Module-III: Strings, Lists, Tuples, and Dictionaries

Strings, Lists, Tuples, and Dictionaries: Strings- Operations, Slicing, Methods, List- Operations, slicing, Methods, Tuple- Operations, Methods, Dictionaries- Operations, Methods, Mutable Vs Immutable, Arrays Vs Lists, Map, Reduce, Filter, Comprehensions.

III.1 Introduction to Strings and operations

A String is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once you have created a string, you cannot change it.

- Strings are used widely in many different applications, such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.

Creating a String in Python: Strings in Python can be created using single quotes or double quotes or even triple quotes. Create a string using single quotes (' '), double quotes (" "), and triple double quotes (""" """). The triple quotes can be used to declare multiline strings in Python.

Example:

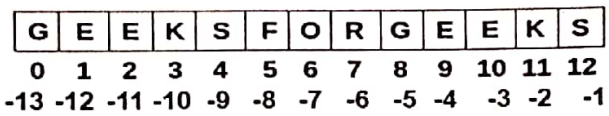
```
# Python Program for Creation of String
# creating a String with single Quotes
String1 = 'Welcome to Python programming'
Print ("String with the use of Single Quotes: ")
print (String1)
# creating a String with double Quotes
String2 = "Welcome to Python programming"
print ("\n String with the use of Double Quotes: ")
Print (String2)
# creating a String with triple Quotes
String3 = '''I'm D Samantha author of this book'''
print("\n String with the use of Triple Quotes: ")
print (String3)
```

Output:

```
String with the use of Single Quotes:
Welcome to Python programming

String with the use of Double Quotes:
Welcome to Python programming

String with the use of Triple Quotes:
Im D Samantha author of this book

Creating a multiline String:
Welcome
To
Python
```

```
# creating String with triple Quotes allows multiple lines
print("\n Creating a multiline String: ")
String3 = '''Welcome
                To
                Python'''
print (String3)
```

Accessing characters in Python String: In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on. While accessing an index out of the range will cause an Index Error. Only Integers are allowed to be passed as an index, float or other types that will cause a Type Error.



Python String indexing

 # Python Program to Access characters of String

```
String1 = "welcome to python"
print("\n First  and nine character of String is using positive indexing : ")
print(String1 [0],String1[8])
print("\n Last character of String using negative index: ")
print(String1[-1])
```

Output:

First and nine character of String is using positive indexing:

w t

Last character of String using negative index:

n

III.2 String Slicing:

In Python, the String Slicing method is used to access a range of characters in the String. Slicing in a String is done by using a Slicing operator, i.e., a colon (:). One thing to keep in mind while using this method is that the string returned after slicing includes the character at the start index but not the character at the last index.

Syntax: [start index :end Index]

Example:

```
# Creating a String
Name = "python programming"
print("Initial String: ")
print(Name)
# Printing 3rd to 12th character
print("\nSlicing characters from 3-12: ")
print(Name[3:12])
```

Output:

Initial String:
python programming
Slicing characters from 3-12:
hon progr

Reversing a Python String: we can also reverse strings in Python. We can reverse a string by using String slicing method. We will reverse a string by accessing the index. We did not specify the first two parts of the slice indicating that we are considering the whole string, from the start index to the last index.

Print (name [::-1]) **output:** Reverse of a string: gnimmargorp nohtyp

III.3 String Methods

- Python has a set of built-in methods that you can use on strings.
- Note: All string methods returns new values. They do not change the original string.

S.No	Method	Description
1	capitalize()	Converts the first character to upper case
2	casefold()	Converts string into lower case
3	center()	Returns a centered string
4	count()	Returns the number of times a specified value occurs in a string
5	encode()	Returns an encoded version of the string
6	endswith()	Returns true if the string ends with the specified value
7	expandtabs()	Sets the tab size of the string
8	find()	Searches the string for a specified value and returns the position of where it was found
9	index()	Searches the string for a specified value and returns the position of where it was found
10	isalnum()	Returns True if all characters in the string are alphanumeric
11	isalpha()	Returns True if all characters in the string are in the alphabet
12	isdecimal()	Returns True if all characters in the string are decimals
13	isdigit()	Returns True if all characters in the string are digits
14	isidentifier()	Returns True if the string is an identifier
15	islower()	Returns True if all characters in the string are lower case
16	isnumeric()	Returns True if all characters in the string are numeric
17	isprintable()	Returns True if all characters in the string are printable

18	isspace()	Returns True if all characters in the string are whitespaces
19	istitle()	Returns True if the string follows the rules of a title
20	isupper()	Returns True if all characters in the string are upper case
21	join()	Joins the elements of an iterable to the end of the string
22	ljust()	Returns a left justified version of the string
23	lower()	Converts a string into lower case
24	replace()	Returns a string where a specified value is replaced with a specified value
25	split()	Splits the string at the specified separator, and returns a list
26	startswith()	Returns true if the string starts with the specified value
27	strip()	Returns a trimmed version of the string
28	swapcase()	Swaps cases, lower case becomes upper case and vice versa
29	title()	Converts the first character of each word to upper case
30	upper()	Converts a string into upper case
31	rfind()	Searches the string for a specified value and returns the last position of where it was found
32	rindex()	Searches the string for a specified value and returns the last position of where it was found
33	rjust()	Returns a right justified version of the string
34	rsplit()	Splits the string at the specified separator, and returns a list
35	rstrip()	Returns a right trim version of the string
36	splitlines()	Splits the string at line breaks and returns a list
37	translate()	Returns a translated string
38	zfill()	Fills the string with a specified number of 0 values at the beginning

1. **capitalize ():** Converts the first character to upper case

Syntax: string.capitalize ()

Example: 1

>>> str="python is a trending programming language"

>>> str.capitalize ()

Output: 'Python is a trending programming language'

2. **casefold ():** Converts string into lower case. This method is similar to the lower() method, but the casefold()method is stronger, more aggressive, meaning that it will convert more characters into lower case, and will find more matches when comparing two strings and both are converted using the casefold()method.

