Section-1: List and Dictionaries: Lists, Defining a list, Dictionaries, Defining a dictionary, Built-in methods for lists and dictionaries, Intro to list comprehension, Basics of nested lists
Section-2: Design with Function: Functions as Abstraction Mechanisms, Design with Recursive Functions, Higher Order Function.
Section-3: Modules: Standard Modules, Packages.

Section-1: List and Dictionaries

Define and compare the properties of collection data types list, tuple, dictionary, and set.

- Python Collections are **container data types**. They are **lists, tuples, sets, and dictionaries**. These are general-purpose built-in data types that are used to store more than one value/element/item.
- The Collections (similar to arrays) are also called **Python data structures or sequences** as they represent more than one value of any data type.

Each of these collection data types has different Properties or Characteristics based on their usage. Following is a comparison of their properties:

Data Type	Ordered?	Mutable (changeable)?	Indexed?	Duplicates allowed?
list	Ordered	Mutable (changeable)	Indexed	Allows Duplicate members
tuple	Ordered	Immutable (unchangeable)	Indexed	Allows Duplicate members
dict	Ordered (>=Py3.7)	Mutable (changeable)	Not Indexed but uses Key	No Duplicates keys; but can have duplicate values
set	unordered	Immutable (unchangeable) However, add & remove of members are possible	Not Indexed	No Duplicates members

List Data Type

List Definition:

The **list is a sequence** of multiple data values of the same or different data types. It is a **versatile data type** exclusive to Python.

➢ It is also called **a collection data type** or **a data structure.**
➢ The **values** in a list are also called **items or elements**.

The list is an **ordered** data **sequence** written in square brackets **[ ]** separated by commas **,** .

List Properties:

A. **Ordered -** The items in a list have a **defined order.** The order of the items will not change. If you add new items to a list, the new items will be placed at the end of the list.
B. **Mutable -** The items in a list are **changeable**. We can update, add, or remove items in a list.
C. **Indexed -** The list items are **indexed**, 1st item has index [0], 2nd item has index [1] and so on.
D. **Allows Duplicates -** Since lists are indexed, lists can have items with the same value.

Syntax - 1: Create List

```
listname = [ item-1, item-2, . . .  item-n ]
```

Syntax - 2: Create List using Constructor

👉 We can use the **list()** constructor to create a list in Python.

```
listname = list ( (  item-1,  item-2, . . .  item-n) )  #notice the 2 parentheses
```

List Literals and Basic Operators:

A list literal is written as a sequence of data values separated by commas enclosed in square brackets [ ] .

 # list of integers

```
>>> [2022, 2027, 2030] # list of integers
```

 # list of strings

```
>>> ["Guntur", "Vijayawada", "Tirupathi"] # list of strings
```

 # empty list

```
>>> [ ] # An empty list
```

We can use other lists as elements in a list, thereby creating a **list of lists**.

 # list of lists

```
>>> [ [50,60,70] , [80,90,100] , ['A','B','C'] ]
```

The Python interpreter **evaluates a list literal**, and each of the elements are also evaluated if required

```
>>> import math
>>> x=25
>>> [x,math.sqrt(x),math.pow(x,2)]
    [25, 5.0, 625.0]
>>>
```

list() and range() functions can build a **list of integers**

```
>>> playerlist = list(range(1,12,1))
>>> playerlist
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

list() function can build a **list from any iterable sequence** such as a string.

```
>>> slist = list("Software")
>>> slist
    ['S', 'o', 'f', 't', 'w', 'a', 'r', 'e']
```

List Access - Accessing Elements in List:

Each element in a List has an index. We can access any item of a List **by its index position**.

Syntax: listname[index]

- **Indexing:** The list elements are indexed starting from 0; which means, the first item in the list is at index 0.
- **Negative Indexing:** Python also supports negative indexing. Negative indexing starts with -1 at the last element in a list. We can use negative indexing without knowing the length of the list to access the last item.

z =	[3,	7,	4,	2]
index	0	1	2	3
negative index	-4	-3	-2	-1

Example: Creating List

```
marks = [50,60,70]
subjects = ["English","Maths","Programming"]
address = [245,"Amaravathi Rd","Guntur",522001]
```

Example: Accessing List

```
print(marks[1]) # 60
print(subjects[1]) # Maths
print(address[2]) # Guntur
```

Built-in List Methods

Python provides several built-in methods to manipulate a list. These include appending, inserting, removing, finding, counting, sorting, and reversing the data elements in a given list.

The built-in methods to perform these actions on a list are shown below. **Note:** When we use these functions, the original **list items are changed** because the lists are mutable.

Built-in List Methods

List Methods	Description	Syntax	Ex: m=[70,60,70,90]	Changed list >>> m
append()	Adds a value at the end of the list	**list.append(value)**	m.append(70)	**>>> m** **[70, 60, 70, 90, 80]**
insert()	Inserts a value at the given index and moves the other values to its right.	**list.insert(index,value)**	m.insert(1,50)	**>>> m** **[70, 50, 60, 70, 90, 80]**
remove()	Deletes a first occurrence of the given value; errors if the value doesn't exist.	**list.remove(value)**	m.remove(90)	**>>> m** **[70, 50, 60, 70, 80]**
reverse()	Reverses the values in a list	**list.reverse()**	m.reverse()	**>>> m** **[80, 70, 60, 50, 70]**
index()	Finds index of a given value in the list	**list.index(value)**	m.index(70)	**1**
sort()	Changes the order of list values into Ascending order	**list.sort()** **list.sort(reverse=True) for Descending**	m.sort()	**>>> m** **[50, 60, 70, 70, 80]**
count()	Returns total number of values in a list	**list.count(value)**	m.count(70)	**2**
pop()	**Deletes & Returns** a value at the given index	**list.pop(index)**	m.pop(4)	**>>> m** **[50, 60, 70, 70]**
extend()	Add the elements of a list (or any iterable), to the end of the current list	**tolist.extend(fromlist)**	grades=["A","B","C"] m.extend(grades)	**>>> m** **[50, 60, 70, 70, 'A', 'B', 'C']**
copy()	Returns a copy of list	**newlist=list.copy()**	new_m = m.copy()	**>>> new_m** **[50, 60, 70, 70, 'A', 'B', 'C']**
clear()	Removes all elements (values) from the list	**list.clear()**	m.clear()	**>>> m** **[ ]**

(Memorize: **AIR RISC PECC**)

Demo on Interactive Shell:

```
>>> m = [70,60,70,90]
>>> m
[70, 60, 70, 90]
>>> m.append(80)
>>> m
[70, 60, 70, 90, 80]
>>> m.insert(1,50)
>>> m
[70, 50, 60, 70, 90, 80]
>>> m.remove(90)
>>> m
[70, 50, 60, 70, 80]
>>> m.reverse()
>>> m
[80, 70, 60, 50, 70]
>>> m.index(70)
1
>>> m.sort()
>>> m
[50, 60, 70, 70, 80]
>>> m.count(70)
2
>>> grades=["A","B","C"]
>>> m.extend(grades)
>>> m
[50, 60, 70, 70, 'A', 'B', 'C']
>>> new_m = m.copy()
>>> new_m
[50, 60, 70, 70, 'A', 'B', 'C']
>>> m.clear()
>>> m
[ ]
```

List Comprehension

Definition:

List comprehension is an easy and shorter syntax to create a new list from an existing list or a string. List comprehension is faster than 'for' loop in processing list items.

Syntax:

```
[expression for element in iterable if condition]
```

List comprehension must be in square brackets [ ]

Part-1: expression - result will be sorted in new list
Part-2: for - one or more for loops on iterable object
Part-3: if - one or more if conditions (optional)

Application: Program to create a list of even numbers upto 10 WITHOUT List Comprehension

```
even_nums = []
for x in range(11):
    if x%2 == 0:
        even_nums.append(x)
print(even_nums)
```

Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

The exact same result can be obtained by using the following list comprehension syntax.

Application: Program to create a list of even numbers upto 10 WITH List Comprehension

```
even_nums = [x for x in range(11) if x%2 == 0]
print(even_nums)
```

Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

Explanation:

1. **for** loop is executed
2. Element **x** would be returned when the condition **if x%2 == 0** evaluates to True.
3. When the condition is True, expression **x** simply stores the value of **x** into a new list.

More examples - List Comprehension

Application: List Comprehension on String List

