Basic Programming with Python

Prepared By:

Professor Dr. Md. Mijanur Rahman
Department of Computer Science & Engineering
Jatiya Kabi Kazi Nazrul Islam University, Bangladesh.
www.mijanrahman.com

4

Data Structures in Python

CONTENTS

4.1 DATA STRUCTURES

In Python, data structures are objects or collections that are used to store and organize data efficiently. These data structures enable you to perform various operations, such as insertion, deletion, traversal, and search on the stored data. Python provides built-in data structures as well as support for creating custom data structures. Some common built-in data structures in Python include lists, tuples, sets, dictionaries, and strings.

- **Lists:** Lists are ordered collections of items, which can be of any data type. Lists are mutable, meaning that you can modify their contents after creation. They are created using square brackets [] and can contain elements separated by commas. Example:

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

- **Tuples:** Tuples are similar to lists, but they are immutable, meaning that their contents cannot be changed after creation. Tuples are created using parentheses () and can contain elements separated by commas. Example:

  ```
  my_tuple = (1, 2, 3, 4, 5)
  ```

- **Sets:** Sets are unordered collections of unique elements. Sets do not allow duplicate elements, and they are mutable. Sets are created using curly braces {} or the set() constructor. Example:

  ```
  my_set = {1, 2, 3, 4, 5}
  ```

- **Dictionaries:** Dictionaries are collections of key-value pairs, where each key is associated with a value. Dictionaries are mutable and unordered. They are created using curly braces {} with key-value pairs separated by colons :. Example:

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

- **Strings:** Strings are sequences of characters. They are immutable, meaning that their contents cannot be changed after creation. Strings can be created using single quotes ' ', double quotes " ", or triple quotes ''' ''' or """ """. Example:

  ```
  my_string = 'Hello, world!'
  ```

In addition to these built-in data structures, Python also supports several other data structures such as arrays, stacks, queues, linked lists, trees, and graphs, which can be implemented using built-in data types or custom classes. Data structures play a crucial role in organizing and manipulating data efficiently in Python programs.

4.2. LIST

In Python, a list is a built-in data structure that represents an ordered collection of items. Lists are mutable, meaning they can be modified after creation. They can contain elements of different data types, including integers, floats, strings, and even other lists. Lists are versatile and widely used in Python programming for storing and manipulating data.

The following are some examples of lists in Python:

```
numbers = [1, 2, 3, 4, 5]                    #List of Numbers
```

```
fruits = ["apple", "banana", "orange", "grape"]   #List of Strings
mixed_data = [1, "hello", 3.14, True]       #List of Mixed Data Types
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]       #List of Lists (Nested Lists)
empty_list = [] # Empty List
repeated_elements = [0] * 5                 #List with Repeated Elements: Creates a list with five 0s
range_list = list(range(1, 10))             #List of Ranges: Creates a list of numbers from 1 to 9
coordinates = [(1, 2), (3, 4), (5, 6)]      #List of Tuples
characters = list("hello")                  #Converts a string into a list of characters
squares = [x**2 for x in range(1, 6)]       #Creates a list of squares of numbers from 1 to 5
```

In Python, a list is a sequence data type. It is an ordered collection of items. Each item in a list has a unique position index, starting from 0. Lists are highly flexible and can store a wide range of data types and combinations thereof. They are one of the most commonly used data structures in Python due to their versatility and ease of use. Following are some key characteristics and operations associated with lists in Python:

1. **Creation:** Lists are created using square brackets [], and elements are separated by commas ,. For example:

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

2. **Indexing:** Elements in a list are indexed starting from 0. We can access individual elements or slices of elements using square bracket notation. For example:

   ```
   print(my_list[0])         # Access the first element
   print(my_list[-1])        # Access the last element
   print(my_list[1:4])       # Access a slice of elements
   ```

3. **Mutability:** Lists are mutable, meaning you can modify their elements after creation. You can change, add, or remove elements from a list. For example:

   ```
   my_list[0] = 10      # Change the value of the first element
   my_list.append(6)   # Add an element to the end of the list
   my_list.remove(3)   # Remove a specific element from the list
   del my_list[1:3]    # Delete a slice of elements from the list
   ```

4. **Length:** You can determine the length of a list (i.e., the number of elements in the list) using the len() function. For example:

   ```
   print(len(my_list))  # Print the number of elements in the list
   ```

5. **List Operations:** List can be concatenated using the "+" operator to create a new list containing elements from both lists and concatenated multiple copies of a list with "*" operator. The membership operators "in" and "not in" work with list object. For example:

   ```
   new_list = [1, 3, 4, 5] + [6, 7, 8]  #Result: [1, 2, 3, 4, 5, 6, 7, 8]
   my_list = ['Hi'] * 4                 #Result: ['Hi', 'Hi', 'Hi', 'Hi']
   7 in [6, 7, 8]                       #Result: True
   ```

6. **Iteration:** You can iterate over the elements of a list using a loop (e.g., for loop) or other iterable constructs. For example:

   ```
   for item in my_list:
   ```

```
    print(item)
```

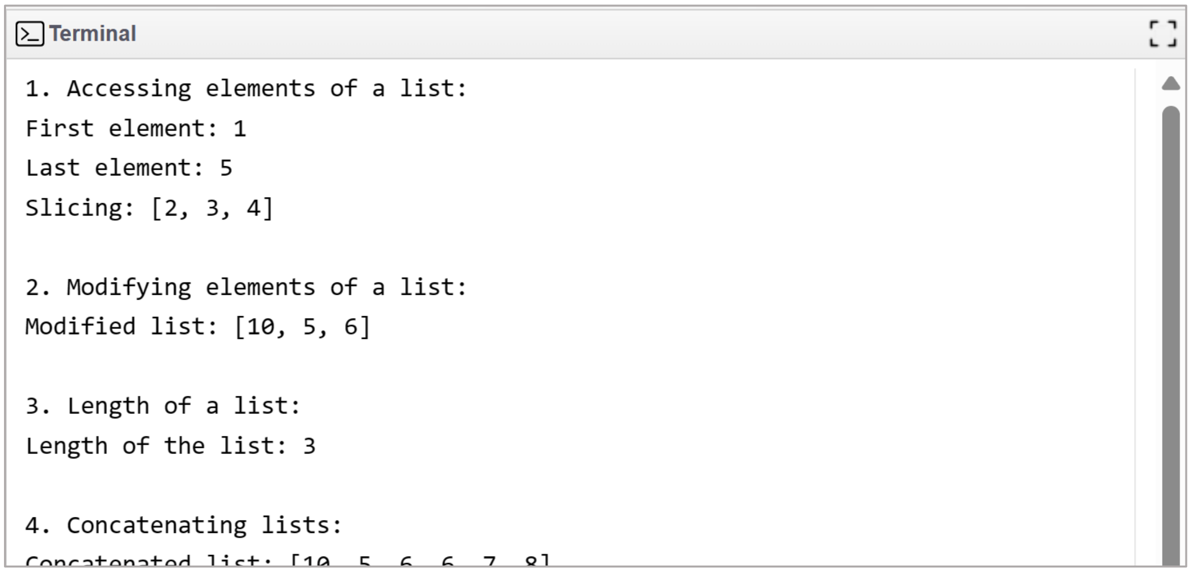
Lists are versatile data structures that can be used in various scenarios, such as storing collections of data, implementing stacks and queues, and more. They are widely used in Python programming due to their flexibility, ease of use, and rich set of operations.

