CHAPTER 3

Lists, Tuples, Sets and Dictionaries

Topics:-

1. **Lists and operations on Lists**
2. **Tuples and operations on Tuples**
3. **Sets and operations on Sets**
4. **Dictionaries and operations on Dictionaries**

1. Lists and operations on Lists

=> Characteristics of List

- **Ordered:**
  Lists maintain the order of elements, meaning that when you insert elements into a list, they remain in that order unless you explicitly modify it.
- **Mutable:**
  Lists can be changed after their creation. You can modify, add, or remove elements.
- **Heterogeneous:**
  A list can contain elements of different data types (e.g., integers, strings, floats, or even other lists).
- **Indexed:**
  Lists are zero-indexed, meaning that the first element is accessed with index 0, the second with index 1, and so on.
- **Allows Duplicates:**
  Lists can contain duplicate elements. There is no restriction on the number of times an element can appear in a list.
- **Arbitrary nesting:**
  Lists can contain other lists as elements, allowing for the creation of multi-dimensional arrays.

Operations on Python Lists

- **Creating a List**

  You can create a list using square brackets [], or with the list() constructor.

  ```
  my_list = [1, 2, 3, 'apple', 5.6]
  ```

- **Accessing Elements**
  Lists are indexed, and you can access elements using their index.

  ```
  print(my_list[0])  # Output: 1
  ```

- **Modifying Elements**

  Lists are mutable, so you can update the values of elements.

  ```
  my_list[1] = 'changed'

  print(my_list)  # Output: [1, 'changed', 3, 'apple', 5.6]
  ```

- **Adding Elements**
  Append: Adds a single element to the end of the list.

  my_list.append(10) # Output: [1, 'changed', 3, 'apple', 5.6, 10]

  Extend: Adds multiple elements to the end of the list.

  my_list.extend([20, 30]) # Output: [1, 'changed', 3, 'apple', 5.6, 10, 20, 30]

- **Inserting Elements**
  You can insert an element at any position using insert().

  my_list.insert(2, 'new')

  print(my_list) # Output: [1, 'changed', 'new', 3, 'apple', 5.6, 10, 20, 30]
- **Removing Elements**
  Remove: Removes the first occurrence of an element.

  my_list.remove('apple') # Removes 'apple'

  Pop: Removes and returns the element at the specified index (default is the last element).

  my_list.pop(2) # Removes 'new' at index 2

  Clear: Empties the entire list.

  my_list.clear() # my_list is now []

- **Length of a List**
  You can get the number of elements using len().

  print(len(my_list)) # Output: 7

- **Slicing a List**
  You can access sublists using slicing.

  sub_list = my_list[1:4] # Returns elements from index 1 to 3

- **Reversing a List**
  Reverse the elements in the list.

  my_list.reverse()

- **Sorting a List**
  You can sort a list in ascending order.
  my_list.sort()

- **Common List Methods**
  count(): Returns the number of times a specific element appears in the list.

  my_list.count(3)

  index(): Returns the index of the first occurrence of an element.

  my_list.index(5.6)

  copy(): Returns a shallow copy of the list.

  new_list = my_list.copy()

2. Tuples and operations on Tuples

=> Tuples are another built-in data structure in Python that, like lists, can hold multiple items. However, they are **immutable**, meaning their elements cannot be modified after creation. They are often used for fixed collections of items where you do not want changes, such as coordinates, or grouped data that shouldn't be altered.

Characteristics of Tuples

- **Ordered:**
  Tuples maintain the order of elements. The order in which you insert elements is preserved, just like lists.
- **Immutable:**
  Once a tuple is created, you cannot modify, add, or remove elements. This immutability makes tuples more secure for storing constant data.
- **Heterogeneous:**
  Tuples can contain elements of different data types, such as integers, strings, lists, or even other tuples.
- **Indexed:**
  Tuples are indexed, meaning elements are accessible by their index, starting from 0.
- **Allows Duplicates:**
  Tuples allow duplicate elements, just like lists. There are no restrictions on repetition of elements.
- **Can Be Nested:**
  Tuples can contain other tuples or lists as elements, enabling multi-level data structures.

Creating a Tuple

You can create a tuple by placing elements inside parentheses () and separating them with commas.

```
my_tuple = (1, 2, 3, 'apple', 4.5)
```

Operations on Tuples

- **Accessing Elements**
  Since tuples are indexed, you can access elements using their index (starting at 0).

  ```
  print(my_tuple[1])  # Output: 2
  ```

- **Slicing a Tuple**
  Like lists, tuples support slicing to extract parts of the tuple.
  `print(my_tuple[1:4])  # Output: (2, 3, 'apple')`
- **Concatenation**
  You can concatenate two or more tuples using the + operator.

  ```
  tuple1 = (1, 2)
  tuple2 = (3, 4)

  new_tuple = tuple1 + tuple2  # Output: (1, 2, 3, 4)
  ```