```
subjects1 = ['Math', 'Chem', 'Python', 'DataS']
subjects2 = [s for s in subjects1 if 'a' in s]
print(subjects2)
```

Output:

```
['Math','DataS']
```

Application: List Comprehension on range for squares

```
squares = [x*x for x in range(11)]
print(squares)
```

Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Nested Lists

Definition: A nested list is a list of lists. In otherwords, a list containing another list (or a sublist) as its element is called a nested list. A nested list can have any number of levels of nesting, i.e., a list can contain another list, which can contain another list, and so on. Nested lists are useful to arrange data in matrix format or a hierarchical structure.

Creating Nested List

A nested list is created by placing sublists separated by comma inside another list.

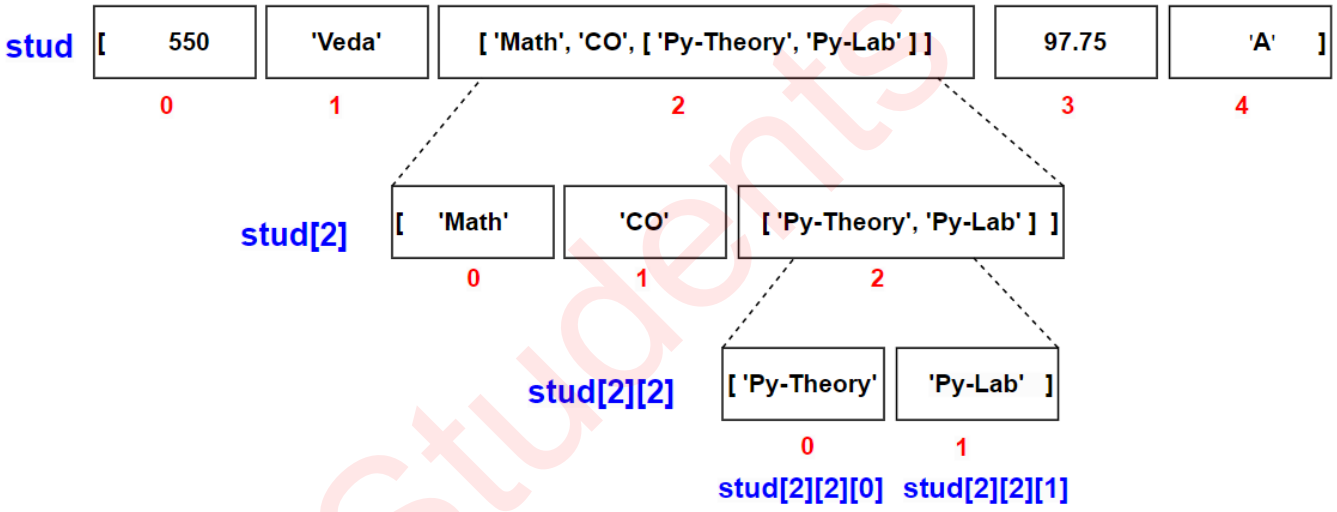
Syntax:

```
nested_list = [ [element11, element12, element13], [element21, element22, element23], . . . ]
```

Example: **stud = [550, 'Veda', ['Math', 'CO', ['Py-Theory','Py-Lab']], 97.75, 'A']**

Access Nested List elements by Index

You can access individual elements in a nested list using multiple indexes as illustrated below:



stud = [550, 'Veda', ['Math', 'CO', ['Py-Theory','Py-Lab']], 97.75, 'A']

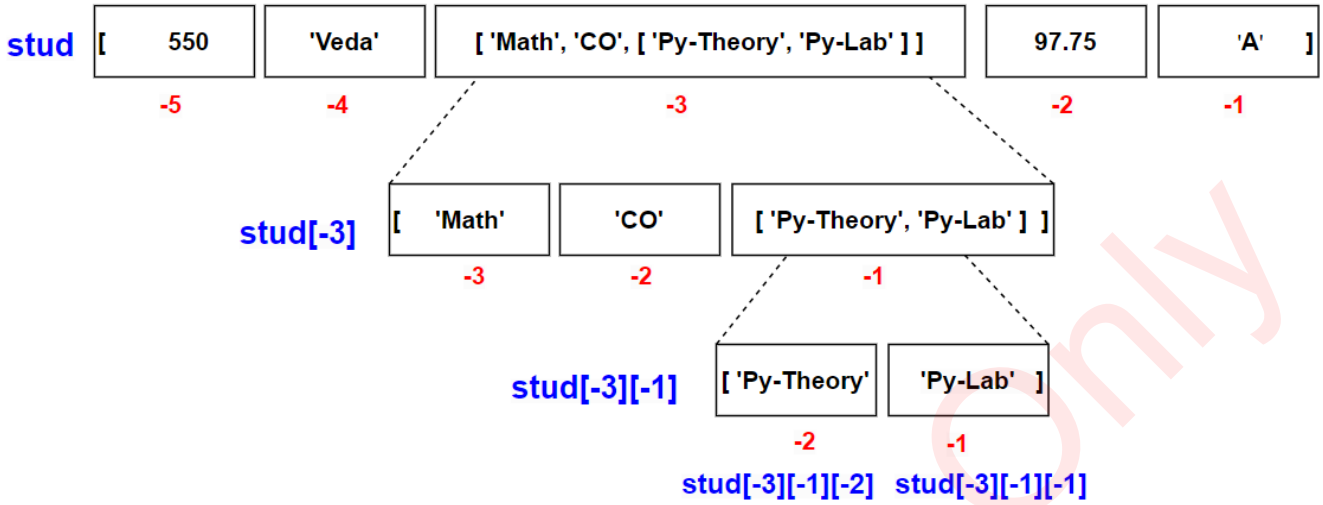
```
print(stud[2])
# Prints [ 'Math', 'CO', [ 'Py-Theory','Py-Lab' ] ]

print(stud[2][2])
# Prints [ 'Py-Theory','Py-Lab' ]

print(stud[2][2][0])
# Prints Py-Theory
```

Negative List Indexing In a Nested List

We can also access a nested list by its negative indexing. Negative indexes count backward from the end of the list. So, stud[-1] is the last item, stud[-2] is the second from last, and so on as illustrated below:




```
stud = [ 550, 'Veda', [ 'Math', 'CO', [ 'Py-Theory','Py-Lab' ] ], 97.75, 'A' ]
print(stud[-3])
# Prints [ 'Math', 'CO', [ 'Py-Theory','Py-Lab' ] ]

print(stud[-3][-1])
# Prints [ 'Py-Theory','Py-Lab' ]

print(stud[-3][-1][-2])
# Prints Py-Theory
```

Application: Changing items in nested list

```
matrix = [ [10,20,30], [40,50] ]
matrix[0][1] = 21  # changes element
print(matrix)      # [[10, 21, 30], [40,50]]
third = [70,80,90]
matrix.append(third) # Appends more items at the end of matrix
print(matrix)        # [[10, 21, 30], [40, 50], [70, 80, 90]]
matrix[1].insert(2,60)  # inserts element 60 at specified index 2
print(matrix)           # [[10, 21, 30], [40, 50, 60], [70, 80, 90]]
matrix.pop(2)        # deletes element at 2
print(matrix)        # [[10, 21, 30], [40, 50, 60]]
print(len(matrix))      # 2, finds length of matrix list
print(len(matrix[1]))   # 3, finds length of matrix element at index 1
```

Iterate through a Nested List

```
marks=[[10,20,30],[40,50,60]]
for mlist in marks:
    for num in mlist:
        print(num, end=' ')
# Prints 10 20 30 40 50 60
```

Application: Add two matrices (nested lists) using nested for loop

```
X = [[1,2,3],
     [4,5,6],
     [7,8,9]]
Y = [[10,20,30],
     [40,50,60],
     [70,80,90]]
result = [[0,0,0],
          [0,0,0],
          [0,0,0]]

for i in range(len(X)):          # iterate through rows
   for j in range(len(X[0])):    # iterate through columns
       result[i][j] = X[i][j] + Y[i][j]

for r in result:                 # prints result matrix
   print(r)
```

Output:

[11, 22, 33]
[44, 55, 66]
[77, 88, 99]

Application: Adds 3D matrices (nested lists)