Example 4.1: Demonstrating various operations on lists in Python.

```
# Create a list
my_list = [1, 2, 3, 4, 5]

# 1. Accessing elements of a list
print("1. Accessing elements of a list:")
print("First element:", my_list[0])   # Accessing the first element
print("Last element:", my_list[-1])    # Accessing the last element
print("Slicing:", my_list[1:4])        # Accessing a slice of elements
print()

# 2. Modifying elements of a list
print("2. Modifying elements of a list:")
my_list[0] = 10          # Changing the value of the first element
my_list.append(6)        # Adding an element to the end of the list
my_list.remove(3)        # Removing a specific element from the list
del my_list[1:3]         # Deleting a slice of elements from the list
print("Modified list:", my_list)
print()

# 3. Length of a list
print("3. Length of a list:")
print("Length of the list:", len(my_list))
print()

# 4. Concatenating lists
print("4. Concatenating lists:")
new_list = my_list + [6, 7, 8]   # Concatenating lists
print("Concatenated list:", new_list)
print()

# 5. Iterating over elements of a list
print("5. Iterating over elements of a list:")
print("Elements of the list:")
for item in my_list:
    print(item)
print()

# 6. Checking if an element exists in a list
print("6. Checking if an element exists in a list:")
print("Is 5 in the list?", 5 in my_list)
print("Is 10 in the list?", 10 in my_list)
print()

# 7. Sorting a list
print("7. Sorting a list:")
my_list.sort()        # Sorting the list in ascending order
print("Sorted list (ascending):", my_list)
my_list.sort(reverse=True)  # Sorting the list in descending order
print("Sorted list (descending):", my_list)
print()

# 8. Reversing a list
print("8. Reversing a list:")
my_list.reverse()      # Reversing the order of elements in the list
print("Reversed list:", my_list)
print()

# 9. Counting occurrences of an element in a list
print("9. Counting occurrences of an element in a list:")
print("Number of occurrences of 5 in the list:", my_list.count(5))
print()

# 10. Clearing a list
print("10. Clearing a list:")
my_list.clear()       # Clearing all elements from the list
print("Cleared list:", my_list)
```

This script demonstrates various operations on lists, including accessing elements, modifying elements, getting the length, concatenating lists, iterating over elements, checking for element existence, sorting, reversing, counting occurrences, and clearing the list. Each operation is performed and the results are printed for demonstration purposes.

Output:

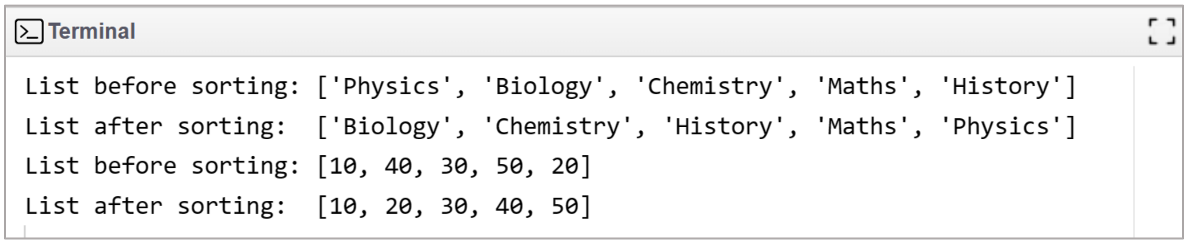


```
1. Accessing elements of a list:
First element: 1
Last element: 5
Slicing: [2, 3, 4]

2. Modifying elements of a list:
Modified list: [10, 5, 6]

3. Length of a list:
Length of the list: 3

4. Concatenating lists:
Concatenated list: [10, 5, 6, 6, 7, 8]
```

Example 4.2: Sorting List Items Alphanumerically in Python.

```
list1 = ['Physics', 'Biology', 'Chemistry', 'Maths', 'History']
print ("List before sorting:", list1)
list1.sort()
print ("List after sorting: ", list1)

list2 = [10, 40, 30, 50, 20]
print ("List before sorting:", list2)
list2.sort()
print ("List after sorting: ", list2)
```

Output:



Terminal

```
List before sorting: ['Physics', 'Biology', 'Chemistry', 'Maths', 'History']
List after sorting:  ['Biology', 'Chemistry', 'History', 'Maths', 'Physics']
List before sorting: [10, 40, 30, 50, 20]
List after sorting:  [10, 20, 30, 40, 50]
```

4.3. TUPLE

In Python, a tuple is a built-in data structure that represents an ordered collection of elements. Tuples are similar to lists, but they are immutable, meaning their elements cannot be modified after creation. Tuples are commonly used for storing heterogeneous data (i.e., data of different types) and for representing fixed-size collections of items.

Following are some examples of tuples in Python:

```
my_tuple = (1, 2, 3, 4, 5)                          # Tuple of Numbers
fruits_tuple = ("apple", "banana", "orange", "grape") # Tuple of Strings
mixed_tuple = (1, "hello", 3.14, True)              # Tuple of Mixed Data Types
single_element_tuple = (1,)                         # Tuple with Single Element
packed_tuple = 1, 2, 3                              # Tuple Packing
a, b, c = packed_tuple                              # Tuple Unpacking
nested_tuple = ((1, 2), (3, 4), (5, 6))             # Tuple of Tuples (Nested Tuples)
repeated_tuple = (0,) * 5                           # Tuple with Repetition: Creates a tuple with five 0s
range_tuple = tuple(range(1, 10))       # Tuple of Ranges: Creates a tuple of numbers from 1 to 9
characters_tuple = tuple("hello")       # Converts a string into a tuple of characters
squares_tuple = tuple(x**2 for x in range(1, 6))  # Creates a tuple of squares of numbers from 1 to 5
```

Tuples are versatile data structures that can store a variety of data types and combinations thereof. They are commonly used in situations where immutability and integrity of data are required. Following are some key characteristics and operations associated with tuples in Python:

- **Creation:** Tuples are created using parentheses () and elements are separated by commas ,. For example:

  ```
  my_tuple = (1, 2, 3, 4, 5)
  ```

- **Indexing:** Elements in a tuple are indexed starting from 0. We can access individual elements or slices of elements using square bracket notation. For example:

```
tup1 = ("Math", "Physics", 65, 85.75)
up2 = (1, 2, 3, 4, 5)
first_element = tup2[0]

print ("Item at 0th index in tup1tup2: ", tup1[0])
print ("Item at index 2 in list2: ", tup2[2])
print ("Items from index 1 to 2 in tup2: ", tup1[1:3])
```

- **Immutable:** Tuples are immutable, meaning once a tuple is created, its elements cannot be modified, added, or removed.

