MODULE 4

INTERMEDIATE ELEMENTS OF PYTHON 1

MODULE OVERVIEW

In this module, we'll explore Python dictionaries, lists, and tuples, which are a powerful and fundamental data structure that allows you to store and organize data or items. Dictionaries are widely used in Python and are an essential part of the language. Lists are versatile and widely used in Python for various purposes. Tuples are used to store collections of items, just like lists, but they are immutable, meaning their elements cannot be changed after they are created.

MODULE LEARNING OBJECTIVES

By the end of this module, you should be able to:

- Understand the differences and similarities between dictionaries, lists, and tuples.
- Create and manipulate dictionaries, lists, and tuples.
- Utilize dictionaries, lists, and tuples for data storage and retrieval.
- Use built-in methods and functions for dictionaries, lists, and tuples.

LEARNING CONTENTS

4.1 Dictionaries

A dictionary in Python is defined using curly braces {}. In Python 3.6 and earlier, dictionaries are unordered collections, meaning the elements are not indexed by numerical positions but in 3.7, they are ordered. When You refer to dictionaries as being ordered, we mean that the contents have a set order that won't change. You can change, add, or remove items after the dictionary has been created.

4.1.1 Creating a Dictionary

You can create a dictionary using the curly braces {} and defining key-value pairs. Keys must be unique, and they are usually strings or numbers. Values can be of any data type, including numbers, strings, lists, or even nested dictionaries. Each element in a dictionary consists of a key and its associated value, separated by a colon.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
```

4.1.2 Dictionary Length

You can determine the length of a dictionary by using a len() function.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
result = len(person)
print(result)
```

Output:

```
2
```

4.1.3 Accessing Dictionary

You can access items of a dictionary by referring to its key name by putting it inside square brackets. There are also several ways to access items by using built-in methods.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
print(person[“name”])
```

Output:

```
John
```

a. Accessing Dictionary using get() method - returns the value of the specified key.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
result = person.get("name")
print(result)
```

Output:

```
John
```

b. Accessing Dictionary using items() method - returns a list containing a tuple for each key value pair.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
result = person.items()
print(result)
```

Output:

```
dict_items([('name' , 'John'), ('age', 30)]
```

c. Accessing Dictionary using keys() method - returns a list containing the dictionary's keys.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
result = person.keys ()
print(result)
```

Output:

```
dict_keys(['name', 'age']
```

d. Accessing Dictionary using values() method - returns a list of all the values in the dictionary.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
result = person.values ()
print(result)
```

Output:

```
dict_keys(['John', 30]
```

4.1.4 Modifying Dictionary

You can modify the value of an existing key or add new key-value pairs to the dictionary using assignment by referring to its key name. In dictionary, you can add, remove, and update items.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
person["age"] = 31
print(result.values)
```

Output:

```
dict_keys(['John', 31]
```

a) Adding a new item – There are ways to add items to a dictionary. First is using assignment, if you are referring a non-existing key, what it does is it will be inserted on the last. The second one is using the update() method.

a.1 Using assignment

Example:

```
person = {
   "name": "John",
   "age": 30,
}
person["sex"] = "Male"
print(person)
```

Output:

```
{'name': 'John', 'age': 30, 'sex': 'Male'}
```

a.2 Using update() method - this only works if the specified key does not exist in the dictionary otherwise, it updates the dictionary with the items from the given argument.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
person.update({"sex": “Male})
print(person)
```

Output:

```
{'name': 'John', 'age': 30, 'sex': 'Male'}
```

```
person = {
   "name": "John",
   "age": 30,
}
person.update({"age": 31})
print(person)
```

```
{'name': 'John', 'age': 31}
```

b) **Changing dictionary items –** There are ways to change items to a dictionary. First is using assignment, refer to its key name and assign the new value to it. The second one is also using the update() method.

b.1 Using assignment

Example:

```
person = {
   "name": "John",
   "age": 30,
}
person["age"] = "31"
print(person)
```

Output:

```
{'name': 'John', 'age': 31}
```

b.2 Using update() method - this updates the dictionary with the items from the given argument.

Example:

```
person = {
   "name": "John",
   "age": 30,
}
person.update({"age": 31})
print(person)
```

Output:

