PYTHON PROGRAMMING

BCA IV SEM(NEP)

NOTES

Prepared By

Mr. Prabhu Kichadi, BE, MTECH

9880437187

UNIT – III

Lists:

Creating Lists; Operations on Lists; Built-in Functions on Lists; Implementation of Stacks and Queues using Lists; Nested Lists. Dictionaries: Creating Dictionaries; Operations on Dictionaries; Built-in Functions on Dictionaries; Dictionary Methods; Populating and Traversing Dictionaries.

Tuples and Sets:

Creating Tuples; Operations on Tuples; Built-in Functions on Tuples; Tuple Methods; Creating Sets; Operations on Sets; Built-in Functions on Sets; Set Methods.

Lists: A list is a mutable ordered sequence of elements. List is a data structure of collection of elements.

Key points about lists:

- **Mutable:** Lists are mutable, meaning you can change their values after they are created.
- **Ordered:** Lists maintain the order in which the elements were added to the list.
- **Indexing**: You can access the elements of a list by their index. The first element has an index of 0.
- **Slicing:** You can extract a subset of elements from a list using slicing notation. Slicing uses a colon to separate the start and end indices.
- A list is an ordered collection of values or items.
- Lists are defined using square brackets, with items separated by commas.

Creating Lists: different ways to create a list in python.

1. **Using square brackets:** You can create a list by enclosing a comma-separated sequence of values inside square brackets. For example:

```
my_list = [1, 2, 3, 4, 5]
```

2. **Using the list() function:** You can create a list by passing a sequence of values to the list() function. For example:

```
my_list = list((1, 2, 3, 4, 5))
```

3. **Using list comprehension:** You can create a list using a list comprehension, which allows you to transform and filter values from another sequence. For example:

```
my_list = [x * 2 for x in range(5)]
```

4. **Using the range() function:** You can create a list of sequential numbers using the range() function, and then convert it to a list using the list() function. For example:

```
my_list = list(range(1, 6))
```

Operations on Lists: there are several operations that you can perform on lists to manipulate them in various ways. Here are some common operations on lists:

1. **Indexing:** You can access an individual item in a list using its index, which starts at 0. For example, to access the first item in a list:
   ```
   my_list = [1, 2, 3]
   print(my_list[0]) # Output: 1
   ```

2. **Slicing:** You can create a new list that contains a subset of the items in the original list by using slicing. Slicing uses a colon to separate the start and end indices, and returns a new list containing the items from the start index up to, but not including, the end index. For example:
   ```
   my_list = [1, 2, 3, 4, 5]
   print(my_list[1:4]) # Output: [2, 3, 4]
   ```

3. **Concatenation:** You can combine two or more lists into a single list using concatenation, which uses the + operator. For example:
   ```
   list1 = [1, 2, 3]
   list2 = [4, 5, 6]
   concatenated_list = list1 + list2
   print(concatenated_list) # Output: [1, 2, 3, 4, 5, 6]
   ```

4. **Repetition:** You can create a new list that contains multiple copies of the items in the original list using repetition, which uses the * operator. For example:
   ```
   my_list = [1, 2, 3]
   repeated_list = my_list * 3
   print(repeated_list) # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
   ```

5. **Length:** You can determine the number of items in a list using the len() function. For example:
   ```
   my_list = [1, 2, 3]
   print(len(my_list)) # Output: 3
   ```

6. **Membership:** You can check if an item is in a list using the in keyword. For example:
   ```
   my_list = [1, 2, 3]
   print(2 in my_list) # Output: True
   print(4 in my_list) # Output: False
   ```

7. **Sorting:** You can sort a list in ascending or descending order using the sort() method. For example:

```
my_list = [3, 1, 2]
my_list.sort() # sorts in ascending order
print(my_list) # Output: [1, 2, 3]

my_list.sort(reverse=True) # sorts in descending order
print(my_list) # Output: [3, 2, 1]
```

8. **Reversing:** You can reverse the order of items in a list using the reverse() method. For example:

```
my_list = [1, 2, 3]
my_list.reverse()
print(my_list) # Output: [3, 2, 1]
```

Built-in Functions on Lists: here are some of the built-in functions available for working with lists in Python:

1. **len()** - Returns the number of elements in the list.
2. **max()** - Returns the maximum value in the list
3. **min()** - Returns the minimum value in the list.
4. **sum()** - Returns the sum of all elements in the list.
5. **sorted()** - Returns a sorted version of the list in ascending order.
6. **reversed()** - Returns a reverse iterator of the list.
7. **list()** - Converts a sequence (tuple, string, etc.) into a list.
8. **count()** - Returns the number of occurrences of a specified element in the list.
9. **index()** - Returns the index of the first occurrence of a specified element in the list.
10. **append()** - Adds an element to the end of the list.
11. **extend()** - Adds the elements of another list (or any iterable) to the end of the list.
12. **insert()** - Inserts an element at a specified position in the list.
13. **remove()** - Removes the first occurrence of a specified element in the list.
14. **pop()** - Removes and returns the last element in the list.
15. **clear()** - Removes all elements from the list.

1. len() : len(): This function returns the number of items in a list. For example:

```
my_list = [1, 2, 3]
print(len(my_list)) # Output: 3
```

2. **max()** - Returns the maximum value in the list.

```
my_list = [1, 3, 2]
print(max(my_list)) # Output: 3
my_list = ['apple', 'banana', 'cherry']
print(max(my_list)) # Output: 'cherry'
```

3. **min()** - Returns the minimum value in the list.
```
my_list = [1, 3, 2]
print(min(my_list)) # Output: 1
my_list = ['apple', 'banana', 'cherry']
print(min(my_list)) # Output: 'apple'
```

4. **sum()** - Returns the sum of all elements in the list.
```
my_list = [1, 2, 3]
print(sum(my_list)) # Output: 6
```

5. **sorted()** - Returns a sorted version of the list in ascending order.
```
my_list = [3, 1, 2]
sorted_list = sorted(my_list)
print(sorted_list) # Output: [1, 2, 3]

my_list = ['banana', 'cherry', 'apple']
sorted_list = sorted(my_list)
print(sorted_list) # Output: ['apple', 'banana', 'cherry']
```

6. **reversed()** - Returns a reverse iterator of the list.
```
my_list = [1, 2, 3]
reversed_list = list(reversed(my_list))
print(reversed_list) # Output: [3, 2, 1]
```

7. **list()** - Converts a sequence (tuple, string, etc.) into a list.
```
my_tuple = (1, 2, 3)
my_list = list(my_tuple)
print(my_list) # Output: [1, 2, 3]
```

8. **count()** - Returns the number of occurrences of a specified element in the list.
```
my_list = [1, 2, 3, 4, 5, 1]
count = my_list.count(1)
print(count) # Output: 2
```

9. **index()** - Returns the index of the first occurrence of a specified element in the list.
```
my_list = [1, 2, 3, 4, 5]
index = my_list.index(3)
print(index) # Output: 2
```

10. **append()** - Adds an element to the end of the list.
```
my_list = [1, 2, 3, 4, 5]
my_list.append(6)
print(my_list) # Output: [1, 2, 3, 4, 5, 6]
```

11. **extend()** - Adds the elements of another list (or any iterable) to the end of the list.
```
my_list = [1, 2, 3, 4, 5]
my_list.extend([6, 7, 8])
print(my_list) # Output: [1, 2, 3, 4, 5, 6, 7, 8]
```

12. **insert()** - Inserts an element at a specified position in the list.
```
my_list = [1, 2, 3]
my_list.insert(1, 4)
print(my_list) # Output: [1, 4, 2, 3]
```

13. **remove()** - Removes the first occurrence of a specified element in the list.
```
numbers = [10, 5, 25, 30, 10]
numbers.remove(10)
print(numbers) # Output: [5, 25, 30, 10]
```

14. **pop()** - Removes and returns the last element in the list, if no index is specified. We can also specify the index too.
```
my_list = [1, 2, 3, 4, 5]
removed_element = my_list.pop(2) # removes and returns the element at index 2
(which is 3)
print(removed_element) # Output: 3
print(my_list) # Output: [1, 2, 4, 5]
```

15. **clear()** - Removes all elements from the list.
```
my_list = [1, 2, 3, 4, 5]
my_list.clear()
print(my_list) # Output: []
```

Implementation of Stacks using Lists:

Stack is a linear data structure, follow the LIFO (Last In First Out) principle, which means that the last element added to the stack is the first element to be removed.

```
class Stack:
  def __init__(self):
    self.stack = []

  def push(self, element):
    self.stack.append(element)

  def pop(self):
    if not self.is_empty():
      return self.stack.pop()
    else:
      return None
    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)


stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print(stack.pop()) # Output: 3
print(stack.pop()) # Output: 2
print(stack.pop()) # Output: 1
print(stack.pop()) # Output: None (stack is empty)
```

In this implementation, the Stack class has four methods:

__init__(self) - Initializes an empty stack.
push(self, element) - Adds an element to the top of the stack.
pop(self) - Removes and returns the top element of the stack. If the stack is empty, returns None.
is_empty(self) - Returns True if the stack is empty, False otherwise.
size(self) - Returns the number of elements in the stack.

Implementation of Queues using Lists:

Queue is a linear data structure which follow the FIFO (First In First Out) principle, which means that the first element added to the queue is the first element to be removed.

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, element):
        self.queue.append(element)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            return None

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)


queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.dequeue()) # Output: 1
print(queue.dequeue()) # Output: 2
print(queue.dequeue()) # Output: 3
print(queue.dequeue()) # Output: None (queue is empty)
```

In this implementation, the Queue class has four methods:

__init__(self) - Initializes an empty queue.
enqueue(self, element) - Adds an element to the back of the queue.
dequeue(self) - Removes and returns the front element of the queue. If the queue is empty, returns None.
is_empty(self) - Returns True if the queue is empty, False otherwise.
size(self) - Returns the number of elements in the queue.

Nested Lists. a nested list is a list that contains other lists as its elements. The elements of a nested list can be accessed using multiple indices.

Ex:

```
my_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In this example, my_list is a list that contains three other lists as its elements. The first element is [1, 2, 3], the second element is [4, 5, 6], and the third element is [7, 8, 9].

We can access the elements of the nested list using multiple indices. For example, to access the first element of the first list, we can use the following code:

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

Dictionaries: In Python, a dictionary is a built-in data structure that represents a collection of key-value pairs. Dictionaries are also known as maps.

Creating Dictionaries; In Python, dictionaries can be created using the following syntax:

my_dict = {key1: value1, key2: value2, key3: value3}

Here's an example of a dictionary created using this syntax:

```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
```

Operations on Dictionaries: common operations on dictionaries in Python:

Accessing values:

```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict["apple"])  # Output: 2
print(my_dict["orange"])  # Output: 4
```

Adding or updating key-value pairs:

```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
my_dict["pear"] = 5
print(my_dict)  # Output: {"apple": 2, "banana": 3, "orange": 4, "pear": 5}

my_dict["apple"] = 6
print(my_dict)  # Output: {"apple": 6, "banana": 3, "orange": 4, "pear": 5}
```

Removing key-value pairs:

```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
del my_dict["banana"]
print(my_dict)  # Output: {"apple": 2, "orange": 4}
```

Checking for existence of keys:

```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print("apple" in my_dict)  # Output: True
print("pear" in my_dict)  # Output: False
```

Looping over keys or values:

```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
for key in my_dict:
        print(key)  # Output: "apple", "banana", "orange"

for value in my_dict.values():
        print(value)  # Output: 2, 3, 4
```

Built-in Functions on Dictionaries:

1. **len() function:** Returns the number of key-value pairs in a dictionary.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(len(my_dict))  # Output: 3
```

2. **keys() method:** Returns a list of all the keys in a dictionary.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.keys())  # Output: ["apple", "banana", "orange"]
```

3. **values() method:** Returns a list of all the values in a dictionary.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.values())  # Output: [2, 3, 4]
```

4. **items() method:** Returns a list of all the key-value pairs in a dictionary as tuples.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.items())  # Output: [("apple", 2), ("banana", 3), ("orange", 4)]
```

5. **get() method:** Returns the value associated with a key, or a default value if the key is not found.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.get("apple"))  # Output: 2
print(my_dict.get("pear", 0))  # Output: 0
```

6. **pop() method:** Removes the key-value pair associated with the specified key and returns the value.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.get("apple"))  # Output: 2
print(my_dict.get("pear", 0))  # Output: 0
```

Dictionary Methods;