- **Repetition**
  You can repeat the elements in a tuple using the * operator.
  `repeated_tuple = tuple1 * 3  # Output: (1, 2, 1, 2, 1, 2)`
- **Length of a Tuple**
  Use len() to find out how many elements are in the tuple.
  `print(len(my_tuple))  # Output: 5`
- **Nested Tuples**

  ```
  print(len(my_tuple))  # Output: 5
  nested_tuple = ((1, 2), (3, 4))
  print(nested_tuple[1][0])  # Output: 3
  ```

- **Tuple Methods**
  - **count():** Returns the number of occurrences of an element in the tuple.
    `print(my_tuple.count(2))  # Output: 1`
  - **index()**: Returns the index of the first occurrence of an element.
    `print(my_tuple.index('apple'))  # Output: 3`

3. Sets and sets operations

=> Sets are an unordered collection of unique elements. They are used to store multiple items in a single variable but, unlike lists or tuples, sets do not allow duplicates and do not maintain order. Sets are primarily used for operations involving membership, union, intersection, and difference.

Characteristics of Sets

- **Unordered:**
  Sets do not maintain the order of elements. When you print or iterate over a set, the order of elements is unpredictable.
- **Mutable:**
  Sets themselves are mutable, meaning you can add or remove elements after the set is created. However, the elements within a set must be immutable (like integers, strings, tuples).
- **No Duplicates:**
  Sets automatically remove duplicate elements. If you add a duplicate element to a set, it will be ignored.
- **Heterogeneous:**
  Like lists and tuples, sets can contain elements of different data types.
- **Unindexed:**
  Sets do not support indexing, so you cannot access elements by their position. You can, however, iterate through a set.

Creating a Set

You can create a set using curly braces {} or the set() constructor. Duplicate elements are automatically removed.

```
my_set = {1, 2, 3, 3, 4}

empty_set = set()            #This is only method for Empty set

print(my_set)  # Output: {1, 2, 3, 4}
```

Operations on Sets

- **Adding Elements**

You can add elements to a set using the add() method.

```
my_set.add(5)

print(my_set)  # Output: {1, 2, 3, 4, 5}
```

- **Removing Elements**
  remove(): Removes the specified element. If the element does not exist, it raises a KeyError.
  my_set.remove(3)
  discard(): Removes the specified element. If the element does not exist, it does not raise an error.
  my_set.discard(10) # No error even if 10 is not in the set
  pop(): Removes and returns an arbitrary element from the set (since sets are unordered). Raises a KeyError if the set is empty.
  my_set.pop()
  clear(): Removes all elements from the set, making it an empty set.
  my_set.clear() # Output: set()
- **Set Operations**
  Python sets support several mathematical operations like union, intersection, difference, and symmetric difference.
  Union (| or union()): Returns a set containing all unique elements from both sets.
  set1 = {1, 2, 3}
  set2 = {3, 4, 5}
  print(set1 | set2) # Output: {1, 2, 3, 4, 5}
  Intersection (& or intersection()): Returns a set containing only the common elements.
  print(set1 & set2) # Output: {3}
  Difference (- or difference()): Returns a set of elements present in the first set but not in the second.
  print(set1 - set2) # Output: {1, 2}
  Symmetric Difference (^ or symmetric_difference()): Returns a set containing elements that are in either of the sets but not in both.
  print(set1 ^ set2) # Output: {1, 2, 4, 5}
- **Set Methods**
  isdisjoint(): Returns True if two sets have no elements in common.
  print(set1.isdisjoint(set2)) # Output: False
  issubset(): Checks if all elements of the first set are present in the second set.
  print(set1.issubset({1, 2, 3, 4})) # Output: True
  issuperset(): Checks if all elements of the second set are present in the first set.
  print({1, 2, 3, 4}.issuperset(set1)) # Output: True

- **Converting Other Data Types to Sets**

  You can convert a list, tuple, or string to a set using the set() constructor. This can be useful to remove duplicates.

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

  my_set = set(my_list)

  print(my_set) # Output: {1, 2, 3, 4}

4. Dictionaries And Dictionaries Operations

=> Dictionaries in Python are unordered collections of items where each item is stored as a **key-value pair**. They are **mutable**, meaning their contents (i.e., the values of the key-value pairs) can be changed after creation.

Characteristics of Dictionaries

- **Key-Value Pairs:**
  Each entry in a dictionary consists of a unique key mapped to a value. Keys must be immutable types (like strings, numbers, or tuples), while values can be of any data type (even lists or other dictionaries).
- **Unordered:**
  Prior to Python 3.7, dictionaries did not maintain the order of insertion. From Python 3.7 onwards, dictionaries maintain insertion order.
- **Mutable:**
  Dictionaries are mutable, meaning you can add, update, or delete key-value pairs after the dictionary is created.
- **Keys Are Unique:**
  Dictionary keys must be unique. If you attempt to create a dictionary with duplicate keys, the last value associated with the key will overwrite the previous one.
- **Heterogeneous:**
  Both keys and values in dictionaries can be of different data types. For example, a key could be a string, and its corresponding value could be a list.

