MASTER OF COMPUTER APPLICATIONS

SEMESTER 1

PYTHON PROGRAMMING

Unit 5

Data Structures

Table of Contents

SL No	Topic		Fig No / Table / Graph	SAQ / Activity	Page No
1	Introduction		-	-	4 - 5
	1.1	Learning Objectives	-	-	
2	Lists		-	-	6 - 8
	2.1	Indexing and Splitting	-	-	
	2.2	Update a List Value	-	-	
3	List Operators		-	-	9 - 16
	3.1	Repetition	-	-	
	3.2	Concatenation	-	-	
	3.3	Membership	-	-	
	3.4	Iteration	-	-	
	3.5	Length	-	-	
	3.6	Adding Elements to a List	-	-	
	3.7	Removing Elements in a List	-	-	
4	Built-in Functions		-	-	17 - 18
	4.1	len(list)	-	-	
	4.2	max(list)	-	-	
	4.3	min(list)	-	-	
	4.4	list(seq)	-	-	
5	Tuple		-	-	19 - 23
	5.1	Creating a Tuple	-	-	
	5.2	Indexing and Slicing	-	-	

1. INTRODUCTION

Unit 4 was a journey through the world of Functions and Modules in Python, where we explored how to create reusable pieces of code to enhance the efficiency and readability of our programs. We delved into defining and calling functions, understanding the scope of variables, and utilizing Python's extensive standard library modules to add powerful functionalities to our projects without reinventing the wheel. By now, you've learned to appreciate the elegance of Python's syntax and the power of modular programming, making your code more organized and manageable.

Now, it's time to dive into Unit 5, where we'll dive into Python's versatile Lists and Tuples. These data structures are the workhorses of Python programming, allowing you to store, access, and manipulate collections of data in intuitive and powerful ways. Understanding lists and tuples is fundamental, as they are used in almost every Python program to organize data, perform iterations, and much more. This unit will equip you with the knowledge to harness these tools effectively, enabling you to manage complex data sets with ease.

As we dive into this new unit, approach the learning process with curiosity and experimentation. Python's interactive nature allows you to try out concepts in real-time, so make the most of it by writing and testing small snippets of code to solidify your understanding of lists and tuples. Pay close attention to indexing and slicing, as these are key to accessing and modifying data within these structures. By the end of this unit, you'll be adept at manipulating lists and tuples, paving the way for more advanced programming techniques in the units to come. Let's get started and explore the dynamic world of Python lists and tuples

1.1 Learning Objectives

At the end of this Unit, you will be able to:

- *Identify the basic characteristics and purposes of lists and tuples in Python.*
- *Explain how lists differ from tuples in terms of mutability and usage.*
- *Demonstrate how to create, access, and modify elements in lists using indexing and slicing techniques.*
- *Compare and contrast the effectiveness of using lists versus tuples for various data storage scenarios.*
- *Assess the impact of list comprehensions on code readability and performance*.

2. LISTS

Lists are sequential data structures that can contain different data types. They are mutable; thus, you can change the content and the indexing of the list while working on it. This feature makes them different from other data structures such as tuples and dictionaries and makes them one of the most versatile data structures in Python.

In a list, each element is assigned a unique identity. This identity refers to the address in the memory where the data is located. The values stored in a list all have different addresses and positions. These positions are known as an index. We will learn more about indexing further in the chapter.

In Python, lists are created by adding the elements of the list inside square brackets []. The elements inside the brackets are separated by commas. A list can have any kind of data type such as integer, float, strings, etc. They can also be any number of elements in a list, the limit depends upon the memory space of the computer system that you are working on.

Example:

```
a_list = []
a_list = [1, 3, 5]
a_list = [1, ‘harish’, 3]
a_list = [44, [76, 88, 99], 45, ‘a’]
```

All the above examples are valid in Python. Thus, a list can have mixed data variables, as well as a list, can be nested into another list. Such lists are called nested list.

2.1. Indexing and Splitting

For accessing all the properties of a list, one must know how to access each element of a list. For this, the elements of the list are provided with an index. As in most programming languages, in Python as well, the indexing starts from 0 (zero). The operator [] is used to access an item in the list.

If you try to access a value of the index in a string that does not exist, then the interpreter will show you an IndexError. Also, one should note that the indexes are always integers numbers. Using float or other data types as indexes will result in an error called TypeError.

Indexing in nested lists is called nested indexing. Examples are shown below.

Example:

```
a_list = [1, 3, ‘a’, ‘b’, [4, 5, ‘list’]]
print (a_list [2])
print (a_list [4] [1])
```

The output of this will be as follows:

```
a
5
```

Python also allows negative indexing of the elements in a list. These are used when the list is to be read from right to left, that is, from the last element of the list to the first. Thus, to reverse a list, you only need to use negative indexing.

In the above example, add the code provided below.

```
print (a_list [-2])
```

The output of the command will be

```
b
```

When accessing a single element of a list, you will use indexing. But what if you need to access more than 1 elements of a list? You can, no doubt, use indexing several times. But if the elements that you want to access are sequences together in the list, then you can simply use slicing or splitting.

Slicing is the method through which you can access ranges of elements. This can be done by separating the indices of the first and last element of the desired range with a colon.