- **Length:** We can determine the length of a tuple (i.e., the number of elements in the tuple) using the len() function. For example:

```
tuple_length = len(my_tuple)
```

- **Iteration:** We can iterate over the elements of a tuple using a loop (e.g., for loop) or other iterable constructs. For example:

```
for item in my_tuple:
print(item)
```

- **Packing and Unpacking:** Tuples support packing and unpacking. Packing is the process of combining multiple values into a single tuple, while unpacking is the process of extracting values from a tuple into individual variables. For example:

```
my_tuple = 1, 2, 3   # Packing
a, b, c = my_tuple   # Unpacking
```

- **Tuple with Single Element:** To create a tuple with a single element, you need to include a trailing comma , after the element to distinguish it from parentheses used for grouping. For example:

```
single_element_tuple = (1,)
```

- **Loop Through Tuple Items:** We can traverse the items in a tuple with for loop construct. The traversal can be done, using tuple as an iterator or with the help of index. For example:

```
tup1 = (25, 12, 10, 30, 10, 100)
for num in tup1:
  print (num, end = ' ')
```

Or

```
indices = range(len(tup1))
for i in indices:
  print ("tup1[{}]: ".format(i), tup1[i])
```

- **Tuple Methods:** Tuples have fewer methods compared to lists since they are immutable. However, they still support methods like count() and index(). For example:

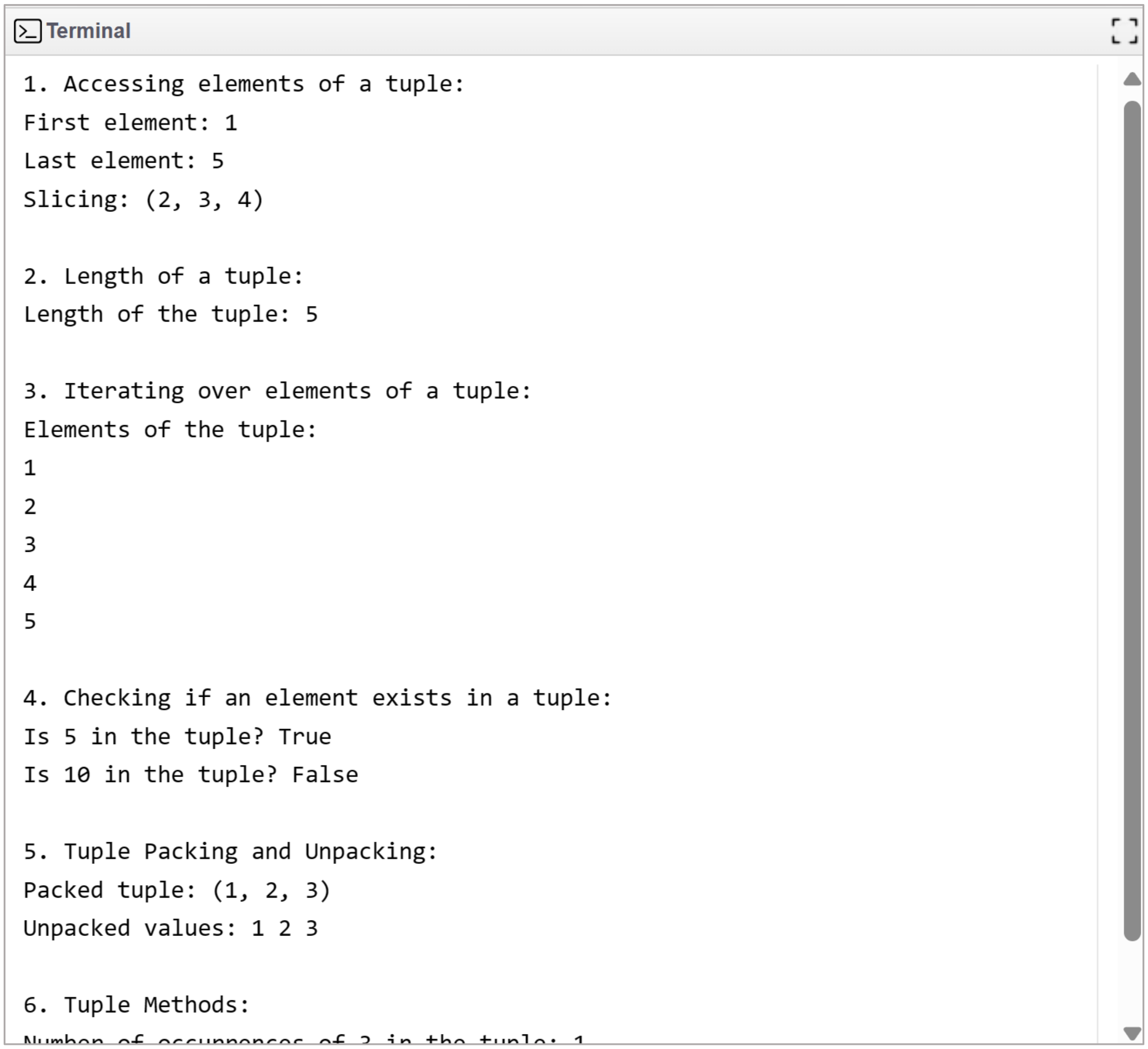
```
count_of_element = my_tuple.count(1)   # Count occurrences of a value
index_of_element = my_tuple.index(2)   # Find index of a value
```