```
{'name': 'John', 'age': 31}
```

c) Removing dictionary items

There are several methods to remove items from a dictionary. The first one is using the pop() method wherein it removes the item with the specified key name. Next is the popitem(), what it does is that it removes the last inserted item. Python also has the del keyword where it can remove item with the specified key name or delete the entire dictionary. Lastly, it has the clear() method where it removes all the items of the dictionary.

c.1. pop()

Example:

```
person = {
   "name": "John",
   "age": 30,
}
person.pop("age")
print(person)
```

Output:

```
{'name': 'John'}
```

c.2. popitem()

Example:

```
person = {
   "name": "John",
   "age": 30,
   "sex": "Male”
}
person.popitem()
print(person)
```

Output:

```
{'name': 'John', 'age': 30}
```

c.3. del

Example 1:

```
person = {
   "name": "John",
   "age": 30,
   "sex": "Male”
}
del person["sex"]
print(person)
```

Output:

```
{'name': 'John', 'age': 30}
```

Example 2:

```
person = {
   "name": "John",
   "age": 30,
   "sex": "Male”
}
del person
print(person)
```

Output:

```
# This will cause an error because
"person" no longer exists.
```

c.4. clear() method

Example:

```
person = {
   "name": "John",
   "age": 30,
   "sex": "Male”
}
person.clear()
print(person)
```

Output:

```
{}
```

4.1.5 Looping a Dictionary

To loop through a dictionary in Python, you can use various methods to access its keys, values, or key-value pairs. Here are three common approaches for looping through a dictionary:

1. **Looping through Keys**
   You can use the keys() method to get an iterable view of the dictionary's keys and then iterate through them using a for loop.

Example:

```
person = {
   "name": "John",
   "age": 30,
   "sex": "Male"
}
for key in person.keys():
   print(key)
```

Output:

```
name
age
sex
```

2. **Looping through Values**
   You can use the values() method to get an iterable view of the dictionary's values and then iterate through them using a for loop.

Example:

```
person = {
   "name": "John",
   "age": 30,
   "sex": "Male"
}
for val in person.values():
   print(val)
```

Output:

```
name
age
sex
```

3. **Looping through Key-Value Pair**

   You can use the items() method to get an iterable view of the dictionary's key-value pairs (as tuples) and then iterate through them using a for loop.

Example:

```
person = {
   "name": "John",
   "age": 30,
   "sex": "Male"
}
for key, value in person.items():
   print(key, ":", value)
```

Output:

```
name : John
age : 30
sex : Male
```

Each approach offers a different way to access and loop through the elements of the dictionary. Depending on your specific use case, you can choose the method that best suits your needs.

4.1.6 Copying Dictionary

In Python, copy() and dict() are two different methods used to create copies of dictionaries. Let's explore each method:

1. **Using copy() method:**

The copy() method is a built-in method available for dictionaries. It creates a copy of the dictionary, meaning it creates a new dictionary object, but it only copies the references to the original dictionary's elements. The new dictionary shares the same objects as the original dictionary.

Here's the syntax:

```
new_dict = original_dict.copy()
```

Example:

```
original_dict = {"a": 1, "b": 2, "c": 3}
shallow_copy_dict = original_dict.copy()

shallow_copy_dict["a"] = 10

print(original_dict)
print(shallow_copy_dict)
```

Output:

```
{'a': 1, 'b': 2, 'c': 3}
{'a': 10, 'b': 2, 'c': 3}
```

As you can see, changing the value of the key "a" in the copy does not affect the original dictionary.

2. **Using dict() constructor:**

The dict() constructor is used to create a new dictionary object from an existing dictionary or other iterable objects. When using a dictionary as the argument to dict(), it creates a copy of the dictionary.

Here's the syntax:

```
new_dict = dict(original_dict)
```

Example:

```
original_dict = {"a": 1, "b": 2, "c": 3}
shallow_copy_dict = dict(original_dict)
print(original_dict)
print(shallow_copy_dict)
```

Output:

```
{'a': 1, 'b': 2, 'c': 3}
{'a': 1, 'b': 2, 'c': 3}
```

Like the previous example, changing the value of the key "a" in the copy does not affect the original dictionary.

Both copy() and dict() methods can be used to create a copy of a dictionary. It's important to remember that a copy only copies the references to the original dictionary's elements, so changes made to mutable objects (like lists or other dictionaries) within the copy will affect the original dictionary.

4.1.6 Nested Dictionary

In Python, a nested dictionary is a dictionary that contains other dictionaries as its values. This allows you to create complex and hierarchical data structures. Nested dictionaries are useful when you need to organize data in multiple levels of key-value pairs.

Let's see an example of a nested dictionary representing information about students:

Example:

```
students = {
   "John": {
      "age": 20,
      "gender": "Male",
      "courses": ["Math", "History", "Science"],
   },
   "Alice": {
      "age": 22,
      "gender": "Female",
      "courses": ["Computer Science",
"English"],
   },
   "Bob": {
      "age": 21,
      "gender": "Male",
      "courses": ["Physics", "Chemistry"],
   }
}
```

In this example, students are a nested dictionary with three students (John, Alice, and Bob) as its keys. Each student's information is represented as a dictionary with keys like "age," "gender," and "courses."

You can access individual elements within the nested dictionary using multiple levels of keys:

```
print(students["John"]["age"])
print(students["Alice"]["gender"])
print(students["Bob"]["courses"])
```

Output

```
20
Female
['Physics', 'Chemistry']
```

Nested dictionaries are useful when you have structured data with multiple levels of information. They allow you to organize and access data in a hierarchical manner, making your code more efficient and readable.

4.2 Lists

In Python, a list is a collection of ordered and mutable elements. Lists are one of the built-in data types and can hold items of different data types, such as integers, strings, floats, and even other lists. Lists are defined using square brackets [ ], and elements within the list are separated by commas. Lists maintain the order of elements as they are added, meaning the order in which you insert elements is preserved. You can change, add, or remove elements from a list after it is created. This flexibility allows for dynamic data manipulation.

4.2.1 Creating a List

You can create a list by enclosing elements within square brackets.

Example:

```
my_list = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "orange"]
mixed_list = [1, "hello", 3.14, True]
empty_list = []
```

4.2.2 List Length

You can determine the length of a list by using a len() function.

Example:

```
my_list = [1, 2, 3, 4, 5]
result = len(person)
print(result)
```

Output:

```
5
```

4.2.3 Accessing List

In Python, you can access individual elements in a list using their indices. Lists are zero-indexed, which means the index of the first element is 0, the index of the second element is 1, and so on. You can also use negative indexing to access elements from the end of the list, where -1 refers to the last element, -2 to the second-to-last element, and so on. An Index Error appears, if you try and access element that does not exist in the list.

Here are different ways to access list items:

a. **Accessing by Positive Index -** Positive value of index means counting forward from beginning of the list.

Example:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[0])
print(my_list [1])
```

Output:

```
1
2
```

b. Accessing by Negative Index- Negative value of index means counting starts from the very last item of the list.

Example:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[-2])
print(my_list [-1])
```

Output:

```
4
5
```

c. Accessing Multiple Items (Slicing)

Slicing in Python is a way to extract a portion of a list (or any sequence) by specifying a start and end index. It creates a new list containing the selected elements, while the original list remains unchanged. Slicing is done using the colon (:) operator.

The syntax for slicing is as follows:

```
new_list = original_list[start_index:end_index]
```

- start_index: The index from which slicing starts (inclusive).
- end_index: The index up to which slicing goes (exclusive).

If you omit the start_index, slicing will start from the beginning of the list, and if you omit the end_index, slicing will go up to the end of the list.

i. Slicing with both start and end index.

Example:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[1:3])
```

Output:

```
2
3
```

ii. Slicing with the start index only.

Example:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[1:])
```

Output:

```
2
3
4
5
```

iii. Slicing with the end index only.

Example:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[:3])
```

Output:

```
1
2
3
```

iv. Slicing without the start and end index.