1. **clear()** method: Removes all key-value pairs from the dictionary.
2. **copy()** method: Returns a shallow copy of the dictionary.
3. **update()** method: Adds the key-value pairs from one dictionary to another. If a key already exists in the destination dictionary, its value is updated.
4. **pop()** method: Removes the key-value pair associated with the specified key and returns the value.
5. **popitem() method**: Removes and returns the last key-value pair inserted into the dictionary.
6. **get():** Returns the value of the specified key. If the key does not exist, it returns the specified default value.
7. **items():** Returns a list of key-value pairs in the dictionary.
8. **keys():** Returns a list of all the keys in the dictionary.

1. **clear()** method: Removes all key-value pairs from the dictionary.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
my_dict.clear()
print(my_dict)  # Output: {}
```

2. **copy()** method: Returns a shallow copy of the dictionary.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
new_dict = my_dict.copy()
print(new_dict)  # Output: {"apple": 2, "banana": 3, "orange": 4}
```

3. **update()** method: Adds the key-value pairs from one dictionary to another. If a key already exists in the destination dictionary, its value is updated.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
new_dict = {"banana": 5, "pear": 6}
my_dict.update(new_dict)
print(my_dict)  # Output: {"apple": 2, "banana": 5, "orange": 4, "pear": 6}
```

4. **pop()** method: Removes the key-value pair associated with the specified key and returns the value.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.pop("banana"))  # Output: 3
print(my_dict)  # Output: {"apple": 2, "orange": 4}
```

5. **popitem() method**: Removes and returns the last key-value pair inserted into the dictionary.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.popitem())  # Output: ("orange", 4)
print(my_dict)  # Output: {"apple": 2, "banana": 3}
```

6. **get():** Returns the value of the specified key. If the key does not exist, it returns the specified default value.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.get("apple"))  # Output: 2
print(my_dict.get("pear", 0))  # Output: 0
```

7. **items():** Returns a list of key-value pairs in the dictionary.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.items())  # Output: [("apple", 2), ("banana", 3), ("orange", 4)]
```

8. **keys():** Returns a list of all the keys in the dictionary.
```
my_dict = {"apple": 2, "banana": 3, "orange": 4}
print(my_dict.keys())  # Output: ["apple", "banana", "orange"]
```

Populating and Traversing Dictionaries.

Populating a dictionary in Python is simply a matter of assigning values to keys using the key-value syntax. Here's an example of creating a dictionary with some initial data:

```
my_dict = {
    "apple": 2,
    "banana": 3,
    "orange": 4
    }
```

To traverse or iterate over the dictionary, you can use a for loop. There are several ways to iterate over the keys, values, or items (key-value pairs) of a dictionary in Python. Here are some examples:

1. Iterating over keys:

```
for key in my_dict:
    print(key)
```

2. Iterating over values:

```
for value in my_dict.values():
  print(value)
```

3. Iterating over items:

```
for key, value in my_dict.items():
    print(key, value)
```

Tuples: a tuple is an ordered collection of elements, similar to a list. However,

- tuples are immutable, meaning that their values cannot be changed once they are created.
- Tuples are defined using parentheses instead of square brackets, and elements are separated by commas.

Creating Tuples:

In Python, you can create a tuple by enclosing a comma-separated sequence of values in parentheses. Here are some examples:

```
# A tuple of integers
my_tuple1 = (1, 2, 3)

# A tuple of strings
my_tuple2 = ("apple", "banana", "cherry")

# A mixed-type tuple
my_tuple3 = (1, "apple", 3.14)

# A single-element tuple (note the trailing comma)
my_tuple4 = ("apple",)

# An empty tuple
my_tuple5 = ()
```

Operations on Tuples: Tuples are similar to lists in many ways, but there are some differences in their behavior. Here are some operations you can perform on tuples in Python:

Indexing: You can access individual elements of a tuple using square brackets and an index:

```
my_tuple = (1, 2, 3)
print(my_tuple[0])  # prints 1
```

Slicing: You can extract a sub-sequence of a tuple using a slice:

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

Concatenation: You can concatenate two tuples using the + operator:

```
my_tuple1 = (1, 2, 3)
my_tuple2 = (4, 5, 6)
my_tuple3 = my_tuple1 + my_tuple2
print(my_tuple3)  # prints (1, 2, 3, 4, 5, 6)
```

Repetition: You can repeat a tuple using the * operator:

```
my_tuple = (1, 2, 3)
my_tuple2 = my_tuple * 3
print(my_tuple2)  # prints (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Length: You can find the length of a tuple using the len() function:

```
my_tuple = (1, 2, 3)
print(len(my_tuple))  # prints 3
```

Membership test: You can check if an element is present in a tuple using the in operator:

```
my_tuple = (1, 2, 3)
print(2 in my_tuple)  # prints True
print(4 in my_tuple)  # prints False
```

Iteration: You can iterate over the elements of a tuple using a for loop:

```
my_tuple = (1, 2, 3)
for element in my_tuple:
        print(element)
```

Built-in Functions on Tuples;

1. **len():** Returns the number of elements in a tuple.
2. **max():** Returns the largest element in a tuple.
3. **min():** Returns the smallest element in a tuple.
4. **sum():** Returns the sum of all elements in a tuple (only works with numeric types).
5. **sorted():** Returns a sorted list of the elements in a tuple.
6. **any():** Returns True if at least one element in a tuple evaluates to True.
7. **all():** Returns True if all elements in a tuple evaluate to True.

1. **len():** Returns the number of elements in a tuple.
```
my_tuple = (1, 2, 3, 4, 5)
print(len(my_tuple))  # prints 5
```

2. **max():** Returns the largest element in a tuple.
```
my_tuple = (1, 2, 3, 4, 5)
print(max(my_tuple))  # prints 5
```

3. **min():** Returns the smallest element in a tuple.
```
my_tuple = (1, 2, 3, 4, 5)
print(min(my_tuple))  # prints 1
```

4. **sum():** Returns the sum of all elements in a tuple (only works with numeric types).
```
my_tuple = (1, 2, 3, 4, 5)
print(sum(my_tuple))  # prints 15
```

5. **sorted():** Returns a sorted list of the elements in a tuple.
```
my_tuple = (5, 3, 1, 4, 2)
sorted_tuple = sorted(my_tuple)
print(sorted_tuple)  # prints [1, 2, 3, 4, 5]
```

6. **any():** Returns True if at least one element in a tuple evaluates to True.
```
my_tuple = (0, False, None, '', [], {})
print(any(my_tuple))  # prints False

my_tuple2 = (0, False, None, '', [], {}, 1)
print(any(my_tuple2))  # prints True
```

7. **all():** Returns True if all elements in a tuple evaluate to True.
```
my_tuple = (1, True, 'hello', [1, 2, 3])
print(all(my_tuple))  # prints True

my_tuple2 = (1, True, 'hello', [1, 2, 3], False)
print(all(my_tuple2))  # prints False
```

Tuple Methods:

Tuples are immutable in Python, which means they cannot be modified once they are created. As a result, there are only two methods defined for tuples:

count(): This method takes one argument and returns the number of times that argument appears in the tuple.

Example:
```
my_tuple = (1, 2, 3, 4, 5, 1, 2, 3)
print(my_tuple.count(1))  # prints 2
```

index(): This method takes one argument and returns the index of the first occurrence of that argument in the tuple. If the argument is not found, it raises a ValueError.

Example:
```
my_tuple = (1, 2, 3, 4, 5, 1, 2, 3)
print(my_tuple.index(4))  # prints 3
```

Note that these methods do not modify the original tuple; they simply provide information about its contents. If you need to modify a tuple, you must first convert it to a list, make your changes, and then convert it back to a tuple.

Sets: a set is an unordered collection of unique elements. The set data type is mutable, which means you can add or remove elements from it.

Here are some key points about sets in Python:

- Sets are enclosed in curly braces {} or can be created using the set() function.
- Sets can only contain immutable (hashable) objects like strings, integers, and tuples (as long as they contain only immutable elements).
- Sets do not allow duplicate elements. If you try to add a duplicate element to a set, it will be ignored.
- Sets are unordered, so you cannot access elements by index. Instead, you can iterate over the set using a loop or use the in operator to check if an element is present in the set.
- Like other collection types, sets support various operations like union, intersection, and difference.