```
M1 = [[[2, 4, 8], [7, 7, 1], [4, 9, 0]], [[5, 0, 0], [3, 8, 6], [0, 5, 8]]]
M2 = [[[3, 8, 0], [1, 5, 2], [0, 3, 9]], [[9, 7, 7], [1, 2, 5], [1, 1, 3]]]
result = [[[0,0,0], [0,0,0], [0,0,0]], [[0,0,0], [0,0,0], [0,0,0]]]
for i in range(0, 2):
    for j in range(0, 3):
        for k in range(0, 3):
            result[i][j][k] = M1[i][j][k] + M2[i][j][k]

for r in result:
   print(r)
```

Output:

[[5, 12, 8], [8, 12, 3], [4, 12, 9]]
[[14, 7, 7], [4, 10, 11], [1, 6, 11]]

 # Application: Program to add two matrices using list comprehension

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

Y = [[10,20,30],
     [40,50,60],
     [70,80,90]]

result = [[X[i][j] + Y[i][j]  for j in range(len(X[0]))] for i in
range(len(X))]

for r in result:
   print(r)
```

Application: Convert nested lits (list of lists) to dictionary

```
capitals = [['India','Delhi'],
            ['USA','Washington'],
            ['Australia','Canberra']]
capitals_dict = {x[0]: x[1] for x in capitals}
print(capitals_dict)
```

Output: {'India': 'Delhi', 'USA': 'Washington', 'Australia': 'Canberra'}

```
states = [['India','Delhi',28],
            ['USA','Washington',50],
            ['Australia','Canberra',6]]
states_dict = {x[0]: x[1:] for x in states}
print(states_dict)
```

Output: {'India': ['Delhi', 28], 'USA': ['Washington', 50], 'Australia': ['Canberra', 6]}

Dictionary Data Type

Dictionary Definition:

A Dictionary organizes data by association, not by position. A Dictionary associates a set of keys with values using **key : value** pairs of association. A value can be **referenced** by using the **key** name.

- Dictionary is a collection of Ordered, Mutable, Unindexed and does NOT allow duplicates.
- Dictionaries are written within curly braces **{ }** in the form of **key : value**.
- Dictionary is useful to access a large amount of data efficiently.

Dictionary Properties:

A. **Ordered -** The dictionary items have a defined order and the order will not change.
B. **Mutable or changeable -** Dictionaries are changeable. We can change, add or remove items after the dictionary has been created.
C. **Not Indexed -** The dictionary elements are NOT indexed; rather, the elements are **referenced** by using the **key** name.
D. **No Duplicates -** Dictionaries cannot have two items with the same key. Duplicate key:value will overwrite existing values.

Syntax - 1: Create Dictionary

```
dictname = { key-1:value-1, key-2:value-2, . . . key-n:value-n }
```

Syntax - 2: Create Dictionary using Constructor

👉 We can use the **dict()** constructor to create a dictionary in Python.

dictname = dict ((key-1:value-1, key-2:value-2, . . . key-n:value-n)) #notice the 2 parentheses

Example: Create Dictionary

```
a = {1:"Abdul",2:"Kalam", "age":60}
cars = {
  "brand": "Hyundai",
  "model": "Creta",
  "year": 2023
}
```

Accessing Dictionary:

Each element in a dictionary is a **key:value pair**. The **key** in each element can be used to access its **value** from the dictionary.

Syntax: dictname[key]

Example: Accessing Dictionary

```
# prints selected key's value
print(cars["brand"])  # Hyundai
print(cars["year"])   # 2023

print("Length of dict",len(cars))      # 3
print("Data type is",type(cars))       # <class 'dict'>
```

Accessing Dictionary KEYS or VALUES:

- **keys()** method is used to access all KEYS of a dictionary
- **values()** method is used to access all VALUES of a dictionary
- **items()** method reads both keys and values from a dictionary

Syntax:

dict.keys()
dict.values()
dict.items()

Example:

```
>>> cars = {
  "brand": "Hyundai",
  "model": "Creta",
  "year": 2023
}

>>> k=cars.keys()
>>> k
dict_keys(['brand', 'model', 'year'])
>>> v=cars.values()
>>> v
dict_values(['Hyundai', 'Creta', 2023])
>>>
>>> i=cars.items()
>>> i
dict_items([('brand', 'Hyundai'), ('model', 'Creta'), ('year', 2023)])
```

Ex: Generate a dictionary of squares up to number 5

```
>>> ds={n:n**2 for n in range(6)}
>>> ds
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Explain the built-in Dictionary Methods in Python

Python provides several built-in methods to manipulate a dictionary. These include accessing, adding, removing, copying, and clearing data elements in a given dictionary.

The built-in methods to perform these actions on a dictionary are shown below. **Note:** The methods pop(), popitem(), update(), and clear() do change **dictionary items** because the dictionaries are mutable.

Dictionary Built-in Methods		
dict Methods	**Description**	**Syntax**
keys()	Returns a list with all keys in a dictionary	**dict.keys()**
values()	Returns a list with all values in a dictionary	**dict.values()**
fromkeys()	Returns a new dictionary with given keys mapped to one value. If the value is not given it defaults to "None". Optional <value> if the key doesn't exist.	**dict.fromkeys(keys, <value>)**
items()	Returns a list with all keys & values in a dictionary	**dict.items()**
get()	Returns the value of the specified key. Optional <value> if the key doesn't exist.	**dict.get(key,<value>)**
pop()	Deletes a value at the given index	**dict.pop(key)**
popitem()	Deletes last key:value pair	**dict.popitem()**
update()	Changes/Adds the specified key:value pair	**dict.update(iterable)** **dict.update(str-key=value)**
copy()	Returns a copy of the dictionary	**newdict = dict.copy()**
clear()	Removes all the elements from the dictionary	**dict.clear()**
setdefault()	Returns the value of a key (if the key is in dictionary). If not, it inserts key with a value to the dictionary.	**dict.setdefault(key, value)**

(Memorize: **KV FIG PPUCCS**)

Application: Built-in methods on dictionary

```
# 1. Syntax:  dict.fromkeys(seq,val) method
countries = {'Ind', 'Eng','Aus'}
batters = [1,2]
# dict.fromkeys(seq,val)
teams = dict.fromkeys(countries,batters)
print(teams)  # {'Aus': [1, 2], 'Ind': [1, 2], 'Eng': [1, 2]}
batters.append(3)
print(teams)  # {'Aus': [1, 2, 3], 'Ind': [1, 2, 3], 'Eng': [1, 2, 3]}
print(teams.get("Aus"))   # [1, 2, 3]
```

```
# 2. Main dictionary built-in methods
grades = {"A":90, "B":80, "C":70,"D":60,"E":50,"F":40}
print(grades.get("B"))  # 80

grades.pop("D") # deletes the given key:value pair
print(grades)   # {'A': 90, 'B': 80, 'C': 70, 'E': 50, 'F': 40}
grades.popitem() # deletes last key:value pair
print(grades)   # {'A': 90, 'B': 80, 'C': 70, 'E': 50}

# updating str_key:value
grades.update(D=60,F=40) # adds new str_key:value pairs
print(grades)   # {'A': 90, 'B': 80, 'C': 70, 'E': 50, 'D': 60, 'F': 40}
grades.update(F=30) # changes value of given str_key
print(grades)   # {'A': 90, 'B': 80, 'C': 70, 'E': 50, 'D': 60, 'F': 30}

# updating num_key:value
sal={1:1000,2:2000}
s3={3:3500,4:4000}
sal.update(s3) # changes sal with s3
print(sal)   # {1:1000,2:2000,3:3500,4:4000}

grades2 = grades.copy()
print(grades2)  # {'A': 90, 'B': 80, 'C': 70, 'E': 50, 'D': 60, 'F': 30}
grades2.clear()
print(grades2)  # { }
```