Syntax: string.casefold ()

Example: 1

>>> str='PYTHON IS A TRENDING PROGRAMMING LANGUAGE'

>>> str.casefold ()

Output: 'python is a trending programming language'

3. **Center ():** The center () method will center align the string, using a specified character (space is default) as the fill character.

Syntax: string.center (length, character)

Length = Required. The length of the returned string.

Character = Optional. The character to fill the missing space on each side. Default is " " (space)

Example-1: Print the word "Engineering", taking up the space of 128 characters, with "Engineering" in the middle:

str1="Engineering"

>>> str1.center (128)

Output: ----------' Engineering '----------

4. **Count ():** The count () method returns the number of times a specified value appears in the string.

Syntax: string. Count (value, start, end)

Value =Required. A String. The string to value to search for

Start = Optional. An Integer. The position to start the search. Default is 0

end = Optional. An Integer. The position to end the search.

Default is the end of the string

Example-1: Return the number of times the value "is" appears in the string:

str2="my favorite subject is programming, python is a programming language"

str2.count ('is')

Output: 2

Example-2: Search from position 10 to 24:

>>>str2='my favorite subject is programming, python is a programming language'

>>>str2.count ('is', 20, 40) **Output:** 1

5. **encode ():** The encode() method encodes the string, using the specified encoding. If no encoding is specified, UTF-8 will be used.

Syntax: string.encode (encoding=encoding, errors=errors)

Encoding = Optional. A String specifying the encoding to use. Default is UTF-8

Errors = Optional. A String specifying the error method.

Legal values are:

- backslashreplace' - uses a backslash instead of the character that could not be encoded
- 'ignore'- ignores the characters that cannot be encoded
- 'name replace'- replaces the character with a text explaining the character
- 'strict' - Default, raises an error on failure
- 'Replace'- replaces the character with a question mark
- 'xmlcharrefreplace' - replaces the character with an xml character

6. **Endswith ():** The endswith () method returns True if the string ends with the specified value, otherwise False.

Syntax: string. Endswith (value, start, end)

Value Required. The value to check if the string ends with

Start Optional. An Integer specifying at which position to start the search

End Optional. An Integer specifying at which position to end the search

Example-1:

txt = "Hello, welcome to my world." x = txt.endswith ("my world.") print(x)

Output: True

7. **expandtabs ():** The expandtabs() method sets the tab size to the specified number of whitespaces.

Syntax: string.expandtabs (tabsize)

Here: tabsize Optional. A number specifying the tabsize. Default tabsize is 8

Example-1: txt = "H\te\tl\tl\to"

x = txt.expandtabs (2)

print(x)

Output: H e l l o

8. **find ():** The find () method finds the first occurrence of the specified value. The find() method returns -1 if the value is not found.

   The find() method is almost the same as the index() method, the only difference is that the index() method raises an exception if the value is not found.

Syntax: string. find(value, start, end)

Here: value required. The value to search for, start Optional. Where to start the search. Default is 0, end Optional. Where to end the search. Default is to the end of the string

Example-1:

txt = "Hello, welcome to my world."

x = txt.find("welcome")

Print(x)

Output: 7

9. Index (): The index () method finds the first occurrence of the specified value. The index () method raises an exception if the value is not found. The index() method is almost the same as the find() method, the only difference is that the find() method returns -1 if the value is not found

Syntax: string. Index (value, start, end)

Here: **value** required. The value to search for, **start** is Optional. Where to start the search. Default is 0, **end** Optional. Where to end the search. Default is to the end of the string

Example-1:

```
txt = "Hello, welcome to my world."
x = txt.index("welcome")
Print(x)
```

Output: 7

10. **Isalnum ():** The isalnum () method returns True if all the characters are alphanumeric, meaning alphabet letter (a-z) and numbers (0-9). Example of characters that are not alphanumeric: (space)!#%&? etc.

Syntax: string. Isalnum ()

Example-1:

```
txt = "Company12"
x = txt.isalnum()
print(x)
```

Output: True

11. **Isalpha ():** The isalpha () method returns True if all the characters are alphabet letters (a-z). Example of characters that are not alphabet letters: (space)!#%&? etc.

Syntax: string. Isalpha ()

Example-1:

```
txt = "CompanyX"
x = txt.isalpha()
print(x)
```

Output: True

12. **Isdecimal ():** The isdecimal () method returns True if all the characters are decimals (0-9). This method is used on Unicode objects.

Syntax: string. Isdecimal ()

Example-1:

```
txt = "\u0033" #unicode for 3
x = txt.isdecimal ()
print(x)
```

Output: True

13. **Isdigit ():** The isdigit () method returns True if all the characters are digits, otherwise False. Exponents, like ², are also considered to be a digit.

Syntax: string. Isdigit ()

Example-1:

```
txt = "50800"
x = txt.isdigit()
print(x)
```

Output: True

14. **Isidentifier ():** The isidentifier () method returns True if the string is a valid identifier, otherwise False. A string is considered a valid identifier if it only contains alphanumeric letters (a-z) and (0-9), or underscores (_). A valid identifier cannot start with a number, or contain any spaces.

Syntax: string.isidentifier ()

Example-1:

```
txt = "Demo"
x = txt.isidentifier ()
print(x)
```

Output: True

15. **Islower ():** The islower () method returns True if all the characters are in lower case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.

Syntax: string. Islower ()

Example-1:

txt = "hello world!"

x = txt.islower()

print(x)

Output: True

16. **Isnumeric ():** The isnumeric () method returns True if all the characters are numeric (0-9), otherwise False. Exponents, like ² and ¾ are also considered to be numeric values.