Creating Sets: In Python, you can create a set using curly braces {} or the set() function. Here are some examples:

```
# creating a set using curly braces
my_set = {1, 2, 3, 4}

# creating a set using set() function
my_set2 = set([1, 2, 3, 4])

# creating an empty set
empty_set = set()
```

Operations on Sets:

Sets in Python support various operations like union, intersection, difference, and symmetric difference. Here are some of the most common operations on sets in Python:

1. **Union |**
2. **Intersection &**
3. **Difference -**
4. **Symmetric Difference ^**
5. **Membership Test in**

1. **Union | :**
   The union of two sets contains all elements that are in either set. You can use the | operator or the union() method to perform the union operation.

   For example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2  # using the | operator
print(union_set)  # Output: {1, 2, 3, 4, 5}

union_set = set1.union(set2)  # using the union() method
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

2. **Intersection & :** The intersection of two sets contains only the elements that are common to both sets. You can use the & operator or the intersection() method to perform the intersection operation.

   For example:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1 & set2  # using the & operator
print(intersection_set)  # Output: {3}

intersection_set = set1.intersection(set2)  # using the intersection() method
print(intersection_set)  # Output: {3}
```

3. **Difference - :**
   The difference between two sets contains only the elements that are in the first set but not in the second set. You can use the - operator or the difference() method to perform the difference operation.

   For example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1 - set2  # using the - operator
print(difference_set)  # Output: {1, 2}

difference_set = set1.difference(set2)  # using the difference() method
print(difference_set)  # Output: {1, 2}
```

4. **Symmetric Difference ^ :**
   The symmetric difference between two sets contains only the elements that are in either of the sets but not in both. You can use the ^ operator or the symmetric_difference() method to perform the symmetric difference operation.

   For example:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_difference_set = set1 ^ set2  # using the ^ operator
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}

symmetric_difference_set = set1.symmetric_difference(set2)  # using the
symmetric_difference() method
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

5. **Membership Test (in) :**
   Membership Test: You can test if an element is a member of a set using the in operator.

   For example:

```
set1 = {1, 2, 3}
print(2 in set1)  # Output: True
print(4 in set1)  # Output: False
```

Built-in Functions on Sets:

len(set): Returns the number of elements in the set.

set() or {}: Creates an empty set.

set(iterable): Creates a set from an iterable (list, tuple, string, etc.).

x in set: Checks if element x is in the set. Returns True if x is in the set, False otherwise.

x not in set: Checks if element x is not in the set. Returns True if x is not in the set, False otherwise.

set1 == set2: Checks if two sets are equal. Returns True if sets are equal, False otherwise.

set1 < set2: Checks if set1 is a proper subset of set2. Returns True if set1 is a proper subset of set2, False otherwise.

set1 <= set2: Checks if set1 is a subset of set2. Returns True if set1 is a subset of set2, False otherwise.

set1 > set2: Checks if set1 is a proper superset of set2. Returns True if set1 is a proper superset of set2, False otherwise.

set1 >= set2: Checks if set1 is a superset of set2. Returns True if set1 is a superset of set2, False otherwise.

Set Methods:

1. **add(element)**: Adds an element to the set.
```
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")
print(fruits) # Output: {'apple', 'banana', 'cherry', 'orange'}
```

2. **clear():** Removes all elements from the set.
```
fruits = {"apple", "banana", "cherry"}
fruits.clear()
print(fruits) # Output: set()
```

3. **copy():** Returns a copy of the set.
```
fruits = {"apple", "banana", "cherry"}
fruits_copy = fruits.copy()
fruits.add("orange")

print(fruits) # Output: {'orange', 'banana', 'apple', 'cherry'}
print(fruits_copy) # Output: {'banana', 'apple', 'cherry'}
```

4. **difference(set):** Returns the set difference of two sets (elements that are in the first set but not in the second set).
```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = set1.difference(set2)
print(set3) # Output: {1, 2}
```

5. **difference_update(set):** Removes all elements of another set from this set.
```
set1 = {1, 2, 3, 4}
set2 = {2, 3}
set1.difference_update(set2)
print(set1) # Output: {1, 4}
```

6. **discard(element):** Removes an element from the set if it is a member. discard() is a set method in Python that removes an element from a set if it is present. If the element is not present, it does nothing.
```
fruits = {"apple", "banana", "cherry"}
fruits.discard("banana")
print(fruits) # Output: {'cherry', 'apple'}
```