```
# 3. Syntax:   dictname.setdefault() method
scores={'Python':90,'DS':70}
# returns value
m=scores.setdefault('Python')
print(m)        # 90
# inserts key with None
scores.setdefault('Math')   # adds 'Math': None
# inserts key with value
scores.setdefault('CO', 80) # adds 'CO': 80
print(scores)   # {'Python': 90, 'DS': 70, 'Math': None, 'CO': 80}
```

Section-2: Design with Functions

Functions improve efficiency and reduce errors because of their,

1. Modularity - Break down a bigger problem into smaller functions (Top-Down Parsing)
2. Reusability - Python uses a **DRY** (Don't Repeat Yourself) principle. It means, **Write a function once**, and **Call the function any time, anywhere**.

Definition:

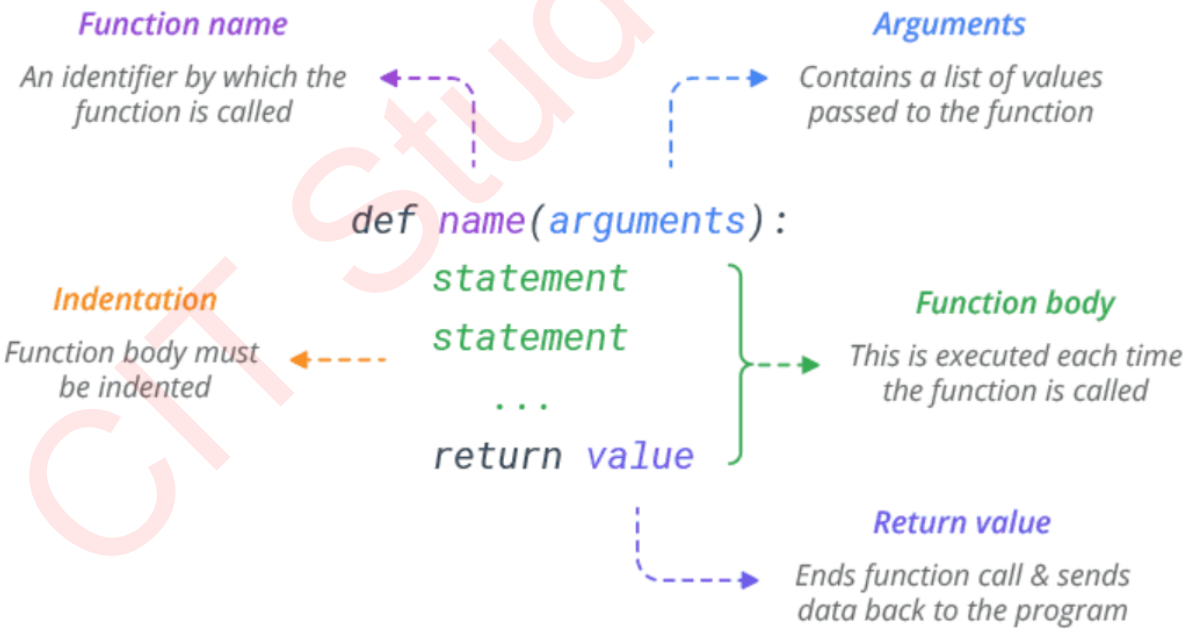
A function in Python is a block of statements that performs a specific task. Functions are useful when we must repeat the same task multiple times without rewriting the code.

- ➢ First, **'def'** keyword is used to **define a function**.
- ➢ Second, the **function must be called** to **run** it,
  - ○ We may pass arguments (values) to the function (optional).
- ➢ The function may return the result back to the calling area (optional).
- ➢ The function may return No value, Single value, or Multiple Values to the calling area (optional).

Syntax:

```
def function-name ( args ):
    statements
    return result
```

def - keyword to define a function
function-name - the name of the function
args - list of values passed into the function (Optional)
return - will send the value back to calling area (Optional)



Functions as Abstraction Mechanism:

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but the inner working is hidden. User is familiar with "what function does" but they don't know "how it does."
In Python, abstraction is used to hide irrelevant data in order to reduce the complexity. It also enhances the application efficiency.

Ex: Users just call a totalSalary() function to get a total salary but do not need to know how to do calculate it.

Categories of functions with examples

The functions in Python are categorized into 4 categories. Categories are decided based on **Argument Passing** and **Return value.**

Category	Pass Arguments form Calling Area to Called Function	Return Value form Called Function to Calling Area	Example
1. No Pass No Return	No	No	**add( ) print(10+5)**
2. No Pass Yes Return	No	Yes	**add( ) return (10+5)**
3. Yes Pass No Return	Yes	No	**add(a, b) print(a+b)**
4. Yes Pass Yes Return	Yes	Yes	**add(a, b): return (a+b)**

1. No Pass, No Return

- No arguments are passed to function from calling area
- No return value sent from the function to calling area

Example:

```
# Function category: No Arguments, No Return value
def add():  # Function Definition
   a,b=10,5
   print("sum = ",a+b)
# main program
add()   # Function call
Output: sum =  15
```

2. No Pass, Yes Return

- No arguments are passed to function from calling area
- The called function returns result back to calling area

Example:

```
# Function category: No Arguments, With Return value
def add():  # Function Definition
   a,b=10,5
   return a+b   # with return value
# main program
sum = add()   # Function call
print("Total = ",sum)
```

Output: **Total = 15**

3. Yes Pass, No Return

- Arguments are passed to function from calling area
- No return value sent from function to calling area

Example:

```
# Function category: With Arguments, No Return value
def add(a,b):  # Function Definition
   print("Total = ",a+b)   # No return value
# main program
add(10,5)   # Function call
```

Output: Total = 15

4. Yes Pass, Yes Return

- Arguments are passed to function from calling area
- Return value is sent from function to calling area

Example:

```
# Function category: With Arguments, With Return value
def add(a,b):  # Function Definition
   return a+b   # with return value
# main program
sum = add(10,5)   # Function call
print("Total = ",sum)
```

Output: Total = 15

Compare a Function, a Fruitful Function, and an Anonymous Function with an example for each.

The functions are, mainly, divided into **2 categories**:

1. Built-in Functions or Standard Library Functions,
2. User-defined Functions.

The user-defined functions are further classified into

- **Functions (Non-fruitful functions or Void functions)**
- **Fruitful functions**
- **Anonymous or Lambda Functions,**
- **also, Recursion Functions.**

Functions or Non-fruitful Functions:

Definition:

A function that doesn't return any value is called a **non-fruitful function** or a void function.

Syntax:

```
def function-name ( args ):
    statements
```

def - keyword to define a function
function-name - the name of the function
args - list of values passed into the function

Example: **Non-Fruitful function**

```
def add(a, b):
  print(a+b)
```

Fruitful functions:

Definition:

A fruitful function in Python is a **function that returns a value** after performing some operation.

We use the **'def'** keyword to define a function. A function takes input arguments (or parameters), processes them, and returns a result. The return value can be of any data type, such as int, float, double, string, or a custom class.
Note: The result is returned to the calling area as a "**fruit**".

Syntax:

```
def function-name ( parameters ):
    statements
    return result
```

def - keyword to define a function
function-name - the name of the function
parameters - list of values received in the function
return - will send the result back to calling area

Example: Fruitful function

```
def add(a,b):
    return a+b
sum = add(50,30)
print(sum)     # 80
```

Anonymous functions or Lambda functions:

Definition:

Anonymous function is a function that has NO NAME when it is defined. It is also called **a lambda** function. The '**lambda' keyword** is used to create the lambda functions. **Lambda functions are restricted to a single code or expression.**

A **lambda** function can take **any number of arguments** but only have **one expression**.

Syntax:

```
lambda arguments : expression
```

Example:

```
# Lambda Function
sum = lambda a,b: a+b
print(sum(10,5))
print(type(sum))     #<class 'function'>
```

What is a lambda function? Describe its characteristics with an example.

A lambda function is a function that has **NO NAME** when it is defined. It is also called **an anonymous** function. We use lambda functions when we need a nameless function for a **short period of time**.

The '**lambda**' is a keyword in Python for defining the anonymous function.

A **lambda** function can take **any number of arguments** but only have **one expression**.

- Lambda functions are stored in a variable and created at run time.
- We can pass the lambda function as an argument to a higher-order function (a function that takes in other functions as arguments).
- Lambda functions can be used inside another function.

Syntax:

```
lambda arguments : expression
```

Example-1: Assigning Lambda to a variable

```
sum = lambda a,b: a+b
print(sum(10,5))
```

Or

Example-2: Not Assigning Lambda to a variable

```
print( (lambda a, b: a + b)(10,5) )
```

=> The advantage of lambda functions is best used when they are used inside another function.

Example-3: * Real use of Lambda function is Inside Another Function *****

