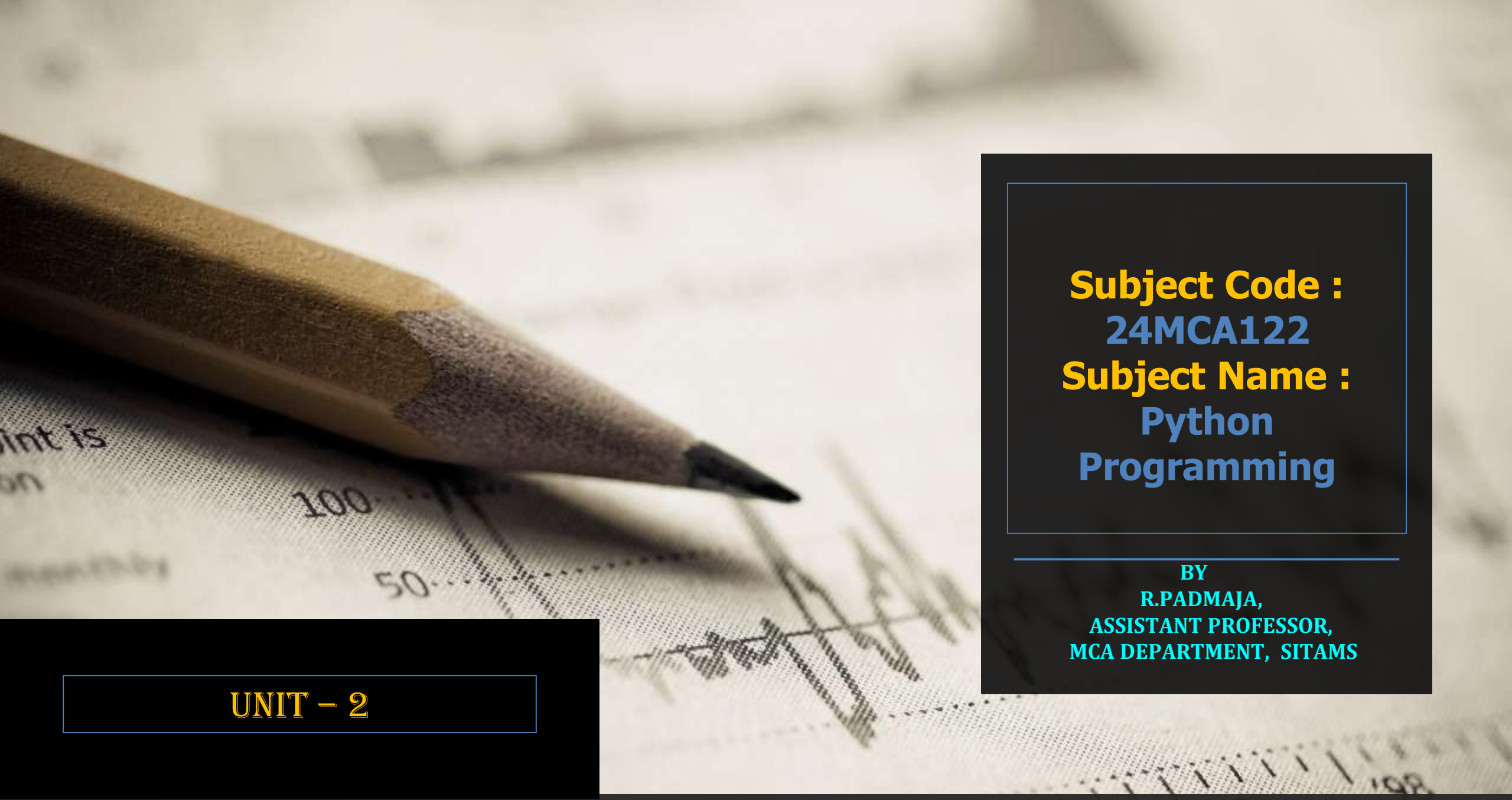


Subject Code : 24MCA122
Subject Name : Python Programming

BY
R.PADMAJA,
ASSISTANT PROFESSOR,
MCA DEPARTMENT, SITAMS

UNIT – 2

1) Functions and Functional Programming
   - 1.1) Function Calls
   - 1.2) Functions that require Arguments
   - 1.3) Functions that return values
2) Recursive Functions
3) Recursive vs Iteration
4) Built-in-functions
   - 4.1) Lists
   - 4.2) Tuples
   - 4.3) sets
   - 4.4) Dictionary

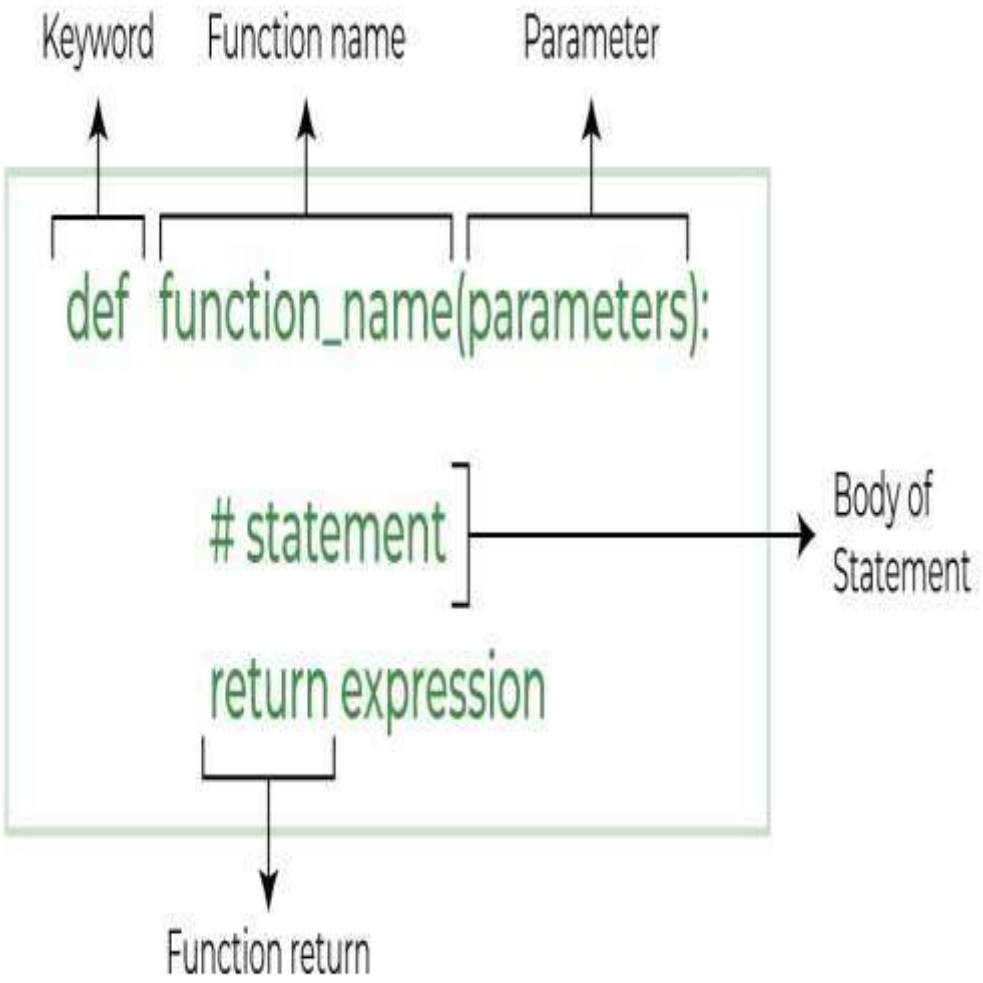
FUNCTIONAL PROGRAMMING

- In Python, a **function is a block of reusable code that performs a specific task**.
- It allows you **to organize your code into modular and reusable units, making your programs more structured and easier to understand.**

Benefits of Using Functions

- ✓ **Increase Code Readability**
- ✓ **Increase Code Reusability**

Python Function Declaration



def: It's the keyword used to define a function in Python.

function_name: It's the name given to the function. You can choose any valid name that follows Python's naming conventions.

parameters: They are optional input values that can be passed to the function for it to perform its tasks . Parameters are placed inside parentheses and separated by commas. If a function doesn't require any input, you can leave the parentheses empty.

Function body: It consists of one or more statements that are indented under the function definition. These statements define the tasks the function performs.