7. **intersection(set):** Returns the intersection of two sets (elements that are common to both sets).
```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = set1.intersection(set2)
```

```
print(set3) # Output: {3}
```

8. **intersection_update(set):** Updates the set with the intersection of itself and another set.

```
set1 = {1, 2, 3, 4}
set2 = {2, 3, 4, 5}
set1.intersection_update(set2)
print(set1) # Output: {2, 3, 4}
```

9. **isdisjoint(set):** Returns True if two sets have no intersection, False otherwise.

```
set1 = {1, 2, 3, 4}
set2 = {5, 6, 7}
set3 = {3, 4, 5}

print(set1.isdisjoint(set2)) # Output: True
print(set1.isdisjoint(set3)) # Output: False
```

10. **issubset(set):** Returns True if another set contains this set, False otherwise.

```
set1 = {1, 2, 3}
set2 = {1, 2, 3, 4, 5}
x = set1.issubset(set2)
print(x) # Output: True
```

11. **issuperset(set): Returns** True if this set contains another set, False otherwise.

```
set1 = {1, 2, 3}
set2 = {1, 2}
x = set1.issuperset(set2)
print(x) # Output: True
```

12. **pop():** Removes and returns an arbitrary element from the set. Raises KeyError if the set is empty.

```
fruits = {"apple", "banana", "cherry"}
x = fruits.pop()
print(x) # Output: 'apple'
print(fruits) # Output: {'banana', 'cherry'}
```

13. **remove(element):** Removes an element from the set. Raises a KeyError if the element is not a member.

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")
print(fruits) # Output: {'apple', 'cherry'}
```

14. **symmetric_difference(set):** Returns the symmetric difference of two sets (elements that are in either set, but not in both).

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = set1.symmetric_difference(set2)
print(set3) # Output: {1, 2, 4, 5}
```

15. **symmetric_difference_update(set):** Updates a set with the symmetric difference of itself and another set. is a set method in Python that updates a set with the symmetric difference of itself and another set. The symmetric difference of two sets is the set of elements that are in either of the sets but not in their intersection.

```
set1 = {1, 2, 3, 4}
set2 = {2, 3, 4, 5}
set1.symmetric_difference_update(set2)
print(set1) # Output: {1, 5}
```

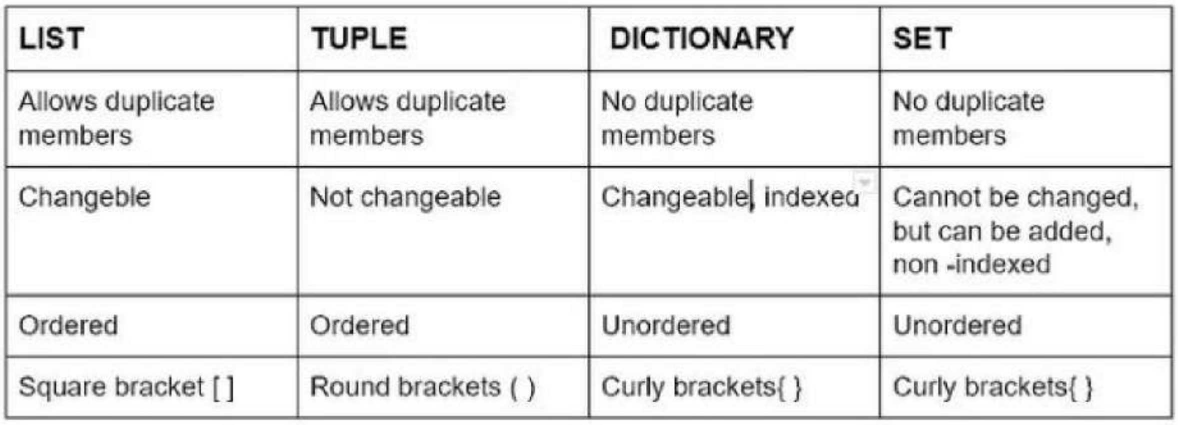
16. **union(set):** Returns the union of two sets (all elements that are in either set).