Syntax: string. Isnumeric ()

Example-1:

txt = "565543"

x = txt.isnumeric()

print(x)

Output: True

17. **Isprintable ():** The isprintable () method returns True if all the characters are printable, otherwise False. Example of none printable character can be carriage return and line feed.

Syntax: string. Isprintable ()

18. **Isspace ():** The isspace () method returns True if all the characters in a string are whitespaces, otherwise False.

Syntax: string. Isspace ()

Example-1:

txt = " "

x = txt.isspace()

print(x)

Output: True

19. **istitle():** The istitle() method returns True if all words in a text start with a upper case letter, AND the rest of the word are lower case letters, otherwise False. Symbols and numbers are ignored.

Syntax: string.istitle ()

Example-1:

```
txt = "Hello, And Welcome To My World!"
x = txt.istitle()
print(x)
```

Output: True

20. **Isupper ():** The isupper () method returns True if all the characters are in upper case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.

Syntax: string. Isupper ()

Example-1:

```
txt = "THIS IS NOW!"
x = txt.isupper()
print(x)
```

Output: True

21. **Join ():** The join () method takes all items in an iterable and joins them into one string.A string must be specified as the separator.

Syntax: string.join (iterable)

Example-1:

```
MyTuple = ("John", "Peter", "Vicky")
x = "#".join (MyTuple)
Print(x)
```

Output: John#Peter#Vicky

22. **Ljust ():** The ljust () method will left align the string, using a specified character (space is default) as the fill character.

Syntax: string. Ljust (length, character)

Here:: length Required. The length of the returned string, **character** Optional. A character to fill the missing space (to the right of the string). Default is " " (space).

Example-1:

```
txt = "banana"
x = txt.ljust(20)
print(x, "is my favorite fruit.")
```

Output: banana is my favorite fruit.

23. **lower ():** The lower () method returns a string where all characters are lower case. Symbols and Numbers are ignored.

Syntax: string. lower ()

Example-1:

```
txt = "Hello my FRIENDS"
x = txt.lower()
print(x)
```

Output: hello my friends

24. **replace():** The replace() method replaces a specified phrase with another specified phrase.

Note: All occurrences of the specified phrase will be replaced, if nothing else is specified.

Syntax: string.replace (oldvalue, newvalue, count)

Here: oldvalue: The string to search for (required)

Newvalue: The string to replace the old value with (required)

Count: A number specifying how many occurrences of the old value you want to replace. Default is all occurrences (optional)

Example-1:

```
txt = "I like bananas"
x = txt.replace ("bananas", "apples")
print(x)
```

Output: I like apples 17 www.Jntufastupdates.com

25. **split():** The split() method splits a string into a list. You can specify the separator, default separator is any whitespace. When maxsplit is specified, the list will contain the specified number of elements plus one.

Syntax: string.split (separator, maxsplit)

separator (Optional): Specifies the separator to use when splitting the string. By default any whitespace is a separator

maxsplit (Optional): Specifies how many splits to do. Default value is -1, which is "all occurrences"

Example-1:

```
txt = "welcome to the jungle"
x = txt.split()
print(x)
```

Output: ['welcome', 'to', 'the', 'jungle']

26. **startswith():** The startswith() method returns True if the string starts with the specified value, otherwise False.

Syntax: string.startswith (value, start, end)

Example-1:

```
txt = "Hello, welcome to my world."
x = txt.startswith("Hello")
print(x)
```

Output: True

27. **strip():** The strip() method removes any leading (spaces at the beginning) and trailing (spaces at the end) characters (space is the default leading character to remove)

Syntax: string.strip (characters)

Characters (Optional): A set of characters to remove as leading/trailing characters.

28. **swapcase ():** The swapcase () method returns a string where all the upper case letters are lower case and vice versa.

Syntax: string.swapcase ()

Example-1:

```
txt = "Hello My Name Is Purple"
x = txt.swapcase ()
Print(x)
```

Output: hELLO mY nAME iS pURPLE

29. **title ():** The title() method returns a string where the first character in every word is upper case. Like a header, or a title. If the word contains a number or a symbol, the first letter after that will be converted to uppercase.

Syntax: string.title ()

Example-1:

```
txt = "Welcome to my world"
x = txt.title()
print(x)
```

Output: Welcome To My World

30. **upper ():** The upper () method returns a string where all characters are in upper case. Symbols and Numbers are ignored.

Syntax: string.upper()

Example-1:

```
txt = "Hello my friends"
x = txt.upper()
print(x)
```

Output: HELLO MY FRIENDS

31. **rfind():** The rfind() method finds the last occurrence of the specified value. The rfind() method returns -1 if the value is not found. The rfind() method is almost the same as the rindex() method.

Syntax: string.rfind (value, start, end)

Example-1:

```
txt = "Mi casa, su casa."
x = txt.rfind("casa")
print(x)
```

Output: 12

32. **rindex():** The rindex() method finds the last occurrence of the specified value. The rindex() method raises an exception if the value is not found. The rindex() method is almost the same as the rfind() method.

Syntax: string.rindex (value, start, end)

33. **rjust():** The rjust() method will right align the string, using a specified character (space is default) as the fill character.

Syntax: string.rjust (length, character)

34. **rsplit():** The rsplit() method splits a string into a list, starting from the right. If no "max" is specified, this method will return the same as the split() method. When maxsplit is specified, the list will contain the specified number of elements plus one.

Syntax: string.rsplit (separator, maxsplit)

35. **rstrip():**The rstrip() method removes any trailing characters (characters at the end a string), space is the default trailing character to remove.

Syntax: string.rstrip(characters)

36. **splitlines():** The splitlines() method splits a string into a list. The splitting is done at line breaks.