One should note that the index used on the left of the colon refers to the first element that you want to include while splitting. Whereas, the index on the right of the colon should be the number of the element that lies after the slice. Go through the example given below to better understand this.

Example:

```
odd_numbers = [1, 3, 5, 7, 9, 11, 13, 15]
print (odd_numbers [1:4])
```

The output of the program will be as follows:

```
3, 5, 7
```

Write another command line given below.

```
print (odd_numbers [0:1])
```

The output will be 1.

Thus, the first index is inclusive in the slicing while the second index is exclusive.

2.2. Update a List Value

As lists are mutable data structures, you can change the values present in the list while it is active. This allows the user to update the list within the program without changing any of the previous code. To update a list value, you will use the assignment operator. The element that needs to be accessed will be naturally called with the help of its index.

Example:

```
even = [2, 4, 6, 8, 10]
even [0] = 0
print (even)
```

The output will be as shown below.

```
0, 4, 6, 8, 10
```

You can also update the values of several elements in a list with the help of slicing. Use the code provided below in the above example and find the output.

```
even [1:4] = [ 2, 4, 6, 8]
```

3. LIST OPERATORS

There are various things through which lists can be used and manipulated. You will find that some of these operations are common with all other sequence data structures. Python offers various methods through which the elements and values of a list can be changed and updated. These were discussed briefly in the previous units as well. Some of the most common operations are explained here.

3.1. Repetition

A list can be repeated n number of times with the help of the operator *. The operator creates a list that has the original list repeated the desired number of times. Here is an example of repetition in lists.

```
mylist = [ 3, 6, 9, 12, 15]

print (mylist * 3)
```

The output of the program will look like this:

```
[3, 6, 9, 12, 15, 3, 6, 9, 12, 15, 3, 6, 9, 12, 15]
```

As you might know that in C when you declare an array, you have to provide the number of elements included in the array during its initialisation. However, in Python, there is no such requirement. There are cases when you need the list to acquire a definite sequence in the memory.

For example, you create an empty list. There are no elements in the list but you need space to store at least 10 elements in the list. You can introduce one element and repeat it using the * operator like [0] * 10.

If you want the interpreter to read that there is nothing in the list, or that the list is empty, then you can right None inside the square brackets of the list. An example is shown below.

Example:

```
mylist = [None] * 10
print (mylist)
```

The output of the program will yield the keyword None printed 10 times.

```
[None, None, None, None, None, None, None, None, None, None]
```

3.2. Concatenation

Two or more lists can be joined together with the operator “+”. This joining of one sequence with another is called concatenation. The lists can have different data types stored in them and can be of different lengths.

Example:

```
lista = [1, 2, 3]
listb = [4, 5, 6]
print (lista + listb)
```

The output of the code will be:

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

3.3. Membership

Membership operators are used to checking whether an element is present in a list or not. If the element is found in the list, then the operator will yield True as the output. If no such element is present in the given list, the output will be false. There are two member operators in Python, in and not in.

Example:

```
a_list = [‘b’, ‘c’, ‘d’, ‘e’, 1, 2, 3]
print (‘a’ in a_list)
print (4 not in a_list)
```

The output of the program will be as follows.

```
False
True
```

Membership operators have multitudes of uses in real life, especially for security purposes. For example, it can be used to check whether the entered password matches the one that is saved.

3.4. Iteration

There are times when you need to manipulate all the elements present in the list with a similar set of codes. In such cases when the code needs to be performed on all elements of the list, the list is iterated. There are various ways you can use iteration in the list. These are explained below.

Using for Loop

One of the easiest ways to iterate over the complete list is through for loop. It is depicted through the example below.

```
list = [1, 2, 3, 4, 5]
for i in list:
    print (i)
```

Output

```
1
2
3
4
5
```

Using loop and range

If you want the iterations to be operated on elements present in a set range, then you can use a loop with range (). It is shown through the example below.

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

length = len (list)

for i in range (length):
    print (list [i])
```

Output:

```
1
2
3
4
5
```

Using While Loop

You can also use the while loop; however, the program may require additional statements.

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

length = len (list)

i = 0

while i< length:

    print (list [i])

i += 1
```

Output:

```
1
2
3
4
5
```

Using list comprehension way

This can be the shortest and most accurate way to iterate over a list.

```
list = [1, 2, 3, 4, 5]
[print (i) for i in list]
```

Output:

```
1
2
3
4
5
```

Using enumerate ()

The method enumerates () is used when you need to convert the list into a list of tuples that can be iterated. Its working is shown through the example.

```
list = [1, 3, 5, 7, 9]
for i, val in enumerate (list):
    print (i, ",", val)
```

Output:

```
0, 1
1, 3
2, 5
3, 7
4, 9
```

Using Numpy

When dealing with lists with n number of dimensions, it is better to use an external library such as NumPy that can speed up the process.

```
import numpy as mylist

a = mylist.arrange(9)

a = a.reshape(3, 3)

for x in mylist.nditer(a):

    print(x)
```

Output:

```
0
1
2
3
4
5
6
7
8
```

3.5. Length

In some cases, you may need to find the length of the list. This can be done through several methods. Without using a special function, you can find the length of the list, that is, the total number of elements includes in a list, in the following way.

```
list = [1, 3, 5, 7, 9]
i = 0
for j in list:
i = i +1
print (str(i))
```

The output will be 5.