Example 4.3: Demonstrating various operations on tuples in Python.

```
# Create a tuple
my_tuple = (1, 2, 3, 4, 5)

# 1. Accessing elements of a tuple
print("1. Accessing elements of a tuple:")
print("First element:", my_tuple[0])    # Accessing the first element
print("Last element:", my_tuple[-1])    # Accessing the last element
print("Slicing:", my_tuple[1:4])        # Accessing a slice of elements
print()

# 2. Length of a tuple
print("2. Length of a tuple:")
print("Length of the tuple:", len(my_tuple))
print()

# 3. Iterating over elements of a tuple
print("3. Iterating over elements of a tuple:")
print("Elements of the tuple:")
for item in my_tuple:
    print(item)
print()

# 4. Checking if an element exists in a tuple
print("4. Checking if an element exists in a tuple:")
print("Is 5 in the tuple?", 5 in my_tuple)
print("Is 10 in the tuple?", 10 in my_tuple)
print()

# 5. Tuple Packing and Unpacking
print("5. Tuple Packing and Unpacking:")
packed_tuple = 1, 2, 3   # Packing
a, b, c = packed_tuple   # Unpacking
print("Packed tuple:", packed_tuple)
print("Unpacked values:", a, b, c)
print()

# 6. Tuple Methods
print("6. Tuple Methods:")
count_of_element = my_tuple.count(3)   # Count occurrences of a value
index_of_element = my_tuple.index(2)   # Find index of a value
print("Number of occurrences of 3 in the tuple:", count_of_element)
print("Index of value 2 in the tuple:", index_of_element)
```

This script demonstrates various operations on tuples, including accessing elements, getting the length, iterating over elements, checking for element existence, tuple packing and unpacking, and using tuple methods. Each operation is performed and the results are printed for demonstration purposes.

Output:



```
1. Accessing elements of a tuple:
First element: 1
Last element: 5
Slicing: (2, 3, 4)

2. Length of a tuple:
Length of the tuple: 5

3. Iterating over elements of a tuple:
Elements of the tuple:
1
2
3
4
5

4. Checking if an element exists in a tuple:
Is 5 in the tuple? True
Is 10 in the tuple? False

5. Tuple Packing and Unpacking:
Packed tuple: (1, 2, 3)
Unpacked values: 1 2 3

6. Tuple Methods:
Number of occurrences of 3 in the tuple: 1
```

4.3.1 Tuple Methods

In Python, a tuple is an immutable sequence of elements. This means that once a tuple is created, its contents cannot be changed or modified. However, tuples do come with a set of methods for various operations, although these methods do not modify the tuple itself but rather return a new tuple or provide information about the existing tuple.

An explanation of some common tuple methods in Python is given below:

- **count(value):** This method returns the number of occurrences of a specified value within the tuple.

```
my_tuple = (1, 2, 2, 3, 4, 2)
print(my_tuple.count(2))  # Output: 3 (since 2 occurs three times in the tuple)
```

- **index(value, [start, [stop]]):** This method returns the index of the first occurrence of a specified value within the tuple. You can also specify optional start and stop parameters to search within a specific range of the tuple.

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple.index(3))  # Output: 2 (since 3 is at index 2 in the tuple)
```

- **len(tuple):** This function returns the number of elements in the tuple. Although len() is not a method specific to tuples, it can be used to obtain the length of any iterable object including tuples.

```
my_tuple = (1, 2, 3, 4, 5)
print(len(my_tuple))  # Output: 5 (since there are five elements in the tuple)
```

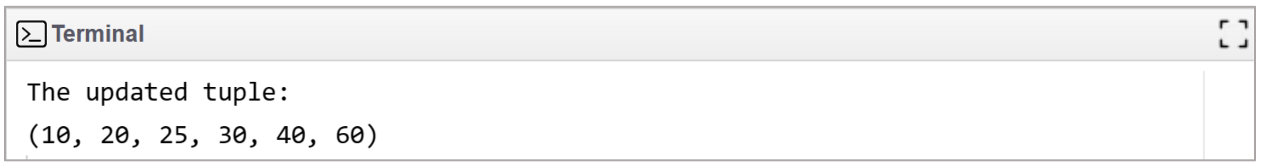
It's important to note that since tuples are immutable, they lack methods that would modify them directly, such as append(), insert(), or remove(). If we need to perform such operations, we would typically convert the tuple to a list, perform the operation, and then convert it back to a tuple if necessary.

Example 4.4: Convert a tuple to a list, perform some list operations, and then convert it back to a tuple in Python.

```
# Given tuple
my_tuple = (10, 20, 30, 40, 50)

# Convert tuple to list
my_list = list(my_tuple)

# Perform list operations
my_list.append(60) #Insert at the last
my_list.insert(2, 25) #Insert at index 2
my_list.remove(50) #Remove item 50

# Convert list back to tuple
new_tuple = tuple(my_list)

print("The updated tuple:")
print(new_tuple)  # Output: (10, 20, 25, 40, 60)
```

In this example, we first convert the tuple my_tuple to a list using the list() function, resulting in my_list. Then, we perform some list operations on my_list such as appending an element (60), inserting an element (25) at index 2, and removing an element (30). Finally, we convert the modified list back to a tuple using the tuple() function, resulting in new_tuple.

Output:



```
The updated tuple:
(10, 20, 25, 30, 40, 60)
```

4.4. SET

In Python, a set is a built-in data structure that represents an unordered collection of unique elements. Sets are mutable, meaning you can modify the contents of a set after creation, but they do not allow duplicate elements. Sets are widely used for operations such as membership testing, removing duplicates from a sequence, and performing set operations like union, intersection, difference, and symmetric difference.

Following are some examples of sets in Python:

```
my_set = {1, 2, 3, 4, 5}                          # Set of Numbers
colors_set = {"red", "green", "blue", "yellow"}   # Set of Strings
mixed_set = {1, "hello", 3.14, True}              # Set of Mixed Data Types
duplicates_set = {1, 2, 2, 3, 3, 4, 5} # Set with Duplicates (Duplicates are automatically removed)
empty_set = set()                                 # Empty Set (Use set() constructor)
squares_set = {x**2 for x in range(1, 6)}         # Creates a set of squares of num
```

Sets are versatile data structures that can store a collection of unique elements. They are particularly useful for eliminating duplicates from a collection, performing mathematical set operations, and testing membership efficiently. Following are some key characteristics and operations associated with sets in Python:

- **Creation:** Sets are created using curly braces {} or the set() constructor. Elements are separated by commas ,. For example:

  ```
  my_set = {1, 2, 3, 4, 5}
  ```

- **Uniqueness:** Sets do not allow duplicate elements. If we try to add a duplicate element to a set, it will be ignored. For example:

  ```
  my_set = {1, 2, 2, 3, 3, 4, 5}  # Duplicate elements are ignored
  ```

- **Membership Testing:** We can efficiently test for membership (i.e., whether an element is present in a set) using the in operator. For example:

  ```
  print(3 in my_set)  # Output: True
  ```

- **Mutability:** Sets are mutable, meaning we can modify their contents after creation. We can add and remove elements from a set. For example:

  ```
  my_set.add(6)      # Add an element to the set
  my_set.remove(3)   # Remove a specific element from the set
  ```

- **Length:** We can determine the number of elements in a set (i.e., its size) using the len() function. For example:

```
set_size = len(my_set)
```

- **Set Operations:** Sets support various set operations such as union (|), intersection (&), difference (-), and symmetric difference (^). For example:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2                    # Union of sets
intersection_set = set1 & set2             # Intersection of sets
difference_set = set1 - set2               # Difference of sets
symmetric_difference_set = set1 ^ set2  # Symmetric difference of sets
```

- **Iterating over Elements:** We can iterate over the elements of a set using a loop (e.g., for loop) or other iterable constructs. For example:

```
for item in my_set:
print(item)
```

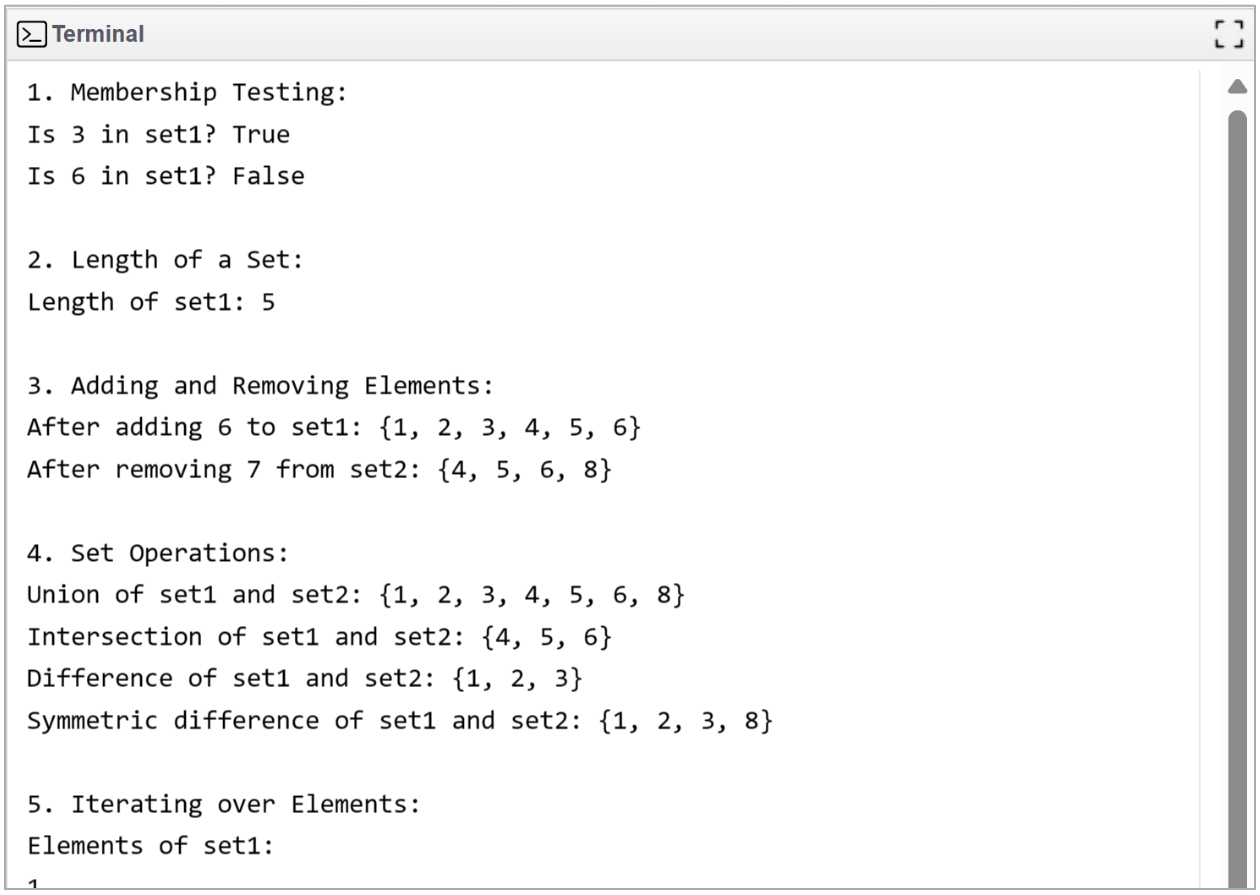
Example 4.5: Demonstrating various operations on sets in Python.

```
# Create two sets
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

# 1. Membership Testing
print("1. Membership Testing:")
print("Is 3 in set1?", 3 in set1)   # Output: True
print("Is 6 in set1?", 6 in set1)   # Output: False
print()

# 2. Length of a Set
print("2. Length of a Set:")
print("Length of set1:", len(set1))
print()

# 3. Adding and Removing Elements
print("3. Adding and Removing Elements:")
set1.add(6)     # Add an element to set1
set2.remove(7)  # Remove an element from set2
print("After adding 6 to set1:", set1)
print("After removing 7 from set2:", set2)
print()

# 4. Set Operations
print("4. Set Operations:")
union_set = set1 | set2          # Union of sets
intersection_set = set1 & set2     # Intersection of sets
difference_set = set1 - set2      # Difference of sets
symmetric_difference_set = set1 ^ set2  # Symmetric difference of sets
```

```
print("Union of set1 and set2:", union_set)
print("Intersection of set1 and set2:", intersection_set)
print("Difference of set1 and set2:", difference_set)
print("Symmetric difference of set1 and set2:", symmetric_difference_set)
print()

# 5. Iterating over Elements
print("5. Iterating over Elements:")
print("Elements of set1:")
for item in set1:
    print(item)
print()

# 6. Clearing a Set
print("6. Clearing a Set:")
set2.clear()  # Clear all elements from set2
print("Cleared set2:", set2)
```

This script demonstrates various operations on sets, including membership testing, getting the length, adding and removing elements, set operations (union, intersection, difference, symmetric difference), iterating over elements, and clearing the set. Each operation is performed, and the results are printed for demonstration purposes.