Return statement: It's optional and is used to specify the value that the function should return when it is called. If a function doesn't have a return statement, it will implicitly return None

Python Functions are classified into

1) **Function with No Arguments**
2) **Function with Arguments**
    1) Default arguments
    2) Keyword arguments
    3) Positional arguments
    4) Arbitrary Keyword arguments
3) **Function with return statement**
4) **Function without Return statement**

1) Function with No Arguments

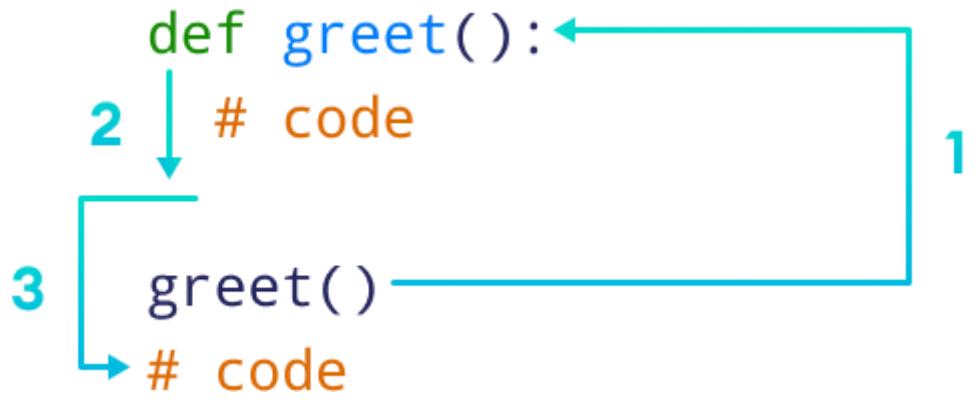
writing Function with No arguments

Sample Code :

```
1  # function definition
2  def greet():
3      print("Welcome to Python Functions")
4  # calling the function with no argument
5  greet()
6  print("outside of the function")
```

Output:

```
Welcome to Python Functions
outside of the function
```



2) Function with Arguments

- **Function with No Argument**

```
1  # function definition
2  def greet():
3      print("hello, smith")
4  #calling function
5  greet()
```

```
hello, smith
```

Function with Argument

```
1  # function definition
2  def greet(x):
3      print("hello,",x)
4  #calling function
5  name = input("Enter the name")
6  greet(name)
```

```
Enter the name Jones
hello,  Jones

Enter the name Ivan
hello,  Ivan
```

2) Function with Arguments

- **Function with No Arguments**

```
1  # function definition
2  def sum():
3      a = int(input("Enter A Value"))
4      b = int(input("Enter the B Value"))
5      c=a+b
6      return c
7  #calling the function
8  res = sum()
9  print("sum of 2 numbers are ", res)
```

```
Enter A Value4
Enter the B Value9
sum of 2 numbers are  13
```

Function with Arguments

```
1  # function definition
2  def sum(a,b):
3      c=a+b
4      return c
5  #calling the function
6  a = int(input("Enter A Value"))
7  b = int(input("Enter the B Value"))
8  res = sum(a,b)
9  print("sum of 2 numbers are ", res)
```

```
Enter A Value8
Enter the B Value4
sum of 2 numbers are  12
```

2) Function with Arguments

2.1) Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments

Code

```
# function with default arguments
def sum(a,b=3):
    c=a+b
    return c
# calling the function
res1 = sum(2)
print("sum of 2 numbers",res1)
res2 = sum(4,5)
print("sum of 2 numbers",res2)
```

Output

```
sum of 2 numbers 5
sum of 2 numbers 9
```

2) Function with Arguments

2.2) Keyword arguments

function by specifying the argument name along with its value (or) The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

Code

```
1  # function with Keyword
2  def greet(name,message):
3      print("hello",name,message)
4  greet("Alice","How are you") # positional Arguments
5  greet(message = "how are u",name="jones") # Keyword Argument
```

Output

```
hello Alice How are you
hello jones how are u
```

2) Function with Arguments

2.3) Positional arguments

Positional arguments, also known as positional parameters, are a type of argument in programming that are passed to a function or method based on their position or order. In contrast to keyword arguments, which are identified by their names, positional arguments rely on their position in the argument list.

Code

```
1  # function with Keyword arguments
2  def greet(name,message):
3      print("hello",name,message)
4  greet("Alice","How are you") # positional Arguments
5  greet(message = "how are u",name="jones") # Keyword Argument
```

Output

```
hello Alice How are you
hello jones how are u
```

2) Function with Arguments

2.4) Arbitrary Keyword arguments

- Sometimes, we do not know in advance the number of arguments that will be passed into a function.
- To handle this kind of situation, we can use arbitrary arguments in Python.
- Arbitrary arguments allow us to pass a varying number of values during a function call.
- We use an asterisk (*) before the parameter name to denote this kind of argument.
- In Python Arbitrary Keyword Arguments, *args, and **kwargs can pass a variable number of arguments to a function using special symbols.
- There are two special symbols:
  - ✓ *args in Python (Non-Keyword Arguments)
  - ✓ **kwargs in Python (Keyword Arguments)

```
# program to find sum of multiple numbers

def find_sum(*numbers):
    result = 0
    for num in numbers:
        result = result + num
    print("Sum = ", result)

# function call with 3 arguments
find_sum(1, 2, 3)

# function call with 2 arguments
find_sum(4, 9)
```

OUTPUT

```
Sum =6
Sum=13
```

FUNCTIONAL PROGRAMMING

Variable Length Arguments

Code

```
1  # function with variable length arguments
2  def MyFun(*argv):
3      for i in argv:
4          print(i)
5  MyFun('geeks','for','geeks')
```

Output

```
geeks
for
geeks
```

Variable Length Keyword Arguments

Code

```
1  # function with variable length Keyword arguments
2  def MyFun(**kwargs):
3      for key,value in kwargs.items():
4          print(key,"=",value)
5  MyFun(first="geeks",mid="for",last="geeks")
```

Output

```
first = geeks
mid = for
last = geeks
```

FUNCTIONAL PROGRAMMING

Function without Return Statement

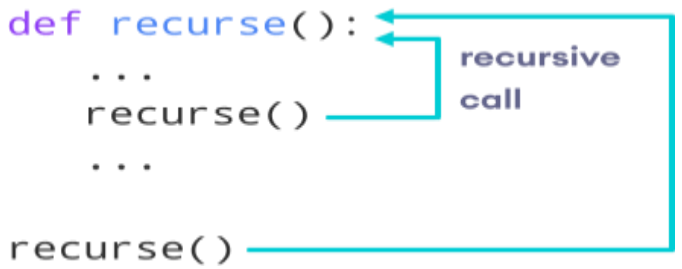
```
1  # function without Return Statement
2  def sum(x,y):
3      print("sum of 2 numbers",x+y)
4  sum(6,8)
```

Function with Return Statement

```
7  #function with Return Statement
8  def Sum2(x,y):
9      z=x+y
10     return z
11 res = Sum2(6,8)
12 print("sum of 2 numbers",res)
```

RECURSIVE FUNCTIONS

- A Function can call other Functions
- It is even possible for a function to call itself and so
- Function that calls itself is called Recursive Function.



- Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6=720

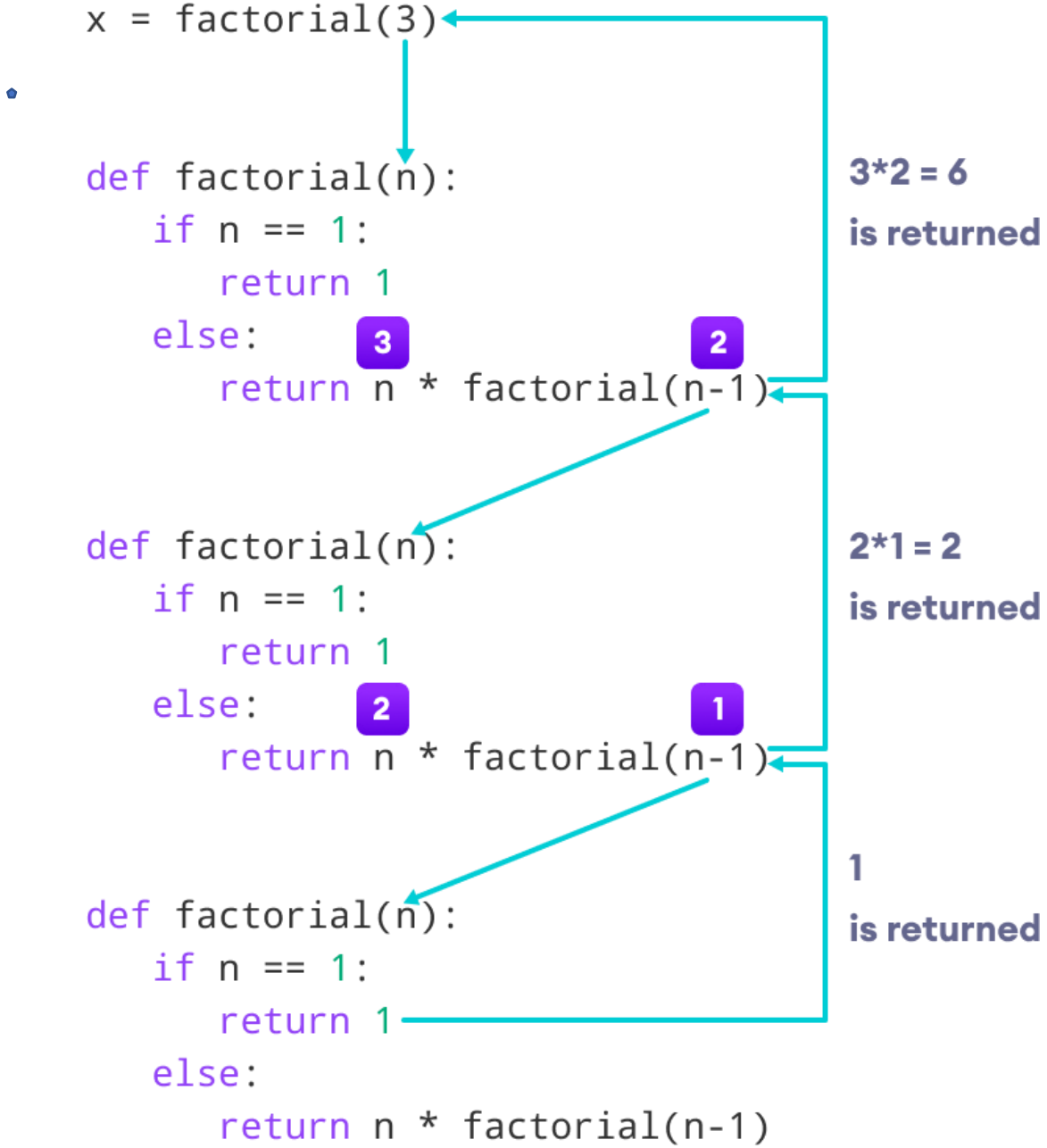
```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

RECURSIVE FUNCTIONS

- In the above example, factorial() is a recursive function as it calls itself.
- When we call this function with a positive integer, it will recursively call itself by decreasing the number.
- Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```

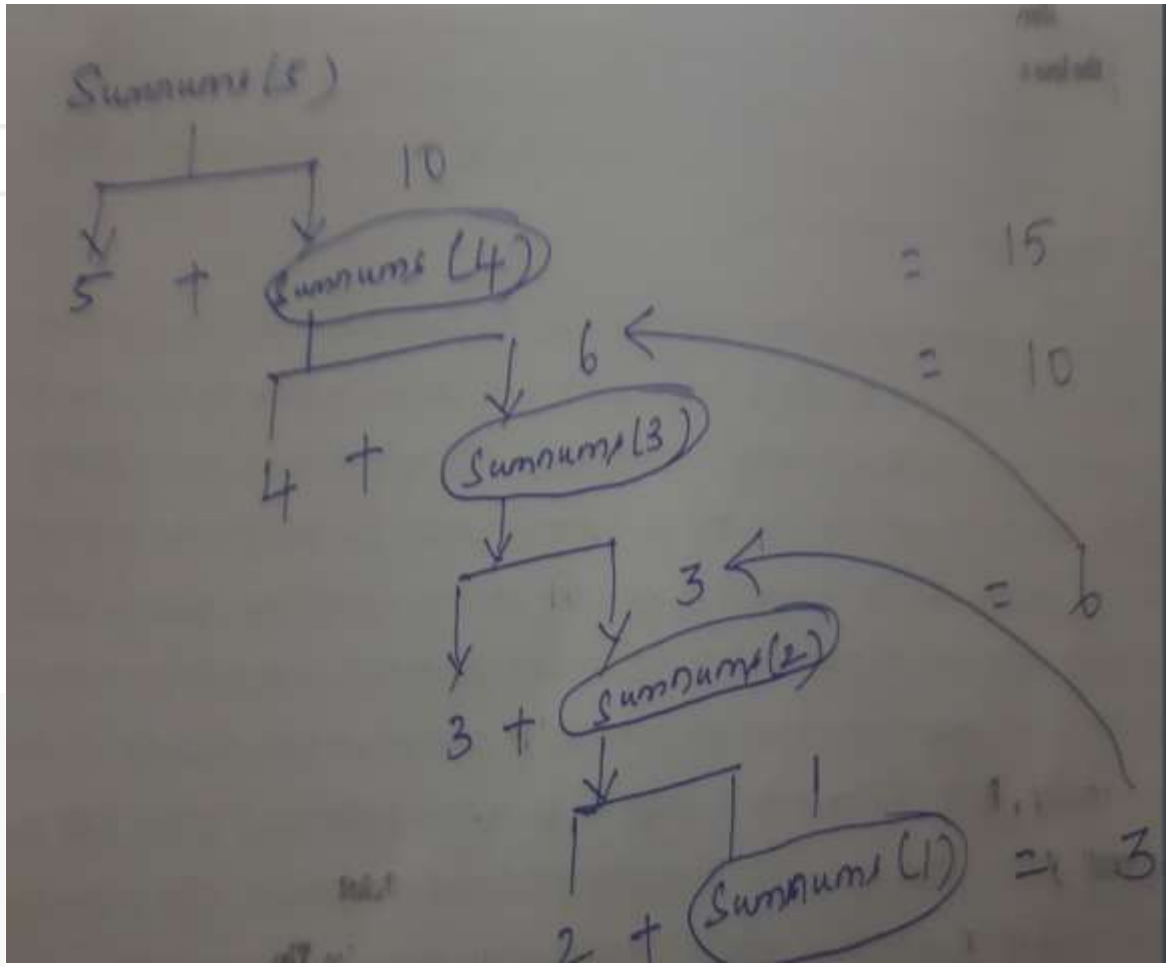
RECURSIVE FUNCTIONS



SUM OF N NATUTAL NUMBERS RECURSIVE FUNCTIONS

Code

```
1 def sumnums(n):
2     if n == 1:
3         return 1
4     return n + sumnums(n - 1)
5 print("Sum of first 6 numbers is ",sumnums(6))
```



Trace Out

```
sumnum(5)
= 5+sumnum(4)                    = 5+10=15
=       4* sumnum(3)             = 4+6=10
=               3*sumnum(2)      = 3+3=6
=                       2*sumnum(1)  = 2+1=3
```

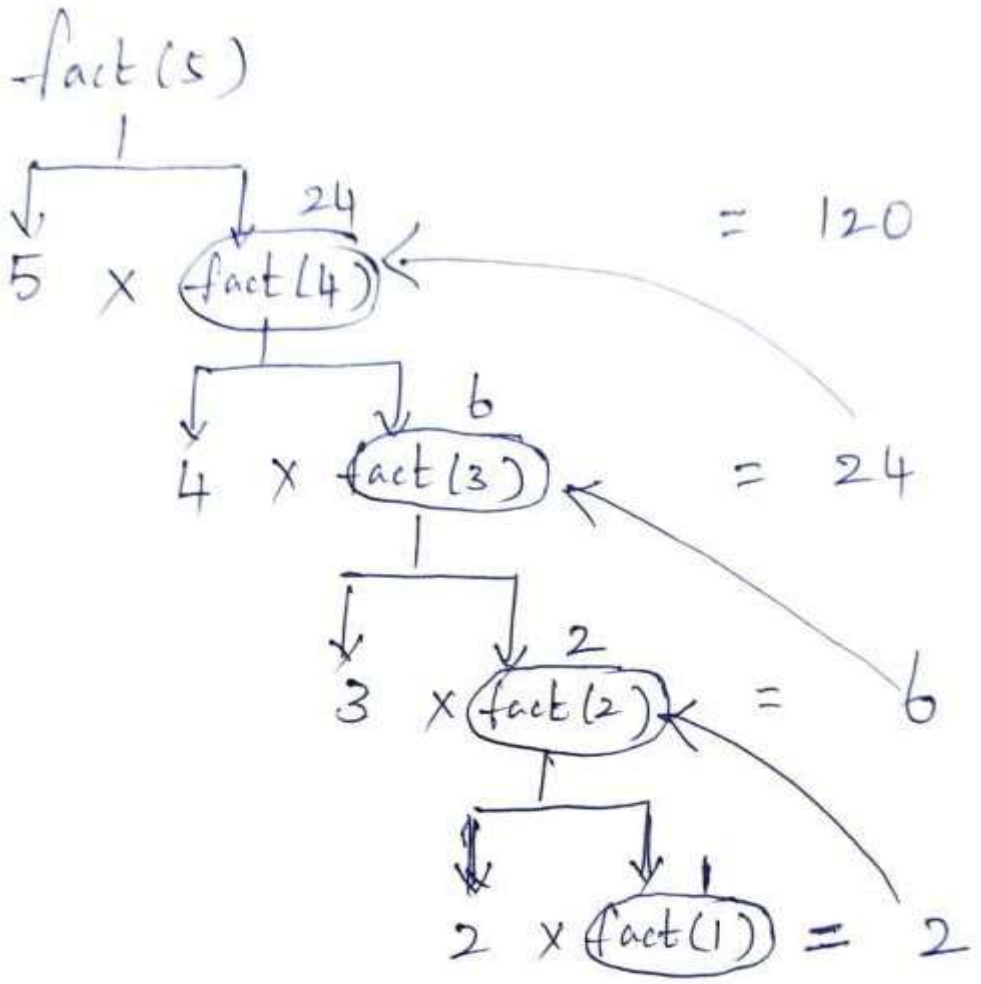
output

Sum of first 5 numbers is 15

RECURSIVE FUNCTIONS

Code

```
1  def fact(x):
2      if x==1: # this is called Base condition
3          return 1;
4      else:
5          return x*fact(x-1)
6  n = int(input("Enter the n Value"))
7  res = fact(n)
8  print("Factorial of n value is",res)
```



Trace Out