```
def power(n):
    return lambda x: x ** n
# SET power value
square = power(2)
cube = power(3)
# SEND base value
print(square(5))
print(cube(5))
```

Output:

25
125

The Characteristics of Lambda Functions:

The lambda function,

1. takes **many arguments** but has only **one expression**.
2. is restricted to **return a single expression**.
3. is used as an **anonymous function inside other functions**.
4. **does not need a return statement**, they always return a single expression.

Applications of Lambda Functions
also, **Applications of Higher Order Functions**

(**Note:** These examples can be written for both Lambda Functions and also for Higher Order Functions)

Lambda functions are used along with built-in functions like **filter(), map(), reduce()** etc.

➔ Lambda with filter()

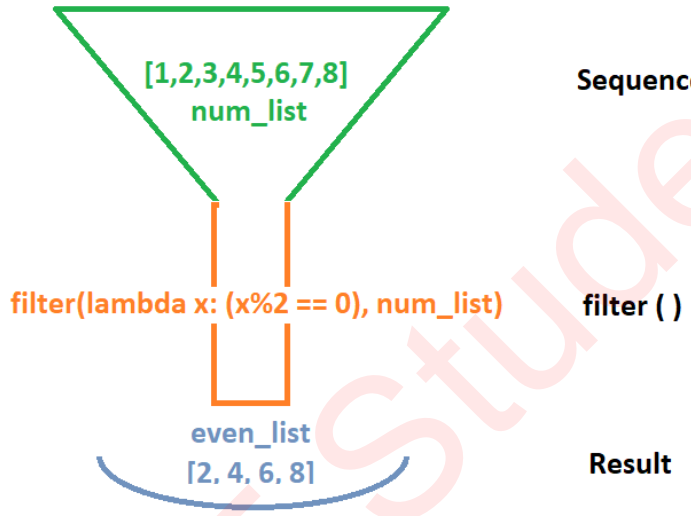
filter() function selects qualified elements **from** an iterable sequence based on the result of a function.

Syntax:

filter(function, iterable)
function - a function iterable - an iterable like sets, lists, tuples, etc.

Example:

```
num_list = [1,2,3,4,5,6,7,8]
even_list = list(filter(lambda x: (x%2 == 0) , num_list))
print(even_list) # [2, 4, 6, 8]
```



➔ Lambda with map():

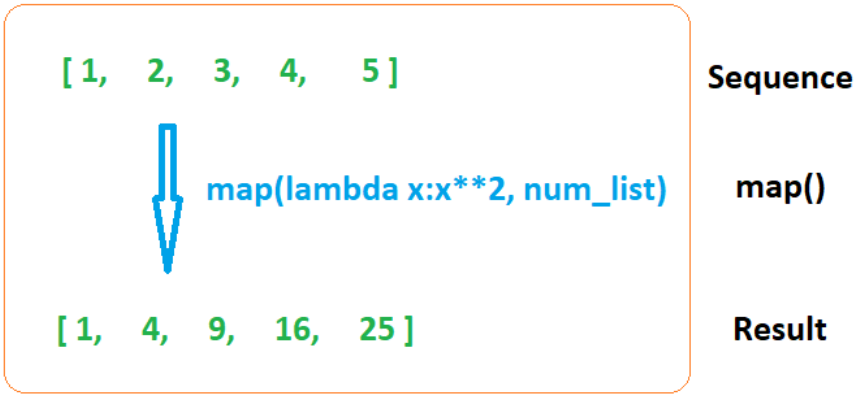
map() method applies a given function **to** each element of an iterable sequence (list, tuple etc.) and returns a sequence containing the results. We can pass more than one iterable sequence to the map() function.

Syntax:

map(function, iterable, ...)
function - a function iterable - an iterable like sets, lists, tuples, etc

Example:

```
num_list = [1,2,3,4,5]
squares_list=list(map(lambda x:x**2, num_list))
print(squares_list)       # [1, 4, 9, 16, 25]
```



➔ **Lambda with reduce():**

reduce() method applies a given function **to** all element of an iterable sequence (list, tuple etc.) and returns a single value. The reduce() method is similar to "for" loop in Python. The reduce() method is optimized and faster than "for" loop.

Note: The reduce() function in python is defined in "functools" module. We need to import "functools" before calling the reduce() function in our program.

Syntax:

```
from functools import reduce

reduce(function, iterable)
```

Example:

```
from functools import reduce
quantity = [10, 20, 30, 40]
result = reduce(lambda x, y: x + y, quantity)
print(result)
```

Output: 100

Explanation: This python program returns the sum of all values in the list as a single value. It uses the lambda function as (((10+20)+30)+40).

Docstring in Functions

Docstrings are enclosed in triple quotes for multi-line descriptions.

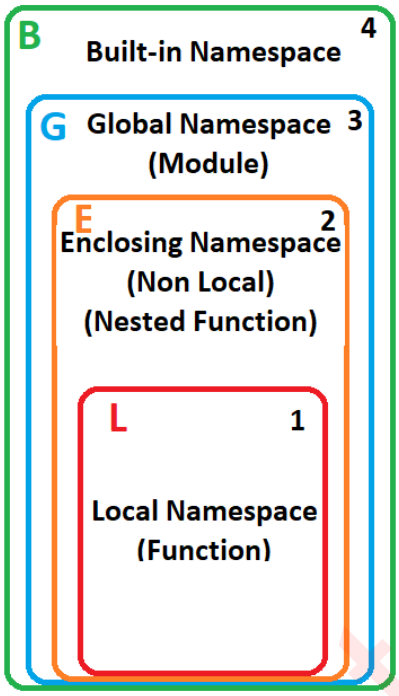
We can attach documentation to a function definition by including a string literal after the function header.

```
>>> def double(n):
...     """Author: Poojitha
...        Version: 1.0
...        This function doubles the given number
...        and prints the result on the screen.
...        syntax: double(number)
...     """
...     print(n*2)
...
...
>>> double(7)
14
>>> help(double)
Help on function double in module __main__:

double(n)
    Author: Poojitha
    Version: 1.0
    This function doubles the given number
    and prints the result on the screen.
    syntax: double(number)
```

Namespace, Scope and Lifetime of variables in Python

👉 A **Namespace** is a collection of defined names along with information about the object that each name refers to. Python has 4 types of namespaces.



👉The **Scope** is the region of the program where a variable is defined and can be accessed.

👉The **Lifetime** is the period of time during which a variable is available in the memory and can be accessed. The lifetime of a variable depends on its scope and how it was defined.

Note: A lifetime of a namespace depends upon the scope of objects, if the scope of an object ends, the lifetime of that namespace also ends.

LEGB Rule: When we read a variable, the Python interpreter will retrieve the variable in by looking up sequentially in the order of LEGB scope. That means, the first occurrence of this variable found in any of the scopes sequentially from, Local -> Enclosed -> Global -> BuiltIn, will be returned

Namespace	Scope (Accessible Area)	Lifetime (Accessible Period of Time)
Built-in Namespace	Python's built-in functions and variables (list, len, pow, round) can be accessed from anywhere in the program without using **import** statements. `>>> dir(__builtins__)`	Throughout the execution of the program. Usually determined by the Python Interpreter.

Global Namespace	Variables defined **outside of any function** have global scope. Accessed from anywhere in the program.	Available in memory during the execution of the whole program. They are only destroyed when the program terminates.
Enclosing Namespace	Variables defined in the outer function of a **nested function** have an enclosing scope. These variables can be accessed by the nested function.	Available during the nested function is being executed. Once the nested function finishes, the variables are destroyed and their memory is freed.
Local Namespace	Variables defined **within a function** have local scope. These variables can only be accessed within that function.	Available only during the execution of a function where they are defined. Once the function finishes, the variables are destroyed and their memory is freed.

The scope and lifetime of variables in Python are used to avoid naming conflicts, better use of memory, and write efficient code.

Example: Local scope variable

```
def rank():
  x = 100    # Local, x can be used only in rank() function
  print(x)
rank()
```

Output: 100

Example: Global scope variable

```
def rank():
    print(x)
    global y     # global keyword makes the variable global
    y=200        # y is global, can be used in all functions
x=100   # x is global, can be used in all functions
rank()
print(y)
```

Output: 100
200

Example: Built-in scope variable