Output:



Terminal

```
1. Membership Testing:
Is 3 in set1? True
Is 6 in set1? False

2. Length of a Set:
Length of set1: 5

3. Adding and Removing Elements:
After adding 6 to set1: {1, 2, 3, 4, 5, 6}
After removing 7 from set2: {4, 5, 6, 8}

4. Set Operations:
Union of set1 and set2: {1, 2, 3, 4, 5, 6, 8}
Intersection of set1 and set2: {4, 5, 6}
Difference of set1 and set2: {1, 2, 3}
Symmetric difference of set1 and set2: {1, 2, 3, 8}

5. Iterating over Elements:
Elements of set1:
```

4.5. DICTIONARY

In Python, a dictionary is a collection of unordered, mutable, and indexed elements. Unlike sequences like lists and tuples, which are indexed by a range of numbers, dictionaries are indexed by keys. Each key in a dictionary is unique and associated with a specific value. Dictionaries are highly flexible and commonly used for storing and retrieving data efficiently.

Following are a few examples demonstrating the use of dictionaries in Python:

1. **Storing Information about a Person:** We can use a dictionary to store various details about a person. For example:

```
person = {
  "name": "Alice",
  "age": 30,
  "gender": "Female",
  "city": "London"
}
```

2. **Mapping Abbreviations to Full Names:** A dictionary can be used to map abbreviations to their corresponding full names. For example:

```
abbreviation_to_full_name = {
  "USA": "United States of America",
  "UK": "United Kingdom",
  "IND": "India"
}
```

3. **Storing Product Information:** We can use a dictionary to store information about products, such as their names, prices, and quantities. For example:

```
product_info = {
  "item1": {"name": "Laptop", "price": 999, "quantity": 10},
  "item2": {"name": "Phone", "price": 499, "quantity": 20},
  "item3": {"name": "Tablet", "price": 299, "quantity": 15}
}
```

4. **Representing Student Grades:** A dictionary can be used to store grades of students. For example:

```
student_grades = {
  "Alice": {"math": 90, "science": 85, "history": 88},
  "Bob": {"math": 85, "science": 92, "history": 80},
  "Charlie": {"math": 88, "science": 90, "history": 82}
}
```

5. **Mapping English Words to Their French Translations:** A dictionary can be used for language translations. For example:

```
english_to_french = {
  "hello": "bonjour",
  "goodbye": "au revoir",
  "thank you": "merci"
}
```

Dictionaries are versatile data structures that can be applied to a wide range of scenarios where key-value pairs are needed for efficient data storage and retrieval.

Following are some key characteristics and operations associated with dictionaries in Python:

1. **Creating a Dictionary:** We can create a dictionary using curly braces {} and specifying key-value pairs separated by colons :. For an example:
   ```
   my_dict = {"name": "John", "age": 30, "city": "New York"}
   ```
2. **Accessing Values:** We can access the value associated with a key using square brackets [] or the get() method. For an example:
   ```
   print(my_dict["name"])  # Output: John
   print(my_dict.get("age"))  # Output: 30
   ```
3. **Adding or Modifying Entries:** We can add new key-value pairs to a dictionary or modify existing ones. For an example:
   ```
   my_dict["gender"] = "Male"  # Adding a new key-value pair
   my_dict["age"] = 31  # Modifying the value of an existing key
   ```
4. **Removing Entries:** We can remove a key-value pair from a dictionary using the del keyword or the pop() method. For an example:
   ```
   del my_dict["city"]  # Removing a specific key-value pair
   my_dict.pop("age")  # Removing a specific key-value pair and returning the value
   ```
5. **Dictionary Methods:** Python dictionaries come with a variety of useful methods for manipulation, such as keys(), values(), and items(). For example:
   ```
   keys = my_dict.keys()  # Returns a view object containing the keys
   values = my_dict.values()  # Returns a view object containing the values
   items = my_dict.items()  # Returns a view object containing the key-value pairs as tuples
   ```
6. **Length of Dictionary:** We can find the number of key-value pairs in a dictionary using the len() function:
   ```
   print(len(my_dict))  # Output: 2 (after removing one entry)
   ```
7. **Checking Key Existence:** We can check whether a key exists in a dictionary using the in keyword. For example:
   ```
   if "name" in my_dict:
      print("Name is present in the dictionary")
   ```

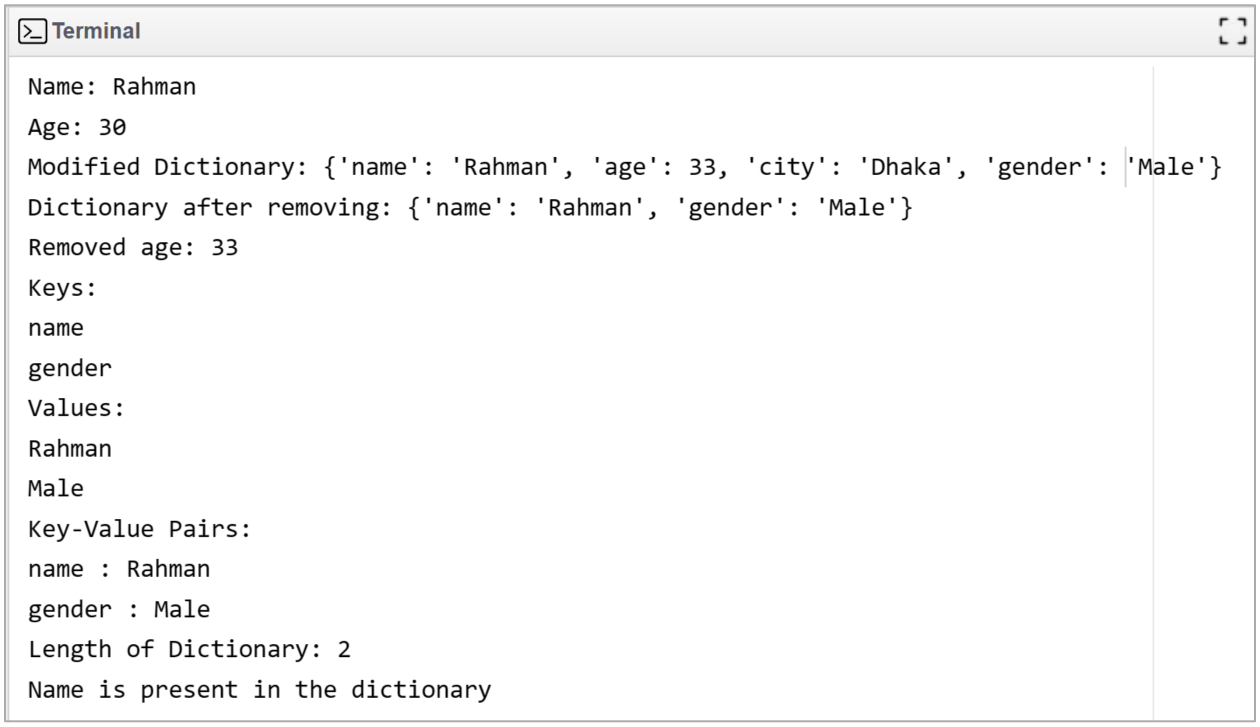
Dictionaries are widely used in Python for tasks such as storing configurations, caching data, and representing structured information. They offer fast lookups and are highly versatile due to their key-value pair structure.