```
fact(5)
= 5*fact(4)                          = 5*24=120
=       4* fact(3)                   = 4*6=24
=               3*fact(2)            = 3*2=6
=                       2*fact(1)    = 2*1=2
```

output

Enter the n Value 5
Factorial of n value is 120

LIST

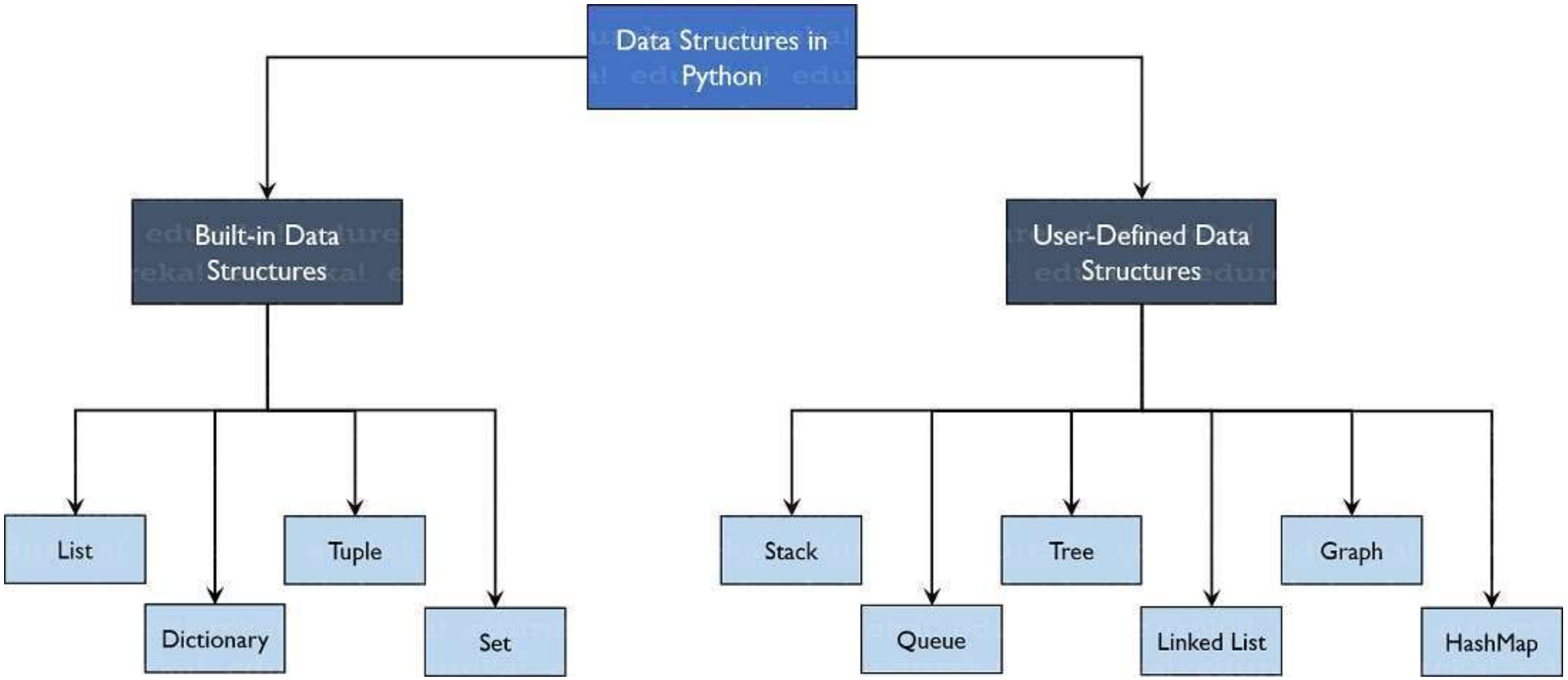
STRUCTURES

- "Data + structures" says it all.
- A data structure structures the data.
- It is a storage unit that organizes and stores data in a way the programmer can easily access.
- In the real world, a programmer/ coder's work involves huge amounts of data.
- Analyzing and using the data is made easy for the programmer when it is arranged the right way.
- Apart from organizing the data, Data structures also makes processing and accessing the data easy.
- Data Structures are a way of organizing data so that it can be accessed more efficiently depending upon the situation.
- Data Structures are fundamentals of any programming language around which a program is built.
- Types of Data Structures in Python are
    1) Built in Data Structures
    2) User Defined Data Structures

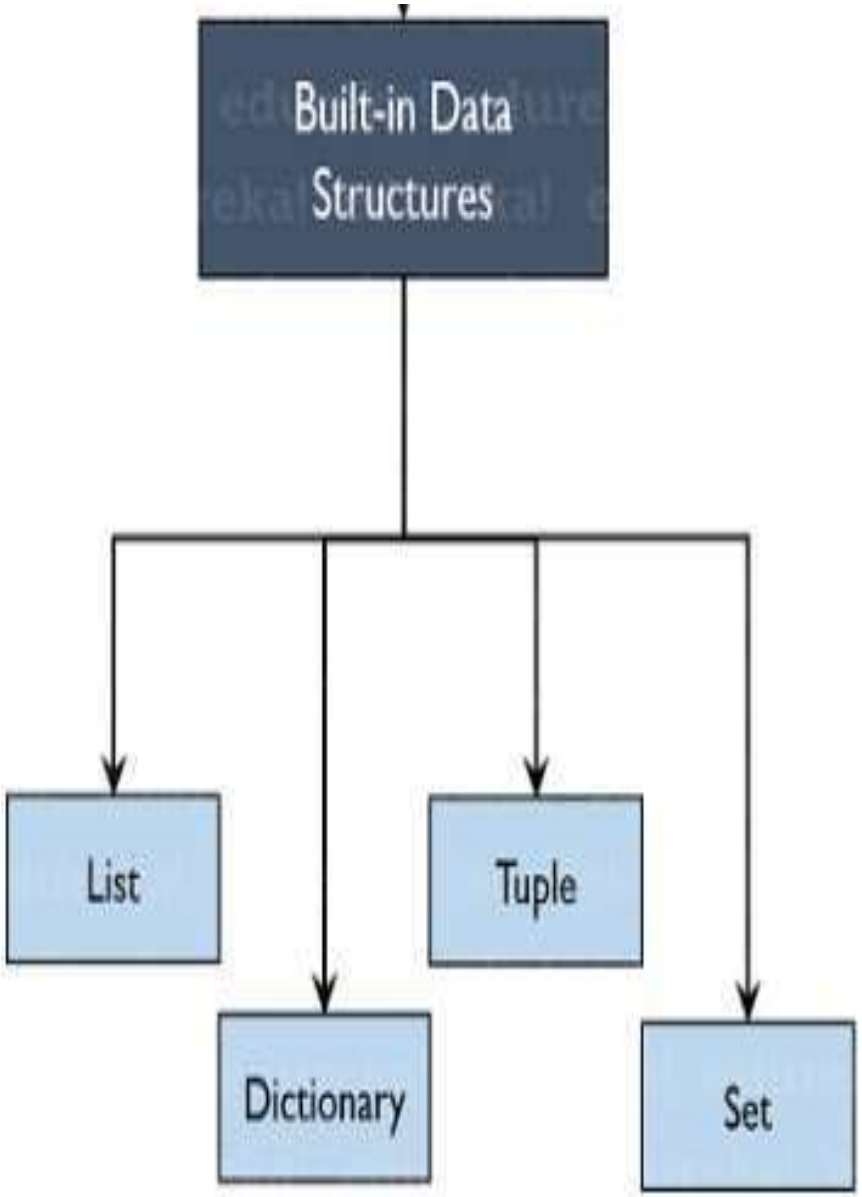
LIST

STRUCTURES

TYPES OF DATA STRUCTURES IN PYTHON



LIST STRUCTURES



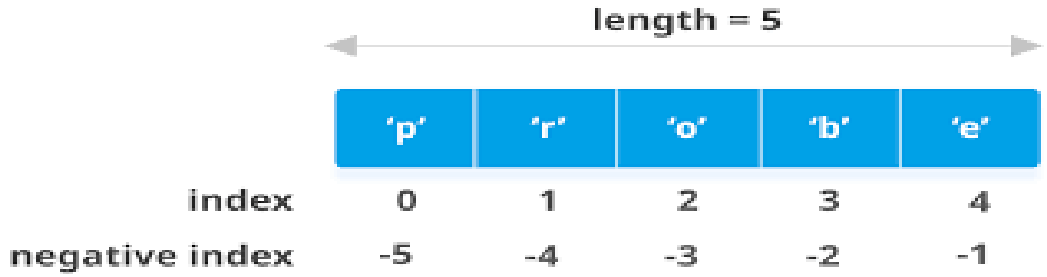
- As the name suggests, these Data Structures are built-in with Python which makes programming easier and helps programmers use them to obtain solutions faster.
- Let's discuss each of them in detail.

LIST

STRUCTURES

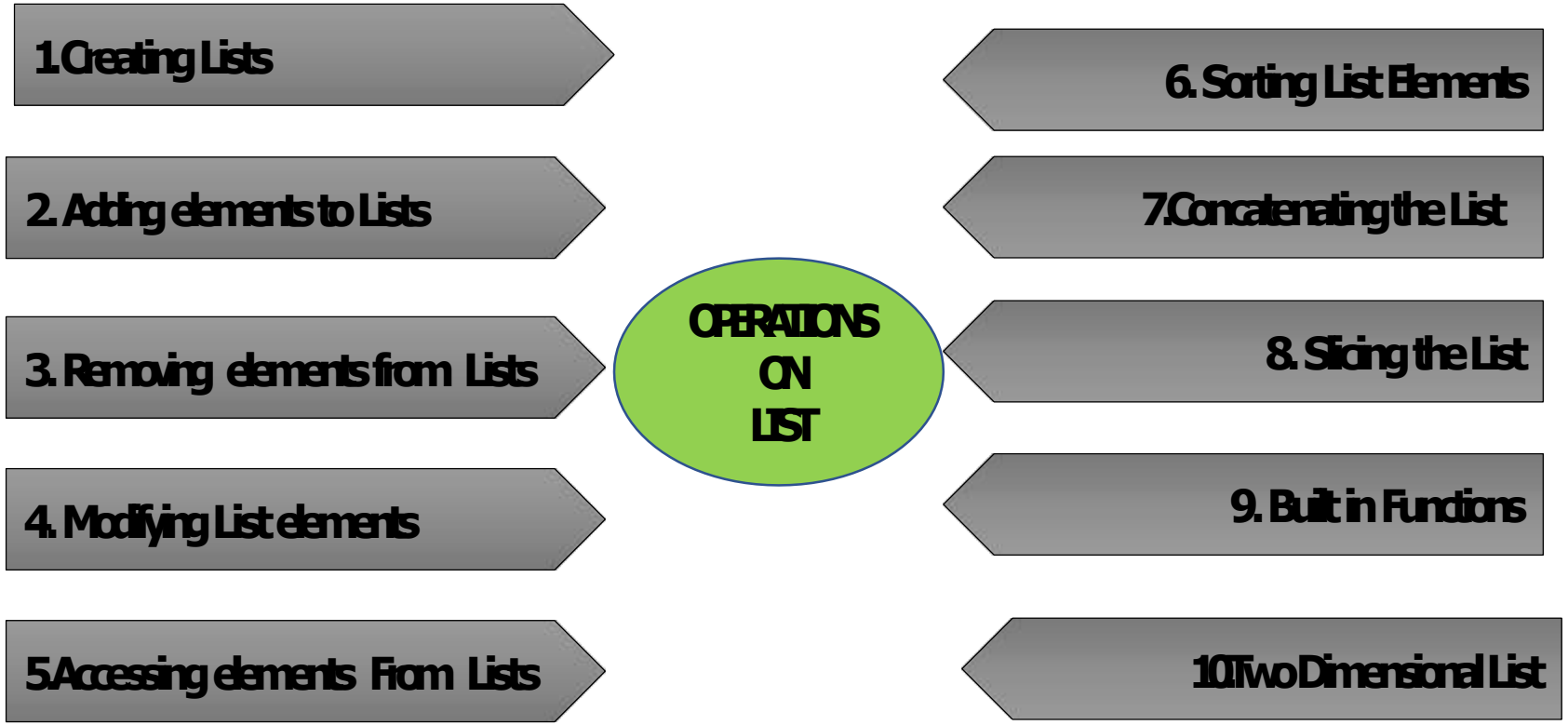
LIST

- List are one of the **Built in Data Structure** in Python
- Lists are **ordered collection of data**
- Lists are **mutable,** meaning that after creation, we can modify their elements.
- Lists allow **duplicate elements**.
- **Lists** are used to store data **of different data types** in a sequential manner.
- There are addresses assigned to every element of the list, which is called **as Index**.
- The index value starts from 0 and goes on until the last element called the **positive index**.
- There is also **negative indexing** which starts from -1 enabling you to access elements from the last to first



LIST

STRUCTURES



LIST

STRUCTURES

1 Creating Lists

- A list is created in Python by placing items inside [], separated by commas. For example,

