MODULE - 4

- **LISTS:** BASIC LIST OPERATIONS, INDEXING AND SLICING IN LISTS, BUILT- IN FUNCTIONS USED ON LISTS, LIST METHODS.
- **DICTIONARIES:** CREATING DICTIONARY, ACCESSING AND MODIFYING KEY: VALUE PAIRS IN DICTIONARIES, BUILT-IN FUNCTIONS USED ON DICTIONARIES, DICTIONARY METHODS.
- **TUPLES AND SETS:** BASIC TUPLE OPERATIONS, INDEXING AND SLICING IN TUPLES, BUILT-IN FUNCTIONS USED ON TUPLES, RELATION BETWEEN TUPLES AND LISTS, RELATION BETWEEN TUPLES AND DICTIONARIES, TUPLE METHODS.
- SETS, SET METHODS, TRAVERSING OF SETS.

LISTS []

- ✓ CREATING LISTS
- ✓ BASIC LIST OPERATIONS
- ✓ INDEXING AND SLICING IN LISTS
- ✓ BUILT-IN FUNCTIONS USED ON LISTS
- ✓ LIST METHODS
- ✓ THE DEL STATEMENT

PYTHON COLLECTIONS (ARRAYS)

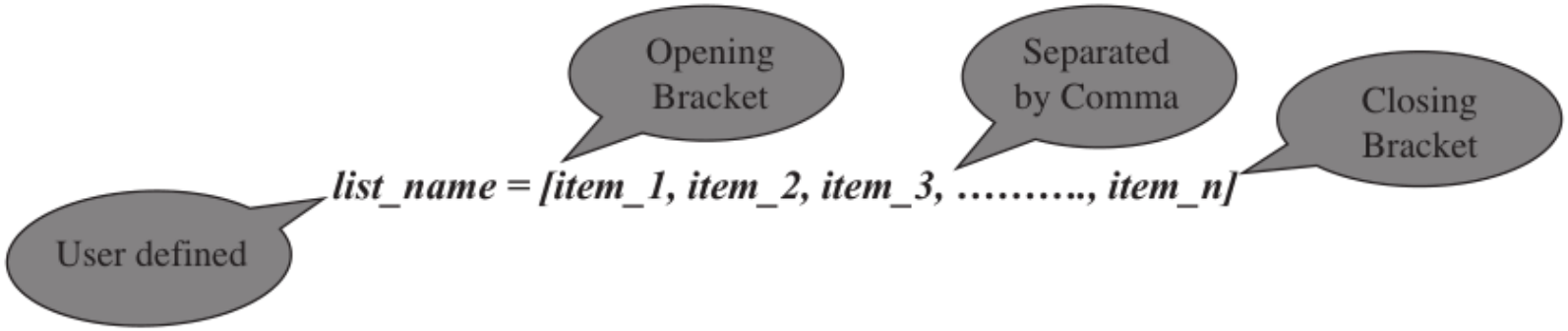
- ✓ THERE ARE FOUR COLLECTION DATA TYPES IN THE PYTHON PROGRAMMING LANGUAGE
  - ✓ **LIST:** IS A COLLECTION WHICH IS ORDERED AND CHANGEABLE. ALLOWS DUPLICATE MEMBERS.
  - ✓ **TUPLE** IS A COLLECTION WHICH IS ORDERED AND UNCHANGEABLE. ALLOWS DUPLICATE MEMBERS.
  - ✓ **SET** IS A COLLECTION WHICH IS UNORDERED AND UNINDEXED. NO DUPLICATE MEMBERS.
  - ✓ **DICTIONARY** IS A COLLECTION WHICH IS UNORDERED, CHANGEABLE AND INDEXED. NO DUPLICATE MEMBERS.

LISTS-INTRODUCTION

- Lists are one of the most flexible data storage formats in python because they can have values added, removed, and changed.
- You can think of the list as a container that holds a number of items.
- Each element or value that is inside a list is called an item. All the items in a list are assigned to a single variable.
- Lists avoid having a separate variable to store each item which is less efficient and more error prone when you have to perform some operations on these items.
- Lists can be simple or nested lists with varying types of values.

CREATING LISTS

Lists are constructed using square brackets [ ] wherein you can include a list of items separated by commas.




```
1  superstore = ["metro", "tesco", "walmart", "kmart", "carrefour"]
2  superstore
```

```
['metro', 'tesco', 'walmart', 'kmart', 'carrefour']
```

You can create an empty list without any items. The syntax is, **list_name = [ ]**

CREATING LISTS

- NUMBER_LIST = [4, 4, 6, 7, 2, 9, 10, 15]
- VALS=[12,'TIGER',10.5, TRUE]
- TYPE(VALS)
- EMPTY_LIST = []
- You can store any item in a list like string, number, object, another variable and even another list.
- You can have a mix of different item types and these item types need not have to be homogeneous.
- For example, you can have a list which is a mix of type numbers, strings and another list itself.

EXAMPLE OF SEQUENCES

Name sequence (C)

Position number of the element within sequence C

C[0]	-45	C[-12]
C[1]	6	C[-11]
C[2]	0	C[-10]
C[3]	72	C[-9]
C[4]	34	C[-8]
C[5]	39	C[-7]
C[6]	98	C[-6]
C[7]	-1345	C[-5]
C[8]	939	C[-4]
C[9]	10	C[-3]
C[10]	40	C[-2]
C[11]	33	C[-1]

BASIC LIST OPERATIONS

```
>>> LIST_1 = [1, 3, 5, 7]
>>> LIST_2 = [2, 4, 6, 8]
>>> LIST_1 + LIST_2
>>> LIST_1 * 3
>>> LIST_1 == LIST_2
>>> 5 IN LIST_1
>>> 10 IN LIST_1
```

THE *LIST()* FUNCTION

- The built-in *list()* function is used to create a list.
- The syntax for *list()* function is, ***LIST([SEQUENCE]),*** Where the sequence can be a string, tuple or list itself. If the optional sequence is not specified then an empty list is created.

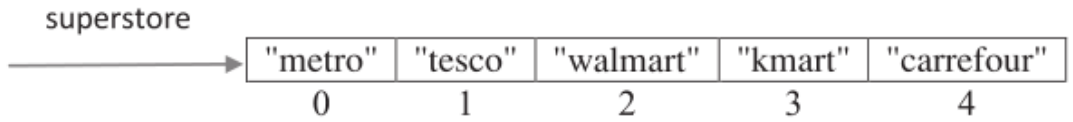
```
>>> QUOTE = "HOW YOU DOING?"

>>> STRING_TO_LIST = LIST(QUOTE)

>>> STRING_TO_LIST

>>> FRIENDS = ["J", "O", "E", "Y"]

>>> FRIENDS + QUOTE

>>> FRIENDS + LIST(QUOTE)
```

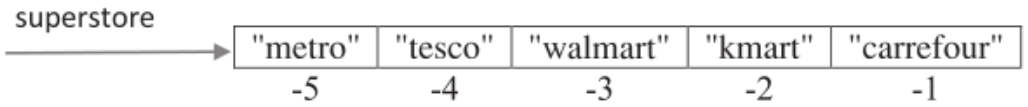
INDEXING AND SLICING IN LISTS

- As an ordered sequence of elements, each item in a list can be called individually, through indexing.
- The expression inside the bracket is called the index.
- Lists use square brackets [ ] to access individual items, with
  - the first item at index 0,
  - the second item at index 1 and so on.
  - The index provided within the square brackets indicates the value being accessed.
- The syntax for accessing an item in a list is, ***LIST_NAME[INDEX]*** , Where index should always be an integer value and indicates the item to be selected

INDEXING AND SLICING IN LISTS

- . For the list *superstore*, the index breakdown is shown below.






```
1  superstore = ["metro", "tesco", "walmart", "kmart", "carrefour"]
2  print(superstore[0])
3  print(superstore[1])
4  print(superstore[2])
5  print(superstore[3])
6  print(superstore[4])
7  print(superstore[9])
```

```
metro
tesco
walmart
kmart
carrefour

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-3ceed56b913f> in <module>
      5 print(superstore[3])
      6 print(superstore[4])
----> 7 print(superstore[9])

IndexError: list index out of range
```

- SUPERSTORE = ["METRO", "TESCO", "WALMART", "KMART", "CARREFOUR"]
- >>> SUPERSTORE[0]
- >>>SUPERSTORE[9]
- >>>SUPERSTORE[-3]

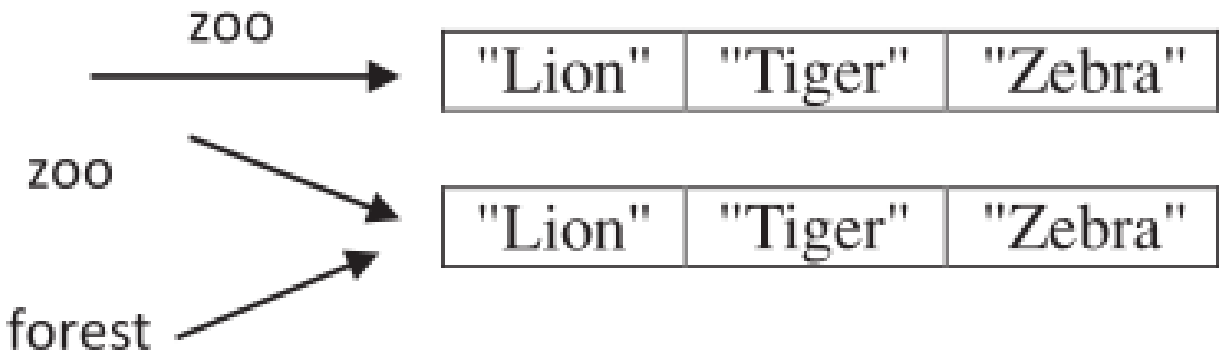
MODIFYING ITEMS IN LISTS

Lists are mutable in nature as the list items can be modified after you have created a list. You can modify a list by replacing the older item with a newer item in its place and without assigning the list to a completely new variable.