Syntax: string.splitlines (keep line breaks)

Example-1:

```
txt = "Thank you for the music\n Welcome to the jungle"
```

```
x = txt.splitlines(True)
print(x)
```

Output: ['Thank you for the music\n', 'Welcome to the jungle']

37. **translate():** The translate() method returns a string where some specified characters are replaced with the character described in a dictionary, or in a mapping table. Use the maketrans() method to create a mapping table. If a character is not specified in the dictionary/table, the character will not be replaced. If you use a dictionary, you must use ASCII codes instead of characters.

Syntax: string.translate(table)

Example-1:

```
txt = "Hello Sam!";
mytable = txt.maketrans("S", "P");
print(txt.translate(mytable));
```

Output: Hello Pam!

38. **zfill():** The zfill() method adds zeros (0) at the beginning of the string, until it reaches the specified length. If the value of the len parameter is less than the length of the string, no filling is done.

Syntax: string.zfill (len)

Example-1:

```
txt = "50"
x = txt.zfill(10)
print(x)
```

Output: 0000000050

III.4 Lists

List operations: List is used to store the group of values & we can manipulate them, in list the values are stores in index format starts with 0. List is mutable object so we can do the manipulations. A python list is enclosed between square ([]) brackets.

- In list insertion order is preserved it means in which order we inserted element same order output is printed.
- A list can be composed by storing a sequence of different type of values separated by commas.
- The list contains forward indexing & backward indexing.

Syntax: <list_name> = [value1, value2, value3,..., value n]

Example:

```
data1= [1, 2, 3, 4]              # list of integers
data2= ['x','y','z']             # list of String
data3= [12.5, 11.6]              # list of floats
data4= []                        # empty list
data5= ['TEC', 10, 56.4,'a']     # list with mixed data types
```

Accessing List data:

- The list items are accessed by referring to the index number.
- Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.
- A range of indexes can be specified by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items.
- Note: The search will start at index 2 (included) and end at index 5 (not included).
- By leaving out the start value, the range will start at the first item.
- By leaving out the end value, the range will go on to the end of the list

Example:

```
x = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
Print(x [1])
Print(x [-1])
```

Output:

Banana

mango

Range of Negative Indexes: Specify negative indexes if you want to start the search from the end of the list.

Example:

```
x = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(x [-4:-1])
```

Output: ['orange', 'kiwi', 'melon']

Check if Item Exists: To determine if a specified item is present in a list use the in keyword.

Example: Check if "apple" is present in the list:

```
fruits= ["apple", "banana", "cherry"]
if "apple" in fruits:
print("Yes, 'apple' is in the fruits list")
```

Output: Yes, 'apple' is in the fruits list

Change Item Value: To change the value of a specific item, refer to the index number.

Example: Change the second item:

```
x = ["apple", "banana", "cherry"]
x[1] = "blackcurrant"
print(x)
```

Output: ['apple', 'blackcurrant', 'cherry']

To insert more than one item, create a list with the new values, and specify the index number where you want the new values to be inserted.

Example: Change the second value by replacing it with two new values:

```
x = ["apple", "banana", "cherry"]
x[1] = ["blackcurrant", "watermelon"]
print(x)
```

Output: ['apple', ['blackcurrant', 'watermelon'], 'cherry']

Change a Range of Item Values: To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

Example: Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon":

```
x = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
x [1:3] = ["blackcurrant", "watermelon"]
print(x)
```

Output: ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']

III.5 List methods

- Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

1. **List append ():** The append () method appends an element to the end of the list.

Syntax: list.append (element)

Here element (Required): An element of any type (string, number, object etc.)

Example: 1 Add an element to the fruits list.

fruits = ['apple', 'banana', 'cherry']

fruits.append ("orange")

Output: ['apple', 'banana', 'cherry', 'orange']

2. **List clear ():** The clear () method removes all the elements from a list.

Syntax: list.clear ()

Example: Remove all elements from the fruits list.

fruits = ['apple', 'banana', 'cherry', 'orange']

fruits. Clear()

Output: [ ]

3. **List copy ():** The copy () method returns a copy of the specified list.

Syntax: list. Copy()

Example: Copy the fruits list.

fruits = ['apple', 'banana', 'cherry', 'orange']

x = fruits.copy ()

Output: ['apple', 'banana', 'cherry']

4. **List count ():** The count () method returns the number of elements with the specified value.

Syntax: list.count (value)

Here, value is required and it any type (string, number, list, tuple, etc.). The value to search for.

Example-2: Return the number of times the value 9 appears in the list.

points = [1, 4, 2, 9, 7, 8, 9, 3, 1]

x = points. Count(9)

Output: 2

5. **List extend ():** The extend () method adds the specified list elements (or any iterable) to the end of the current list.

Syntax: list.extend (iterable)

Example – 1: Add the elements of cars to the fruits list.

fruits = ['apple', 'banana', 'cherry']

cars = ['Ford', 'BMW', 'Volvo']

fruits.extend (cars)

Output: ['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']

6. **List index ():** The index () method returns the position at the first occurrence of the specified value.

   Note: The index () method only returns the first occurrence of the value.

Syntax: list. Index (element)

Example – 1: What is the position of the value "cherry".

fruits = ['apple', 'banana', 'cherry']

x = fruits.index ("cherry")

Output: 2

7. **List insert ():** To insert a new list item, without replacing any of the existing values, we can use the insert () method. The insert () method inserts an item at the specified index:

Syntax: list. Insert (position, element)

Here,

Position = A number specifying in which position to insert the value

Element = an element of any type (string, number, object etc.)

Example: Insert "watermelon" as the third item.

x = ["apple", "banana", "cherry"]

x.insert (2, "watermelon")

print(x)

Output: ['apple', 'banana', 'watermelon', 'cherry']

8. **List pop ():** The pop () method removes the element at the specified position. The pop () method returns removed value. Position is a number specifying the position of the element you want to remove, default value is -1, which returns the last item

   Syntax: list.pop (position)

Example - 1: Remove the second element of the fruit list:

```
fruits = ['apple', 'banana', 'cherry']
fruits.pop (1)
```

Output: ['apple', 'cherry']

9. **List remove ():** The remove () method removes the first occurrence of the element with the specified value.