```
1  l1 = [1,4,8] # creating a list with 3 integer elements
2  print('List Elements are ',l1)
```

Output:

```
List Elements are  [1, 4, 8]
```

- A list can have any number of items and they may be of different types (integer, float, string, etc.). For example,

```
1  l1 = [1,'Hello',8,3.4] # creating a list with different types of  elements
2  print('List Elements are ',l1)
```

Output:

```
List Elements are  [1, 'Hello', 8, 3.4]
```

- A List can be Created with Duplicate evalues

```
1  l1 = [1,'Hello',8,3.4,1] # creating a list with duplicate elements
2  print('List Elements are ',l1)
```

Output:

```
List Elements are  [1, 'Hello', 8, 3.4, 1]
```

LIST

STRUCTURES

2. Adding elements to Lists

List has 3 Methods to Add Elements

1) **append** → **To Add Single element at the end of the List**
2) **extend** → **To Add Multiple Elements at the end of the List**
3) **insert** → **To Add Element at a specified Position in the List**

append

Example

```
1  l1 = [2,3,4,5]
2  print("Before Appending ",l1)
3  l1.append(6)
4  print("After Appending ",l1)
```

Output

```
Before Appending  [2, 3, 4, 5]
After Appending  [2, 3, 4, 5, 6]
```

extend

Example

```
1  l1 = [2,3,4,5]
2  print("Before applying Extending ",l1)
3  l1.extend([5,6,7])
4  print("After applying Extending",l1)
```

Output

```
Before applying Extending  [2, 3, 4, 5]
After applying Extending [2, 3, 4, 5, 5, 6, 7]
```

insert

Example

```
1  l1 = [2,3,4,5]
2  print("Before applying Insert ",l1)
3  l1.insert(2,20)
4  print("After inserting element 20 at 2 nd position",l1)
```

Output

```
Before applying Insert  [2, 3, 4, 5]
After inserting element 20 at 2 nd position [2, 3, 20, 4, 5]
```

Extend can also be used to concatenate two Lists

```
1  l1 = [1,3,5]
2  print("Even Numbers List",l1)
3  l2 = [2,4,6]
4  print("Odd Numbers List",l2)
5  l1.extend(l2)
6  print("Odd and Even Numbers ",l1)
```

```
Even Numbers List [1, 3, 5]
Odd Numbers List [2, 4, 6]
Even and Odd Numbers  [1, 3, 5, 2, 4, 6]
```

LIST

STRUCTURES

3. Removing elements from Lists

- To delete elements, use the *del* keyword which is built-in into Python but this does not return anything back to us.
- To get the element back, use the pop() function which takes the index value.
- To remove an element by its value, use the remove() function.

```
l1=[1,2,3,4,5,6,7,8,9]
print('Original Listt ',l1)
# to delete element at 1st position but it does not return the element
del l1[1]
print('after deleting an element at 1st pos using del keyword',l1)
# # to remove element at 4th position and returning the element
ret_ele = l1.pop(4)
print('After deleting an element at 4th pos using pop function',l1)
print('removed element',ret_ele)
l1.remove(7)
print('After removing element 7 using remove function',l1)
```

Output

```
Original Listt  [1, 2, 3, 4, 5, 6, 7, 8, 9]
after deleting an element at 1st pos using del keyword [1, 3, 4, 5, 6, 7, 8, 9]
After deleting an element at 4th pos using pop function [1, 3, 4, 5, 7, 8, 9]
removed element 6
After removing element 7 using remove function [1, 3, 4, 5, 8, 9]
```

LIST

STRUCTURES

4. Modifying List elements

- The List Elements can be modified using Index of the element

```
1   l1= ['john','Smith','Ivan','Bayross','Korth']
2   print('Original List :', l1)
3   l1[0] = 'Leon' # to Modify element at 0th Position
4   print('After Modifying element at 0th position',l1)
5   l1[2:4]='James','Gosling','Rossum'# modifying elements from 2 to 4 position
6   print("After Modifying elements from 2nd to 4th position",l1)
```

Output

```
Original List : ['john', 'Smith', 'Ivan', 'Bayross', 'Korth']
After Modifying element at 0th position ['Leon', 'Smith', 'Ivan', 'Bayross', 'Korth']
After Modifying elements from 2nd to 4th position ['Leon', 'Smith', 'James', 'Gosling', 'Rossum', 'Korth']
```

LIST

STRUCTURES

5 Accessing elements From Lists

- **For Loop** - One way to Access elements from the Lists
- **Index** - Another Way to Access Elements from the Lists

Code

```
l1=[20,25,30,35,40]
print('Original List',l1)
# Accessing Using For Loop
print("Accessing List Elements using For Loop")
for i in l1:
    print(i)
# Accessing List Elements Using Index
print('Accessing all elements',l1) #access all elements
print('Accessing element at index 3 is ', l1[3]) #access index 3 element
print('Accessing elements from 0 to 1 and exclude 2 is ',l1[0:2]) #access elements from 0 to 1 and exclude 2
print('Accessing elements in reverse',l1[::-1]) #access elements in reverse
```

Output

```
Original List [20, 25, 30, 35, 40]
Accessing List Elements using For Loop
20
25
30
35
40
Accessing all elements [20, 25, 30, 35, 40]
Accessing element at index 3 is  35
Accessing elements from 0 to 1 and exclude 2 is  [20, 25]
Accessing elements in reverse [40, 35, 30, 25, 20]
```

LIST

STRUCTURES

6.Sorting List Elements

List Elements can be sorted using 2 ways

- **Sorted –is a function –where Sorting does not done at In-Place**
- **Sort –is a List method –where Sorting done at In-Place**

Code

```
l1=[80,40,90,10,120,30]
print('Original List',l1)
# sorting using sorted function
sorted = sorted(l1)
print('sorted using sorted function original',l1,'sorted',sorted)
#sorting using sort function
l1.sort()
print('After Sorting using sort method',l1)
```

Output

```
Original List [80, 40, 90, 10, 120, 30]
sorted using sorted function original [80, 40, 90, 10, 120, 30] sorted [10, 30, 40, 80, 90, 120]
After Sorting using sort method [10, 30, 40, 80, 90, 120]
```

LIST

STRUCTURES

7. Concatenating one or More List Elements

+Operator is used to concatenate one or more List Elements

Code

```
1   l1=[1,2,3]
2   l2=[4,5,6]
3   l3=[7,8,9]
4   l4=['hello','hai']
5   l5 = l1+l2+l3+l4
6   print('original First List',l1)
7   print('original Second List',l2)
8   print('original Third List',l3)
9   print('original Fourth List',l4)
10  print('After Concatenation First,second,Third and Fourth List',l5)
```

Output

```
original First List [1, 2, 3]
original Second List [4, 5, 6]
original Third List [7, 8, 9]
original Fourth List ['hello', 'hai']
After Concatenation First,second,Third and Fourth List [1, 2, 3, 4, 5, 6, 7, 8, 9, 'hello', 'hai']
```

LIST

STRUCTURES

8. Slicing List Elements

Positive Indexing	0	1	2	3	4	5	6	7	8
List Elements	20	25	30	35	40	45	50	55	60
Negative Indexing	-9	-8	-7	-6	-5	-4	-3	-2	-1

Index is used to Slice the List Elements

Code