- >>> FAUNA = ["PRONGHORN", "ALLIGATOR", "BISON"]
- >>> FAUNA[0] = "GROUNDHOG"
- >>> FAUNA[-1] = "BEAVER"

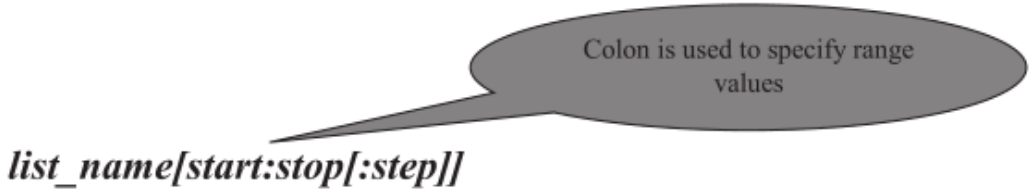
When you assign an existing list variable to a new variable, an assignment (=) on lists does not make a new copy. Instead, assignment makes both the variable names point to the same list in memory.

```
>>> ZOO = ["LION", "TIGER", "ZEBRA"]
>>> FOREST = ZOO
>>> ID(FOREST)
>>> ID(ZOO)
>>>ZOO[0] = "FOX“
>>> ZOO
>>> FOREST
>>> TYPE(ZOO)
```



SLICING IN LISTS

- Slicing of lists is allowed in python wherein a part of the list can be extracted by specifying index range along with the colon (:) operator which itself is a list.
- The syntax for list slicing is,

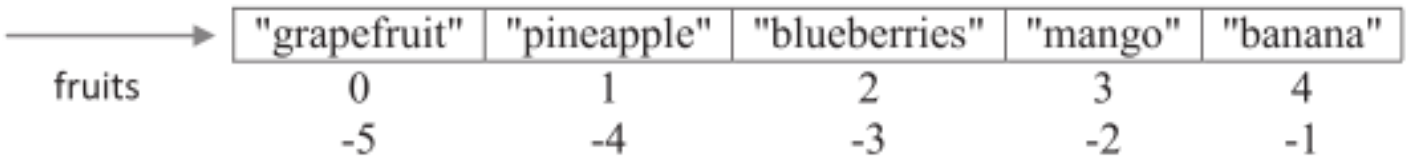


Where both start and stop are integer values (positive or negative values).

- List slicing returns a part of the list from the start index value to stop index value which includes the start index value but excludes the stop index value.
- Step specifies the increment value to slice by and it is optional.

SLICING IN LISTS

- >>> FRUITS = ["GRAPEFRUIT", "PINEAPPLE", "BLUEBERRIES", "MANGO", "BANANA"]
- >>> FRUITS[1:3]
- >>> FRUITS[:3]
- >>> FRUITS[2:]
- >>> FRUITS[1:4:2]
- >>> FRUITS[:]
- >>> FRUITS[::2]
- >>> FRUITS[-3:-1]



BUILT-IN FUNCTIONS USED ON LISTS

Built-In Functions Used on Lists

Built-In Functions	Description
len()	The *len()* function returns the numbers of items in a list.
sum()	The *sum()* function returns the sum of numbers in the list.
any()	The *any()* function returns *True* if any of the Boolean values in the list is *True*.
all()	The *all()* function returns *True* if all the Boolean values in the list are *True*, else returns *False*.
sorted()	The *sorted()* function returns a modified copy of the list while leaving the original list untouched.

BUILT-IN FUNCTIONS USED ON LISTS

```
>>> LAKES = [‘DAL', ‘KRISHNA', ‘SARAKKI', ‘BELLANDUR', ‘UTTARAHALLI']
>>> LEN(LAKES)
>>> NUMBERS = [1, 2, 3, 4, 5]
>>> SUM(NUMBERS)
>>> MAX(NUMBERS)
>>> MIN(NUMBERS)
>>> ANY([1, 1, 0, 0, 1, 0])
>>> ANY([ 0, 0, 0, 0])
>>> ANY([ 0, 1, 0, 0])
>>> ALL([1, 1, 1, 1])
>>> LAKES_SORTED_NEW = SORTED(LAKES)
```

LIST METHODS

- The list size changes dynamically whenever you add or remove the items and there is no need for you to manage it yourself.
- dir(list) displays all the methods associated with the *list* by passing the list function to *dir()*.

['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

Various List Methods

List Methods	Syntax	Description
append()	list.append(item)	The *append()* method adds a single item to the end of the list. This method does not return new list and it just modifies the original.
count()	list.count(item)	The *count()* method counts the number of times the item has occurred in the list and returns it.
insert()	list.insert(index, item)	The *insert()* method inserts the item at the given index, shifting items to the right.
extend()	list.extend(list2)	The *extend()* method adds the items in list2 to the end of the list.
index()	list.index(item)	The *index()* method searches for the given item from the start of the list and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the list then *ValueError* is thrown by this method.
remove()	list.remove(item)	The *remove()* method searches for the first instance of the given item in the list and removes it. If the item is not present in the list then *ValueError* is thrown by this method.
sort()	list.sort()	The *sort()* method sorts the items *in place* in the list. This method modifies the original list and it does not return a new list.
reverse()	list.reverse()	The *reverse()* method reverses the items *in place* in the list. This method modifies the original list and it does not return a new list.
pop()	list.pop([index])	The *pop()* method removes and returns the item at the given index. This method returns the rightmost item if the index is omitted.

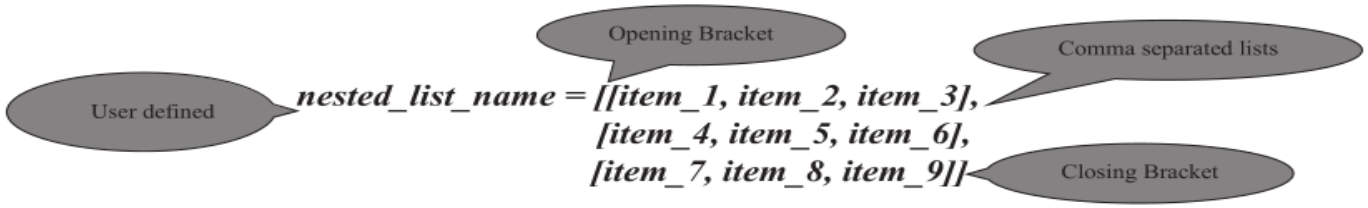
Note: Replace the word *"list"* mentioned in the syntax with your *actual list name* in your code.

POPULATING LISTS WITH ITEMS

```
>>> CITIES = ["OSLO", "DELHI", "WASHINGTON", "LONDON", "SEATTLE", "PARIS", "WASHINGTON"]

>>> CITIES.COUNT('SEATTLE')

>>> CITIES.COUNT('WASHINGTON')

>>> CITIES.INDEX('LONDON')

>>> CITIES.REVERSE()

>>> CITIES.APPEND('BENGALURU')

>>> CITIES.SORT()

>>> CITIES.POP()

>>> CITIES.POP(2)

>>> MORE_CITIES = ["BOMBAY", "COPENHAGEN"]

>>> CITIES.EXTEND(MORE_CITIES)

>>> CITIES.REMOVE("BOMBAY")
```

NESTED LISTS

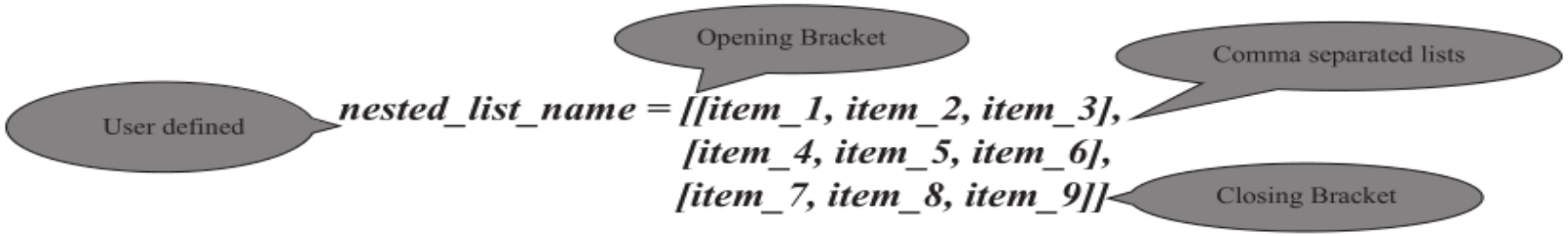
- The syntax for nested lists is,



- Each list inside another list is separated by a comma. For example,
- >>>ASIA = [["INDIA", "JAPAN", "KOREA"],["SRILANKA", "MYANMAR", "THAILAND"],["CAMBODIA", "VIETNAM", “BANGLADESH"]]
- ASIA[0]
- ASIA[0][1]
- ASIA[1][2] = "PHILIPPINES"
- ASIA
- >>> VALS=[12,'TIGER',10.5, TRUE]
- >>> NUMBERS = [1, 2, 3, 4, 5]
- >>> NEWLIST=[VALS,NUMBERS]

NESTED LISTS

- The syntax for nested lists is,



- Each list inside another list is separated by a comma. For example,

```
1 asia = [["India", "Japan", "Korea"],
2         ["Srilanka", "Myanmar", "Thailand"],
3         ["Cambodia", "Vietnam", "Israel"]]
4 print(asia[0])
5 print(asia[0][1])
6 asia[1][2] = "Philippines"
7 print(asia)
```

```
['India', 'Japan', 'Korea']
Japan
[['India', 'Japan', 'Korea'], ['Srilanka', 'Myanmar', 'Philippines'], ['Cambodia', 'Vietnam', 'Israel']]
```