Example:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[:])
```

Output:

```
1
2
3
4
5
```

v. Slicing with negative index (Make sure the start index is less than the end index)

Example:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[-4:-1])
```

Output:

```
2
3
4
```

vi. Slicing with step size

Example:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[0:4:2])
```

Output:

```
1
3
```

Remember that if you try to access an index that is out of range, Python will raise an IndexError. So, ensure that the index you use is within the valid range of the list. Accessing list items by their indices is a fundamental operation in Python and allows you to work with individual elements to perform various operations and computations.

4.2.4 Modifying List

In Python, lists are mutable, which means you can modify them after they are created. You can add, update, or remove elements from a list. Here are some common ways to modify a list:

a) Adding List Elements

In Python, you can add elements to a list using various methods:

a.1) Using **append()** method - The append() method adds an element to the end of the list.

Example:

```
my_list = [1, 2, 3, 4, 5]
my_list.append(6)
print(my_list)
```

Output:

```
[1, 2, 3, 4, 5, 6]
```

a.2) Using **insert()** method - The insert() method allows you to add an element at a specific index in the list (Syntax: yourList.insert(index, value)).

Example:

```
my_list = [1, 2, 3, 4, 5]
my_list.insert(5, 6)
print(my_list)
```

Output:

```
[1, 2, 3, 4, 5, 6]
```

a.3) Using **`+`** operator - You can use the + operator to concatenate lists, effectively adding elements from one list to another.

Example:

```
my_list = [1, 2, 3, 4, 5]
addToList = [6, 7,8]
final_list = my_list + addToList
print(final_list)
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

a.4) Using **extend()** method - The extend() method adds elements from another iterable (e.g., another list) to the end of the list.

Example:

```
my_list = [1, 2, 3, 4, 5]
addToList = [6, 7,8]
final_list.extend(addToList)
print (final_list)
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Remember that the method you choose to add elements to a list depends on your specific use case and the desired result. The append() method is typically used to add a single element to the end of the list, while insert() allows you to add an element at a specific position. The extend() method is useful for adding multiple elements from another iterable.

b) Updating List Elements

In Python, you can update elements in a list by assigning new values to the specific elements using their indices. Lists are mutable, so you can modify individual elements after the list is created.

Example:

```
my_list = [1, 2, 3, 4, 5]
my_list[2] = 10
print (my_list)
```

Output:

```
[1, 2, 10, 4, 5]
```

In this example, index 2 is used to access the element `2` and assigned the new value `10` to it. As a result, the list my_list is updated with the new value, and the element at index 2 is now `10`.

You can also change a range of list items through slicing. To change a range of items in a list, you can use slicing along with assignment. Slicing allows you to extract a portion of the list, and then you can assign new values to that portion to update it.

Example:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8]
my_list[2:5] = [10, 20, 30]
print (my_list)
```

Output:

```
[1, 2, 10, 20, 30, 6, 7, 8]
```

Keep in mind that the length of the slice and the length of the list you assign must be the same. If the lengths are different, the list will be expanded or reduced accordingly. In case there are so many values provided than the range of indices, what happens is that the excess values will be added to the list before the end index. On the other hand, if we replace a larger range with a smaller list, the list reduces.

Example 1:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8]
my_list[2:5] = [10, 20, 30, 40, 50]
print (my_list)
```

Output:

```
[1, 2, 10, 20, 30, 40, 50, 6, 7, 8]
```

Example 2:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8]
my_list[2:5] = [10]
print (my_list)
```

Output:

```
[1, 2, 10, 6, 7, 8]
```

c) Removing List Elements

In Python, you can remove elements from a list using various methods. Here are some common ways to remove list elements:

1. Using **remove()** method - The remove() method allows you to remove the first occurrence of a specific element from the list.

   Example:

   ```
   my_list = [1, 2, 3, 4, 5, 6, 7, 8]
   my_list.remove(6)
   print (my_list)
   ```

   Output:

   ```
   [1, 2, 3, 4, 5, 7, 8]
   ```

2. Using **pop()** method - The pop() method removes and returns the element at the specified index. If no index is provided, it removes the last element.

   Example 1:

   ```
   my_list = [1, 2, 3, 4, 5, 6, 7, 8]
   my_list.pop(5)
   print (my_list)
   ```

   Output:

   ```
   [1, 2, 3, 4, 5, 7, 8]
   ```

   Example 2:

   ```
   my_list = [1, 2, 3, 4, 5, 6, 7, 8]
   my_list.pop()
   print (my_list)
   ```

   Output:

   ```
   [1, 2, 3, 4, 5, 6, 7]
   ```

3. Using **del** keyword - The del keyword is used to delete elements from the list by specifying the index or a range of indices.