3.6. Adding Elements to a List

To add elements more items to the list, we can use the method append (). Through this method, new items can be added to the end of the list.

Example:

```
even = [2, 4, 6, 8]
even.append (10)
print (even)
```

The output will be:

```
[2, 4, 6, 8, 10]
```

Using the method extend (), you can add several items to the end of the list.

Example:

```
odd = [1, 3, 5, 7]
th=[9, 11, 13]
odd.extend (th)
print (odd)
```

The output of such a program will be:

```
[1, 3, 5, 7, 9, 11, 13]
```

If you want to add an element at a desired location in the list, then the method insert () is used.

Example:

```
odd = [1, 3, 5, 7]
odd.insert (1, 2)
print (odd)
```

Note that in the function insert (), the first number represents the index where the new element is to be included and the second number represents the value that needs to be included.

The output of this program will be:

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

3.7. Removing Elements in a List

The keyword del is used to delete an element or several elements from the list. You can also use the keyword to delete the entire list. Here is an example that details the different ways the del keyword can be used.

```
list1 = [1, 3, 5, 7, 9, 11]
# Using del to remove the element at index 2
```

```
del list1[2]
print(list1)  # Output will be [1, 3, 7, 9, 11]

# Using del to remove a slice from index 1 to 3 (exclusive)
del list1[1:3]
print(list1)  # Output will be [1, 9, 11]

# Using remove to remove the first occurrence of element 9
list1.remove(9)
print(list1)  # Output will be [1, 11]

# Using pop to remove and return the element at index 1 (the last element in this case)
removed_element = list1.pop(1)
print("Removed element:", removed_element)  # Output will be Removed element: 11
print(list1)  # Output will be [1]

# Deleting the entire list
del list1

# Trying to print the list after deletion will cause an error
# Uncommenting the following line will raise a NameError
# print(list1)  # NameError: name 'list1' is not defined
```

The output will be as follows:

```
After deleting element at index 2: [1, 3, 7, 9, 11]
After deleting a slice from index 1 to 3: [1, 9, 11]
After removing element 9: [1, 11]
Removed element: 11
Final list: [1]
```

Other methods can be used to delete an element from the list. These include remove (), pop (), and clear (). remove () is used to delete a given item. pop (i) is used to delete an item at the given index. the clear () method is used to empty the complete list.

4. BUILT-IN FUNCTIONS

Python provides various in-built functions that can be used to perform various processes on the list and its elements. You can use these functions to ease the code or to know more about the given list. The prominent functions in Python for lists are explained below.

4.1. len(list)

To determine the length, that is, the number of elements present in the list, the len () function is used.

Example:

```
odd = [1, 3, 5, 7]
print (len (odd))
```

The output will be 4.

4.2. max(list)

The function max () finds the largest value in a list.

Example:

```
odd = [1, 3, 5, 7]
print (max (odd))
```

The output will be 7.

4.3. min(list)

The min () function returns the smallest element present in the given list.

Example:

```
odd = [1, 3, 5, 7]
print (min (odd))
```

The output will be 1.

4.4. list(seq)

There are times when you will need to change a string or tuple into the list. In Python, you can do that with the help of the list () function. The constructor returns a list, if there are no parameters given, then it will return an empty list.

```
string1 = "python"
print (list (string1))
```

The output will be:

```
['p', 'y', 't', 'h', 'o', 'n']
```

5. TUPLE

A tuple is an ordered collection of items and is a part of the standard Python language. Programmers can use a tuple to store multiple items or objects in a single variable. The tuple is like a list in many aspects and is one of Python's built-in data types. However, unlike a list, tuples are immutable. That means we cannot change the elements once we assign them to a tuple, but we can reassign or delete an entire tuple. The immutable nature of tuples helps programmers manipulate them faster when compared to a list. As the elements are intended to stay constant throughout the program, using tuple prevents any accidental data collection changes.

Another feature that differentiates a list and a tuple is how they are enclosed. We enclose the items of a list within square brackets ("[...]"). On the other hand, we use parenthesis or round bracket ("(...)") to enclose the elements. Although we use parenthesis in a tuple, we refer to the index number of the elements (enclosed in square brackets) to access them, similar to lists and strings.

5.1. Creating a Tuple

Tuples collect and hold their elements inside a parenthesis. They are characterized by immutable nature . Tuple has the capacity to hold multiple elements under a single variable name.

```
# tuple with parenthesis
l = (a, c, e)
```

You can also make tuples without using the parenthesis, but it a good practice to use them.

```
# tuple without parenthesis
l = q, f
print(l)
(q, f)
```

For the most part, tuples include more than one item. However, the comma must be used even when creating tuples with single elements.

```
# tuple with a single element
```

```
single = (a, )
```

As a tuple is indexed, it allows programmers to use a single element multiple time.

```
# tuple with duplicated values
my_tup = ('pineapple', 'apple', 'banana', 'banana', 'apple')
print (my_tup)
```

You can also create tuples with a mix of various data types or nested elements.

```
# tuple with mixed data types
a=12
my_tup = (a, 'Hi', 3.4)
print(my_tup)
# nested tuple
my_tup = ('horse', [2, 7, 6], (1, 4, 2))
print(my_tup)
```

5.2. Indexing and Slicing

In Python, we use indexing and slicing to access elements in sequence data types like a tuple. Indexing is used to access individual items by using their index numbers; whereas, slicing is used to access a sequence of elements.