Syntax: list.remove (element)

Example: Remove the "banana" element of the fruit list.

```
fruits = ['apple', 'banana', 'cherry']
fruits.remove ("banana")
```

Output: ['apple', 'cherry']

10. **List reverse ():** The reverse () method reverses the sorting order of the elements.

Syntax: list.reverse ()

Example: Reverse the order of the list1.

```
List1= ["banana", "Orange", "Kiwi", "cherry"]
List1.reverse()
print(List1)
```

Output: ['cherry', 'Kiwi', 'Orange', 'banana']

Loop Through a List:

- We can loop through the list items by using a for loop.
- Print all items in the list, one by one.
- Use the range() and len() functions to print all items by referring to their index number:

Example – 1:

```
list = ["apple", "banana"]
for x in list:
    print(x)
```

Output:

apple

banana

Using a While Loop: Use the len() function to determine the length of the list, then start at 0and loop your way through the list items by referring to their indexes. Remember to increase the index by 1 after each iteration.

Example – 1:

```
list = ["apple", "banana", "cherry"]
i = 0
while i < len(list):
        print(list[i])
        i = i + 1
```

Output:

apple

banana

cherry

Looping Using List Comprehensive: It offers the shortest syntax for looping through lists.

Example – 1:

```
thislist = ["apple", "banana", "cherry"]
[print x for x in thislist]
```

Output:

apple

banana

cherry

Sort Lists: Sort List Alphanumerically. List objects have a sort () method that will sort the list alphanumerically, ascending, by default. To sort descending, use the keyword argument reverse = True.

Example -1:

List1= ["orange", "mango", "kiwi", "pineapple", "banana"]

List1.sort()

print (List1)

Output: ['banana', 'kiwi', 'mango', 'orange', 'pineapple']

III.6 Introduction to Tuples in python

Tuple is a collection of objects separated by commas. In some ways, a tuple is similar to a Python list in terms of indexing, nested objects, and repetition but the main difference between both is Python tuple is immutable, unlike the Python list which is mutable.

Tuples in Python are similar to Python lists but not entirely. Tuples are immutable and ordered and allow duplicate values. Some Characteristics of Tuples in Python.

- We can find items in a tuple since finding any item does not make changes in the tuple.
- One cannot add items to a tuple once it is created.
- Tuples cannot be appended or extended.
- We cannot remove items from a tuple once it is created.

Creating Python Tuples: There are various ways by which you can create a tuple in Python. They are as follows:

1. Using round brackets
2. With one item
3. Tuple Constructor

1. **Create Tuples using Round Brackets ():** To create a tuple we will use () operators and a sequence of data enclosed in a parenthesis

   T = ("one", "Two", "Three")

   print(T)

   Output: ('one', 'Two', 'Three')

2. Create a Tuple with One Item: Python 3.11 provides us with another way to create a Tuple.

   Values: tuple [int | float, ...] = (1,2,4,1.2,2.3)

   print(values)

   Output: (1, 2, 4, 1.2, 2.3)

3. Tuple Constructor in Python: To create a tuple with a Tuple constructor, we will pass the elements as its parameters.

   Tuple_constructor = tuple(("dsa", "26evelopment", "deep learning"))

   print(tuple_constructor)

   Output: ('dsa', 'developement', 'deep learning')

Accessing Values in Python Tuples: Tuples in Python provide two ways by which we can access the elements of a tuple.

1. Using a positive index
2. Using a negative index

Python Access Tuple using a Positive Index: Using square brackets we can get the values from tuples in Python. The positive index values starts from 0 to N-1

Example:

```
T = ("star", "for", "hero")
print("Value in T[0] = ", T[0])
print("Value in T[2] = ", T[2])
```

Output: Value in T[0] = star

Value in T[2] = hero

Access Tuple using Negative Index: The negative index is used to access all elements of a tuple in reverse order (from last to first)

Example:

```
T = ("star", "for", "hero")
print("Value in T[-1] = ", T[-1])
print("Value in T[-2] = ", T[-2])
```

Output: Value in T[-1] = hero

Value in T[-2] = for

III.7 Operations on Tuples

The different operations related to tuples in Python:

1. Concatenation
2. Nesting
3. Repetition
4. Slicing
5. Deleting
6. Finding the length

1. **Concatenation:** To Concatenation of Python Tuples, we will use plus operators (+).

   Example:

```
# concatenating 2 tuples
tuple1 = (0, 1, 2, 3)
tuple2 = ('python', 'programming')
# Concatenating above two
print(tuple1 + tuple2)
```

   Output: (0, 1, 2, 3, 'python', 'programming')

2. **Nesting:** A nested tuple in Python means a tuple inside another tuple.

   Example:

```
# concatenating 2 tuples
tuple1 = (0, 1, 2, 3)
tuple2 = ('python', 'programming')
# Nesting two tuples
t=(tuple1 , tuple2)
print(t)
```

Output: ((0, 1, 2, 3), ('python', 'programming'))

3. **Repetition:** We can create a tuple of multiple same elements from a single element in that tuple.

   Example:
   ```
   # create a tuples with a repetation
   t = ( 'programming',)*3
   print(t)
   ```

   Output: ('programming', 'programming', 'programming')

4. **Slicing:** Slicing a Python tuple means dividing a tuple into small tuples using the indexing method.

   Syntax: [start-index: end-index]

   Example:
   ```
   # slicing
   tuple1 = (0 ,1, 2, 3)
   print(tuple1[1:])
   print(tuple1[::-1])
   print(tuple1[2:4])
   ```

   Output:
   ```
   (1, 2, 3)
   (3, 2, 1, 0)
   (2, 3)
   ```

5. **Deletion:** In python, we are deleting a tuple using 'del' keyword. The output will be in the form of error because after deleting the tuple, it will give a NameError.