TWO-DIMENSIONAL LISTS

- Two-dimensional list:

A list that contains other lists as its elements

  - Also known as nested list
  - Common to think of two-dimensional lists as having rows and columns
  - Useful for working with multiple sets of data

- To process data in a two-dimensional list need to use two indexes
- Typically use nested loops to process

A two-dimensional list

	Column 0	Column 1
Row 0	'Joe'	'Kim'
Row 1	'Sam'	'Sue'
Row 2	'Kelly'	'Chris'

Subscripts for each element of the scores list

	Column 0	Column 1	Column 2
Row 0	scores[0][0]	scores[0][1]	scores[0][2]
Row 1	scores[1][0]	scores[1][1]	scores[1][2]
Row 2	scores[2][0]	scores[2][1]	scores[2][2]

THE DEL STATEMENT

- You can remove an item from a list based on its index rather than its value.
- The difference between del statement and pop() function is that the del statement does not return any value while the pop() function returns a value.
- The del statement can also be used to remove slices from a list or clear the entire list

```
1  a = [5, -8, 99.99, 432, 108, 213]
2  del a[0]
3  print(a)
4  del a[2:4]
5  print(a)
6  del a[:]
7  print(a)
```

```
>>> a = [5, -8, 99.99, 432, 108, 213]
>>> del a[0]
>>> a
>>> del a[2:4]
>>> a
>>> del a[:]
>>> a
```

THE DEL STATEMENT

- You can remove an item from a list based on its index rather than its value.
- The difference between del statement and pop() function is that the del statement does not return any value while the pop() function returns a value.
- The del statement can also be used to remove slices from a list or clear the entire list

```
1  a = [5, -8, 99.99, 432, 108, 213]
2  del a[0]
3  print(a)
4  del a[2:4]
5  print(a)
6  del a[:]
7  print(a)
```

```
[-8, 99.99, 432, 108, 213]
[-8, 99.99, 213]
[]
```

SUMMARY

- Lists are a basic and useful data structure built into the python language.
- Built-in functions include
    - *Len()*, which returns the length of the list;
    - *Max()*, which returns the maximum element in the list;
    - *Min()*, which returns the minimum element in the list and
    - *Sum()*, which returns the sum of all the elements in the list.
- An individual elements in the list can be accessed using the index operator [].
- lists are mutable sequences which can be used to add, delete, sort and even reverse list elements.
- The *sort()* method is used to sort items in the list.
- The *split()* method can be used to split a string into a list.
- Nested list means a list within another list.

program to dynamically build user input as a list

```
list_items = input("enter list items separated by a space ").split()
print(f"list items are {list_items}")
items_of_list = []
total_items = int(input("enter the number of items "))
for i in range(total_items):
    item = input("enter list item: ")
    items_of_list.append(item)
print(f"list items are {items_of_list}")
```

TRAVERSING OF LISTS

```
#program to illustrate traversing of lists using the for loop


fast_food = ["waffles", "sandwich", "burger", "fries"]
for each_food_item in fast_food:
    print(f"i like to eat {each_food_item}")
for each_food_item in ["waffles", "sandwich", "burger", "fries"]:
    print(f"i like to eat {each_food_item}")
```

#program to display the index values of items in list

```
silicon_valley = ["google", "amd", "yahoo", "cisco", "oracle"]

for index_value in range(len(silicon_valley)):

    print(f"the index value of '{silicon_valley[index_value]}' is {index_value}")
```

WRITE PYTHON PROGRAM TO SORT NUMBERS IN A LIST IN ASCENDING ORDER USING BUBBLE SORT BY PASSING THE LIST AS AN ARGUMENT TO THE FUNCTION CALL

```
def bubble_sort(list_items):
    for i in range(len(list_items)):
        for j in range(len(list_items)-i-1):
            if list_items[j] > list_items[j+1]:
                temp = list_items[j]
                list_items[j] = list_items[j+1]
                list_items[j+1] = temp
    print(f"the sorted list using bubble sort is {list_items}")
items_to_sort = [5, 4, 3, 2, 1]
bubble_sort(items_to_sort)
```

FIND MEAN, VARIANCE AND STANDARD DEVIATION OF LIST NUMBERS

```python
import math
def statistics(list_items):
    mean = sum(list_items)/len(list_items)
    print(f"Mean is {mean}")

    variance = 0
    for item in list_items:
        variance += (item-mean)**2
    variance /= len(list_items)
    print(f"Variance is {variance}")

    standard_deviation = math.sqrt(variance)
    print(f"Standard Deviation is {standard_deviation}")

statistics([1, 2, 3, 4])
```

```
Mean is 2.5
Variance is 1.25
Standard Deviation is 1.118033988749895
```

```python
#Input Five Integers (+ve and -ve). Find the Sum of Negative Numbers,
#Positive Numbers and Print Them. Also, Find the Average of All the
#Numbers and Numbers Above Average

def find_sum(list_items):
    positive_sum = 0
    negative_sum =0

    for item in list_items:
        if item > 0:positive_sum = positive_sum + item
        else:negative_sum = negative_sum + item

    average = (positive_sum + negative_sum) / len(list_items)

    print(f"Sum of Positive numbers in list is {positive_sum}", )
    print(f"Sum of Negative numbers in list is {negative_sum}")
    print(f"Average of item numbers in list is {average}")
    print("Items above average are")

    for item in list_items:
        if item > average:print(item)

find_sum([-2, 4.2, 5, 6, -3])
```

```
Sum of Positive numbers in list is 15.2
Sum of Negative numbers in list is -5
Average of item numbers in list is 2.04
Items above average are
4.2
5
6
```

```python
# Write Python Program to Add Two Matrices
matrix_1 = [[1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]]
matrix_2 = [[1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]]
matrix_result = [[0, 0, 0],
                 [0, 0, 0],
                 [0, 0, 0]]

for rows in range(len(matrix_1)):
    for columns in range(len(matrix_2[0])):

        matrix_result[rows][columns] = matrix_1[rows][columns] + matrix_2[rows][columns]

print("Addition of two matrices is")
for items in matrix_result:
    print(items)
```

```
Addition of two matrices is
[2, 4, 6]
[8, 10, 12]
[14, 16, 18]
```

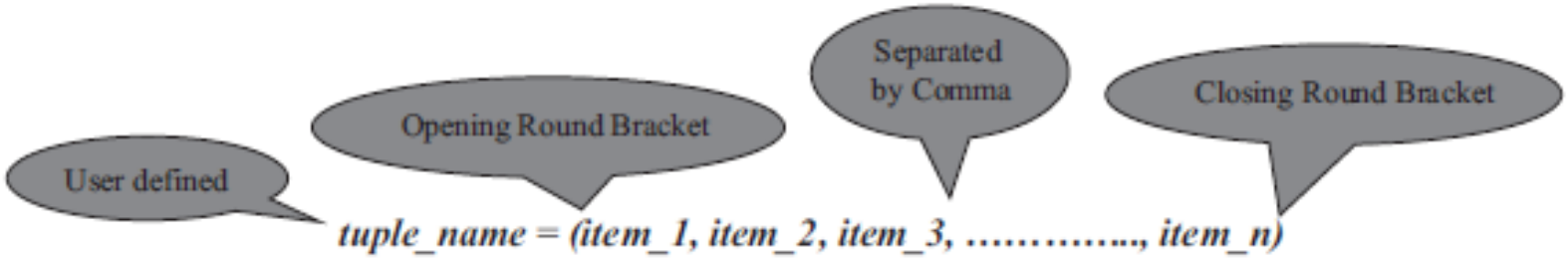
```
# WRITE PYTHON PROGRAM TO ADD TWO MATRICES

MATRIX_1 = [[1, 2, 3], [4, 5, 6],[7, 8, 9]]

MATRIX_2 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

MATRIX_RESULT = [[0, 0, 0], [0, 0, 0],[0, 0, 0]]

FOR ROWS IN RANGE(LEN(MATRIX_1)):

    FOR COLUMNS IN RANGE(LEN(MATRIX_2[0])):

        MATRIX_RESULT[ROWS][COLUMNS] = MATRIX_1[ROWS][COLUMNS] + MATRIX_2[ROWS][COLUMNS]

PRINT("ADDITION OF TWO MATRICES IS")

FOR ITEMS IN MATRIX_RESULT:

    PRINT(ITEMS)
```

```
#WRITE PYTHON PROGRAM TO PERFORM QUEUE OPERATIONS
FROM COLLECTIONS IMPORT DEQUE
DEF QUEUE_OPERATIONS():
    QUEUE = DEQUE(["ERIC", "JOHN", "MICHAEL"])
    PRINT(F"QUEUE ITEMS ARE {QUEUE}")
    PRINT("ADDING FEW ITEMS TO QUEUE")
    QUEUE.APPEND("TERRY")
    QUEUE.APPEND("GRAHAM")
    PRINT(F"QUEUE ITEMS ARE {QUEUE}")
    PRINT(F"REMOVED ITEM FROM QUEUE IS {QUEUE.POP()}")
    PRINT(F"REMOVED ITEM FROM QUEUE IS {QUEUE.POPLEFT()}")
    PRINT(F"QUEUE ITEMS ARE {QUEUE}")

QUEUE_OPERATIONS()
```

TUPLES ( ) : CREATING TUPLES

- In mathematics, a tuple is a finite ordered list (sequence) of elements. A tuple is defined as a data structure that comprises an ordered, finite sequence of immutable, heterogeneous elements that are of fixed sizes.
- A tuple is a finite ordered list of values of possibly different types which is used to bundle related values together without having to create a specific type to hold them.
- Tuples are immutable
- Once a tuple is created, you cannot change its values. A tuple is defined by putting a comma-separated list of values inside parentheses ( ).
- Each value inside a tuple is called an item.