In both indexing and slicing, the index operator "[]" is incorporated to access an element. The program then fetches the respective item from the given tuple (counting from the left). Indexing is limited only to the number of items in the tuple. If you try to retrieve an element beyond the index range: the program will raise an IndexError.

Using float or other data types will result in a TypeError. Hence, only integers are allowed in indexing.

```
my_tuple = ('my', 'name', 'is')
print(my_tuple[1])
```

Similarly, if you wish to retrieve items from a nested tuple, you use nested indexing.

```
# nested tuple
nes_tuple = ("horse", [2, 7, 6], (1, 5, 8))
```

```
# nested index
print(nes_tuple[0][3]) # s is retrieved
print(nes_tuple[2][2]) # 8 is retrieved
```

In slicing, we use the slicing operator (:) to retrieve a range of elements. To understand how the range works, imagine that the index is present between the elements instead. To access a range, we use the index that will slice that particular range.

Example:

```
slice_tuple = ('l', 'o', 'n', 'g', 'i', 't', 'u', 'd', 'e')
print(slice_tuple[1:4])
# elements 2nd to 4th are sliced
# output will be ( 'o', 'n', 'g' )
```

You can also access a range from the beginning or end to a particular.

```
slice_tuple = ('l', 'o', 'n', 'g', 'i', 't', 'u', 'd', 'e')
print(slice_tuple[6:])
# elements from 7th to end are sliced
# output will be ( 'u', 'd', 'e' )
slice_tuple = ('l', 'o', 'n', 'g', 'i', 't', 'u', 'd', 'e')
print(slice_tuple[:6])
# elements from beginning to 5th are sliced
# output will be ( 'l', 'o', 'n', 'g', 'i', 't' )
```

5.3. Negative Indexing

In Python, the sequence can also have negative indexing. The last item in the sequence is assigned -1 index, the second to last has -2 index, etc.

```
# Using negative indexing
my_tuple = ('l', 'o', 'n', 'g', 'i', 't', 'u', 'd', 'e')
# For output: 'i'
print(my_tuple[-5])
```

Slicing using the negative index:

```
my_tuple = ('l', 'o', 'n', 'g', 'i', 't', 'u', 'd', 'e')
```

```
# elements 3rd to 5th
print(my_tuple[-3:-5])
# Output: ()
```

5.4. Delete a tuple

As discussed earlier, tuples are unchangeable sequences. It explains the unvarying nature of the elements in the tuple. However, we can delete a tuple entirely by using the "del" keyword.

```
# Deleting a tuple
my_tuple = ('l', 'o', 'n', 'g', 'i', 't', 'u', 'd', 'e')
del my_tuple
```

In the above example, if you try to print the tuple "my_tuple," the program will throw a Name Error.

5.5. Basic Operations of a tuple

We can perform some basic operations in a tuple as follows:

A. **Addition:** Using the addition operator (+), we can concatenate two or more tuples.

```
# '+' will join the three tuples given
t = (2, 5, 0) + (1, 3) + (4, 5, 6)
print (t)
Output:
(2, 5, 0, 1, 3, 4, 5, 6)
```

B. **Multiplication:** We use the multiplication operator to (*) to create a repetitive tuple sequence. When a tuple is multiplied with an integer 'x', it creates a new tuple with all the initial tuple elements repeated 'x' times.

```
# Multiplying a tuple with an integer
t = ('l', 'a')
print (t*3)

Output:

( 'l', 'a', 'l', 'a', 'l', 'a' )
```

C. **Using "in" Keyword:** We can check the existence of an element in a tuple using the "in" keyword. If the element exists, the program will return 'True'; otherwise, it returns 'False'.

```
#Using "in" keyword
my_tuple = ('l', 'o', 'n', 'g', 'i', 't', 'u', 'd', 'e')
print('n' in my_tuple) # Output: True
print('a' in my_tuple) # Output: False
```

D. **min() and max() Function:** Using the min() and max() function, we can determine the tuple's minimum and maximum values.

```
#Using min() and max() Function
my_tuple = (1,2,3,4,5,6)
print (max(my_tuple)) # Output: 6
print (min(my_tuple)) # Output: 1
```

E. **Iterating in a Tuple:** We can use the for loop to iterate through a tuple as follows:

```
# Using for loop to iterate through a tuple
for name in ('Joshua', 'Mike'):
   print("Hello", name)
Output
Hello Joshua
Hello Mike
```

6. SETS

Unlike a tuple, a Set is an unorganized and unindexed collection of objects. The items in a set do not have a defined order and may appear in a different order every time we use them. Hence, they cannot be accessed or referred to using a key or index. Every value in a set is unique and unchangeable. Set elements are unordered. That is, there is no specific order in which the elements appear. Set features restrict duplicate values. In addition, we can add additional elements to the existing set. Set hold its elements in curly brackets "{...}".

6.1. Creating a set

As mentioned above, we can create a set by enclosing elements in curly brackets separated by a comma.