```
from math import pi
def pival():
   print('Local scope: ', pi)
print('Global scope: ', round(pi))
pival()
```

Output:
Global scope: 3
Local scope: 3.141592653589793

Example: Enclosed scope variable (also called Non-Local scope)

```
# Enclosed/Non-Local scope in child()
def parent():
   a = 10
   def child():
       print('child ', a) #10, a has Enclosed or Non-Local scope in child()
   child()
parent()
```

Output:
Child 10
Parent 10
 # Note: a in child() is neither Global nor Local. Hence, a is Enclosed in child()

```
# Local scope in child()
def parent():
   a = 10
   def child():
       a = 20
       print('Child ', a) #20, a has Local scope in child() as per LEGB rule
   child()
   print('Parent ',a)
parent()
```

Output:
Child 20
Parent 10

What are the different types of arguments (or parameter passing) in Python functions? Justify with suitable examples

The Function Arguments in Python are also called Formal arguments. We can call a function by using the following 4 types of formal arguments.

1. **Required arguments (Positional arguments)**
2. **Keyword arguments (Named arguments)**
3. **Default arguments**
4. **Variable-length arguments (or Arbitrary arguments)**

1. Required Arguments or Positional Arguments:

Explanation:

Required or Positional arguments are values assigned to the arguments by their position when the function is called. Ex: 1st value to 1st argument, 2nd value to 2nd argument, and so on.

> 👉Values must be required for all arguments according to their position. We must pass values in the same sequence defined in a function definition.

➔ By default, Python functions are called by using the positional arguments.

Application:

```
# Required arguments
def student(name, marks):
    print('Details:', name, marks)

# Function call
student('Sumanth', 15)
```

Output:

Details: Sumanth 15

2. Keyword Arguments:

Explanation:

The **Keyword Argument** is also called a **Named Argument**. We can change the sequence of keyword arguments by **using their name in function calls**. When we call functions in this way, the **order** (position) of the arguments **can be changed**.

Application:

```
# Keyword arguments
def student(name, marks):
    print('Details:', name, marks)
```

```
# Function Call: both Keyword arguments
student(name='Varma', marks=14)

# Function Call: 1 positional and 1 keyword
student('Venu', marks=12)

# Function Call: both Keyword arguments in different order
student(marks=15, name='Varshitha')
```

Output:
Details: Varma 14
Details: Venu 12
Details: Varshitha 15

3. Default Arguments:

Explanation:
The function **arguments can have default values**. We can assign default values to the arguments using the '=' (assignment) operator when defining a function. We can set a default value to any number of arguments.

- The default value will be used if we do not pass a value to that argument.
- If we pass a value, then the passed value will override the default value.

Application:

```
def student(name, marks, college="CIT"):
    print('Details:', name, marks, college)

# Passed only the required arguments
student('Vasanthi', 95)
```

Output:

```
Details: Vasanthi 95 CIT
```

4. Variable-length Arguments or Arbitrary Arguments:

Explanation:

We use **variable-length arguments** if we do not know the number of arguments to pass into a function. We can pass multiple arguments into a function. Internally all these values are represented in the form of a **tuple**.

Python has 2 types of Variable-length arguments as follows:

Type	Variable-length Positional Arguments	Variable-length Keyword Arguments
Declaration	**(*args)** * followed by 1 argument name	**(**kwargs)** ** followed by 1 argument name
Syntax	**def f-name(*args):** **statements**	**def f-name(**kwargs):** **statements**
Explanation	• Asterisk operator(*) is used • We can pass multiple positional arguments to a function	• Unpacking operator(**) is used • We can pass multiple keyword arguments to a function • The kwargs are accessed using key-value pair (same as accessing a Dictionary).

Examples:

Application: Variable-length Positional Arguments (*args)

```
def add(*scores):
    sum = 0
    for i in scores:
        sum += i
    print("Total= ", sum)

# main program
# Function called with variable arguments
sum = add(60,50)    # 110
sum = add(60,50,70)      # 180
```

Output:

```
Total=  110
Total=  180
```

Application: Variable-length **Keyword Arguments (**kwargs)**

```
def totalmarks(**sub_marks):
    total = 0
    for i in sub_marks:
        # get subject name
        sub = i
        # get subject value
        marks = sub_marks[i]
        total = total+marks
        print(sub, "=", marks)
    print("Total (Variable KW Args)=",total)
# pass multiple keyword arguments
totalmarks(math=60, chem=50, python=70)
totalmarks(chem=50, math=60, python=70)
```

Output:

```
math = 60
chem = 50
python = 70
Total= 180
```

How to pass a list into a function? Explain with an example program.

We can pass any data type (list, dict, str, number, etc) as an argument into a function.

A list is a sequence or collection of many elements of the same or different data types.
When we pass a list into a function as an argument.
The passed list is still treated as a list inside the function.

Example:

```
def semester(subjects):
  for s in subjects:
    print(s)

# Passing list into a function
subjects = ["Chem", "Math", "Python"]   # List defined
semester(subjects)  # Passed list into the function
```

Output:
Chem
Math
Python

What is recursion in Python? Write a program to find the factorial of a given number using recursion.

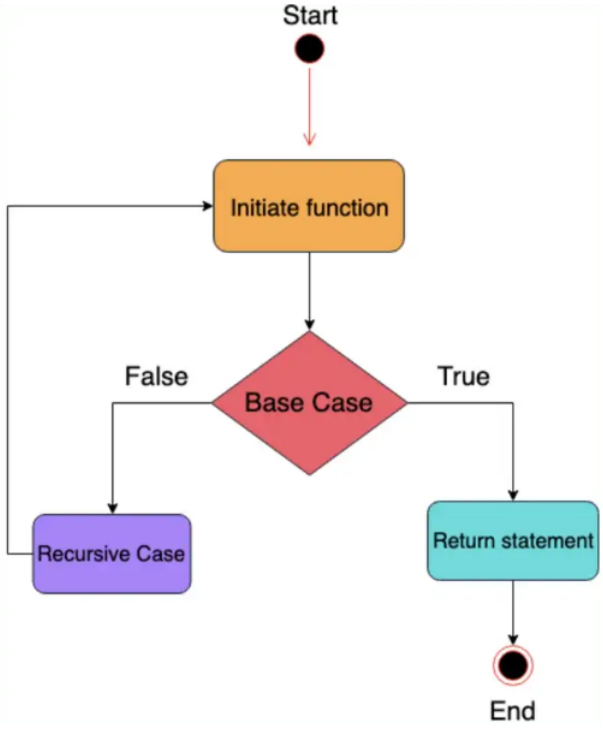
Definition:

> **A function called by itself repetitively is called a recursive function. The function call is termed a recursive call.**

The recursive function will call itself multiple times until a condition is satisfied. The recursive functions should be used very carefully because, when a function is called by itself it enters into the infinite loop. And when a function enters into the infinite loop, the function execution never gets completed.

👉We MUST define the condition to exit from the function call so that the recursive function gets terminated.

Flowchart of Recursive Function:



The recursive function has two main parts in its body,

1. **the base case** (condition) and
2. **the recursive case (recursive function call).**

The flow of execution of Recursive Function:

- first, the program checks the base case condition.
- If it is TRUE, the function returns and quits;
- otherwise, the recursive case is executed by calling the function recursively.

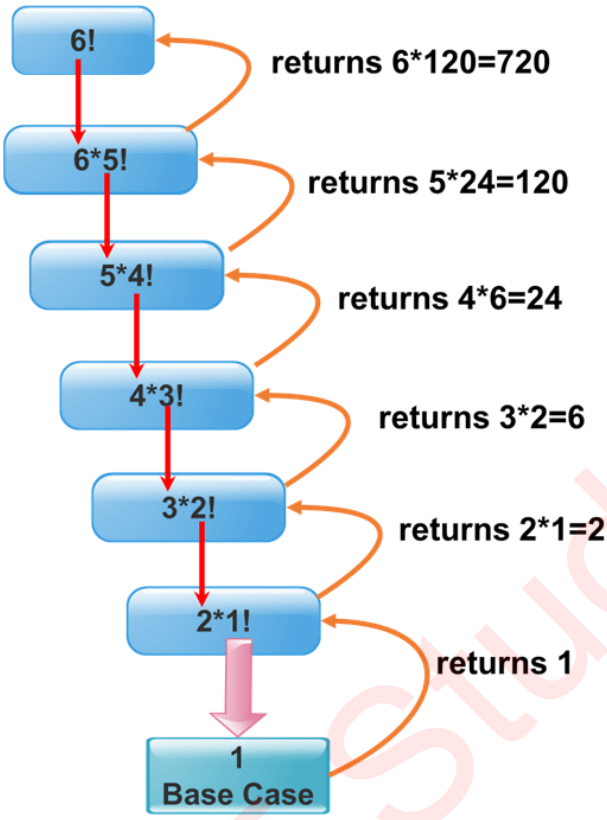
Syntax: Recursion in a Python

```
def recursive_function(argument)
{    # base case condition
     if base_case == True:
        return result
     # recursive case
     else:
        return recursive_function(argument) #recursive call
}
```

Application: Python Program to Find Factorial of a given integer number

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)  # Recursive call
print("Factorial is: ", fact(6))    # First Function call
```

Output: **Factorial is: 720**



Ex: The recursive function call execution **to find factorial of 6**.