Example 4.6: Demonstrating various operations on dictionaries in Python.

```
# Creating a dictionary
my_dict = {"name": "Rahman", "age": 30, "city": "Dhaka"}

# Accessing values
print("Name:", my_dict["name"])  # Output: Rahman
print("Age:", my_dict.get("age"))  # Output: 30

# Adding or modifying entries
my_dict["gender"] = "Male"  # Adding a new key-value pair
my_dict["age"] = 33  # Modifying the value of an existing key
print("Modified Dictionary:", my_dict)

# Removing entries
del my_dict["city"]  # Removing a specific key-value pair
removed_age = my_dict.pop("age")  # Removing a specific key-value pair and returning the value
print("Dictionary after removing:", my_dict)
print("Removed age:", removed_age)

# Iterating through a dictionary
print("Keys:")
for key in my_dict:
    print(key)

print("Values:")
for value in my_dict.values():
    print(value)

print("Key-Value Pairs:")
for key, value in my_dict.items():
    print(key, ":", value)

# Length of dictionary
print("Length of Dictionary:", len(my_dict))

# Checking key existence
if "name" in my_dict:
    print("Name is present in the dictionary")
```

This program demonstrates various operations, such as creating a dictionary, accessing values, adding/modifying/removing entries, iterating through the dictionary, finding the length of the dictionary, and checking the existence of keys.

Output:



```
Name: Rahman
Age: 30
Modified Dictionary: {'name': 'Rahman', 'age': 33, 'city': 'Dhaka', 'gender': 'Male'}
Dictionary after removing: {'name': 'Rahman', 'gender': 'Male'}
Removed age: 33
Keys:
name
gender
Values:
Rahman
Male
Key-Value Pairs:
name : Rahman
gender : Male
Length of Dictionary: 2
Name is present in the dictionary
```

4.6. STRING

In Python, a string is a sequence of characters enclosed within either single quotes (''), double quotes (" "), or triple quotes (''' ''' or """ """). Strings are immutable, meaning they cannot be changed once created. Here's a simple example:

```
my_string = "Hello, World!"
```

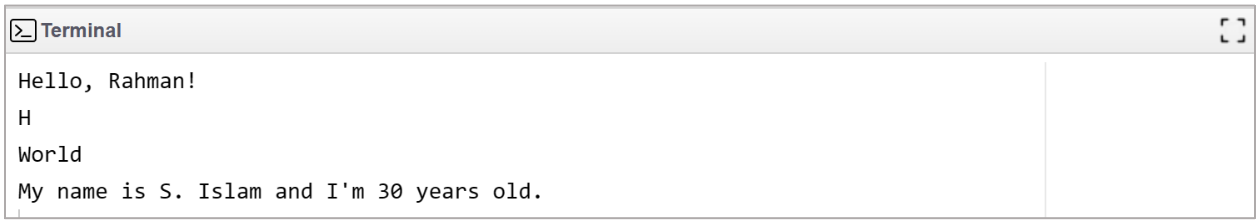
We can perform various operations on strings, such as concatenation, slicing, and formatting. Python provides a rich set of methods to manipulate strings, including functions for searching, replacing, splitting, and joining.

Example 4.7: Perform various operations on strings, such as concatenation, slicing, and formatting in Python.

```
# Concatenation
greeting = "Hello"
name = "Rahman"
message = greeting + ", " + name + "!"
print(message)  # Output: Hello, Rahman!

# Slicing
my_string = "Hello, World!"
print(my_string[0])      # Output: H
print(my_string[7:12])   # Output: World
```

```
# Formatting
name = "S. Islam"
age = 30
formatted_string = "My name is {} and I'm {} years old.".format(name, age)
print(formatted_string)   # Output: My name is Bob and I'm 30 years old.
```

Output:



```
Hello, Rahman!
H
World
My name is S. Islam and I'm 30 years old.
```

4.6.1 Creating Python Strings

In Python, we can create a string by enclosing characters within either single quotes (''), double quotes (" "), or triple quotes (''' ''' or """ """). Following are examples of each method:

Using Single Quotes:

```
my_string = 'Hello, World!'
```

Using Double Quotes:

```
my_string = "Hello, World!"
```

Using Triple Quotes (for multiline strings):

```
my_string = '''Hello,
World!'''
```

or

```
my_string = """Hello,
World!"""
```

All these methods create the same string "Hello, World!". The choice between single, double, or triple quotes depends on the preference and the specific requirements of the string, such as whether it contains single or double quotes within it or if it spans multiple lines.

4.6.2 Accessing Values in Strings

In Python, we can access individual characters or substrings within a string using indexing and slicing. The following are some examples that show how we can do it:

Accessing Individual Characters: We can access individual characters in a string using square brackets [] and providing the index of the character we want to access. Indexing starts at 0 for the first character.

```
my_string = "Hello, World!"

# Accessing the first character
```

```
first_character = my_string[0]  # 'H'

# Accessing the last character
last_character = my_string[-1]  # '!'

# Accessing a character at a specific index
character_at_index_7 = my_string[7]  # 'W'
```

Slicing Substrings: We can extract a substring from a string using slicing. Slicing allows us to specify a range of indices to extract characters between those indices.

```
my_string = "Hello, World!"

# Slicing to get a substring
substring1 = my_string[7:12]  # 'World'

# Slicing with negative indices
substring2 = my_string[-6:-1]  # 'World'

# Omitting start or end index
substring3 = my_string[:5]  # 'Hello'
substring4 = my_string[7:]  # 'World!'
```

Using Step in Slicing: We can also specify a step value to extract characters at regular intervals within the specified range.

```
my_string = "Hello, World!"

# Slicing with step
every_other_character = my_string[::2]  # 'Hlo ol!'
```

Accessing Characters in Nested Structures: If we have a nested structure like a list of strings, you can access characters in a similar manner.

```
my_list = ["Hello", "World"]
first_char_second_word = my_list[1][0]  # 'W'
```

4.7.3 String Special Operators

In Python, special operators, also known as string operators, allow us to perform various operations on strings, such as concatenation, repetition, and membership testing. The following are some commonly used string operators in Python:

Concatenation (+): The concatenation operator (+) joins two strings together.

```
string1 = "Hello"
string2 = "World"
concatenated_string = string1 + " " + string2  # Output: "Hello World"
```

Repetition (*): The repetition operator (*) repeats a string a specified number of times.

```
string = "Hello"
```

```
repeated_string = string * 3  # Output: "HelloHelloHello"
```

Membership (in, not in): The membership operators (in, not in) test whether a substring exists within a string.

```
string = "Hello, World!"
is_hello_present = "Hello" in string  # Output: True
is_world_present = "World" not in string  # Output: False
```

Indexing ([]): Indexing allows you to access individual characters or substrings within a string.

```
string = "Hello, World!"
first_character = string[0]  # Output: "H"
substring = string[7:12]  # Output: "World"
```

Slicing ([start:end:step]): Slicing extracts a substring from a string with the specified start and end indices, and optionally a step value.

```
string = "Hello, World!"
substring = string[7:12]  # Output: "World"
```

String Formatting (% and .format()): String formatting allows you to insert values into a string in a specified format.

```
name = "Alice"
age = 30
formatted_string = "My name is %s and I'm %d years old." % (name, age)
# Output: "My name is Alice and I'm 30 years old."

name = "Bob"
age = 25
formatted_string = "My name is {} and I'm {} years old.".format(name, age)
# Output: "My name is Bob and I'm 25 years old."
```

4.7.4. String Formatting Operator

In Python, string formatting operators are used to insert values into a string in a specified format. There are several ways to format strings in Python, including the old-style % formatting and the newer str.format() method. Additionally, Python 3.6 introduced f-strings for string interpolation. The following examples illustrate how we can use each method:

Old-Style % Formatting: This method uses the % operator to format strings.

```
name = "Alice"
age = 30
formatted_string = "My name is %s and I'm %d years old." % (name, age)
# Output: "My name is Alice and I'm 30 years old."
```

str.format() Method: This method uses the format() method to insert values into a string.

```
name = "Bob"
age = 25
```

```
formatted_string = "My name is {} and I'm {} years old.".format(name, age)
# Output: "My name is Bob and I'm 25 years old."
```

Formatted String Literals (f-strings): Introduced in Python 3.6, f-strings provide a concise way to embed expressions inside string literals.

```
name = "Charlie"
age = 35
formatted_string = f"My name is {name} and I'm {age} years old."
# Output: "My name is Charlie and I'm 35 years old."
```

Each of these methods provides a way to format strings in Python, and the choice between them often depends on personal preference and the specific requirements of the formatting task.

4.7 ARRAYS

In Python, arrays are a fundamental data structure provided by the built-in array module. However, the more commonly used data structure for storing sequences of elements is the list.

Lists: Lists are mutable sequences, meaning you can change their elements after they are created. Lists can contain elements of different data types and can grow or shrink dynamically.

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

We can perform various operations on lists, such as appending, inserting, removing, slicing, and iterating.

Arrays: Arrays in Python are provided by the array module. Unlike lists, arrays are homogeneous collections of items (i.e., all elements must be of the same type). Arrays are more memory-efficient than lists for storing large sequences of primitive data types like integers or floats. To use arrays, we need to import the array module:

```
from array import array
```

Then, we can create an array by specifying the type code and the initial values:

```
my_array = array('i', [1, 2, 3, 4, 5])
```

Here, 'i' is the type code for integers. Other type codes include 'f' for floats, 'd' for doubles, 'b' for bytes, etc.

The following is an overview of some basic operations you can perform on arrays in Python using both the built-in array module and NumPy:

Creating Arrays:

Using the array module:

```
from array import array
my_array = array('i', [1, 2, 3, 4, 5])
```

Using NumPy:

```
import numpy as np
```

```
my_array = np.array([1, 2, 3, 4, 5])
```

Accessing Elements:

Using indexing:

```
print(my_array[0])  # Accessing the first element
```

Using slicing:

```
print(my_array[1:3])  # Slicing from index 1 to 2
```

Appending Elements: Using the append() method (for lists and NumPy arrays):

```
my_list.append(6)
my_array = np.append(my_array, 6)
```

Concatenating Arrays: Using the + operator (for lists) or concatenate() function (for NumPy arrays):

```
combined_list = my_list1 + my_list2
combined_array = np.concatenate((array1, array2))
```

Element-wise Operations: For NumPy arrays, you can perform element-wise arithmetic operations:

```
array_sum = array1 + array2
array_product = array1 * array2
```

Reshaping: For NumPy arrays, you can reshape arrays into different dimensions:

```
reshaped_array = np.reshape(my_array, (2, 3))  # Reshape into 2 rows and 3 columns
```

Stacking: For NumPy arrays, you can stack arrays vertically or horizontally:

```
vertical_stack = np.vstack((array1, array2))
horizontal_stack = np.hstack((array1, array2))
```

Arrays have similar operations as lists, but they do not have as many built-in methods. However, arrays are more efficient for numerical computations and are often used in scenarios where performance is critical. In most cases, lists are more commonly used due to their flexibility and the wide range of built-in operations they support. Arrays are typically used in specific situations where performance is a concern or when dealing with large datasets of homogeneous data.

4.7.1 Operations in Array Data Structures

Traversing: Traversing an array involves accessing each element of the array one by one. This can be done using a loop, such as a for loop or a while loop.

```
from array import *
array1 = array('i', [10,20,30,40,50])
for x in array1:
 print(x)
```

Searching: Searching for an element in an array involves iterating through the array and checking if the desired element is present. We can use various searching algorithms like linear search or binary search.

```
from array import *
array1 = array('i', [10,20,30,40,50])
print (array1.index(40))
```

Inserting: Inserting an element into an array involves adding a new element at a specific position in the array. This might require shifting elements to make space for the new element.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.insert(1,60)
for x in array1:
 print(x)
```

Deleting: Deleting an element from an array involves removing an element from a specific position in the array. This might require shifting elements to fill the gap left by the deleted element.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.remove(40)
for x in array1:
   print(x)
```

Sorting: Sorting an array involves arranging the elements of the array in ascending or descending order. We can use built-in sorting algorithms like sorted() or sort() method for lists, or more efficient algorithms like QuickSort or MergeSort.

```
sorted_array = sorted(my_array)  # Creates a new sorted array
my_array.sort()  # Sorts the array in-place
```

Updating: Updating an element in an array involves changing the value of an existing element at a specific position in the array.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1[2] = 80
for x in array1:
   print(x)
```

⌘⌘⌘