A set can store any number of items, and they can be of different data types (boolean, integer, string, etc.). However, a set cannot have a mutable element like lists or dictionaries.

```
# Set with mixed data types
my_set = {5.0, "H1", (1, 2, 3)}
print(my_set)
```

6.2. set() Method

The set() method or set construct is used to convert an iterable sequence into a set. We can also create sets using the built-in set() function in Python. It follows the following syntax.

```
set<iterable>
```

The "iterable" in the above syntax may be a list, a dictionary, a string, or a tuple. We can create an empty set using the set() method if we do not specify any argument/iterable in it.

```
# Creating an empty set
a = set()
```

We can create a set from a string in the following manner.

```
#Set from a string
my_set = (set('Language'))
print(my_set)
```

Output:

```
{ 'L', 'n', 'u', 'a', 'g', 'e' }
```

We can create a set from a tuple in the following manner.

```
my_set = (set(('a', 'b', 'o', 'a', 'r', 'd')))
print(my_set)
```

Output:

```
{'a', 'b', 'd', 'o', 'r'}
```

We can create a set from a list in the following manner.

```
#Set from a List
my_set = set([1, 2, 3, 2])
print(my_set)
```

Output:

```
{1,2,3}
# The last '2' is not included as a set does not allow duplicates.
```

6.3. add()

Although sets are mutable, they are unorganized. Hence, indexing has no meaning, and we cannot use indexing or slicing to access elements in sets.

We can add a new item in a set using the add() method. The add() method is used with the following syntax:

```
set.add(elem)
```

Where "elem" is the element that we want to add. We cannot get back a set if we use the add() method while creating a new set.

```
noneValue = set().add(elem)
```

The above expression will return "none" as that is the return type of add().

Let us look at an example of how to add an element to a set.

```
letter = {'c', 'o', 'p', 'p', 'e'}
# adding 'r'
letter.add( 'r' )
print('The letters are:', letter)
```

Output:

```
The letters are: {'p', 'c', 'r', 'e', 'o'}
# The order of the letters may be different.
```

We can add a tuple to a set as:

```
# adding a tuple to a set
letter = {'c', 'o', 'p'}
# the tuple
tup = ('p', 'e', 'r')
# using add()
letter.add(tup)
print('The letters are:', letter)
```

Output:

```
The letters are: { ('p', 'e', 'r'), 'c', 'o', 'p'}
```

6.4. update()

The update() method is similar to the add() method. However, we can add multiple elements at the same time using the update() method. The update() method can accept any iterable like tuples, lists, strings, or other sets. It has the following syntax:

```
set.update(iterable1, iterable2, iterable3)
```

Similar to the add() method, update() does not return any value. Let us see how the update() is used to add another set to an existing one.

```
letter = {'p', 'y', 't'}
let = {'t', 'h', 'o', 'n'}
# using update()
```

```
letter.update(let)
print ("The word is ", letter)
```

Output:

```
The word is {'p', 'y', 't', 'h', 'o', 'n' }
```

6.5 discard()

In Python, the library function discard() is programmed to eliminate specific elements from the set, given the elements are available in the set. Discard() function will not display error if the element specified in the discard() function is not available in the iterable unlike remove() function.

Syntax:

```
set.discard(x)
```

Where "x" is the element to be removed, the discard() method does not have a return value.

We can use discard() in the following ways:

Method 1:

```
num = {2, 6, 4, 5, 3}
num.discard(3)
print('numbers = ', num)
```

Output:

```
numbers = {2, 6, 4, 5}
```

The output remains the same even if we use the code below:

```
num.discard(15)
print('numbers = ', num)
```

Output:

```
numbers = {2, 6, 4, 5}
```

Method 2:

```
num = {2, 6, 4, 5, 3}
print(num.discard(3))
# The above expression returns "None"
print('numbers = ', num)
```

Output:

```
None

numbers = {2, 4, 5, 6}
```

7. DICTIONARY

Dictionaries are central data structures in Python that store an ordered collection of items. Each item in a dictionary is mutable and identified using a key name (or key) with an associated value. Unlike an index, a key can be of any data type, such as string, float, integer, etc., making it more flexible.

Dictionaries can store any number of items and are also referred to as maps, hashmaps, lookup tables, or associative arrays. In the real world perspective, dictionaries can be associated with phonebooks. They help you quickly retrieve information, such as a phone number, with a specific key (person's name).

7.1. Creating a Dictionary

As mentioned above, each element in a dictionary is tied to a key. The corresponding key is used to get/view that particular element in the dictionary. Compared to conventional programming languages, Python offers the flexibility to select the desired key for elements. However, the key must be unique and immutable data types such as string, number, or tuple. On the other hand, the item associated with the key may be of any data type.

A dictionary has the following syntax:

variable = {key1 : item1, key2 : item2...}

The key and its value are separated by a colon (:), the unit items are separated by commas, and the whole sequence is enclosed within curly brackets.

We can create an empty dictionary in the following manner:

empty = {}

7.2. Accessing Values of a Dictionary

Unlike other ordered data structures that use indexing, dictionaries use keys to access or retrieve items. We can access the values in the dictionary by using the square bracket ("[...]") or the get() method.

If a value does not exist in the dictionary and we use the square brackets, the program raises a KeyError. In contrast, the get() method returns a "None" value.