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = set1.union(set2)
print(set3) # Output: {1, 2, 3, 4, 5}
```

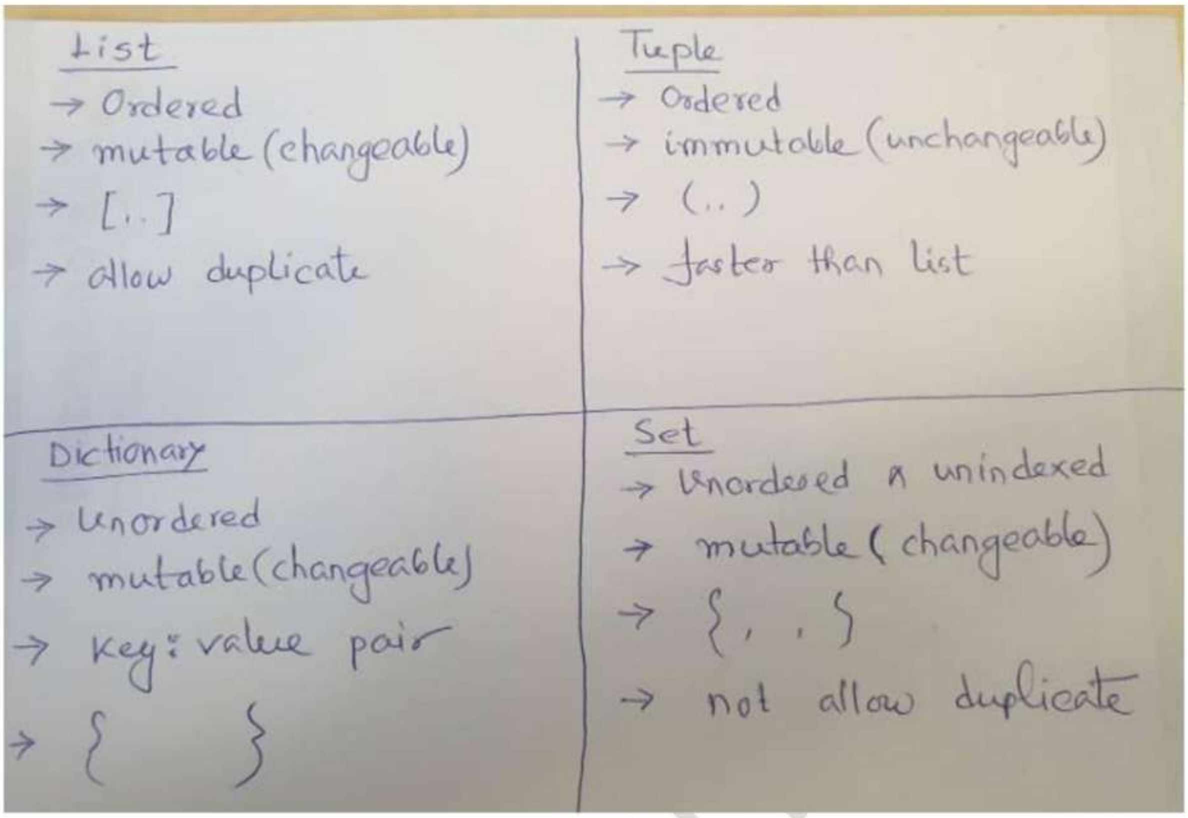
17. **update(set):** Updates the set with the union of itself and another set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set1.update(set2)
print(set1) # Output: {1, 2, 3, 4, 5}
```

Differences:



LIST	TUPLE	DICTIONARY	SET
Allows duplicate members	Allows duplicate members	No duplicate members	No duplicate members
Changeble	Not changeable	Changeable, indexed	Cannot be changed, but can be added, non -indexed
Ordered	Ordered	Unordered	Unordered
Square bracket [ ]	Round brackets ( )	Curly brackets{ }	Curly brackets{ }



List	Tuple
→ Ordered	→ Ordered
→ mutable (changeable)	→ immutable (unchangeable)
→ [..]	→ (..)
→ allow duplicate	→ faster than list

Dictionary	Set
→ Unordered	→ Unordered & unindexed
→ mutable (changeable)	→ mutable (changeable)
→ key: value pair	→ {, ,}
→ { }	→ not allow duplicate

Note: Refer all python Lab journal programs on this unit.