   Note: Remove individual tuple elements is not possible, but we can delete the whole Tuple using Del keyword.

   Syntax: del <name of tuple>

   Example:
   ```
   # Deletion of a tuple
   tuple1 = (0 ,1, 2, 3)
   del tuple1
   ```

6. **Length of Tuple:** To find the length of a tuple, we can use Python's len() function and pass the tuple as the parameter.

Example:
```
# Finding a length of a tuple
tuple1 = (0 ,1, 2, 3)
print(len(tuple1))
```

Output: 4

III.8 Methods in tuples

Python has two built-in methods that you can use on tuples.

1. Count()
2. Index()

1. Count() Method: The count () method of Tuple returns the number of times the given element appears in the tuple.

Syntax: tuple.count (element)

Example:
```
Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
res = Tuple1.count(3)
print('Count of 3 in Tuple1 is:', res)
```

Output: Count of 3 in Tuple1 is: 3

2. **Index() Method:** The Index () method returns the first occurrence of the given element from the tuple.

Syntax: tuple.index (element, start, end)

Parameters:

Element: The element to be searched.

Start (Optional): The starting index from where the searching is started

End (Optional): The ending index till where the searching is done

Example:
```
Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
res = Tuple1.index(3)
print('First occurrence of 3 is', res)
```

Output: First occurrence of 3 is 3

III.9 Dictionaries

Dictionaries

- In Python, a dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'.
- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is unordered, it means that the items does not have a defined order, you cannot refer to an item by using an index.
- It is changeable, means that we can change, add or remove items after the dictionary has been created.
- It does not allow duplicates means cannot have two items with the same key.
- Dictionaries are written with curly brackets, and have keys and values
- It can be referred by using the key name.
- The values in dictionary items can be of any data type.

Syntax: dictionary name = {key1 : value1, key2 : value2, ...}

Example-1: Create and print a dictionary.

```
dict1 = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
print(dict1)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

Different ways to create Python Dictionary: In python dictionaries are created with {} and with a dict() constructor with key-value pairs specified within curly braces and as a list of tuples.

The dict is a predefined constructor in python it accept key-value pairs as list or tuples

Example:

```
Dict = {}
print("Empty Dictionary: ")
print(Dict)
Dict = dict({1: 'hai', 2: 'hello', 3: 'how'})
print("\nDictionary with the use of dict(): ")
print(Dict)

Dict = dict([(1, 'star'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

III.10 Dictionary methods

Python Dictionary Methods: Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key. with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

1. **Dictionary clear ():** The clear () method removes all the elements from a dictionary.

Syntax: dictionary. Clear()

Example: Remove all elements from the car list.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
car.clear ()
print(car)
```

Output: {}

2. **Dictionary copy ():** You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be reference to dict1, and changes made in dict1 will automatically also be made in dict2. The copy () method returns a copy of the specified dictionary.

Syntax: dictionary.copy ()

Example: Copy the car dictionary.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.copy ()
print(x)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

3. **Dictionary fromkeys ():** The fromkeys () method returns a dictionary with the specified keys and the specified value.

Syntax: dict.fromkeys (keys, value)

keys :An iterable specifying the keys of the new dictionary

Value (Optional): The value for all keys. Default value is None

Example: 1 Create a dictionary with 3 keys, all with the value 0.

```
x = ('key1', 'key2', 'key3')
y = 0
d = dict.fromkeys(x, y)
print(d)
```

Output: ['key1': 0, 'key2': 0, 'key3': 0]

4. **Dictionary get ():** The get () method returns the value of the item with the specified key.

Syntax: dictionary.get (keyname)

Example-1: Get the value of the "model" item.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.get("model")
print(x)
```

Output: Mustang

5. **Dictionary items ():** The items () method returns a view object. The view object contains the key-value pairs of the dictionary, as tuples in a list. The view object will reflect any changes done to the dictionary.

Syntax: dictionary.items ()

Example-1: Return the dictionary's key-value pairs:

```
car = {
```

"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.items()
print(x)
Output: dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

6. **Dictionary keys ():** The keys () method returns a view object. The view object contains the keys of the dictionary, as a list. The view object will reflect any changes done to the dictionary.

Syntax: dictionary.keys ()

Example – 1: Return the keys.

car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.keys()
print(x)
Output: dict_keys(['brand', 'model', 'year'])

7. **Dictionary pop ():** The pop () method removes the specified item from the dictionary. The value of the removed item is the return value of the pop () method.

Syntax: dictionary.pop (keyname, default value)

Example: 1 Remove "model" from the dictionary.

car = {
"brand": "Ford",
"model": "Mustang",

```
"year": 1964
}
car.pop("model")
print(car) Output: {'brand': 'Ford', 'year': 1964}
```

8. **Dictionary popitem ():** The popitem () method removes the item that was last inserted into the dictionary. In versions before 3.7, the popitem () method removes a random item. The removed item is the return value of the popitem () method, as a tuple.

Syntax: dictionary.popitem ()

Example: 1 Remove the last item from the dictionary.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
car.popitem ()
print(car)
```

Output: {'brand': 'Ford', 'model': 'Mustang'}

9. **Dictionary setdefault ():** The setdefault () method returns the value of the item with the specified key. If the key does not exist, insert the key, with the specified value.

Syntax: dictionary.setdefault (keyname, value)

Example: 1 Get the value of the "model" item.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
```

```
x = car.setdefault ("model", "Bronco")
print(x)
```

Output: Mustang

10. **Dictionary update():** The update() method inserts the specified items to the dictionary. The specified items can be a dictionary, or an iterable object with key value pairs.

Syntax: dictionary.update (iterable)

Example: Insert an item to the dictionary.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
car.update ({"color": "White"})
print(car)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'}

11. **Dictionary values ():** The values () method returns a view object. The view object contains the values of the dictionary, as a list. The view object will reflect any changes done to the dictionary, see example below.