Example:

```
# retrieving elements
my_dict = {'name': 'Phil', 'age': 50, 'country': 'America'}
# Accessing value using []
print(my_dict['name']) #Output: Phil
# Accessing value using get()
print(my_dict.get('country')) #Output: America
```

7.3. Adding and Deleting Values in a Dictionary

In Python, we can append a new element/item or alter the existing value using the assignment operator "=". To add or append a new item, a new key and associated value should be constructed in the syntax. In order to alter the existing element describes the new value for the existing key of the dictionary in the syntax. .

```
# Adding and updating values in a dictionary
my_dict = {'name': 'Phil', 'age': 50}
# Adding new key : value pair
my_dict['country'] = 'America'
# "country" is the key and "America" is its value
print(my_dict)
```

Output:

```
{'name': 'Phil', 'age': 50, 'country': 'America'}
```

```
# Changing an existing value
my_dict['age'] = 53
print(my_dict)
```

Output:

```
{'name': 'Phil', 'age': 53, 'country': 'America'}
```

We can delete the items from a dictionary using the del keyword with the key enclosed in square brackets.

```
my_dict = {10 : 'rose', 20 : 'lily', 30 : 'tulip', 40 : 'lotus', 50 : 'marigold'}
# Deleting item using del keyword
del my_dict [30]
print(my_dict)
```

Output:

```
{10: 'rose', 20: 'lily', 40: 'lotus', 50: 'marigold'}
```

If you wish to remove an item and return the value associated with it, you can use the pop() method.

```
my_dict = {10 : 'rose', 20 : 'lily', 30 : 'tulip', 40 : 'lotus', 50 : 'marigold'}
# Deleting item using pop() method
print(my_dict.pop(40))
```

Output:

```
lotus
```

The popitem() method is used to delete the last item in the dictionary and return its key and value. Using the clear() method, delete all the items in a dictionary.

```
my_dict = {10 : 'rose', 20 : 'lily', 30 : 'tulip', 40 : 'lotus', 50 : 'marigold'}
# Deleting item using popitem() method
print(my_dict.popitem())
```

Output:

```
(50, 'marigold')
```

```
# Deleting all the items
print(my_dict.clear())
```

Output:

None

7.4. Iteration in a Dictionary

Similar to lists, we can retrieve multiple elements from a dictionary by using the for loop.

```
# Retrieving values by using for loop
my_dict = {'name': 'Phil', 'country': 'America'}
for i in my_dict:
    print ("Key: " + i + " and Value: " + my_dict[i]);
# Using the variable 'i' directly will return only the key in the dictionary and not the value.
# The key is used just like an index
```

Output:

```
Key: name and Value: Phil
Key: country and Value: America
```

You cannot directly concatenate an integer with a string when using the for loop. However, we can use the format() method in the following manner.

```
my_dict = {'name': 'Phil', 'age': 50, 'country': 'America'}
for i in my_dict:
print ("Key: {0} and Value: {1}".format(i, my_dict[i]));
```

Output:

```
Key: name and Value: Phil
Key: age and Value: 50
Key: country and Value: America
```

7.5. Built-In Functions of Dictionary

Python offers some functions in a dictionary to perform tasks.

all():

The python library function ‘all()’ checks whether all entries or items in the iterable is true. If the iterable contains completely true element, the any() function returns ‘true’ whereas it returns a 'false' if there is any one false element in the iterable object, that is iterable contains a 0 or false. If there is no element in the object, then the all() returns "true".

Syntax:

```
all(iterable)
---
my_dict = {10 : 'rose', 20 : 'lily', 30 : 'tulip', 40 : 'lotus', 50 : 'marigold'}
print(all(my_dict))
```

Output :

```
True
# All keys are True
---
my_dict = {0 : 'rose', 20 : 'lily', 30 : 'tulip', 40 : 'lotus', 50 : 'marigold'}
print(all(my_dict))
```

Output :

```
False
# 0 is False
```

any(): The python library function 'any()' checks whether minimum one entry/item in the iterable is true. If the iterable contains atleast one true element, the any() function returns 'true' whereas it returns a 'false' if there is no true element in the iterable object. That is the iterable contain only either 0 or false.

Syntax:

```
any(iterable)
---
my_dict = {10 : 'rose', 20 : 'lily', 30 : 'tulip', 40 : 'lotus'}
print(any(my_dict))
```

Output :

```
True
# All keys are True
```

```
my_dict = {}
print(any(my_dict))
```

Output :

```
False
```

# Empty dictionary

len(): It returns the number of items (length) in a dictionary.

```
my_dict = {10 : 'rose', 20 : 'lily', 30 : 'tulip', 40 : 'lotus'}
print(len(my_dict))
```

Output :

```
4
```

cmp(): Compares the items of two dictionaries. However, this function is not available in Python 3.

sorted(): Python has the library function to easily sort any sequence, ‘sorted()’. Sorted() function gives sorted output and it does not alter the original sequence.