```
l1=[20,25,30,35,40,45,50,55,60]
print('To Access first element of the List',l1[0])
print('To Access 2nd to 4th by excluding 4th element',l1[2:4])
print('To Access 5th to 8th by excluding 8th element',l1[5:8])
print('To Access first 3 elements',l1[:3])
print('To Access first 7 elements by skipping 2 elements',l1[:7:2])
print('To Accessing Last 2 elements',l1[-1:-3:-1])
print('To Access List Elements in Original Order',l1[1::1])
print('To Access List Elements in Reverse Order',l1[-1::-1])
```

Output

```
To Access first element of the List 20
To Access 2nd to 4th by excluding 4th element [30, 35]
To Access 5th to 8th by excluding 8th element [45, 50, 55]
To Access first 3 elements [20, 25, 30]
To Access first 7 elements by skipping 2 elements [20, 30, 40, 50]
To Accessing Last 2 elements [60, 55]
To Access List Elements in Original Order [25, 30, 35, 40, 45, 50, 55, 60]
To Access List Elements in Reverse Order [60, 55, 50, 45, 40, 35, 30, 25, 20]
```

LIST

STRUCTURES

9. Built In Functions/other Methods

Built in Functions

1) **len** — Helps to find Number of elements in a List

2) **Sorted** — Helps to sort the List Elements (Not In-Place)

Other Methods

1) **Count** — Helps to find Number of Occurrences of the Given Element

2) **Index** — Helps to find the Index of the specified element

Code

```
1  l1=[34,56,34,12,78,10]
2  length = len(l1)# to find number of elements in a list
3  print('No.of Elements in the list is',length)
4  sorted = sorted(l1)
5  print('original List',l1,'sorted list',sorted)
6  count = l1.count(34)
7  print('No. of Occurences of 34 in List is',count)
8  pos = l1.index(56)
9  print('Position of 56 in the List is ',pos)
```

Output

```
No.of Elements in the list is 6
original List [34, 56, 34, 12, 78, 10] sorted list [10, 12, 34, 34, 56, 78]
No. of Occurences of 34 in List is 2
Position of 56 in the List is  1
```

LIST

STRUCTURES

10. Two Dimensional Lists

- Lists are a very widely use data structure in python.
- They contain a list of elements separated by comma.
- But sometimes lists can also contain lists within them.
- These are called nested lists or multidimensional lists.

Code

```
1  l1=[[10,20,30],[2,3],[6,7,8]]
2  print(l1)
3  l1.append([11,22,33])
4  print('After appending ',l1)
5  print('Accessing first element of 2D List',l1[0])
6  for i in range(len(l1)):
7      for j in range(len(l1[i])):
8          print(l1[i][j],end=" ")
9      print()
```

Output

```
[[10, 20, 30], [2, 3], [6, 7, 8]]
After appending  [[10, 20, 30], [2, 3], [6, 7, 8], [11, 22, 33]]
Accessing first element of 2D List [10, 20, 30]
10 20 30
2 3
6 7 8
11 22 33
```

ITERATING OVER A LISTS IN PYTHON

Iterate over a list in Python

- There are multiple ways to iterate over a list in Python.
  1) Using For Loop
  2) For Loop and Range
  3) Using a While Loop

1) Using For Loop

Simple For Loop can be used to iterate over a list in python

Code

```
list = [1, 3, 5, 7, 9]

# Using for loop
for i in list:
    print(i)
```

Output

```
1
3
5
7
9
```

ITERATING OVER A LISTS IN PYTHON

Iterate over a list in Python

2) For Loop and Range

Code

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]
# Iterating the index
for i in range(len(list)):
    print(list[i])
```

Output

```
1
3
5
7
9
```

3) Using While Loop

Code

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]

# Getting length of list
length = len(list)
i = 0

# Iterating using while loop
while i < length:
    print(list[i])
    i += 1
```

Output

```
1
3
5
7
9
```

LIST - SUMMARY

Void Methods of List

1) sort — list.sort()
2) remove — list.remove(element)
3) insert — list.insert(pos, element)
4) append — list.append(element)
5) extend — list.extend([elements])

Return Type Methods of List

1) pop() — element = list.pop(position)
2) count() — var = list.count(element)
3) index() — var = list.index(element)

List functions

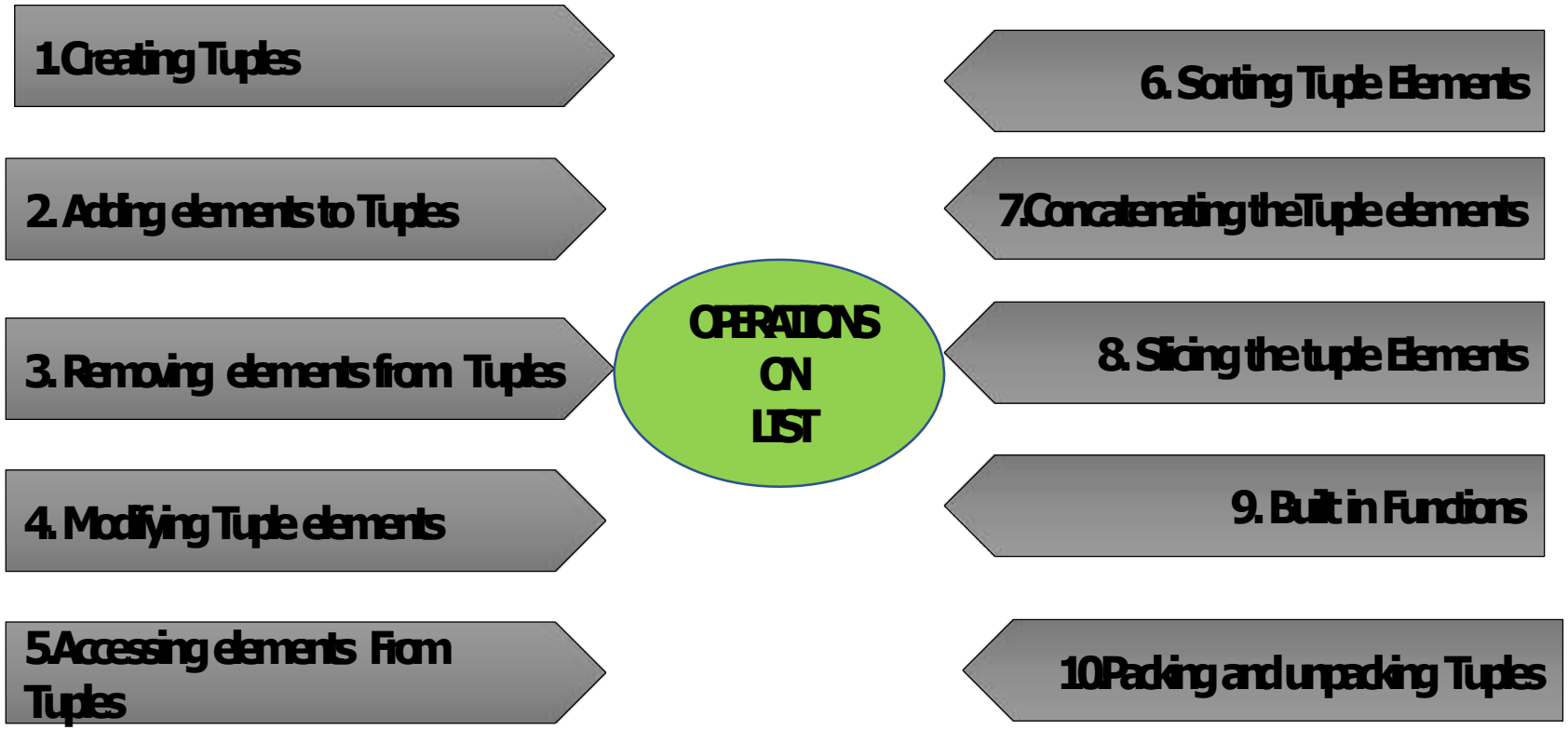
1) len — var = len(list)
2) sorted — var = sorted(list)

TUPLE STRUCTURES

- **Tuple is a Built in Data Structure in Python**
- **Tuple items are Ordered, means that the items have a defined order, and that order will not change.**
- **Tuple items are Unchangeable (Immutable), meaning that we cannot change, add or remove items after the tuple has been created**
- **Tuple allows Duplicates**

TUPLE

STRUCTURES



TUPLE

1) Creating a Tuple

- A tuple is created by placing all the items (elements) inside parentheses (), separated by commas.
- The parentheses are optional, however, it is a good practice to use them.
- A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Different types of tuples

# Empty tuple
my_tuple = ()
print(my_tuple)

# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

Output

```
()
(1, 2, 3)
(1, 'Hello', 3.4)
('mouse', [8, 4, 6], (1, 2, 3))
```

TUPLE

2) Adding elements to a Tuple

- Since tuples are immutable, they do not have a built-in append() method, but there are other ways to add items to a tuple.

1) **Convert into a list**: convert tuple into a list,, add item(s), and convert it back into a tuple.