Flow of recursion:

fact(6)
6 * fact(5)
6 * 5 * fact(4)
6 * 5 * 4 * fact(3)
6 * 5 * 4 * 3 * fact(2)
6 * 5 * 4 * 3 * 2 * fact(1)
6 * 5 * 4 * 3 * 2 * 1 = **720**

Application: Find Fibonacci Series of a given number of terms using Recursion Function

```
def fib(i):
    if (i == 0):
        return 0
    if (i == 1):
        return 1
    return fib(i - 1) + fib(i - 2)

n=int(input("Enter terms for Fibonacci series: "))
for i in range (n):
      print(fib(i),end=" ")
```

Output: Enter terms for Fibonacci series: 7
0 1 1 2 3 5 8

Advantages of Recursive Functions:

1. We can Reduce the length of the code,
2. We can Improve the Readability of code,
3. We can Solve complex problems.

Disadvantages of Recursive Functions:

1. Need more memory and time for execution,
2. Debugging is difficult.

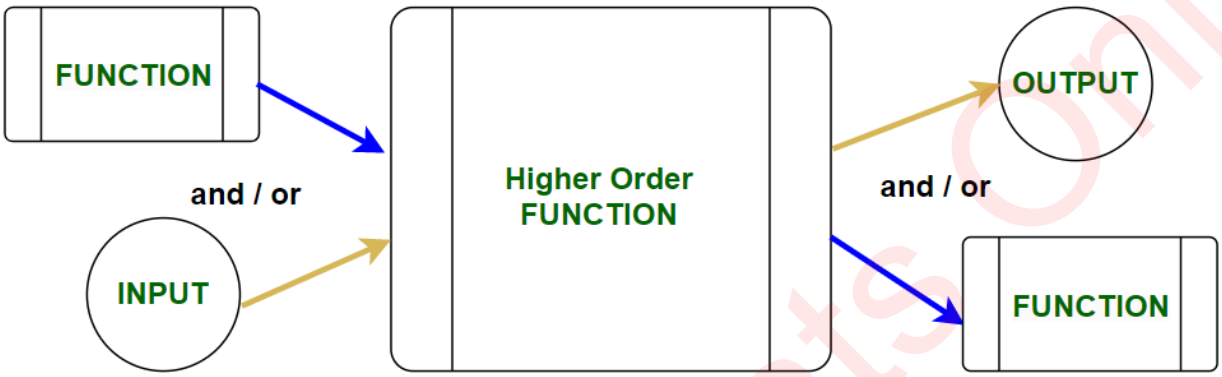
What are Higher Order Functions? Explain them with an example program.

Definition:

In Python, a higher-order function is a function that takes one or more functions as arguments or returns a function as its result.

Hence, the functions that operate with another function are known as Higher-order Functions.

Note: The higher-order functions can manipulate other functions by treating them like any other variable.



Characteristics of higher-order functions:

- A function is an instance of the Object type.
- We can store the function in a variable.
- We can pass the function as a parameter to another function.
- We can return the function from a function.
- We can store them in data structures such as hash tables, and lists, etc.

Function as an object:

In Python, a function can be assigned to a variable. This assignment does not call the function, instead, a reference to that function is created.

Example:

```
# Function as an object variable
def caps(name):
    return name.upper()
# Assigning function to a variable
upper_name = caps
print("Function object upper_name = ", upper_name('Dhanush'))
```

Output:

Function object upper_name = DHANUSH

Explanation:
In this example, the caps() function is assigned to a variable called upper_name. The upper_name is a function object that points to the function caps().

Passing Function as an argument to other function:
Functions are like objects in Python. Therefore, the functions can be passed as arguments to other functions. In the following example, we have created a function speak() that takes another function as an argument.

Example:
```
# Function passed as an argument to other functions
def caps(name):
    return name.upper()
def smalls(name):
    return name.lower()
def convert(farg):
    # storing the function in a variable
    result = farg("Function passed as an argument.")
    print(result)
convert(caps)
convert(smalls)
```

Output:
FUNCTION PASSED AS AN ARGUMENT.
function passed as an argument.

Returning function:
Since the functions are objects, we can return a function from another function.
Example:
```
# Functions that return another function
def add_salary(s):      # s=1000
    def add_bonus(b):   # b=100
        return s + b
    return add_bonus
sal = add_salary(1000) # sal = add_bonus
print("Total Salary & Bonus is ",sal(100))
```

Output:
Total Salary & Bonus is 1100

Functions Return Multiple Values: A function in Python can also return **Multiple values.**
[Note: Multiple values cannot be returned from a function in C, C++, or Java. **]**

1. **Return value as Tuple** - A Tuple is a sequence of items separated by a comma with or without (). Tuples are Ordered, Immutable, Indexed, Allows Duplicates.

```
def detailsTuple():
    semester = "CIT Sem-2 2023"
    students = 600
    #return semester, students # Return a tuple without ()
    return (semester, students)  # Return a tuple with ()
# Returns as Tuple (Method Call)
sem, std = detailsTuple() # Assigning returned tuple
print(sem, std)
Output:   CIT Sem-2 2023 600
```

2. **Return value as List** - List is a sequence/collection of values of different data types enclosed in [ ]. Tuples are Ordered, Mutable, Indexed, Allows Duplicates.

```
def detailsList():
    semester = "CIT Sem-2 2023"
    students = 600
    return [semester, students] # Returns a list
# Returns as List   (Method Call)
slist = detailsList()
print(slist)
Output:   ['CIT Sem-2 2023', 600]
```

3. **Return value as Dict** - A Dictionary is a sequence/collection of Key-Value pairs of different data types enclosed in { }. Dict is Ordered, Mutable, Not-Indexed, No Duplicate Keys but Duplicate Values are allowed.

```
def detailsDict():
    d = dict()
    d["semester"] = "CIT Sem-2 2023"
    d["students"] = 600
    return d # Returns the dictionary
# Returns as Dictionary   (Method Call)
sdict = detailsDict() # Assigning returned dictionary
print(sdict)
Output:   {'semester': 'CIT Sem-2 2023', 'students': 600}
```

4. **Return value as Object** - We can create a class (similar to a struct in C) to hold multiple values and return an object of the class.

```
# Function returns multiple values using Class-Object
class FirstYear:
    def __init__(self):
        self.semester = "CIT Sem-2 2023"
        self.students = 600

# Method returns multiple values using Class-Object
def details():
    return FirstYear()  # Returns Class-Object

s = details()
print(s.semester)
print(s.students)

Output:  CIT Sem-2 2023
         600
```

5. **Return value using yield** - The yield keyword generates a sequence of values, one value at a time. To return multiple values from a generator function, you can use the **yield** keyword to yield each value in each turn and continues until the generator function completes execution or encounters a return statement.

```
# Function returns multiple values using yield
def get_data():
    yield 522007
    yield 'CIT'
    yield [60,80,100]

# Method call
data = get_data()
print(next(data))  # Prints 522007
print(next(data))  # Prints 'CIT'
print(next(data))  # Prints [60,80,100]
Output:  522007
         CIT
         [60, 80, 100]
```

Section-3: Module

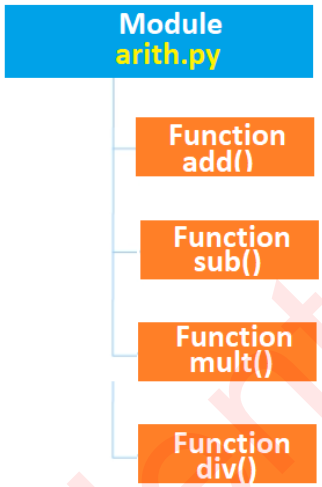
What are Modules and Packages in Python?

