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.

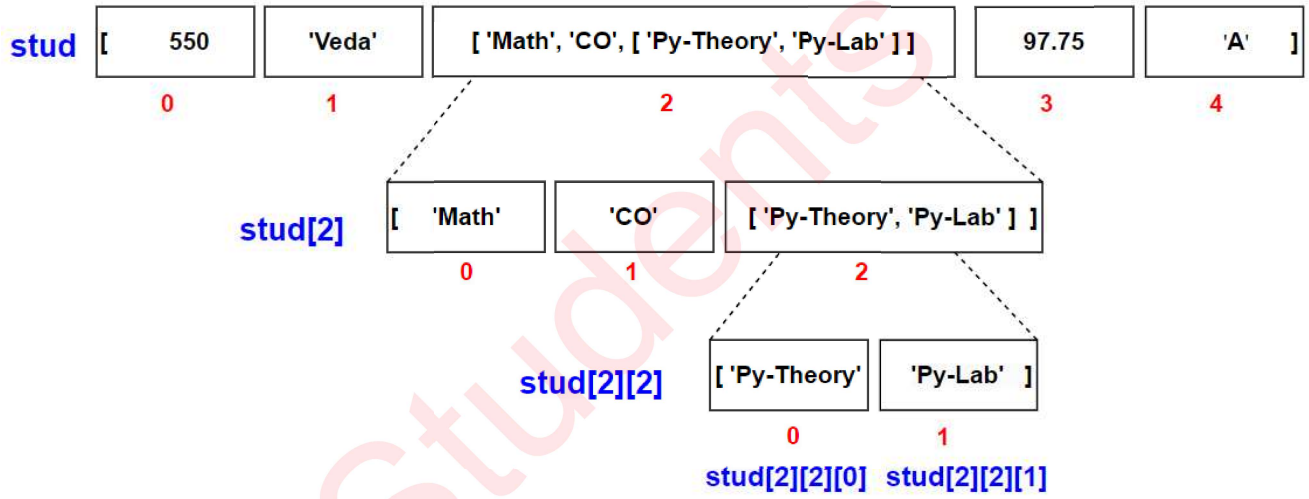
Syntax:

```
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']

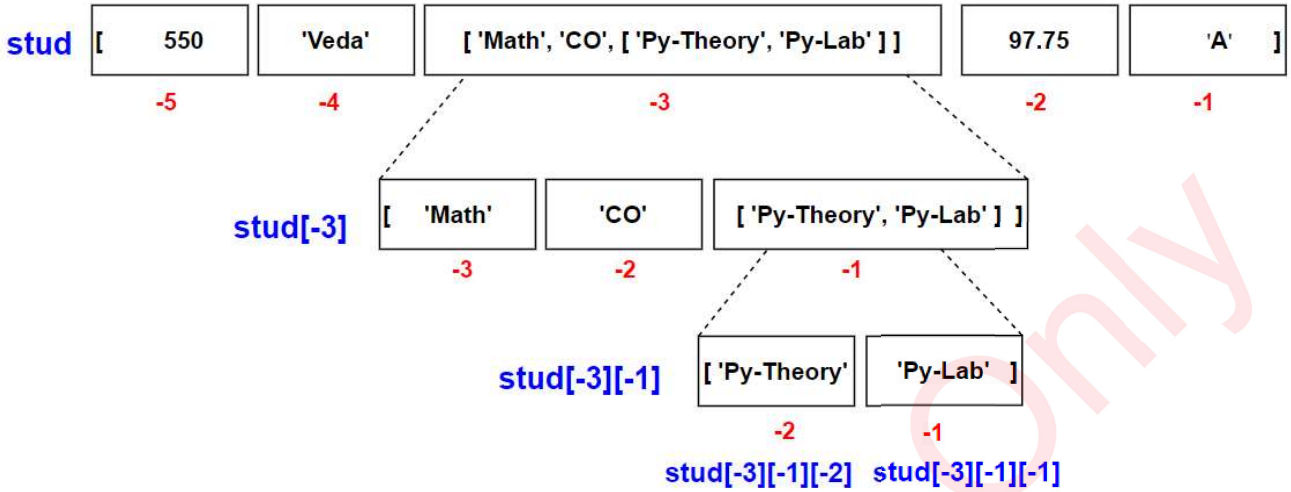
```
print(stud[2])
# Prints [ 'Math', 'CO', [ 'Py-Theory','Py-Lab' ] ]

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}
```