python means iterating through all the elements of the list, one by one, to perform some operation or gather information from each element.
- **List Operations and Slices:**In Python, lists are versatile data structures that allow you to store and manipulate collections of items.
- **List Functions:**In Python, lists are versatile data structures, and there are many built-in functions that you can use to manipulate and work with lists effectively.

Try this: Predict the output

```
# Python program to print Even Numbers in a List

# list of numbers
list1 = [10, 21, 4, 45, 66, 93]

# iterating each number in list
for num in list1:

        # checking condition
        if num % 2 == 0:
                print(num, end=" ")
```

In this session we are going to understand tuples, features of tuples, creating tuples, accessing values in tuples, various operations of tuples, tuple functions, indexing and slicing in tuples.

II. What is a Tuple?

A tuple is an ordered and immutable collection or sequence of elements in Python. Immutable means that once a tuple is created, its elements cannot be modified, added, or removed. Tuples are typically enclosed in parentheses (), although the parentheses are optional in many cases. Elements within a tuple are separated by commas.

Features of Tuples

Tuples in Python have several features that make them a useful data structure in various programming scenarios:

- **Ordered:** Tuples are ordered collections, which means that the order in which elements are added to the tuple is preserved. You can access elements by their positions (indices).

- **Immutable:** Tuples are immutable, meaning their elements cannot be modified, added, or removed after the tuple is created. This immutability makes tuples suitable for representing data that should not change.

- **Heterogeneous:** Tuples can contain elements of different data types. You can mix integers, floats, strings, and other data types within a single tuple.

- **Indexing:** You can access elements within a tuple using zero-based indexing, just like in lists. For example, my_tuple[0] accesses the first element of the tuple.

- **Iteration:** Tuples can be iterated using loops, making it easy to process each element in the tuple sequentially.

- **Packing and Unpacking:** You can create a tuple by packing values together, and you can also unpack a tuple into individual variables. For example, x, y = (3, 4) packs the values into a tuple and then unpacks them into x and y variables.

- **Multiple Data Types:** Tuples can hold elements of different data types in a single tuple. This flexibility allows you to create tuples that represent complex data structures.

- **Comparatively Faster:** Accessing elements in a tuple is generally faster than in a list because tuples are immutable, and their elements are stored in a more memory-efficient way.

- **Used as Dictionary Keys:** Tuples are often used as keys in dictionaries because they are immutable and can represent pairs or combinations of values.

- **Function Return Values:** Tuples are commonly used in functions to return multiple values. A function can return a tuple of results, which can be unpacked by the caller.

- **Memory Efficient:** Tuples consume less memory compared to lists because they don't require space for dynamic resizing.

III. Creating Tuples

Tuples in Python can be created in several ways:

Using Parentheses: The most common way to create a tuple is by enclosing comma-separated values in parentheses. Here's an example:

```
my_tuple = (1, 2, 3)
```

Using the tuple() Constructor: You can use the tuple() constructor to create a tuple from an iterable, such as a list or a string:

```
my_list = [4, 5, 6]
tuple_from_list = tuple(my_list)

my_string = "abc"
tuple_from_string = tuple(my_string)
```

Creating a Singleton Tuple: To create a tuple with a single element, you need to include a comma after the element, even if you don't use parentheses:

```
single_element_tuple = (42,)  # A tuple with one element
```

Empty Tuple: You can create an empty tuple by using empty parentheses:

```
empty_tuple = ()
```

Here are examples of each of these methods:

```
# Using parentheses
tuple1 = (1, 2, 3)

# Using the tuple() constructor
list1 = [4, 5, 6]
tuple2 = tuple(list1)

string1 = "abc"
tuple3 = tuple(string1)

# Creating a singleton tuple
singleton_tuple = (42,)

# Creating an empty tuple
empty_tuple = ()

print(tuple1)
print(tuple2)
print(tuple3)
print(singleton_tuple)
print(empty_tuple)
```

All of these methods create tuples with different values, and once created, the elements within a tuple cannot be modified.