SYNTAX FOR CREATING A TUPLE IS




```
>>> internet = ("cern", "timber", "www", 1980)
>>> type(internet)
<class 'tuple'>
>>> f1 = "ferrari", "redbull", "mercedes", "williams", "renault"
>>> f1
('ferrari', 'redbull', 'mercedes', 'williams', 'renault')
>>> type(f1)
<class 'tuple'>
```

EMPTY AND NESTED TUPLE

- You can create an empty tuple without any values. The syntax is, ***Tuple_name = ()***
- You can store any item of type string, number, object, another variable, and even another tuple itself. You can have a mix of different Types of items in tuples, and they need not be homogeneous.

```
>>> Air_force = ("f15", "f22a", "f35a")
>>> Fighter_jets = (1988, 2005, 2016, air_force)
>>> Fighter_jets
```

BASIC TUPLE OPERATIONS

```
>>> TUPLE_1 = (2, 0, 1, 4)
 >>> TUPLE_2 = (2, 0, 1, 9)
>>> TUPLE_1 + TUPLE_2
(2, 0, 1, 4, 2, 0, 1, 9)
>>> TUPLE_1 * 3
(2, 0, 1, 4, 2, 0, 1, 4, 2, 0, 1, 4)
 >>> TUPLE_1 == TUPLE_2
FALSE
```

```
>>> TUPLE_ITEMS = (1, 9, 8, 8)
>>> 1 IN TUPLE_ITEMS
TRUE
 >>> 25 IN TUPLE_ITEMS
FALSE
>>> TUPLE_1 = (9, 8, 7)
>>> TUPLE_2 = (9, 1, 1)
 >>> TUPLE_1 > TUPLE_2
TRUE
 >>> TUPLE_1 != TUPLE_2
TRUE
```

THE *TUPLE()* FUNCTION

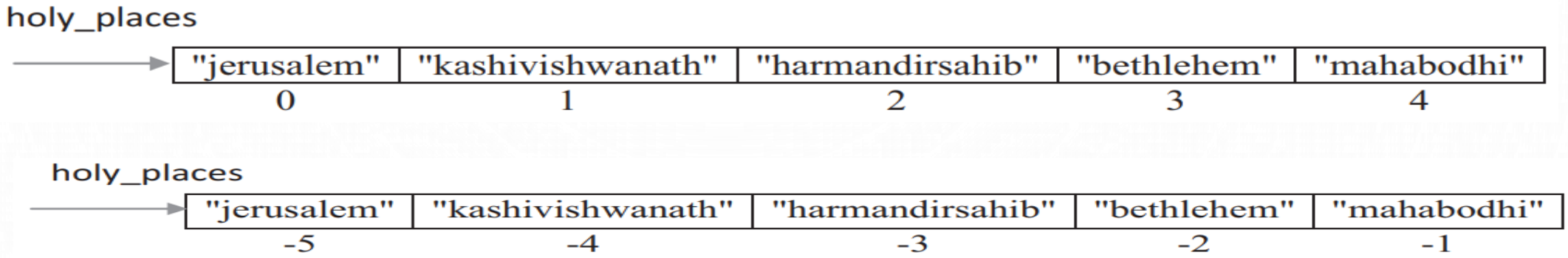
The built-in *tuple()* function is used to create a tuple. The syntax for the *tuple()* function is, ***tuple([sequence])***

Where the sequence can be a number, string or tuple itself.

```
>>> norse = "Vikings”
>>> String_to_tuple = tuple(norse)
>>> Zeus = ["g", "o", "d", "o", "f", "s", "k", "y"]
 >>> List_to_tuple = tuple(zeus)
>>> String_to_tuple + "scandinavia"
>>> String_to_tuple + tuple("scandinavia")
>>> Letters = ("a", "b", "c")
 >>> Numbers = (1, 2, 3)
>>> Nested_tuples = (letters, numbers)
>>> tuple("wolverine")
```

INDEXING AND SLICING IN TUPLES

- The syntax for accessing an item in a tuple is, ***tuple_name[index]*** **w**here index should always be an integer value and indicates the item to be selected.




```
>>> Holy_places = ("jerusalem", "kashivishwanath", "harmandirsahib", "bethlehem", "mahabodhi")
>>> Holy_places[0]
>>> Holy_places[5]
>>> Holy_places[0]==holy_places[-5]
```

SLICING IN TUPLES

- Slicing of tuples is allowed in python wherein a part of the tuple can be extracted by specifying an index range along with the colon (:) operator, which itself results as tuple type.
- The syntax for tuple slicing is, ***Tuple_name[start:stop[:step]]*** where both start and stop are integer values (positive or negative values).
- Tuple slicing returns a part of the tuple from the start index value to stop index value, which includes the start index value but excludes the stop index value. The step specifies the increment value to slice by and it is optional.

colors →

"v"	"i"	"b"	"g"	"y"	"o"	"r"
0	1	2	3	4	5	6
-7	-6	-5	-4	-3	-2	-1

```
>>> colors = ("v", "i", "b", "g", "y", "o", "r")
>>> colors[1:4]
>>> colors[:5]
>>> colors[3:]
>>> colors[::]
>>> colors[:]
>>> colors[1:5:2]
>>> colors[::-1]
>>> colors[-5:-2]
```

SLICING IN TUPLES

```
1  colors = ("v", "i", "b", "g", "y", "o", "r")
2  print(colors)
3  print(colors[1:4])
4  print(colors[:5])
5  print(colors[3:])
6  print(colors[:])
7  print(colors[::])
8  print(colors[1:5:2])
9  print(colors[::2])
10 print(colors[::-1])
11 print(colors[-5:-2])
```

```
('v', 'i', 'b', 'g', 'y', 'o', 'r')
('i', 'b', 'g')
('v', 'i', 'b', 'g', 'y')
('g', 'y', 'o', 'r')
('v', 'i', 'b', 'g', 'y', 'o', 'r')
('v', 'i', 'b', 'g', 'y', 'o', 'r')
('i', 'g')
('v', 'b', 'y', 'r')
('r', 'o', 'y', 'g', 'b', 'i', 'v')
('b', 'g', 'y')
```

BUILT-IN FUNCTIONS USED ON TUPLES

Built-In Functions Used on Tuples

Built-In Functions	Description
len()	The *len()* function returns the numbers of items in a tuple.
sum()	The *sum()* function returns the sum of numbers in the tuple.
sorted()	The *sorted()* function returns a sorted copy of the tuple as a list while leaving the original tuple untouched.

For example,

>>> years = (1987, 1985, 1981, 1996)

>>> len(years)

>>> sum(years)

>>> sorted_years = sorted(years)

>>> sorted_years

```
1  years = (1987, 1985, 1981, 1996)
2  print(len(years))
3  print(sum(years))
4  sorted_years = sorted(years)
5  print(sorted_years)
```

```
4
7949
[1981, 1985, 1987, 1996]
```

RELATION BETWEEN TUPLES AND LISTS

- Though tuples may seem similar to lists, they are often used in different situations and for different purposes.
- Tuples are immutable, and usually, contain a heterogeneous sequence of elements that are accessed via unpacking or indexing.
- Lists are mutable, and their items are accessed via indexing.
- Items cannot be added, removed or replaced in a tuple.

```
1 coral_reef = ("great_barrier", "ningaloo_coast", "amazon_reef", "pickles_reef")
2 coral_reef[0] = "pickles_reef"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-33d925c9c166> in <module>
      1 coral_reef = ("great_barrier", "ningaloo_coast", "amazon_reef", "pickles_reef")
----> 2 coral_reef[0] = "pickles_reef"

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

EXAMPLE

```
1 coral_reef = ("great_barrier", "ningaloo_coast", "amazon_reef", "pickles_reef")
2 coral_reef_list = list(coral_reef)
3 print(coral_reef_list)
```

```
['great_barrier', 'ningaloo_coast', 'amazon_reef', 'pickles_reef']
```

- If an item within a tuple is mutable, then you can change it.
- Consider the presence of a list as an item in a tuple, then any changes to the list get reflected on the overall items in the tuple.

```
1 german_cars = ["porsche", "audi", "bmw"]
2 european_cars = ("ferrari", "volvo", "renault", german_cars)
3 print(european_cars)
4 european_cars[3].append("mercedes")
5 print(german_cars)
6 print(european_cars)
```

```
('ferrari', 'volvo', 'renault', ['porsche', 'audi', 'bmw'])
['porsche', 'audi', 'bmw', 'mercedes']
('ferrari', 'volvo', 'renault', ['porsche', 'audi', 'bmw', 'mercedes'])
```

EXAMPLE

1. >>> German_cars = ["porsche", "audi", "bmw"]

2. >>> European_cars = ("ferrari", "volvo", "renault", german_cars)

3. >>> European_cars

('Ferrari', 'volvo', 'renault', ['porsche', 'audi', 'bmw'])

4. >>> European_cars[3].Append("mercedes")

5. >>> German_cars

['Porsche', 'audi', 'bmw', 'mercedes']

6. >>> European_cars

('Ferrari', 'volvo', 'renault', ['porsche', 'audi', 'bmw', 'mercedes'])

7>>>coral_reef = ("great_barrier", "ningaloo_coast", "amazon_reef", "pickles_reef")

>>> coral_reef[0] = "pickles_reef“

>>>coral_reef_list = list(coral_reef)

TUPLE METHODS

>>> DIR(TUPLE)

Various Tuple Methods

Tuple Methods	Syntax	Description
count()	tuple_name.count(item)	The *count()* method counts the number of times the item has occurred in the tuple and returns it.
index()	tuple_name.index(item)	The *index()* method searches for the given item from the start of the tuple and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the tuple, then *ValueError* is thrown by this method.

Note: Replace the word *"tuple_name"* mentioned in the syntax with your *actual tuple name* in your code.