   Example 1:

   ```
   my_list = [1, 2, 3, 4, 5, 6, 7, 8]
   del my_list[5]
   print (my_list)
   ```

   Output:

   ```
   [1, 2, 3, 4, 5, 7, 8]
   ```

   Example 2:

   ```
   my_list = [1, 2, 3, 4, 5, 6, 7, 8]
   del my_list[1:3]
   print (my_list)
   ```

   Output:

   ```
   [1, 4, 5, 6, 7, 8]
   ```

   Example 3:

   ```
   my_list = [1, 2, 3, 4, 5, 6, 7, 8]
   del my_list
   print (my_list)
   ```

   Output:

   ```
   # This will cause an error because
   "my_list" no longer exists.
   ```

4. Using **clear()** method - The clear() method removes all elements from the list, making it empty.

Example:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8]
my_list.clear()
print (my_list)
```

Output:

```
[]
```

Remember that when using remove(), pop(), or del, the list is modified in place, and the elements are permanently removed from the list. If you need to keep the removed elements or avoid modifying the original list, consider creating a copy of the list before performing the removal operations.

d) count() method

In Python, the count() method is a built-in function that allows you to count the occurrences of a specific element in a list. It returns the number of times the specified element appears in the list. The count() method is useful for determining the frequency of specific elements in a list, which can be helpful in various data analysis and processing tasks.

The syntax of the count() method is as follows:

```
list.count(element)
```

Where:

- **list**: The list in which you want to count occurrences.
- **element**: The element whose occurrences you want to count.

Example:

```
numbers = [1, 2, 2, 3, 4, 2, 5]
count_of_twos =
numbers.count(2)
print(count_of_twos)
```

Output:

```
3
```

In this example, the count() method is used to count the occurrences of element 2 in the numbers list. It returns the value 3, indicating that the number 2 appears three times in the list.

4.2.5 Looping List

Looping over lists is a common operation in Python, allowing you to process each element in the list. There are several ways to loop through a list, including using for loops and while loops.

1. Using for loops - You can use a for loop to iterate through each element in the list.

Example:

```
my_list = [1, 2, 3]
for val in my_list:
   print(val)
```

Output:

```
1
2
3
```

You can also use enumerate() function if you need both the index and the element from the list.

Example:

```
my_list = [1, 2, 3]
for index, val in enumerate(my_list):
   print(f"Index: {index}, Value: {val}")
```

Output:

```
Index: 0, Value: 1
Index: 1, Value: 2
Index: 2, Value: 3
```

2. Using while loops - You can use a while loop to iterate over a list, but it requires managing an index variable.

Example:

```
index = 0
my_list = [1, 2, 3]
while index < len(my_list):
   print(my_list[index])
   index += 1
```

Output:

```
1
2
3
```

Each of these methods allows you to iterate over the elements in a list and perform specific operations on them. The choice of loop depends on the specific use case and the desired behavior. In most cases, using a for loop or list comprehension is more convenient and Pythonic for looping through lists.

4.2.6 Sorting List

In Python, you can sort lists using the built-in sorted() function or the list's sort() method. Sorting a list arranges its elements in ascending or descending order, depending on your requirements.

1. Using sorted() function - The sorted() function returns a new sorted list without modifying the original list.

Example:

```
my_list = [5, 1, 9, 6, 8]
sortedList = sorted(my_list)
print("Original List:", my_list)
print("Sorted List:", sortedList)
```

Output:

```
Original List: [5, 1, 9, 6, 8]
Sorted List: [1, 5, 6, 8, 9]
```

By default, sorted() sorts the list in ascending order. If you want to sort in descending order, you can use the reverse=True parameter:

Example:

```
my_list = [5, 1, 9, 6, 8]
sortedList = sorted(my_list, reverse=True)
print("Original List:", my_list)
print("Sorted List:", sortedList)
```

Output:

```
Original List: [5, 1, 9, 6, 8]
Sorted List: [9, 8, 6, 5, 1]
```

2. Using sort() method - The sort() method sorts the list in place, modifying the original list.

Example:

```
my_list = [5, 1, 9, 6, 8]
my_list.sort()
print(my_list)
```

Output:

```
[1, 5, 6, 8, 9]
```

Just like the sorted() method, sort() also sorts the list in ascending order by default. For descending order, you can use the reverse=True parameter.

Example:

```
my_list = [5, 1, 9, 6, 8]
my_list.sort(reverse=True)
print(my_list)
```

Output:

```
[9, 8, 6, 5, 1]
```

Both sorted() and sort() methods work for lists of various data types, including numbers, strings, and custom objects. They use the default ordering for each data type. For more complex sorting requirements, you can use the key parameter with custom comparison functions.

Remember that the sort() method modifies the list in place, while sorted() returns a new sorted list, leaving the original list unchanged. Choose the method that best suits your needs based on whether you want to keep the original list or create a new sorted list.

4.2.7 Joining List

There are several ways to join lists in Python, depending on your specific requirements and the desired result. Each of these methods can be useful in different situations. The choice of method depends on whether you want to create a new list, modify an existing list, or use a more concise approach. Here are some common methods:

a) Using **`+`** operator - You can use the + operator to concatenate two or more lists into a new list.

Example:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = list1 + list2
print(list3)
```

Output:

```
[1, 2, 3, 4, 5, 6]
```

b) Using **extend()** method - The extend() method appends elements from one list to the end of another list, modifying the original list.

Example:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)
print(list1)
```

Output:

```
[1, 2, 3, 4, 5, 6]
```

c) Using **append()** and a **loop** - You can use a loop to append each element from one list to another.

Example:

```
list1 = [1, 2, 3]
for item in list2:
   list1.append(item)
print(list1)
```

Output:

```
[1, 2, 3, 4, 5, 6]
```

4.3 Tuples

In Python, a tuple is a built-in data type used to represent an ordered collection of elements. Tuples are like lists, but they have a few key differences. The most significant difference between lists and tuples is that tuples are immutable, meaning once they are created, their elements cannot be changed, added, or removed. In contrast, lists are mutable, and you can modify their elements after creation. Tuples are defined using parentheses ().

4.3.1 Creating a Tuple

You can create a tuple by enclosing elements within parenthesis. Tuples can hold elements of different data types, like lists. They are useful when you have a collection of items that should remain constant or when you want to ensure data integrity and prevent accidental modification.

Example:

```
myNumberTuple = (1, 2, 3, 4, 5)
fruits = ("apple", "banana", "orange")
mixedTuples = ("apple", 1, True)
emptyTuple = ()
```

If you want to create a tuple with only one item, you need to add a comma after the item, otherwise, Python will not recognize it as tuple.

Example:

```
myTuple1= ("apple")
myTuple2 =("apple",)
print(type(myTuple1))
print(type(myTuple2))
```

Output

```
<class 'str'>
<class 'tuple'>
```

4.3.2 Tuple Length

You can determine the length of a tuple by using a len() function.

Example:

```
myTuple = (1, 2, 3, 4, 5)
result = len(myTuple)
print(result)
```

Output:

```
5
```

4.3.3 Accessing Tuple

Accessing elements in a tuple is like accessing elements in a list, using zero-based indexing. Since tuples are immutable, you can retrieve elements, but you cannot modify, add, or remove elements after the tuple is created.

Here's how you can access elements in a tuple:

Example:

```
myTuple = (1, 2, 3, 4, 5)
print(myTuple[1])
print(myTuple[3])
```

Output:

```
2
4
```

You can also use negative indexing to access elements from the end of the tuple.

Example:

```
myTuple = (1, 2, 3, 4, 5)
print(myTuple[-1])
print(myTuple[-3])
```

Output:

```
5
3
```

It is also possible to access range of elements of a tuple just like lists.

Example:

```
myTuple = (1, 2, 3, 4, 5)
print(myTuple[1:3])
```

Output:

```
2
3
```

4.3.4 Modifying Tuple

In Python, tuples are immutable, which means you cannot modify their elements after they are created. Once a tuple is defined, its elements cannot be changed, added, or removed. If you attempt to modify a tuple, you'll encounter an error.

Here's an example to demonstrate that tuples are immutable:

Example:

```
myTuple = (1, 2, 3, 4, 5)
myTuple[0] = 8
```

Output:

```
# TypeError: 'tuple' object does
not support item assignment
```

a) Adding Tuple Elements

Since tuples are immutable, they do not have a built-in append() method, but there are other ways to add items to a tuple. But there are ways to make it possible.

a.1. **Convert into a list** - You can convert it into a list, add your item(s), and convert it back into a tuple.

Example:

```
myTuple = (1, 2, 3, 4, 5)
temp = list(myTuple)
temp.append(6)
myTuple = tuple(temp)
print(myTuple)
```

Output:

```
(1, 2, 3, 4, 5, 6)
```

a.2. **Add tuple to a tuple** - You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple.

Example:

```
myTuple = (1, 2, 3, 4, 5)
nextTuple = (6, 7, 8)
myTuple += nextTuple
print(myTuple)
```

Output:

```
(1, 2, 3, 4, 5, 6, 7, 8)
```

b) Changing Tuple Elements

Since tuple has no way to change its elements, you can apply the same workaround that you did when you are adding items.

Example:

```
myTuple = (1, 2, 3, 4, 5)
temp = list(myTuple)
temp[1] = 9
myTuple = tuple(temp)
print(myTuple)
```

Output:

```
(1, 9, 3, 4, 5)
```

c) **Removing Tuple Elements**

Since tuple has no way to remove element/s, you can apply the same workaround that you did when you are adding or changing items.

Example:

```
myTuple = (1, 2, 3, 4, 5)
temp = list(myTuple)
temp.remove(3)
myTuple = tuple(temp)
print(myTuple)
```

Output:

```
(1, 2, 4, 5)
```

4.3.5 Unpacking a Tuple

Unpacking a tuple in Python allows you to extract individual elements from a tuple and assign them to separate variables. This can be done using a simple assignment statement with the variables on the left-hand side and the tuple on the right-hand side.

Here's an example of unpacking a tuple:

Example:

```
numbers = (10, 20)
x, y = numbers
print(x)
print(y)
```

Output:

```
10
20
```

In this example, we have a tuple called numbers with two elements: 10 and 20. We then unpack the elements of the tuple into the variables x and y. The value 10 is assigned to the variable x, and 20 is assigned to the variable y.

The number of variables on the left-hand side of the assignment must match the number of elements in the tuple; otherwise, you'll get a ValueError. For example, if you try to unpack a tuple with more elements than variables:

Example:

```
numbers = (10, 20, 30)
x, y = numbers
```

Output:

```
# ValueError: too many values to
unpack
```

4.3.6 Looping a Tuple

Looping through a tuple in Python is like looping through a list. Since tuples are iterable, you can use loops to access each element in the tuple one by one. Looping through a tuple allows you to process each element or perform specific operations on them. The method you choose depends on whether you need the index along with the elements or just the elements themselves. The most common approach is to use a for loop to iterate through the tuple elements directly. Here are a few ways to loop through a tuple:

a. Using a for loop

Example:

```
numbers = (10, 20, 30)
for num in numbers:
    print(num)
```

Output:

```
10
20
30
```

You can also use the enumerate() function if you need both index and the element of a tuple.

Example:

```
numbers = (10, 20, 30)
for index, num in enumerate(numbers):
    print(f"Index: {index}, Number: {num}")
```

Output:

```
Index: 0, Number: 10
Index: 1, Number: 20
Index: 2, Number: 30
```

b. Using while loop

Example:

```
index = 0
numbers = (10, 20, 30)
while index < len(numbers):
   print(numbers [index])
   index += 1
```

Output:

```
10
20
30
```

4.3.7 Joining Tuples

In Python, you can join two or more tuples to create a new tuple by using the + operator. This operation creates a new tuple that contains all the elements from both tuples. Tuples are immutable, so their elements cannot be modified once they are created. By using the + operator to join tuples, you are effectively creating a new tuple instead of modifying the existing ones.

Here's how you can join tuples using the + operator:

Example:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
tuple3 = tuple1 + tuple2

print(tuple3)
```

Output:

```
(1, 2, 3, 4, 5, 6)
```

In this example, tuple3 is a new tuple that contains all the elements from both tuple1 and tuple2. You can also join multiple tuples together in a single expression.

Example:

```
tuple1 = (1, 2)
tuple2 = (3, 4)
tuple3 = (5, 6)
combined_tuple = tuple1 + tuple2 + tuple3
print(combined_tuple)
```

Output:

```
(1, 2, 3, 4, 5, 6)
```

It's important to note that the original tuples (tuple1, tuple2, and tuple3) remain unchanged. When you join tuples using the + operator, you are creating a new tuple that combines the elements from the original tuples.