IV. Accessing Values in Tuples

You can access values in tuples by using indexing, slicing, or iterating through the tuple. Here's how you can access values in a tuple:

Indexing: You can access individual elements of a tuple by their position (index) within the tuple. Indexing in Python is zero-based, so the first element is at index 0, the second element is at index 1, and so on.

```
my_tuple = (10, 20, 30, 40, 50)

# Accessing individual elements
first_element = my_tuple[0]  # Retrieves the first element (10)
second_element = my_tuple[1]  # Retrieves the second element (20)
```

Negative Indexing: You can also use negative indexing to access elements from the end of the tuple. -1 represents the last element, -2 the second-to-last, and so on.

```
last_element = my_tuple[-1]  # Retrieves the last element (50)
second_last_element = my_tuple[-2]  # Retrieves the second-to-last element (40)
```

Slicing: Slicing allows you to access a range of elements from the tuple. You specify the start and end indices (exclusive) separated by a colon :.

```
my_tuple = (10, 20, 30, 40, 50)

# Slicing to get a subset of elements
subset = my_tuple[1:4]  # Retrieves elements from index 1 to 3 (20, 30, 40)
```

Iterating: You can iterate through the elements of a tuple using a for loop or other iteration methods.

```
my_tuple = (10, 20, 30, 40, 50)

# Iterating through elements
for item in my_tuple:
    print(item)
```

Remember that tuples are immutable, so you cannot modify their elements directly. If you need to modify a tuple, you would need to create a new tuple with the desired changes.

Here's an example that demonstrates some of the features of tuples:

```
# Creating a tuple
my_tuple = (1, "Hello", 3.14)

# Accessing elements
print("Accessing value at index 0")
print(my_tuple[0])  # Output: 1

# Iterating through a tuple
print("Accessing all the values")
for item in my_tuple:
    print(item)


# Immutable nature
# This would result in an error:
print(“Attempt to modify value at index zero”)
my_tuple[0] = 42
```

Output:

```
Accessing value at index 0
1
Accessing all the values
1
Hello
3.14
Attempt to modify value at index zero
Traceback (most recent call last):
  File "C:\Users\Dipankar\Desktop\code\sd.py", line 17, in <module>
    my_tuple[0] = 42
TypeError: 'tuple' object does not support item assignment
```

V. Various Operations of Tuples

Tuples in Python support various operations and methods that allow you to work with them effectively. Here are some common operations and methods associated with tuples:

Concatenation: You can concatenate two or more tuples using the + operator to create a new tuple that contains elements from all the input tuples.

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
result_tuple = tuple1 + tuple2  # (1, 2, 3, 4, 5, 6)
```

Repetition: You can repeat a tuple by using the * operator. This creates a new tuple with repeated elements.

```
tuple1 = (1, 2)
repeated_tuple = tuple1 * 3  # (1, 2, 1, 2, 1, 2)
```

Membership Test: You can use the in and not in operators to check if an element exists in a tuple.

```
my_tuple = (1, 2, 3)
result1 = 2 in my_tuple  # True
result2 = 4 not in my_tuple  # True
```

Length: You can determine the number of elements in a tuple using the len() function.

```
my_tuple = (10, 20, 30, 40)
length = len(my_tuple)  # 4
```

Minimum and Maximum: You can find the minimum and maximum values in a tuple using the min() and max() functions.

```
my_tuple = (10, 5, 30, 15)
minimum = min(my_tuple)  # 5
maximum = max(my_tuple)  # 30
```

Count: You can count the occurrences of a specific element in a tuple using the count() method.

```
my_tuple = (1, 2, 2, 3, 2)
count = my_tuple.count(2)  # 3 (number of times 2 appears)
```

Index: You can find the index (position) of the first occurrence of a specific element in a tuple using the index() method.

```
my_tuple = (10, 20, 30, 20, 40)
index = my_tuple.index(20)  # 1 (index of the first occurrence of 20)
```

Sorting: You can create a sorted version of a tuple using the sorted() function, which returns a new sorted list.