```
1  channels = ("ngc", "discovery", "animal_planet", "history", "ngc")
2  print(channels.count("ngc"))
3  print(channels.index("history"))
```

```
2
3
```

TUPLE PACKING AND UNPACKING

- The statement t = 12345, 54321, 'hello!' Is an example of tuple packing.

```
1  t = 12345, 54321, 'hello!'
2  print(t)
```

```
(12345, 54321, 'hello!')
```

- The reverse operation of tuple packing is also possible. This operation is called tuple unpacking and works for any sequence on the right-hand side.

```
1  x, y, z = t
2  print(x)
3  print(y)
4  print(z)
```

```
12345
54321
hello!
```

Program to iterate over items in tuples using for loop

```
ocean_animals = ("electric_eel", "jelly_fish", "shrimp", "turtle", "blue_whale")

for i in ocean_animals:

    print(f"{i} is an ocean animal")
```

USING ZIP() FUNCTION

- The zip() function makes a sequence that aggregates elements from each of the iterables (can be zero or more).
- The syntax for zip() function is, ***Zip(*iterables)***
- An iterable can be a list, string, or dictionary. It returns a sequence of tuples, where the i-th tuple contains the i-th element from each of the iterables.

```
1  x = [1, 2, 3]
2  y = [4, 5, 6]
3  zipped = zip(x, y)
4  print(list(zipped))
```

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

```
>>> X = [1, 2, 3]

>>> Y = [4, 5, 6]

>>> ZIPPED = ZIP(X, Y)

>>> LIST(ZIPPED)

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

EXAMPLE 2:

```
USN=("17EC001","17EC002","17EC003","17EC004")

NAME=("ABC","DEF","IJK","LMN")

USN_NAME = LIST(ZIP(USN, NAME))

PRINT(USN_NAME)
```

EXAMPLE

- To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

```
1  questions = ('name', 'quest', 'favorite color')
2  answers = ('lancelot', 'the holy grail', 'blue')
3  for q, a in zip(questions, answers):
4      print(f'What is your {q}? It is {a}.')
```

```
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

POPULATING TUPLES WITH ITEMS

```
TUPLE_ITEMS = ()
TOTAL_ITEMS = INT(INPUT("ENTER THE TOTAL NUMBER OF ITEMS: "))
FOR I IN RANGE(TOTAL_ITEMS):
    USER_INPUT = INT(INPUT("ENTER A NUMBER: "))
    TUPLE_ITEMS += (USER_INPUT,)
PRINT(F"ITEMS ADDED TO TUPLE ARE {TUPLE_ITEMS}")


LIST_ITEMS = []
TOTAL_ITEMS = INT(INPUT("ENTER THE TOTAL NUMBER OF ITEMS: "))
FOR I IN RANGE(TOTAL_ITEMS):
    ITEM = INPUT("ENTER AN ITEM TO ADD: ")
    LIST_ITEMS.APPEND(ITEM)
ITEMS_OF_TUPLE = TUPLE(LIST_ITEMS)
PRINT(F"TUPLE ITEMS ARE {ITEMS_OF_TUPLE}")
```

Write python program to swap two numbers without using intermediate/temporary variables. Prompt the user for input

```
a = int(input("enter a value for first number "))

b = int(input("enter a value for second number "))

b, a = a, b

print("after swapping")

print(f"value for first number {a}, value for second number {b}")
```

Program to demonstrate the return of multiple values from a function

```
def return_multiple_items():
    monument = input("which is your favorite monument? ")
    year = input("when was it constructed? ")
    return monument, year
mnt, yr = return_multiple_items()
print(f"my favorite monument is {mnt} and it was constructed in {yr}")
result = return_multiple_items()
print(f"my favorite monument is {result[0]} and it was constructed in {result[1]}")
```

Write python code to sort a sequence of names according to their alphabetical order without using sort() function

```
DEF READ_LIST_ITEMS():
    PRINT("ENTER NAMES SEPARATED BY A SPACE")
    LIST_ITEMS = INPUT().SPLIT()
    RETURN LIST_ITEMS
DEF SORT_ITEM_LIST(ITEMS_IN_LIST):
    N = LEN(ITEMS_IN_LIST)
    FOR I IN RANGE(N):
        FOR J IN RANGE(1, N-I):
            IF ITEMS_IN_LIST[J-1] > ITEMS_IN_LIST[J]:
                (ITEMS_IN_LIST[J-1], ITEMS_IN_LIST[J]) = (ITEMS_IN_LIST[J], ITEMS_IN_LIST[J-1])
    PRINT("AFTER SORTING")
    PRINT(ITEMS_IN_LIST)


ALL_ITEMS = READ_LIST_ITEMS()
SORT_ITEM_LIST(ALL_ITEMS)
```

DICTIONARIES

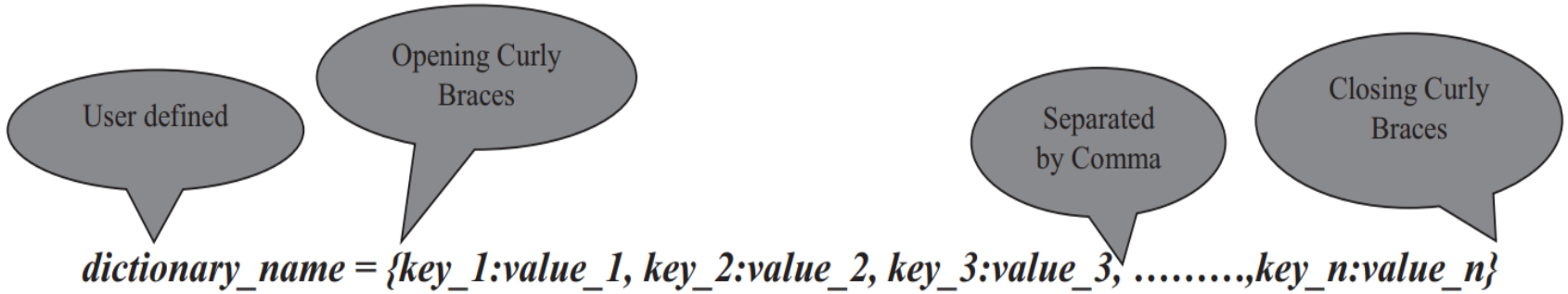
- Creating dictionary
- Accessing and modifying key : value pairs in dictionaries
- Built-in functions used on dictionaries
- Dictionary methods
- The del statement,

CREATING DICTIONARY

- A dictionary is a collection of an unordered set of key:value pairs, with the requirement that the keys are unique within a dictionary.
- Dictionaries are constructed using curly braces { }, wherein you include a list of key:value pairs separated by commas. Also, there is a colon (:) separating each of these key and value pairs, where the words to the left of the colon operator are the keys and the words to the right of the colon operator are the values.
- Unlike lists, which are indexed by a range of numbers, dictionaries are indexed by keys. Here a key along with its associated value is called a ***key:value*** pair.
- Dictionary keys are case sensitive.
- Dictionary keys are immutable type and can be either a string or a number. Since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend(), you cannot use lists as keys.
- Duplicate keys are not allowed in the dictionary.

CREATING DICTIONARIES

- The syntax for creating a dictionary is



- fish = {"g": "goldfish", "s":"shark", "n": "needlefish", "b":"barramundi","m":"mackerel"}

DICTIONARIES AND THEIR ASSOCIATED VALUES

```
>>> mixed_dict = {"portable":"laptop", 9:11, 7:"julius"}
>>> type(mixed_dict)
```

creating empty dictionaries:

```
>>> empty_dictionary = {}
>>> type(empty_dictionary)
```

```
>>> pizza = {"pepperoni":3, "calzone":5, "margherita":4}
>>> fav_pizza = {"margherita":4, "pepperoni":3, "calzone":5}
>>> pizza == fav_pizza
```

DICTIONARIES AND THEIR ASSOCIATED VALUES

```
>>> mixed_dict = {"portable":"laptop", 9:11, 7:"julius"}
>>> mixed_dict
{'portable': 'laptop', 9: 11, 7: 'julius'}
>>> type(mixed_dict)
<class 'dict'>
```

creating empty dictionaries:

```
>>> empty_dictionary = {}
>>> type(empty_dictionary)
<class 'dict'>
>>> pizza = {"pepperoni":3, "calzone":5, "margherita":4}
>>> fav_pizza = {"margherita":4, "pepperoni":3, "calzone":5}
>>> pizza == fav_pizza
```

ACCESSING AND MODIFYING KEY:VALUE PAIRS IN DICTIONARIES

- the syntax for accessing the value for a key in the dictionary is, ***dictionary_name[key]***
- the syntax for modifying the value of an existing key or for adding a new key:value pair to a dictionary is, ***dictionary_name[key] = value***
- if the key is already present in the dictionary, then the key gets updated with the new value.
- if the key is not present then the new key:value pair gets added to the dictionary.

ADDING AND MODIFYING KEY:VALUE PAIRS

```
>>> renaissance = {"giotto":1305, "donatello":1440, "michelangelo":1511,"botticelli":1480, "clouet":1520}

>>> renaissance["giotto"] = 1310

>>> renaissance["leonardo"]

>>> renaissance["michelangelo"]

>>> renaissance["leonardo"] = 1503

>>> renaissance
```

ADDING AND MODIFYING KEY:VALUE PAIRS

```
>>> renaissance = {"giotto":1305, "donatello":1440, "michelangelo":1511,"botticelli":1480, "clouet":1520}
>>> renaissance["giotto"] = 1310
>>> renaissance
>>> renaissance["leonardo"]
traceback (most recent call last):
  file "<pyshell#13>", line 1, in <module>  renaissance["leonardo"]
keyerror: 'leonardo'
>>> renaissance["michelangelo"]
>>> renaissance["leonardo"] = 1503
>>> renaissance
{'giotto': 1310, 'donatello': 1440, 'michelangelo': 1511, 'botticelli': 1480, 'clouet': 1520, 'leonardo': 1503}
```

CHECK FOR THE PRESENCE OF A KEY IN THE DICTIONARY

- You can check for the presence of a key in the dictionary using in and not in membership operators. It returns either a boolean true or false value.