Modules in Python:

👉A Python **Module** is a Python File (**. py**) that contains **collection of Functions** and Global variables.

A module is a simple Python file with several functions that can be used to provide different functionalities in a program. The Python modules serve as a ready-made library available to programmers and users.

In the following example, we created a **module "arith.py"**, imported that module into a **program "arith_calc.py"**.



File name: arith.py

```
# Arithetic module
def add(a,b):
    print(a+b)
def sub(a,b):
    print(a-b)
def mult(a,b):
    print(a*b)
def div(a,b):
    print(a/b)
```

File name: arith_calc.py

```
# Importing our own Module arith.py
import arith
arith.add(10,5)
arith.sub(10,5)
arith.mult(10,5)
arith.div(10,5)
```

Output:

```
15
5
50
2.0
```

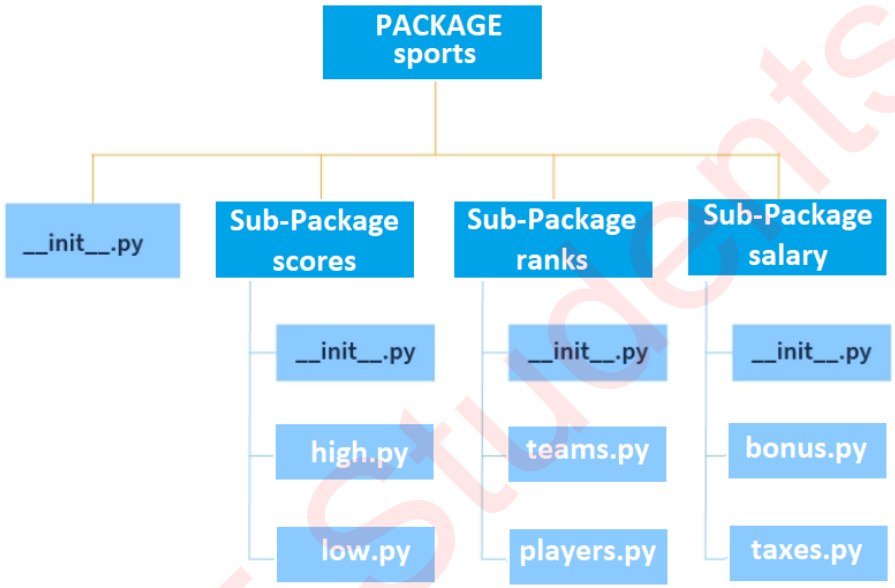
Packages in Python:

👉A Python **Package** contains,

1. **Collection of Sub-Packages or Modules** (related modules are put in a same package) and
2. **__init__.py** file through which Python interpretes it as a package. This file stores contents of the package. **Note:** The **__init__.py** Python file works as a **Constructor** for the Python Package.

👉When a module from an external package is required in a program, that package can be imported and its modules can be used.

👉Python packages ensure modularity by dividing the packages into sub-packages, making the Python project easier to maintain.




```
# math package
import math
# Square root
sqrt_val = math.sqrt(16)
print(sqrt_val)  # Output: 4.0
```

a. How to import specific attributes from a module into the current Namespace. Illustrate it with an appropriate code.

1. Importing Module From a Package:

Packages help in reusability of code. We use "**import**" statement in Python program to use a module. We can import only a module from a package using a dot operator (.).

Syntax:

```
import module1[, module2,... moduleN]

import package.sub-package.module

from package import module
```

Examples:

```
Import arith

import sports.ranks

from sports import ranks
```

2. Importing specific attribute from a Module:

To access a function called **top_rank()** of the "**rank**" module, you use the following code:
Writing.Book.edit.plagiarism_check()

```
import sports.ranks

sports.ranks.top_rank()
```

b. Briefly describe and illustrate any three Python packages with an example program.

The commonly used packages in Python are **NumPy, Pandas, SciPy, Matplotlib, Selenium, math, random or statistics,** etc.

Let's briefly describe and illustrate 3 packages:

1. **Math -** for mathematical operations**,**
2. **Random -** random number generation**, and**
3. **Statistics -** statistical analysis**.**

1. **Math:**

The math package is a built-in package that provides various mathematical functions and constants. It is a library of basic and advanced mathematical functions for arithmetic, trigonometry, logarithms, exponentials, etc.

- It helps us write prgrams with ease and efficiency.
- We need to import the math package before using its functions.

Here's an example:

```
# Module math
import math
# Square root
sqrt_val = math.sqrt(16)
print(sqrt_val)  # Output: 4.0

# Trigonometric functions
sin_val = math.sin(math.pi / 2)
print(sin_val)  # Output: 1.0

# Logarithm
log_val = math.log(10, 2)
print(log_val)  # Output: 3.3219280948873626

# Constants
print(math.pi)      # Output: 3.141592653589793
print(math.e)       # Output: 2.718281828459045
print(math.inf)     # Output: inf
print(math.nan)     # Output: nan
```

2. Random Package:

The random package is a built-in package in Python that is used to produce pseudo-random numbers. The numbers are not exactly random but are generated by computer algorithms.

- The Python Random Module can be used to generate a random integer, choose a random element from a list, rearrange items randomly, etc.
- We need to import random at the beginning of our code to use the Python Random Module.

Here's an example:

```
# Module random
import random
# Random integer between a range
rand_int = random.randint(1, 10)
```

```
print(rand_int)  # Output: 4

# Random float between 0 and 1
rand_float = random.random()
print(rand_float)  # Output: 0.6627291929320501

# Random selection from a list
my_list = [1, 2, 3, 4, 5]
rand_choice = random.choice(my_list)
print(rand_choice)  # Output: 3
```

3. Statistics Package:
The statistics package provides statistical functions to analyze and calculate mathematical statistics of numeric data. The statistics module was new in Python 3.4.

Here's an example:

```
# Module statistics
import statistics

# Mean
data = [1, 2, 3, 4, 5]
mean_val = statistics.mean(data)
print(mean_val)  # Output: 3

# Median
median_val = statistics.median(data)
print(median_val)  # Output: 3

# Standard deviation
std_dev = statistics.stdev(data)
print(std_dev)  # Output: 1.5811388300841898

# Variance
variance = statistics.variance(data)
print(variance)  # Output: 2.5
```

What is PIP? Explain how to install packages using PIP with at least 2 examples.

- Python packages are published to the **PyPI** (Python Package Index). PyPI hosts an extensive collection of packages, including development frameworks, tools, and libraries.
- **PIP (P**referred **I**nstaller **P**rogram**)** is the package manager that maintains packages from **PyPI** on our PC. Use **pip3** if you installed Python from the Python website or the Microsoft Store.

Requirement:
Before installing Python packages, make sure Python is installed on your machine.

➔ **To install a package using pip3:**
open a Terminal on Command Prompt on Windows
Syntax:

```
pip3 install {package_name}
```
{package_name} refers to a package you want to install.

➔ To install the **numpy** package, you would type:
pip3 install numpy

➔ To install the **scipy** package, you would type:
pip3 install scipy

➔ **To list all the installed packages:**
pip3 freeze
Or
pip3 list

```
Package            Version
------------------------ -----------
numpy              1.24.1
scipy              1.10.1
```

Note: If the package has dependencies (i.e., it requires other packages for it to function), pip3 will automatically install them as well.

➔ **To use a package in Python program:**
Once the installation is complete, you can import the package into your Python code.

Ex: If you installed the **numpy** package, you could import it and use it as follows.

```
import numpy as np
arr = np.array(["I", "love", "Python", "package", "management"])
```

➔ **To update a package to the latest version:**
pip3 install --upgrade {package_name}

Ex:
To update the **numpy** package to the latest version, use the following command:
pip3 install --upgrade numpy

➔ **To uninstall a package:**
pip3 uninstall {package_name}

Ex:
pip3 uninstall numpy