```
my_tuple = (4, 1, 3, 2)
sorted_tuple = tuple(sorted(my_tuple))  # (1, 2, 3, 4)
```

Slicing: You can create subsets of a tuple using slicing, as mentioned earlier.\

```
my_tuple = (1, 2, 3, 4, 5)
subset = my_tuple[1:4]  # (2, 3, 4)
```

Unpacking: You can unpack a tuple into separate variables for further processing.

```
my_tuple = (10, 20, 30)
a, b, c = my_tuple  # a=10, b=20, c=30
```

These operations and methods make tuples a versatile and powerful data structure in Python for handling ordered and immutable collections of data.

VI. Tuple Functions

Tuples in Python don't have many built-in methods since they are immutable, but they support a few useful functions and methods for various operations. Here are some common functions and methods associated with tuples:

len(): The len() function is used to find the number of elements in a tuple.

```
my_tuple = (1, 2, 3, 4)
length = len(my_tuple)  # 4
```

max() and min(): The max() function returns the maximum value in a tuple, and the min() function returns the minimum value.

```
my_tuple = (10, 5, 30, 15)
maximum = max(my_tuple)  # 30
minimum = min(my_tuple)  # 5
```

sum(): The sum() function calculates the sum of all elements in a tuple.

```
my_tuple = (1, 2, 3, 4, 5)
total = sum(my_tuple)  # 15
```

sorted(): The sorted() function returns a sorted list from the elements of a tuple.

```
my_tuple = (4, 1, 3, 2)
sorted_list = sorted(my_tuple)  # [1, 2, 3, 4]
```

count(): The count() method is used to count the number of occurrences of a specific element in a tuple.

```
my_tuple = (1, 2, 2, 3, 2)
count = my_tuple.count(2)  # 3 (number of times 2 appears)
```

index(): The index() method is used to find the index (position) of the first occurrence of a specific element in a tuple.

```
my_tuple = (10, 20, 30, 20, 40)
index = my_tuple.index(20)  # 1 (index of the first occurrence of 20)
```

These functions and methods are particularly useful for performing various operations on tuples, such as finding statistics (e.g., minimum, maximum, and sum), searching for specific elements, and sorting elements when needed.

VII. Indexing and Slicing in Tuple

In Python, you can access elements in a tuple using indexing and slicing, just like you would with lists and other sequences. Here's how indexing and slicing work with tuples.

Indexing: Indexing allows you to access individual elements within a tuple.
The indexing is zero-based, meaning that the first element has an index of 0, the second has an index of 1, and so on.
Here's an example of indexing a tuple:

```
my_tuple = (10, 20, 30, 40, 50)

first_element = my_tuple[0]   # Access the first element (10)
second_element = my_tuple[1]  # Access the second element (20)
third_element = my_tuple[2]   # Access the third element (30)
```

Negative Indexing: You can also use negative indices to access elements from the end of the tuple.
-1 represents the last element, -2 represents the second-to-last element, and so on.

Here's an example of negative indexing:

```
my_tuple = (10, 20, 30, 40, 50)

last_element = my_tuple[-1]      # Access the last element (50)
second_last_element = my_tuple[-2]  # Access the second-to-last element (40)
```

Slicing: Slicing allows you to create a subset of a tuple by specifying a range of indices. The syntax for slicing is start:stop:step, where start is the starting index (inclusive), stop is the stopping index (exclusive), and step is the step size (optional).
Here's an example of slicing a tuple:

```
my_tuple = (10, 20, 30, 40, 50)

subset1 = my_tuple[1:4]    # Creates a subset from index 1 to 3 (20, 30, 40)
subset2 = my_tuple[::2]    # Creates a subset with a step of 2 (10, 30, 50)
```

Slicing can be particularly useful when you want to extract a portion of a tuple or create a new tuple with specific elements.

Keep in mind that tuples are immutable, so you can't change the values of individual elements using indexing or slicing. If you need to modify a tuple, you'll need to create a new tuple with the desired values.

Exercise:

Use GPT to write a program:

1.Hi! Can you create a program using tuple that manages student records, including student names, exam scores, and final grades.

After the ILT, the student has to do a 30 min live lab. Please refer to the Day 11 Lab doc in the LMS.