The syntax is:

sorted(iterable, key, reverse)

The parameter ‘iterable’ is mandatory. Iterable disclose the object to be sorted. While key and reverse parameters are optional. If the reverse parameter is given value true, then reverse sorting takes place..

```
# Sorting the key as a list
my_dict = {10 : 'rose', 20 : 'lily', 30 : 'tulip', 40 : 'lotus'}
print(sorted(my_dict))
```

Output:

```
[10, 20, 30, 40]
```

```
# Using the reverse parameter

my_dict = {10 : 'rose', 20 : 'lily', 30 : 'tulip', 40 : 'lotus'}

print(sorted(my_dict, reverse = True))

Output:
[40, 30, 20, 10]
# Reverse key order
---
# Using the key function
my_dict = {'10' : 'rose', '2000' : 'lily', '30' : 'tulip', '400' : 'lotus'}
print(sorted(my_dict, key = len))

Output:

['10', '30', '400', '2000']

# Thelen function sorts the keys based on their length
```

Items(): The items() function returns a view object that displays a list of dictionary's key-value tuple pairs. The view object contains the key-value pairs in the form of tuples, and any changes made to the dictionary will reflect in the view object.

Example:

```
# Define a dictionary
student_scores = {'John': 85, 'Emma': 92, 'Kelly': 78}

# Use the items() function
items_view = student_scores.items()

# Display the items of the dictionary
print(items_view)


# Output: dict_items([('John', 85), ('Emma', 92), ('Kelly', 78)])
```

In this example, items_view will contain the items of student_scores in the form of key-value pairs. If we add or remove an item from student_scores, items_view will automatically update to reflect these changes.

keys(): The keys() function returns a view object that displays a list of all the keys in the dictionary. Like the items() function, this view object is dynamic and reflects changes made to the dictionary.

Example:

```
# Define a dictionary
student_ages = {'John': 17, 'Emma': 18, 'Kelly': 16}

# Use the keys() function
keys_view = student_ages.keys()

# Display the keys of the dictionary
print(keys_view)

# Output: dict_keys(['John', 'Emma', 'Kelly'])
```

8. SUMMARY

In Unit 5, we dove into the world of Lists, Tuples, and more, expanding our Python toolkit. Let's unpack what we've learned in a more conversational tone.

Starting with Lists, these versatile data structures became our best friends for holding an ordered collection of items. Unlike their immutable cousin, the Tuple, lists are all about flexibility. We can add, remove, or change their elements, making them perfect for tasks where our data is dynamic. Remember how we played around with indexing and slicing? Accessing elements, whether it's pulling out a single item or a range, became a breeze. And let's not forget the fun we had updating lists, where we could change the list's content on the fly, making our code adaptable.

But then, we encountered the more reserved Tuples. These immutable collections taught us the value of constants in programming. Once a tuple is created, it's set in stone, which is great for ensuring that certain data remains unchanged throughout the program. We also looked at how tuples use indexing and slicing, similar to lists, but with the added security of immutability. It's like having a safe where you can view your valuables through a glass window, but you can't touch them.

Now, let's chat about Sets. If Lists and Tuples were about order and structure, Sets threw that out the window. They're all about uniqueness and unorder. Trying to find or remove duplicates from a list? A set has got your back. And with operations like union, intersection, and difference, sets became our go-to for handling unique collections of items.

Dictionaries were next, and boy, were they a game-changer! Think of them as personal assistants, storing information in a key-value pair format. Need to look up something? Just ask for it by its key, and voilà, the value appears. Adding and updating entries were straightforward, empowering us to build more complex and organized data structures.

Throughout Unit 5, we also got hands-on with various built-in functions and operations for each data type, enhancing our ability to manipulate and interrogate our data structures effectively. Whether it was iterating over items, checking for membership, or employing functions like len(), max(), and min(), we gained a toolkit that made working with data in Python not just efficient but also enjoyable.

9. GLOSSARY

- List: An ordered, mutable collection of elements enclosed in square brackets []. Lists can contain items of different data types.
- Tuple: An ordered, immutable collection of elements enclosed in parentheses (). Tuples support mixed data types and nesting but cannot be modified after creation.
- Indexing: The process of accessing individual items in a list or tuple using their position, starting from 0 for the first element.
- Slicing: A method to extract a portion of a list or tuple using a range of indices, specified with a colon : separator.
- Negative Indexing: Accessing elements from the end of a list or tuple, with -1 representing the last element.
- Set: An unordered collection of unique elements enclosed in curly braces {}. Sets do not support duplicate values and are mutable.
- Dictionary: A collection of key-value pairs enclosed in curly braces {}. Keys must be unique and immutable.
- Mutable: A property of data structures that allows their content to be changed after creation. Lists, Dictionary and sets are mutable.
- Immutable: A property of data structures that prevents their content from being changed after creation. Tuples are immutable.
- Membership Operators: Operators (in, not in) used to check if a value exists within a list, tuple, or set.
- Iteration: The process of looping through each item in a list, tuple, set, or dictionary to perform operations.
- Concatenation: The operation of joining two lists or tuples using the + operator.
- Repetition: Repeating the elements of a list or tuple a specified number of times using the * operator.
- add() Method: A set method used to add a single element to a set.
- update() Method: A set method used to add multiple elements to a set.
- discard() Method: A set method used to remove a specified element from a set without raising an error if the element does not exist.

- len() Function: A built-in function that returns the number of elements in a list, tuple, set, or dictionary.
- max() Function: A built-in function that returns the largest value in a list, tuple, or set.
- min() Function: A built-in function that returns the smallest value in a list, tuple, or set.
- del Keyword: Used to delete elements in a list, an entire list, tuple, set, or specific key-value pairs in a dictionary.