```
t1 = ("apple", "banana", "cherry")
y = list(t1)
y.append("orange")
t1 = tuple(y)
```

2) **Add tuple to a tuple**. add tuples to tuples, create a new tuple with the item(s), and add it to the existing tuple:

```
t2 = ("apple", "banana", "cherry")
y = ("orange",)
t2 += y
print(t2)
```

TUPLE

3) Removing Elements from the Tuple

You cannot remove elements from the Tuple

TUPLE

4. Modifying Tuple elements

- Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.
- It is possible to convert the **tuple into a list**, **change the list**, and convert the **list back into a tuple.**

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

TUPLE

5Accessing elements From Tuples

- **For Loop** - One way to Access elements from the Lists
- **Index** - Another Way to Access Elements from the Lists

```
1   t1 =('hello','hai','smith','jones',3,4,5,6,7)
2   print('Original List',t1)
3
4   # Accessing the elements
5   for i in t1:
6       print(i)
7
8   # accessing 3rd element
9   print('3rd element',t1[3])
10
11  #Accessing  elements from 2nd to 5th
12  print('from 2nd to 6th ',t1[2:6])
13
14  # Accessing last 3 elements
15  print('Accessing last 3 elements',t1[-1:-4:-1])
16
17  #Accessing elements from first to last
18  print('Accessing from first to last',t1[1:])
```

TUPLE

6. Sorting Tuple Elements

- **Sorted** built in function is used to sort tuple elements
- To sort in descending order, pass reverse=True to sorted() function.

```
1  t1=(2,3,5,10,12)
2  print('Original Tuple',t1)
3  asc = sorted(t1)
4  print("In Ascending Order",asc)
5  des = sorted(t1,reverse=True)
6  print('In Descending Order',des)
```

OUTPUT

```
Original Tuple (2, 3, 5, 10, 12)
In Ascending Order [2, 3, 5, 10, 12]
In Descending Order [12, 10, 5, 3, 2]
```

TUPLE

9, Built In functions

Tuple Functions	Description
len(tuple)	Gives Number of elements in the tuple
max(tuple)	Returns the largest number from the tuple
min(tuple)	Returns the smallest number from the tuple
sum(tuple)	Returns the sum of tuple elements

```
1   t1=(2,3,5,10,12,5)
2   print('Original Tuple',t1)
3
4   # to find largest number in the tuple
5   print("Largest number in the tuple is ",max(t1))
6
7   # to find smallest number in the tuple
8   print("smallest number in the tuple is ",min(t1))
9
10  # to find number of elements in the tuple
11  print("Number of elements in the tuple is ",len(t1))
12
13  # to sum the elements of the tuple
14  print("sum of tuple elements are", sum(t1))
```

output

```
Original Tuple (2, 3, 5, 10, 12, 5)
Largest number in the tuple is  12
smallest number in the tuple is  2
Number of elements in the tuple is  6
sum of tuple elements are 37
```

TUPLE

Packing and unpacking Tuple

When a tuple is created, we normally assign values to it. This is called "packing" a tuple:

```
fruits = ("apple", "banana", "cherry")
(a,b,c) = fruits
print(a)
print(b)
print(c)
```

output

apple
banana
cherry

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(a, b, *c) = fruits
print(a)
print(b)
print(c)
```

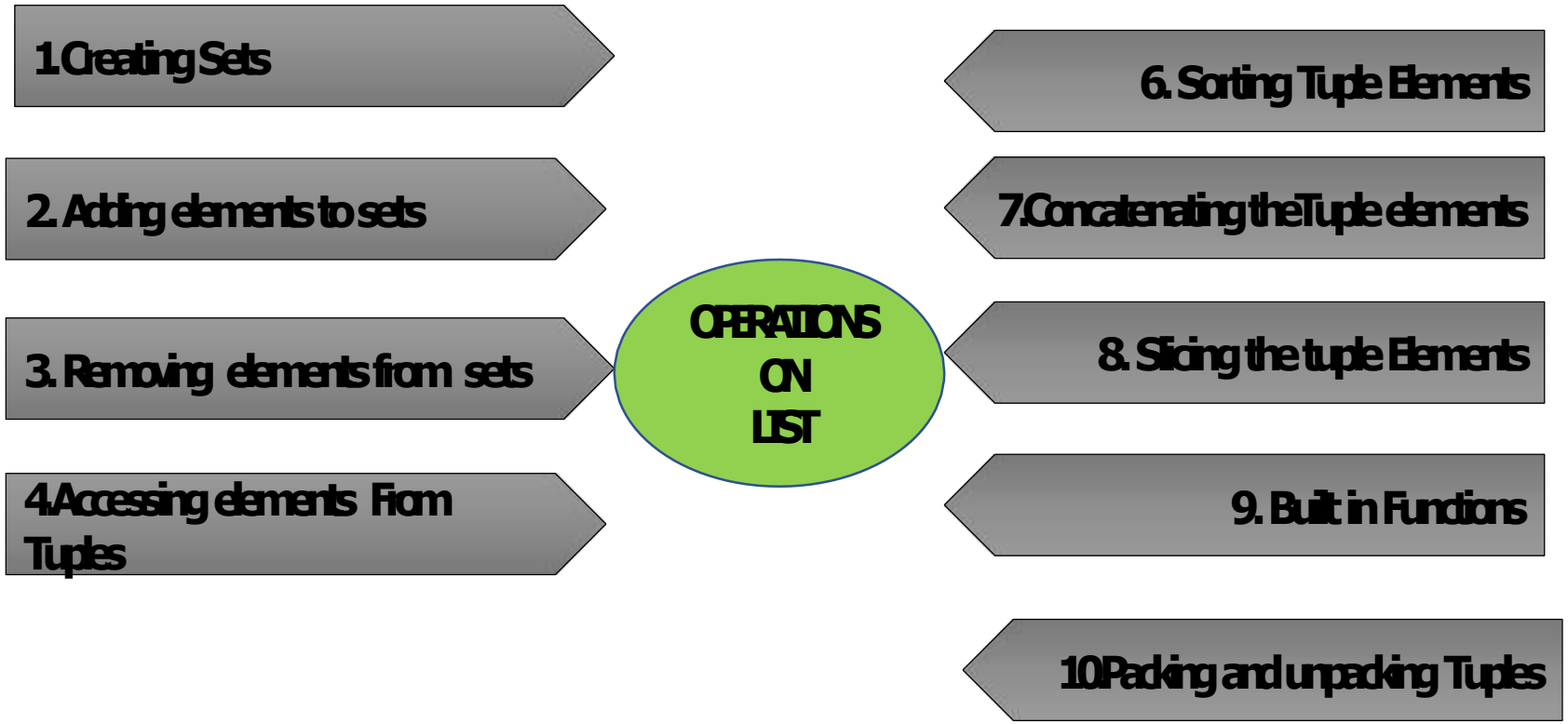
output

apple
banana
['cherry', 'strawberry', 'raspberry']

SET

- **SET is a Built in Data Structure in Python**
- **SET items are UnOrdered, means that the items will not have a defined order, and that order will change.**
- **SET items are Unchangeable (Immutable), meaning that we cannot change, add or remove items after the tuple has been created**
- **SET do not allow Duplicates**

SET


1.Creating Sets
2. Adding elements to sets
3. Removing elements from sets
4.Accessing elements From Tuples
OPERATIONS ON LIST
6. Sorting Tuple Elements
7.Concatenating the Tuple elements
8. Slicing the tuple Elements
9. Built in Functions
10.Packing and unpacking Tuples

TUPLE

Creating sets

Sets are created by placing all the elements inside curly braces {}, separated by comma.

```
1   student_id = {110,111,112,114}
2   print('student ids are ',student_id)
```

Output

```
student ids are  {112, 114, 110, 111}
```

TUPLE

Adding elements to sets

Elements can be added to sets using add() method

```
student_id = {110,111,112,114}
print('student ids are ',student_id)
student_id.add(115)
print("after adding new student is",student_id)
```

Output

```
student ids are  {112, 114, 110, 111}
after adding new student is {110, 111, 112, 114, 115}
```

TUPLE

Accessing elements from the sets

Elements can be accessed through For loop

```
student_id = {110,111,112,114}
print('student ids are ',student_id)

# one way of accessing Set Elements
student_id.add(115)
print("after adding new student is",student_id)

#another way of accessing Set Elements
for i in student_id:
    print(i)
```

Output

```
student ids are  {112, 114, 110, 111}
after adding new student is {110, 111, 112, 114, 115}
110
111
112
114
115
```

TUPLE

Remove element from the set

- remove() and discard() methods are used to remove elements from the set

```
student_id = {110,111,112,114}
print('Before Removing student ids are ',student_id)
student_id.remove(111)
print("After removing element",student_id)
student_id.discard(112)
print("After Removing element from set",student_id)
```

Output

Before Removing student ids are {112, 114, 110, 111}
After removing element {112, 114, 110}
After Removing element from set {114, 110}

PYTHON DATA STRUCTURES SUMMARY

- **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, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

SETS IN PYTHON

Creating Sets

You can create a set by using the set() constructor or by using curly braces {}.

```
# Using curly braces
another_set = {1, 2, 3, 4}
print(another_set)  # Output: {1, 2, 3, 4}

# Creating an empty set
empty_set = set()
print(empty_set)  # Output: set()
```

Adding Elements

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

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

Removing Elements

Elements can be removed using the remove() or discard() methods. The remove() method will raise a KeyError if the element is not found, while discard() will not.

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

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

# Discarding an element not in the set
my_set.discard(5)  # No error
```

Set Operations

Python sets support several standard set operations:

Union (| or union()): Combines all elements from both sets.

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

Intersection (& or intersection()): Returns elements that are common to both sets.

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

Difference (- or difference()): Returns elements that are in the first set but not in the second set.

```
difference_set = set1 - set2
# or
difference_set = set1.difference(set2)
print(difference_set)  # Output: {1, 2}
```

Symmetric Difference (^ or symmetric_difference()): Returns elements that are in either of the sets, but not in both.

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

Checking Membership

You can check if an element is in a set using the in keyword.