```
>>> Clothes = {"rainy":"raincoats", "summer":"tees", "winter":"sweaters"}
>>> "Spring" in clothes
>>> "Spring" not in clothes
```

THE DICT() FUNCTION

- The built-in *dict()* function is used to create dictionary.
- The syntax for ***dict()*** function when the optional keyword arguments used is, *Dict([**kwarg])*

```
numbers = dict(one=1, two=2, three=3)
print(numbers)
```

```
{'one': 1, 'two': 2, 'three': 3}
```

- The syntax for ***dict()*** function when iterables used is, *Dict(iterable[, **kwarg])*
- You can specify an iterable containing exactly two objects as tuple, the key and value in the ***dict()*** function.

>>>dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

BUILT-IN FUNCTIONS USED ON DICTIONARIES

Built-In Functions Used on Dictionaries

Built-in Functions	Description
len()	The *len()* function returns the number of items (*key:value* pairs) in a dictionary.
all()	The *all()* function returns Boolean True value if all the keys in the dictionary are True else returns False.
any()	The *any()* function returns Boolean True value if any of the key in the dictionary is True else returns False.
sorted()	The *sorted()* function by default returns a list of items, which are sorted based on dictionary keys.

- In python, any non-zero integer value is true, and zero is interpreted as false.
- The sorted() function returns the sorted list of keys by default in ascending order without modifying the original ***key:value*** pairs

EXAMPLE

```
>>> presidents = {"washington":1732, "jefferson":1751, "lincoln":1809, "roosevelt":1858, "eisenhower":1890}
>>> len(presidents)
>>> sorted(presidents)
>>> sorted(presidents, reverse = True)
>>> sorted(presidents.values())
>>> sorted(presidents.items())
>>> dict_func1 = {0:True, 2:False}
>>> all(dict_func1)
>>> dict_func2 = {1:true, 2:false}
>>> all(dict_func2)
>>> any(dict_func)
>>> dict_func3 = {0:true, 0:false}
>>> any(dict_func3)
```

DICTIONARY METHODS >>> DIR(DICT)

Various Dictionary Methods

Dictionary Methods	Syntax	Description
clear()	dictionary_name.clear()	The *clear()* method removes all the *key:value* pairs from the dictionary.
fromkeys()	dictionary_name.fromkeys(seq [, value])	The *fromkeys()* method creates a new dictionary from the given sequence of elements with a value provided by the user.
get()	dictionary_name.get(key [, default])	The *get()* method returns the value associated with the specified key in the dictionary. If the key is not present then it returns the default value. If default is not given, it defaults to *None*, so that this method never raises a *KeyError*.
items()	dictionary_name.items()	The *items()* method returns a new view of dictionary's key and value pairs as tuples.
keys()	dictionary_name.keys()	The *keys()* method returns a new view consisting of all the keys in the dictionary.
pop()	dictionary_name.pop(key[, default])	The *pop()* method removes the key from the dictionary and returns its value. If the key is not present, then it returns the default value. If default is not given and the key is not in the dictionary, then it results in *KeyError*.
popitem()	dictionary_name.popitem()	The *popitem()* method removes and returns an arbitrary (key, value) tuple pair from the dictionary. If the dictionary is empty, then calling *popitem()* results in *KeyError*.
setdefault()	dictionary_name.setdefault (key[, default])	The *setdefault()* method returns a value for the key present in the dictionary. If the key is not present, then insert the key into the dictionary with a default value and return the default value. If key is present, *default* defaults to None, so that this method never raises a *KeyError*.
update()	dictionary_name.update([other])	The *update()* method updates the dictionary with the *key:value* pairs from *other* dictionary object and it returns *None*.
values()	dictionary_name.values()	The *values()* method returns a new view consisting of all the values in the dictionary.

Note: Replace the word "*dictionary_name*" mentioned in the syntax with your *actual dictionary name* in your code.

DICTIONARY METHODS >>> DIR(DICT)

Dictionary Methods	Syntax	Description
clear()	dictionary_name.clear()	The *clear()* method removes all the *key:value* pairs from the dictionary.
fromkeys()	dictionary_name.fromkeys(seq [, value])	The *fromkeys()* method creates a new dictionary from the given sequence of elements with a value provided by the user.
get()	dictionary_name.get(key [, default])	The *get()* method returns the value associated with the specified key in the dictionary. If the key is not present then it returns the default value. If default is not given, it defaults to *None*, so that this method never raises a *KeyError*.
items()	dictionary_name.items()	The *items()* method returns a new view of dictionary's key and value pairs as tuples.
keys()	dictionary_name.keys()	The *keys()* method returns a new view consisting of all the keys in the dictionary.
pop()	dictionary_name.pop(key[, default])	The *pop()* method removes the key from the dictionary and returns its value. If the key is not present, then it returns the default value. If default is not given and the key is not in the dictionary, then it results in *KeyError*.

DICTIONARY METHODS >>> DIR(DICT)

popitem()	dictionary_name. popitem()	The *popitem()* method removes and returns an arbitrary (key, value) tuple pair from the dictionary. If the dictionary is empty, then calling *popitem()* results in *KeyError*.
setdefault()	dictionary_name. setdefault (key[, default])	The *setdefault()* method returns a value for the key present in the dictionary. If the key is not present, then insert the key into the dictionary with a default value and return the default value. If key is present, *default* defaults to None, so that this method never raises a *KeyError*.
update()	dictionary_name. update([other])	The *update()* method updates the dictionary with the *key:value* pairs from *other* dictionary object and it returns *None*.
values()	dictionary_name. values()	The *values()* method returns a new view consisting of all the values in the dictionary.

Note: Replace the word *"dictionary_name"* mentioned in the syntax with your *actual dictionary name* in your code.

DICTIONARY METHODS

```python
box_office_billion = {
    "avatar": 2009,
    "titanic": 1997,
    "starwars": 2015,
    "harrypotter": 2011,
    "avengers": 2012,
}
box_office_billion_fromkeys = box_office_billion.fromkeys(box_office_billion)
print(box_office_billion_fromkeys)
box_office_billion_fromkeys = box_office_billion.fromkeys(
    box_office_billion, "billion_dollar"
)
print(box_office_billion_fromkeys)
print(box_office_billion.get("frozen"))
print(box_office_billion.get("frozen", 2013))
print(box_office_billion.keys())
print(box_office_billion.values())
print(box_office_billion.items())
box_office_billion.update({"frozen": 2013})
print(box_office_billion)
box_office_billion.setdefault("minions")
print(box_office_billion)
box_office_billion.setdefault("ironman", 2013)
print(box_office_billion)
print(box_office_billion.pop("avatar"))
print(box_office_billion.popitem())
box_office_billion.clear()
print(box_office_billion)
```

```
{'avatar': None, 'titanic': None, 'starwars': None, 'harrypotter': None, 'avengers': None}
{'avatar': 'billion_dollar', 'titanic': 'billion_dollar', 'starwars': 'billion_dollar', 'harrypotter': 'billion_dollar', 'aveng
ers': 'billion_dollar'}
None
2013
dict_keys(['avatar', 'titanic', 'starwars', 'harrypotter', 'avengers'])
dict_values([2009, 1997, 2015, 2011, 2012])
dict_items([('avatar', 2009), ('titanic', 1997), ('starwars', 2015), ('harrypotter', 2011), ('avengers', 2012)])
{'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011, 'avengers': 2012, 'frozen': 2013}
{'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011, 'avengers': 2012, 'frozen': 2013, 'minions': None}
{'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011, 'avengers': 2012, 'frozen': 2013, 'minions': None, 'ir
onman': 2013}
2009
('ironman', 2013)
{}
```

EXAMPLE

```
>>>box_office_billion = {"avatar":2009, "titanic":1997, "starwars":2015, "harrypotter":2011, "avengers":2012}

>>>boxbox_office_billion_fromkeys = box_office_billion.fromkeys(box_office_billion)

>>>id(box_office_billion_fromkeys)

>>>box_office_billion_fromkeys = box_office_billion.fromkeys(box_office_billion, "billion_dollar")

>>> box_office_billion.keys()

>>> box_office_billion.values()

>>> box_office_billion.items()

>>> box_office_billion.update({"frozen":2013})

>>> box_office_billion.setdefault("minions")

>>> box_office_billion.setdefault("ironman", 2013)
```

EXAMPLE

```
>>> box_office_billion.pop("avatar")
>>> box_office_billion.popitem()
>>> box_office_billion.clear()
>>> print(box_office_billion.get("frozen"))
>>> box_office_billion.get("frozen",2013)
>>> box_office_billion.get("avatar")
>>> box_office_billion.get("avs")
>>> box_office_billion.get("avs", "not found")
>>>box_office_billion.get("avatar", "not found")
```

EXAMPLE

```
>>>studs={"ec101":"abc", "ec102":“xyz", "b_sec":['ajay','abhishek','amar','atish'], 'ece':{"6a":‘raj',"6b":‘ram',"6c":‘bheem', "6d":'sham'}}
>>> studs['ec102']
>>> studs['b_sec']
>>> studs['ece']
>>> keys=[10,20,30,40]
>>> values=['bsc','bcom','arts','masscom']
>>> degrees=zip(keys,values)
>>> degrees=dict(zip(keys,values))
>>> degrees
>>> del degrees[10]
```

- if you want to convert *dict_keys*, *dict_values*, and *dict_items* data types returned from *keys()*, *values()*, and *items()* methods to a true list then pass their list, such as returned values, to *list()* function

```
>>>list(box_office_billion.keys())
>>> list(box_office_billion.values())
>>> list(box_office_billion.items())
```

THE DEL STATEMENT

- To delete the key:value pair, use the del statement followed by the name of the dictionary along with the key you want to delete.

del dict_name[key]

```
1  animals = {"r":"raccoon", "c":"cougar", "m":"moose"}
2  print(animals)
3  del animals["c"]
4  print(animals)
```

```
{'r': 'raccoon', 'c': 'cougar', 'm': 'moose'}
{'r': 'raccoon', 'm': 'moose'}
```

Write python program to generate a dictionary that contains (i: i*i) such that i is a number ranging from 1 to n.

Write python program to generate a dictionary that contains (i: i*i) such that i is a number ranging from 1 to n.