10. SELF-ASSESSMENT QUESTIONS

1. In Python, lists are created by placing elements inside square brackets [ ], separated by ______.
2. The process of accessing multiple elements in a sequence is known as ______.
3. In Python, the first element of a list has an index of ______.
4. Tuples are ______ data structures, which means their elements cannot be changed once assigned.
5. To add an element to a set, the ______ method is used.
6. Dictionaries in Python store data in key-value pairs, where each key must be ______.
7. A single element in a tuple can be defined by following the element with a ______.
8. The ______ function is used to determine the number of items in a list.
9. Which of the following is used to access a range of elements in a list?
   A) ()
   B) []
   C) {}
   D) <>
10. What will be the output of the following code snippet? a_list = [1, 2, 3]; print(a_list[-1])
    A) 1
    B) 2
    C) 3
    D) Error
11. Which method is used to add an item to the end of a list?
    A) append()
    B) extend()
    C) insert()
    D) add()
12. How can you create a set with the elements 'a', 'b', and 'c'?
    A) set('a', 'b', 'c')
    B) {'a', 'b', 'c'}
    C) set(['a', 'b', 'c'])

D) Both B and C are correct.

13. What will be the output of the following code snippet? my_tuple = (1, 2, 3); print(my_tuple[1])
    A) 1
    B) 2
    C) 3
    D) (2, 3)
14. In a dictionary, keys must be:
    A) Immutable
    B) Mutable
    C) Of any data type
    D) None of the above
15. To remove an item from a set, you can use the _______ method.
    A) remove()
    B) discard()
    C) delete()
    D) Both A and B are correct.

11. TERMINAL QUESTIONS

1. Explain the difference between lists and tuples in Python with examples.
2. Describe how you can convert a list to a tuple and vice versa.
3. Discuss the importance of negative indexing in Python sequences and provide examples where it can be particularly useful.
4. How can you use slicing to reverse a list or a tuple in Python? Provide code examples.
5. What is the significance of the None value in lists, and how is it different from an empty list?
6. Explain with examples how the extend() method differs from the append() method when used with lists.
7. In what scenarios would you prefer using a tuple over a list in Python?
8. Discuss the concept of immutability in Python with reference to tuples. How does it affect the way tuples are used?
9. How can sets be used to efficiently remove duplicates from a list? Provide a code snippet to illustrate this.
10. Describe the key characteristics that differentiate sets from lists and tuples in Python.
11. Write a Python function that takes a list of numbers and returns a new list containing only the even numbers from the original list.
12. Create a Python program that takes a tuple of strings and returns a concatenated string made up of the elements of the tuple.
13. Develop a Python script that combines two lists by alternatingly taking elements. For example, list_combine(['a','b'], [1,2]) should return ['a', 1, 'b', 2].
14. Write a Python function that accepts a set and returns a sorted list of elements from that set.
15. Implement a Python program that demonstrates the use of nested tuples and accesses elements from the nested tuples.
16. Create a Python script that converts a multi-dimensional list into a flat list.
17. Write a Python function that removes all occurrences of a specified value from a list without using the built-in remove() method.
18. Write a Python function to count the occurrences of each element in a list and return the result as a dictionary.

19. Implement a Python script that demonstrates the use of enumerate() in a list to access both the index and the value of each element.

12. ANSWERS

Self-Assessment Questions

1. commas
2. slicing
3. 0
4. immutable
5. add
6. unique
7. comma
8. len
9. B) []
10. C) 3
11. A)append()
12. D) Both B and C are correct.
13. 2
14. A)Immutable
15. D) Both A and B are correct.

Terminal Questions

1. Refer to "2. Lists" for the explanation and examples differentiating lists and tuples.
2. For converting between lists and tuples, see examples in "5.1 Creating a Tuple" and "2. Lists".
3. Negative indexing is explained in "2.1 Indexing and Splitting" and "5.3 Negative Indexing".
4. Slicing to reverse sequences is covered in "2.1 Indexing and Splitting" for lists and "5.2 Indexing and Slicing" for tuples.
5. The usage of None in lists is discussed in "3. List Operators".
6. Differences between extend() and append() methods are explained in "3.6 Adding Elements to a List".
7. Preferences for tuples over lists due to immutability are detailed in "5. Tuple".

8. The concept of immutability with reference to tuples is elaborated in "5. Tuple".
9. Removing duplicates using sets is demonstrated in "6. Sets".
10. Key differences between sets, lists, and tuples are outlined in "6. Sets".
11. For even number filtering, use list comprehensions as shown in "3.4 Iteration".
12. String concatenation from tuples can be derived from "5.2 Indexing and Slicing".
13. Combining lists alternately is a practical application of list indexing covered in "2. Lists".
14. Converting a set to a sorted list involves basic set operations and the sorted() function, see "6. Sets".
15. Accessing elements in nested tuples is explained in "5.2 Indexing and Slicing".
16. Flattening a list involves nested loops or comprehensions, similar to those in "3.4 Iteration".
17. Removing occurrences without remove() would use list comprehensions, refer to "3.4 Iteration".
18. Counting occurrences and returning a dictionary is related to "7. Dictionary" and list handling in "2. Lists".
19. The use of enumerate() is akin to list iteration methods in "3.4 Iteration".