```
my_set = {1, 2, 3}
print(2 in my_set)  # Output: True
print(5 in my_set)  # Output: False
```

Iterating Through a Set

You can iterate through the elements of a set using a for loop.

```
my_set = {1, 2, 3}
for elem in my_set:
    print(elem)
```

Built-in Functions

Some useful built-in functions for sets include:

- len(set): Returns the number of elements in the set.
- min(set): Returns the smallest element in the set.

- max(set): Returns the largest element in the set.
- sum(set): Returns the sum of all elements in the set.

```
my_set = {1, 2, 3, 4}
print(len(my_set))  # Output: 4
print(min(my_set))  # Output: 1
print(max(my_set))  # Output: 4
print(sum(my_set))  # Output: 10
```

PROGRAM

```
# Creating sets
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
print("Set1:", set1)
print("Set2:", set2)

# Adding elements to a set
set1.add(5)
print("Set1 after adding 5:", set1)

# Removing elements from a set
set1.remove(1)  # Raises KeyError if 1 is not in the set
print("Set1 after removing 1:", set1)

# Discarding elements from a set
set1.discard(2)  # Does not raise an error if 2 is not in the set
print("Set1 after discarding 2:", set1)

# Union of sets
union_set = set1 | set2
print("Union of Set1 and Set2:", union_set)
```

```python
# Intersection of sets
intersection_set = set1 & set2
print("Intersection of Set1 and Set2:", intersection_set)

# Difference of sets
difference_set = set1 - set2
print("Difference of Set1 and Set2:", difference_set)

# Symmetric difference of sets
symmetric_difference_set = set1 ^ set2
print("Symmetric Difference of Set1 and Set2:",
symmetric_difference_set)

# Checking membership
print("Is 3 in Set1?", 3 in set1)
print("Is 1 in Set1?", 1 in set1)

# Iterating through a set
print("Elements in Set1:")
for elem in set1:
    print(elem)

# Using built-in functions
print("Length of Set1:", len(set1))
print("Minimum in Set1:", min(set1))
print("Maximum in Set1:", max(set1))
print("Sum of elements in Set1:", sum(set1))
```

OUTPUT

```
Set1: {1, 2, 3, 4}
Set2: {3, 4, 5, 6}
Set1 after adding 5: {1, 2, 3, 4, 5}
Set1 after removing 1: {2, 3, 4, 5}
```

```
Set1 after discarding 2: {3, 4, 5}
Union of Set1 and Set2: {3, 4, 5, 6}
Intersection of Set1 and Set2: {3, 4, 5}
Difference of Set1 and Set2: set()
Symmetric Difference of Set1 and Set2: {6}
Is 3 in Set1? True
Is 1 in Set1? False
Elements in Set1:
3
4
5
Length of Set1: 3
Minimum in Set1: 3
Maximum in Set1: 5
Sum of elements in Set1: 12
```

DICTIONARY IN PYTHON

In Python, a dictionary is an unordered collection of items where each item consists of a key-value pair.

Dictionaries are mutable, which means they can be changed after creation. They are also dynamic, meaning they can grow and shrink as needed.

Creating Dictionaries

You can create a dictionary using curly braces {} or the dict() constructor.

```
# Using curly braces
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

```
# Using the dict() constructor
another_dict = dict(name='Bob', age=30, city='San Francisco')
print(another_dict)  # Output: {'name': 'Bob', 'age': 30, 'city': 'San Francisco'}

# Creating an empty dictionary
empty_dict = {}
print(empty_dict)  # Output: {}
```

Accessing Values

You can access the value associated with a specific key using square brackets [] or the get() method.

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

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

# Using get() with a default value
print(my_dict.get('country', 'Not Found'))  # Output: Not Found
```

Adding and Updating Items

You can add new key-value pairs or update existing ones using the square brackets [].

```
# Adding a new key-value pair
my_dict['country'] = 'USA'
print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York', 'country': 'USA'}
```

```
# Updating an existing key-value pair
my_dict['age'] = 26
print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
```

Removing Items

You can remove items using the del statement, the pop() method, or the popitem() method.

```
# Using pop() method
# Using del statement
del my_dict['city']
print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'country': 'USA'}

my_dict.pop('age')
print(age)  # Output: 26
print(my_dict)  # Output: {'name': 'Alice', 'country': 'USA'}

# Using popitem() method (removes the last inserted item)
item = my_dict.popitem()
print(item)  # Output: ('country', 'USA')
print(my_dict)  # Output: {'name': 'Alice'}
```

Dictionary Methods

Python provides several methods to work with dictionaries:

- keys(): Returns a view object containing the keys.
- values(): Returns a view object containing the values.

- items(): Returns a view object containing the key-value pairs.
- update(): Updates the dictionary with elements from another dictionary or an iterable of key-value pairs.
- clear(): Removes all items from the dictionary.

```
# Using keys(), values(), and items() methods
print(my_dict.keys())#Output:dict_keys(['name','age','city'])
print(my_dict.values())#Output:dict_values['Alice',25,'New York']
print(my_dict.items())  # Output: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])

# Using update() method
my_dict.update({'age': 27, 'city': 'Los Angeles'})
print(my_dict)  # Output: {'name': 'Alice', 'age': 27, 'city': 'Los Angeles'}

# Using clear() method
my_dict.clear()
print(my_dict)  # Output: {}
```

Iterating Through a Dictionary

You can iterate through a dictionary using a for loop.

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

# Iterating through keys
for key in my_dict:
    print(key, my_dict[key])
    # Output:
    # name Alice
```

```
    # age 25
    # city New York

# Iterating through values
for value in my_dict.values():
    print(value)
    # Output:
    # Alice
    # 25
    # New York

# Iterating through key-value pairs
for key, value in my_dict.items():
    print(key, value)
    # Output:
    # name Alice
    # age 25
    # city New York
```

Built-in Functions

Some useful built-in functions for dictionaries include:

- **len(dict): Returns the number of items in the dictionary.**
- **sorted(dict): Returns a sorted list of the dictionary's keys.**

```
print(len(my_dict))  # Output: 3
print(sorted(my_dict))  # Output: ['age', 'city', 'name']
```

PROGRAM

```python
# Creating dictionaries
dict1 = {'name': 'Alice', 'age': 25, 'city': 'New York'}
dict2 = dict(name='Bob', age=30, city='San Francisco')
print("Dictionary 1:", dict1)
print("Dictionary 2:", dict2)

# Accessing values
print("Name in dict1:", dict1['name'])
print("Age in dict1:", dict1.get('age'))
print("Country in dict1:", dict1.get('country', 'Not
Found'))

# Adding and updating items
dict1['country'] = 'USA'
print("Dict1 after adding country:", dict1)
dict1['age'] = 26
print("Dict1 after updating age:", dict1)

# Removing items
del dict1['city']
print("Dict1 after deleting city:", dict1)
age = dict1.pop('age')
print("Popped age:", age)
print("Dict1 after popping age:", dict1)
item = dict1.popitem()
print("Popped item:", item)
print("Dict1 after popping item:", dict1)
```

```python
# Dictionary methods
dict1 = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print("Keys in dict1:", dict1.keys())
print("Values in dict1:", dict1.values())
print("Items in dict1:", dict1.items())

# Updating dictionary
dict1.update({'age': 27, 'city': 'Los Angeles'})
print("Dict1 after update:", dict1)

# Clearing dictionary
dict1.clear()
print("Dict1 after clearing:", dict1)

# Iterating through a dictionary
dict1 = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print("Iterating through keys:")
for key in dict1:
    print(key, dict1[key])

print("Iterating through values:")
for value in dict1.values():
    print(value)

print("Iterating through items:")
for key, value in dict1.items():
    print(key, value)
```

```
# Built-in functions
print("Length of dict1:", len(dict1))
print("Minimum key in dict1:", min(dict1))
print("Maximum key in dict1:", max(dict1))
print("Sorted keys in dict1:", sorted(dict1))
```

OUTPUT

Dictionary 1: {'name': 'Alice', 'age': 25, 'city': 'New York'}
Dictionary 2: {'name': 'Bob', 'age': 30, 'city': 'San Francisco'}
Name in dict1: Alice
Age in dict1: 25
Country in dict1: Not Found
Dict1 after adding country: {'name': 'Alice', 'age': 25, 'city': 'New York', 'country': 'USA'}
Dict1 after updating age: {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
Dict1 after deleting city: {'name': 'Alice', 'age': 26, 'country': 'USA'}
Popped age: 26
Dict1 after popping age: {'name': 'Alice', 'country': 'USA'}
Popped item: ('country', 'USA')
Dict1 after popping item: {'name': 'Alice'}
Keys in dict1: dict_keys(['name', 'age', 'city'])
Values in dict1: dict_values(['Alice', 25, 'New York'])
Items in dict1: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])
Dict1 after update: {'name': 'Alice', 'age': 27, 'city': 'Los Angeles'}
Dict1 after clearing: {}
Iterating through keys:

name Alice

age 25

city New York

Iterating through values:

Alice

25

New York

Iterating through items:

name Alice

age 25

city New York

Length of dict1: 3

Minimum key in dict1: age

Maximum key in dict1: name

Sorted keys in dict1: ['age', 'city', 'name']

		Mutable/ Immutable	**Ordered/ Unordered**	**Allows /do not Allow Duplicates**
LIST	Items in [ ]	Mutable	Ordered	Allows
TUPLE	Items in ( )	Immutable	Ordered	Allows
SET	Items in { }	Mutable	Unordered	Do not allow
DICTIONARY	Items in key-value pairs	Mutable	Unordered	Allows