```
number = int(input("enter a number "))

create_number_dict = dict()

for i in range(1, number+1):

    create_number_dict[i] = i * i

print("the generated dictionary of the form (i: i*i) is")

print(create_number_dict)
```

```
1  def main():
2      number = int(input("Enter a number "))
3      create_number_dict = dict()
4      for i in range(1, number + 1):
5          create_number_dict[i] = i * i
6      print("The generated dictionary of the form (i: i*i) is")
7      print(create_number_dict)
8
9
10 if __name__ == "__main__":
11     main()
```

```
Enter a number 3
The generated dictionary of the form (i: i*i) is
{1: 1, 2: 4, 3: 9}
```

PROGRAM TO DYNAMICALLY BUILD USER INPUT AS A LIST

```
print("method 1: building dictionaries")
build_dictionary = {}
for i in range(0, 2):
    dic_key = input("enter key ")
    dic_val = input("enter val ")
    build_dictionary.update({dic_key: dic_val})
print(f"dictionary is {build_dictionary}")
```

```
print("method 2: building dictionaries")
build_dictionary = {}
for i in range(0, 2):
    dic_key = input("enter key ")
    dic_val = input("enter val ")
    build_dictionary[dic_key] = dic_val
print(f"dictionary is {build_dictionary}")
```

```
print("method 3: building dictionaries")
build_dictionary = {}
i = 0
while i < 2:
    dict_key = input("enter key ")
    dict_val = input("enter val ")
    build_dictionary.update({dict_key: dict_val})
    i = i + 1
print(f"dictionary is {build_dictionary}")
```

Program to dynamically build user input as a list

```python
def main():
    print("Method 1: Building Dictionaries")
    build_dictionary = {}
    for i in range(0, 2):
        dic_key = input("Enter key ")
        dic_val = input("Enter val ")
        build_dictionary.update({dic_key: dic_val})
    print(f"Dictionary is {build_dictionary}")

    print("Method 2: Building Dictionaries")
    build_dictionary = {}
    for i in range(0, 2):
        dic_key = input("Enter key ")
        dic_val = input("Enter val ")
        build_dictionary[dic_key] = dic_val
    print(f"Dictionary is {build_dictionary}")

    print("Method 3: Building Dictionaries")
    build_dictionary = {}
    i = 0
    while i < 2:
        dict_key = input("Enter key ")
        dict_val = input("Enter val ")
        build_dictionary.update({dict_key: dict_val})
        i = i + 1
    print(f"Dictionary is {build_dictionary}")


if __name__ == "__main__":
    main()
```

```
Method 1: Building Dictionaries
Enter key microsoft
Enter val windows
Enter key canonical
Enter val ubuntu
Dictionary is {'microsoft': 'windows', 'canonical': 'ubuntu'}
Method 2: Building Dictionaries
Enter key apple
Enter val macos
Enter key canonical
Enter val ubuntu
Dictionary is {'apple': 'macos', 'canonical': 'ubuntu'}
Method 3: Building Dictionaries
Enter key microsoft
Enter val windows
Enter key apple
Enter val macos
Dictionary is {'microsoft': 'windows', 'apple': 'macos'}
```

Write a python program to input information for *n* number of students as given below:

a. Name

b. Registration number

c. Total marks.

The user has to specify a value for *n* number of students.

The program should output the registration number and marks of a specified student given his name

```
def student_details(number_of_students):
    student_name = {}
    for i in range(0, number_of_students):
        name = input("enter the name of the student ")
        registration_number = input("enter student's registration number ")
        total_marks = input("enter student's total marks ")
        student_name[name] = [registration_number, total_marks]
    student_search = input('enter name of the student you want to search ')
    if student_search not in student_name.keys():
        print('student you are searching is not present in the class')
    else:
        print("student you are searching is present in the class")
        print(f"student's registration number is
{student_name[student_search][0]}")
        print(f"student's total marks is {student_name[student_search][1]}")
number_of_students = int(input("enter the number of students "))
student_details(number_of_students)
```

```python
def student_details(number_of_students):
    student_name = {}
    for i in range(0, number_of_students):
        name = input("Enter the Name of the Student ")
        registration_number = input("Enter student's Registration Number ")
        total_marks = input("Enter student's Total Marks ")
        student_name[name] = [registration_number, total_marks]
    student_search = input('Enter name of the student you want to search ')
    if student_search not in student_name.keys():
        print('Student you are searching is not present in the class')
    else:
        print("Student you are searching is present in the class")
        print(f"Student's Registration Number is {student_name[student_search][0]}")
        print(f"Student's Total Marks is {student_name[student_search][1]}")
def main():
    number_of_students = int(input("Enter the number of students "))
    student_details(number_of_students)
if __name__ == "__main__":
    main()
```

```
Enter the number of students 2
Enter the Name of the Student jack
Enter student's Registration Number 1AIT18CS05
Enter student's Total Marks 89
Enter the Name of the Student jill
Enter student's Registration Number  1AIT18CS08
Enter student's Total Marks 91
Enter name of the student you want to search jill
Student you are searching is present in the class
Student's Registration Number is  1AIT18CS08
Student's Total Marks is 91
```

PROGRAM TO ILLUSTRATE TRAVERSING OF KEY:VALUE PAIRS IN DICTIONARIES USING FOR LOOP

```python
currency = {
    "India": "Rupee",
    "USA": "Dollar",
    "Russia": "Ruble",
    "Japan": "Yen",
    "Germany": "Euro",
}


def main():
    print("List of Countries")
    for key in currency.keys():
        print(key)
    print("List of Currencies in different Countries")
    for value in currency.values():
        print(value)
    for key, value in currency.items():
        print(f"'{key}' has a currency of type '{value}'")


if __name__ == "__main__":
    main()
```

```
List of Countries
India
USA
Russia
Japan
Germany
List of Currencies in different Countries
Rupee
Dollar
Ruble
Yen
Euro
'India' has a currency of type 'Rupee'
'USA' has a currency of type 'Dollar'
'Russia' has a currency of type 'Ruble'
'Japan' has a currency of type 'Yen'
'Germany' has a currency of type 'Euro'
```

Write python program to check for the presence of a key in the dictionary and to sum all its values

```python
historical_events = {
    "apollo11": 1969,
    "great_depression": 1929,
    "american_revolution": 1775,
    "berlin_wall": 1989,
}


def check_key_presence():
    key = input("Enter the key to check for its presence ")
    if key in historical_events.keys():
        print(f"Key '{key}' is present in the dictionary")
    else:
        print(f"Key '{key}' is not present in the dictionary")


def sum_dictionary_values():
    print("Sum of all the values in the dictionary is")
    print(f"{sum(historical_events.values())}")


def main():
    check_key_presence()
    sum_dictionary_values()


if __name__ == "__main__":
    main()
```

```
Enter the key to check for its presence apollo11
Key 'apollo11' is present in the dictionary
Sum of all the values in the dictionary is
7662
```

Write a program that accepts a sentence and calculate the number of digits, uppercase and lowercase letters.

```
def main():
    sentence = input("Enter a sentence ")
    construct_dictionary = {"digits": 0, "lowercase": 0, "uppercase": 0}
    for each_character in sentence:
        if each_character.isdigit():
            construct_dictionary["digits"] += 1
        elif each_character.isupper():
            construct_dictionary["uppercase"] += 1
        elif each_character.islower():
            construct_dictionary["lowercase"] += 1
    print("The number of digits, lowercase and uppercase letters are")
    print(construct_dictionary)


if __name__ == "__main__":
    main()
```

```
Enter a sentence I am Time, the great destroyer of the world - Bhagavad Gita 11.32
The number of digits, lowercase and uppercase letters are
{'digits': 4, 'lowercase': 42, 'uppercase': 4}
```

Write python program to count the number of times an item appears in the list

```python
novels = ["gone_girl", "davinci_code", "games_of_thrones", "gone_girl", "davinci_code"]


def main():
    count_items = dict()
    for book_name in novels:
        count_items[book_name] = count_items.get(book_name, 0) + 1
    print("Number of times a novel appears in the list is")
    print(count_items)


if __name__ == "__main__":
    main()
```

```
Number of times a novel appears in the list is
{'gone_girl': 2, 'davinci_code': 2, 'games_of_thrones': 1}
```

Write python program to count the number of times each word appears in a sentence

```
1  def main():
2      count_words = dict()
3      sentence = input("Enter a sentence ")
4      words = sentence.split()
5      for each_word in words:
6          count_words[each_word] = count_words.get(each_word, 0) + 1
7      print("The number of times each word appears in a sentence is")
8      print(count_words)
9
10
11 if __name__ == "__main__":
12     main()
```

```
Enter a sentence Everyone needs a little inspiration from time to time
The number of times each word appears in a sentence is
{'Everyone': 1, 'needs': 1, 'a': 1, 'little': 1, 'inspiration': 1, 'from': 1, 'time': 2, 'to': 1}
```

Write python program to count the number of characters in a string using dictionaries. Display the keys and their values in alphabetical order.