Syntax: dictionary.values ()

Example: 1 Return the values.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.values ()
```

```
print(x)
```

Output: dict_values(['Ford', 'Mustang', 1964])

12. del: The del keyword removes the item with the specified key name.

Example: 1

```
thisdict = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
del thisdict["model"]
print(thisdict)
```

Output: {'brand': 'Ford', 'year': 1964}

Check if Key Exists: To determine if a specified key is present in a dictionary use the in keyword.

Example: 1 Check if "model" is present in the dictionary.

```
thisdict = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
if "model" in thisdict:
print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Output: Yes, 'model' is one of the keys in the thisdict dictionary

Dictionary Length: To determine how many items a dictionary has, use the len () function:

Example: Print the number of items in the dictionary.

```
print(len(thisdict))
```

Output: 3

Loop through a Dictionary: You can loop through a dictionary by using a for loop. When looping through a dictionary, the return values are the keys of the dictionary, but there are methods to return the values as well.

Example: 1 Print all key names in the dictionary, one by one.

```
thisdict = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
for x in thisdict:
print(x)
```

Output:

brand
model
year

Nested Dictionaries: A dictionary can contain dictionaries, this is called nested dictionaries.

Example: 1 Create a dictionary that contain three dictionaries.

```
myfamily = {
"child1" : {
"name" : "Emil",
"year" : 2004
},
"child2" : {
"name" : "Tobias",
"year" : 2007
},
```

"child3" : {
"name" : "Linus",
"year" : 2011
}
}
Output: {'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}

III.11 Mutable Vs Immutable

- In Python, everything is an object. An object has its own internal state. Some objects allow you to change their internal state and others don't.
- In Python, Every variable in Python holds an instance of an object. There are two types of objects in Python i.e. Mutable and Immutable objects. Whenever an object is instantiated, it is assigned a unique object id.
- The type of the object is defined at the runtime and it can't be changed afterward. However, its state can be changed if it is a mutable object.
- An object whose internal state can be changed is called a mutable object, while an object whose internal state cannot be changed is called an immutable object.

Immutable Objects in Python: Immutable Objects are of in-built datatypes like int, float, bool, string, Unicode, and tuple. In simple words, an immutable object can't be changed after it is created.

- **The following are examples of immutable objects:**
  - Numbers (int, float, bool,…)
  - Strings
  - Tuples
  - Frozen sets

Mutable Objects in Python: Mutable Objects are of type Python list, Python dict, or Python set. Custom classes are generally mutable.

- **And the following are examples of mutable objects:**

  Lists

  Sets

  Dictionaries

User-defined classes can be mutable or immutable, depending on whether their internal state can be changed or not.

Python's Mutable vs Immutable

- Mutable and immutable objects are handled differently in Python. Immutable objects are quicker to access and are expensive to change because it involves the creation of a copy. Whereas mutable objects are easy to change.
- The use of mutable objects is recommended when there is a need to change the size or content of the object.
- Exception: However, there is an exception in immutability as well. We know that a tuple in Python is immutable. But the tuple consists of a sequence of names with unchangeable bindings to objects

III.12 Arrays Vs Lists

Arrays: An array is a collection of items stored at contiguous memory locations. The array The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

When to Use Python Arrays :

- Lists are built into the Python programming language, whereas arrays aren't. Arrays are not a built-in data structure, and therefore need to be imported via the array module in order to be used.
- Arrays of the array module are a thin wrapper over C arrays, and are useful when you want to work with homogeneous data.
- They are also more compact and take up less memory and space which makes them more size efficient compared to lists.
- If you want to perform mathematical calculations, then you should use NumPy arrays by importing the NumPy package

How to Use Arrays in Python: In order to create Python arrays, you'll first have to import the array module which contains all the necessary functions.

There are three ways you can import the array module:

1. By using import array at the top of the file. This includes the module array. You would then go on to create an array using array.array().

```
import array
#how you would create an array
array.array()
```

2. Instead of having to type array.array() all the time, you could use import array as arr at the top of the file, instead of import array alone. You would then create an array by typing arr.array(). The arr acts as an alias name, with the array constructor then immediately following it.

```
import array as arr
#how you would create an array
arr.array()
```

3. Lastly, you could also use from array import *, with * importing all the functionalities available. You would then create an array by writing the array() constructor alone.

```
from array import *
#how you would create an array
array()
```

How to Define Arrays in Python: Once you've imported the array module, you can then go on to define a Python array. The general syntax for creating an array looks like this:

variable_name = array(typecode,[elements])

- variable_name would be the name of the array.
- The typecode specifies what kind of elements would be stored in the array. Whether it would be an array of integers, an array of floats or an array of any other Python data type. Remember that all elements should be of the same data type.
- Inside square brackets you mention the elements that would be stored in the array, with each element being separated by a comma. You can also create an empty array by just writing variable_name = array(typecode) alone, without any elements.

Below is a typecode table, with the different typecodes that can be used with the different data types when defining Python arrays:

TYPECODE	C TYPE	PYTHON TYPE	SIZE
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	wchar_t	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Example:

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers)
```

output: array('i', [10, 20, 30])

- First we included the array module, in this case with import array as arr .
- Then, we created a numbers array.
- We used arr.array() because of import array as arr .
- Inside the array() constructor, we first included i, for signed integer. Signed integer means that the array can include positive and negative values. Unsigned integer, with H for example, would mean that no negative values are allowed.
- Lastly, we included the values to be stored in the array in square brackets.

How to Find the Length of an Array in Python: To find out the exact number of elements contained in an array, use the built-in len() method.

Example:

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(len(numbers))
```

output: 3

Array Indexing and How to Access Individual Items in an Array in Python

- Each item in an array has a specific address. Individual items are accessed by referencing their index number.Indexing in Python, and in all programming languages and computing in general, starts at 0. It is important to remember that counting starts at 0 and not at 1.
- To access an element, you first write the name of the array followed by square brackets. Inside the square brackets you include the item's index number.The general syntax would look something like this:

Example:

```
array_name[index_value_of_item]
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers[0]) # gets the 1st element
print(numbers[1]) # gets the 2nd element
print(numbers[2]) # gets the 3rd element
```

output:

```
10
20
30
```

How to Search Through an Array in Python: You can find out an element's index number by using the index() method. You pass the value of the element being searched as the argument to the method, and the element's index number is returned.