Creating a Dictionary

You can create a dictionary using curly braces {} with key-value pairs separated by a colon :. Multiple pairs are separated by commas.

my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

my_dict = dict(name='Alice', age=25, city='New York')

Accessing Dictionary Elements

- **Accessing Values by Key**

You can access the value associated with a key using square brackets [].

print(my_dict['name']) # Output: Alice

- **Using get() Method**

The get() method retrieves the value of a key. If the key does not exist, it returns None (or a specified default value).

print(my_dict.get('age')) # Output: 25

print(my_dict.get('salary', 'Not Found')) # Output: Not Found

Dictionary Operations

- **Adding or Updating Key-Value Pairs**

You can add new key-value pairs or update an existing key's value.

my_dict['salary'] = 50000 # Adding a new key-value pair

my_dict['age'] = 30 # Updating an existing key's value

- **Removing Elements**

pop(): Removes and returns the value for the specified key. Raises a KeyError if the key is not found.
age = my_dict.pop('age')
print(age) # Output: 30
popitem(): Removes and returns the last key-value pair as a tuple (insertion order is respected in Python 3.7+).
print(my_dict.popitem()) # Output: ('salary', 50000)
del: Deletes a key-value pair from the dictionary.
del my_dict['city']
clear(): Removes all key-value pairs, making the dictionary empty.
my_dict.clear()

- **Checking for Key Existence**
  You can check whether a key exists in the dictionary using the in keyword.
  print('name' in my_dict) # Output: True

- **Iterating Over a Dictionary**
  You can iterate over the keys, values, or key-value pairs in a dictionary.
  Iterating Over Keys:
  for key in my_dict:
  print(key)
  Iterating Over Values:
  for value in my_dict.values():
  print(value)
  Iterating Over Key-Value Pairs:
  for key, value in my_dict.items():
  print(f'{key}: {value}')

Dictionary Methods

- **keys():** Returns a view object containing all keys in the dictionary.
  print(my_dict.keys()) # Output: dict_keys(['name', 'city'])
- **values():** Returns a view object containing all values in the dictionary.
  print(my_dict.values()) # Output: dict_values(['Alice', 'New York'])
- **items():** Returns a view object containing all key-value pairs as tuples
  print(my_dict.items()) # Output: dict_items([('name', 'Alice'), ('city', 'New York')])
- **update():** Updates the dictionary with elements from another dictionary or an iterable of key-value pairs.
  my_dict.update({'age': 28, 'salary': 60000})

Created By Dhruv Prajapati~

Data Structure	Methods /Operations	List	Tuple	Set	Dictionary
Creation		my_list = [1, 2, 3]	my_tuple = (1, 2, 3)	my_set = {1, 2, 3}	my_dict = {'a': 1, 'b': 2}
Access		my_list[0]	my_tuple[0]	N/A (Sets are unordered)	my_dict['a']
Add Elements		my_list.append(4)	N/A (Tuples are immutable)	my_set.add(4)	my_dict['c'] = 3
Update Elements		my_list[1] = 5	N/A (Tuples are immutable)	N/A (No element replacement)	my_dict['a'] = 4
Remove Elements		my_list.remove(2)	N/A (Tuples are immutable)	my_set.remove(3)	my_dict.pop('a')
		my_list.pop()		my_set.discard(3)	del my_dict['a']
		my_list.clear()		my_set.clear()	my_dict.clear()
Length		len(my_list)	len(my_tuple)	len(my_set)	len(my_dict)
Concatenation		my_list + [4, 5]	my_tuple + (4, 5)	N/A	N/A
Membership		3 in my_list	3 in my_tuple	3 in my_set	'a' in my_dict
Slicing		my_list[1:3]	my_tuple[1:3]	N/A	N/A
Iterating		for x in my_list:	for x in my_tuple:	for x in my_set:	for k, v in my_dict.items():
Sorting		my_list.sort()	N/A (Tuples are immutable)	N/A (Sets are unordered)	N/A
Reversing		my_list.reverse()	N/A (Tuples are immutable)	N/A	N/A
Copying		my_list.copy()	my_tuple[:] (via slicing)	my_set.copy()	my_dict.copy()
Index		my_list.index(3)	my_tuple.index(3)	N/A	N/A
Count		my_list.count(2)	my_tuple.count(2)	N/A	N/A
Union		N/A	N/A	my_set.union(other_set)	N/A
Intersection		N/A	N/A	my_set.intersection(other_set)	N/A
Difference		N/A	N/A	my_set.difference(other_set)	N/A
Symmetric Diff.		N/A	N/A	my_set.symmetric_difference(other_set)	N/A
Dictionary Methods		N/A	N/A	N/A	my_dict.keys(), my_dict.values(), my_dict.items()
Get Value		N/A	N/A	N/A	my_dict.get('a')
Default Value		N/A	N/A	N/A	my_dict.setdefault('a', default_value)
Update		N/A	N/A	N/A	my_dict.update(other_dict)