```python
def construct_character_dict(word):
    character_count_dict = dict()
    for each_character in word:
        character_count_dict[each_character] = (
            character_count_dict.get(each_character, 0) + 1
        )
    sorted_list_keys = sorted(character_count_dict.keys())
    for each_key in sorted_list_keys:
        print(each_key, character_count_dict.get(each_key))


def main():
    word = input("Enter a string ")
    construct_character_dict(word)


if __name__ == "__main__":
    main()
```

```
Enter a string massachussetts
a 2
c 1
e 1
h 1
m 1
s 5
t 2
u 1
```

Program to demonstrate nested dictionaries

```python
student_details = {
    "name": "jasmine",
    "registration_number": "1AIT18CS05",
    "sub_marks": {"python": 95, "java": 90, ".net": 85},
}


def nested_dictionary():
    print(f"Student Name {student_details['name']}")
    print(f"Registration Number {student_details['registration_number']}")
    average = sum(student_details["sub_marks"].values()) / len(
        student_details["sub_marks"]
    )
    print(f"Average of all the subjects is {average}")


def main():
    nested_dictionary()


if __name__ == "__main__":
    main()
```

```
Student Name jasmine
Registration Number 1AIT18CS05
Average of all the subjects is 90.0
```

SETS

- Python also includes a data type for sets.
- A set is an unordered collection with no duplicate items.
- Primary uses of sets include membership testing and eliminating duplicate entries.
- Sets support mathematical operations, such as union, intersection, difference, and symmetric difference.
- Curly braces **{ }** or the set() function can be used to create sets with a comma-separated list of items inside curly brackets **{ }**.
- A set is a collection of unique items. Duplicate items will be removed
- To create empty set you have to use set(), not **{ }** as the latter creates an empty dictionary
- Sets are mutable.
- Indexing is not possible in sets, since set items are unordered. You cannot access or change an item of the set using indexing or slicing.

EXAMPLE

```
>>> basket = {'apple', 'orange', 'apple',
'pear', 'orange', 'banana'}
>>> basket
>>> len(basket)
>>> sorted(basket)
>>> 'orange' in basket
>>> 'crabgrass' in basket
>>> a = set('abracadabra')
>>> b = set('alacazam')
```

```
>>> a
>>> b
>>> a - b
>>> b-a
>>> a | b
>>> a & b
>>> a ^ b
```

Various Set Methods

Set Methods	Syntax	Description
add()	set_name.add(*item*)	The *add()* method adds an *item* to the set *set_name*.
clear()	set_name.clear()	The *clear()* method removes all the items from the set *set_name*.
difference()	set_name.difference(**others*)	The *difference()* method returns a new set with items in the set *set_name* that are not in the *others* sets.
discard()	set_name.discard(item)	The *discard()* method removes an *item* from the set *set_name* if it is present.
intersection()	set_name.intersection(**others*)	The *intersection()* method returns a new set with items common to the set *set_name* and all *others* sets.
isdisjoint()	set_name.isdisjoint(*other*)	The *isdisjoint()* method returns True if the set *set_name* has no items in common with *other* set. Sets are disjoint if and only if their intersection is the empty set.
issubset()	set_name.issubset(*other*)	The *issubset()* method returns True if every item in the set *set_name* is in *other* set.
issuperset()	set_name.issuperset(*other*)	The *issuperset()* method returns True if every element in *other* set is in the set *set_name*.
pop()	set_name.pop()	The method *pop()* removes and returns an arbitrary item from the set *set_name*. It raises KeyError if the set is empty.
remove()	set_name.remove(*item*)	The method *remove()* removes an *item* from the set *set_name*. It raises KeyError if the *item* is not contained in the set.
symmetric_ difference()	set_name. symmetric_difference(other)	The method *symmetric_difference()* returns a new set with items in either the set or other but not both.
union()	set_name.union(**others*)	The method *union()* returns a new set with items from the set *set_name* and all *others* sets.
update()	set_name.update(**others*)	Update the set *set_name* by adding items from all *others* sets.

Note: Replace the words *"set_name", "other" and "others"* mentioned in the syntax with your *actual set names* in your code.

SET METHODS >>> DIR(SET)

Set Methods	Syntax	Description
add()	set_name.add(*item*)	The *add()* method adds an *item* to the set *set_name*.
clear()	set_name.clear()	The *clear()* method removes all the items from the set *set_name*.
difference()	set_name.difference(**others*)	The *difference()* method returns a new set with items in the set *set_name* that are not in the *others* sets.
discard()	set_name.discard(item)	The *discard()* method removes an *item* from the set *set_name* if it is present.
intersection()	set_name.intersection(**others*)	The *intersection()* method returns a new set with items common to the set *set_name* and all *others* sets.
isdisjoint()	set_name.isdisjoint(*other*)	The *isdisjoint()* method returns True if the set *set_name* has no items in common with *other* set. Sets are disjoint if and only if their intersection is the empty set.

SET METHODS >>> dir(set)

issubset()	set_name.issubset(*other*)	The *issubset()* method returns True if every item in the set *set_name* is in *other* set.
issuperset()	set_name.issuperset(*other*)	The *issuperset()* method returns True if every element in *other* set is in the set *set_name*.
pop()	set_name.pop()	The method *pop()* removes and returns an arbitrary item from the set *set_name*. It raises KeyError if the set is empty.
remove()	set_name.remove(*item*)	The method *remove()* removes an *item* from the set *set_name*. It raises KeyError if the *item* is not contained in the set.
symmetric_ difference()	set_name. symmetric_difference(other)	The method *symmetric_difference()* returns a new set with items in either the set or other but not both.
union()	set_name.union(**others*)	The method *union()* returns a new set with items from the set *set_name* and all *others* sets.
update()	set_name.update(**others*)	Update the set *set_name* by adding items from all *others* sets.

Note: Replace the words "*set_name*", "*other*" *and* "*others*" mentioned in the syntax with your *actual set names* in your code.

VARIOUS SET METHODS

```
>>> european_flowers = {"sunflowers", "roses", "lavender", "tulips", "goldcrest"}
>>> american_flowers = {"roses", "tulips", "lilies", "daisies"}
>>> american_flowers.add("orchids")
>>> american_flowers
>>> american_flowers.difference(european_flowers)
>>> american_flowers-european_flowers
>>> american_flowers.intersection(european_flowers)
>>> american_flowers & european_flowers
>>> american_flowers.isdisjoint(european_flowers)
>>> american_flowers.issuperset(european_flowers)
>>> american_flowers.issubset(european_flowers)
```

VARIOUS SET METHODS

```
>>> american_flowers.symmetric_difference(european_flowers)
>>> american_flowers ^ european_flowers
>>> american_flowers.union(european_flowers)
>>> american_flowers | european_flowers
>>> american_flowers.update(european_flowers)
>>> american_flowers
>>> american_flowers.discard("roses")
>>> american_flowers
>>> european_flowers.pop()
>>> american_flowers.clear()
>>> american_flowers
```

TRAVERSING OF SETS

program to iterate over items in sets using *for* loop

```
warships = {"u.s.s._arizona", "hms_beagle", "ins_airavat", "ins_hetz"}
for i in warships:
    print(f"{i} is a warship")
```

WRITE A FUNCTION WHICH RECEIVES A VARIABLE NUMBER OF STRINGS AS ARGUMENTS. FIND UNIQUE CHARACTERS IN EACH STRING

```
def find_unique(*all_words):
    for i in all_words:
        unique_list = list(set(i))
        print(f"unique characters in the word {i} are {unique_list}")
find_unique("egg", "immune", "feed", "vacuum", "goddessship")
```

WRITE A PYTHON PROGRAM THAT ACCEPTS A SENTENCE AS INPUT AND REMOVES ALL DUPLICATE WORDS. PRINT THE SORTED WORDS

```
sentence = input("enter a sentence ")

words = sentence.split()

print(f"the unique and sorted words are {sorted(list(set(words)))}")
```

FROZENSET

- A frozenset is basically the same as a set, except that it is immutable.
- Once a frozenset is created, then its items cannot be changed. Since they are immutable, they can be used as members in other sets and as dictionary keys.
- The frozensets have the same functions as normal sets, except none of the functions that change the contents (update, remove, pop,etc.) are available.

- >>> dir(frozenset)

```
>>> fs = frozenset(["g", "o", "o", "d"])
>>> fs
>>> animals = set([fs, "cattle", "horse"])
>>> animals
>>> official_languages_world = {"english":59, "french":29, "spanish":21}
>>> frozenset(official_languages_world)
>>> frs = frozenset(["german"])
>>> official_languages_world = {"english":59, "french":29, "spanish":21, frs:6}
```

SUMMARY

- Tuple is an immutable data structure comprising of items that are ordered and heterogeneous.
- Tuples are formed using commas and not the parenthesis.
- Indexing and slicing of items are supported in tuples.
- Tuples support built-in functions such as len(), min(), and max().
- The set stores a collection of unique values and are not placed in any particular order.
- Add an item to the set using add() method and remove an item from the set using the remove() method.
- The *for* loop is used to traverse the items in a set.
- The *issubset()* or *issuperset()* method is used to test whether a set is a superset or a Subset of another set.
- Sets also provide functions such as union(), intersection(), difference(), and symmetric_difference().