Example:

```
import array as arr
numbers = arr.array('i',[10,20,30])
#search for the index of the value 10
print(numbers.index(10))
```

Output: 0

- If there is more than one element with the same value, the index of the first instance of the value will be returned

How to Loop through an Array in Python : You've seen how to access each individual element in an array and print it out on its own. If you want to print each value one by one, This is where a loop comes in handy. You can loop through the array and print out each value, one-by-one, with each loop iteration.

Example:

```
import array as arr
numbers = arr.array('i',[10,20,30])
for number in numbers:
  print(number)
```

Output: 10
20
30

You could also use the range() function, and pass the len() method as its parameter. This would give the same result as above:

Example:

```
import array as arr
values = arr.array('i',[10,20,30])
#prints each individual value in the array
for value in range(len(values)):
  print(values[value])
```

Output: 10
20
30

How to Slice an Array in Python: To access a specific range of values inside the array, use the slicing operator, which is a colon :

- When using the slicing operator and you only include one value, the counting starts from 0 by default. It gets the first item, and goes up to but not including the index number you specify.

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
```

```
#get the values 10 and 20 only
print(numbers[:2])  #first to second position
```

Output : array('i', [10, 20])

Methods For Performing Operations on Arrays in Python

- Arrays are mutable, which means they are changeable. You can change the value of the different items, add new ones, or remove any you don't want in your program anymore.
- Let's see some of the most commonly used methods which are used for performing operations on arrays.

How to Change the Value of an Item in an Array: You can change the value of a specific element by speficying its position and assigning it a new value:

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#change the first element
#change it from having a value of 10 to having a value of 40
numbers[0] = 40
print(numbers)
```

Output: array('i', [40, 20, 30])

How to Add a New Value to an Array: To add one single value at the end of an array, use the append() method:

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#add the integer 40 to the end of numbers
numbers.append(40)
```

```
print(numbers)
```

Output: array('i', [10, 20, 30, 40])

Be aware that the new item you add needs to be the same data type as the rest of the items in the array.

Extend method: If you want to add more than one value to the end an array. Use the extend() method, which takes an iterable (such as a list of items) as an argument. Again, make sure that the new items are all the same data type.

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#add the integers 40,50,60 to the end of numbers
#The numbers need to be enclosed in square brackets
numbers.extend([40,50,60])
print(numbers)
```

Output: array('i', [10, 20, 30, 40, 50, 60])

Insert method: The insert() method, is to add an item at a specific position. The insert() function takes two arguments: the index number of the position the new element will be inserted, and the value of the new element.

Example:

```
#add the integer 40 in the first position remember indexing starts at 0
import array as arr
numbers = arr.array('i',[10,20,30]) #original array
numbers.insert(0,40)
print(numbers)
```

Output: array('i', [40, 10, 20, 30])

How to Remove a Value from an Array : To remove an element from an array, use the remove() method and include the value as an argument to the method.

Example:

```
import array as arr
```

```
#original array
numbers = arr.array('i',[10,20,30])
numbers.remove(10)
print(numbers)
```

Output: array('i', [20, 30])

With remove(), only the first instance of the value you pass as an argument will be removed.

Pop method: You can also use the pop() method, and specify the position of the element to be removed:

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30,10,20])
#remove the first instance of 10
numbers.pop(0)
print(numbers)
```

Output: array('i', [20, 30, 10, 20])

Difference Between List and Array in Python

List	Array
Can consist of elements belonging to different data types	Only consists of elements belonging to the same data type
No need to explicitly import a module for the declaration	**Need to explicitly import the array module for declaration**
Cannot directly handle arithmetic operations	Can directly handle arithmetic operations
Preferred for a shorter sequence of data items	Preferred for a longer sequence of data items
Greater flexibility allows easy modification (addition, deletion) of data	Less flexibility since addition, and deletion has to be done element-wise
The entire list can be printed without any explicit looping	A loop has to be formed to print or access the components of the array

Consume larger memory for easy addition of elements	Comparatively more compact in memory size
Nested lists can be of variable size	Nested arrays has to be of same size.
Can perform direct operations using functions like: count() , sort()	Need to import proper modules to perform these operations.
Example: my_list = [1, 2, 3, 4]	**Example:** import array arr = array.array('i', [1, 2, 3])

III.13 Map method in python

The map() function in Python: The map() function (which is a built-in function in Python) is used to apply a function to each item in an iterable (like a Python list or dictionary). It returns a new iterable (a map object) that you can use in other parts of your code.

The general syntax: map(function, iterable, [iterable1, iterable2, ...])

Let's see an example: imagine you have a list of numbers, and you want to create a new list with the cubes of the numbers in the first list. A traditional approach would involve using the for loop

The map() function simplifies and reduce the code in trditional aproach :

```
org_list = [1, 2, 3, 4, 5]
# define a function that returns the cube of `num`
def cube(num):
  return num**3
fin_list = list(map(cube, org_list))
print(fin_list) # [1, 8, 27, 64, 125]
```

- Instead of writing a separate function to calculate the cube of a number, we can use a lambda expression in its place. Here's how you'd do that:

```
fin_list = list(map(lambda x:x**3, org_list))
```

```
print(fin_list) # [1, 8, 27, 64, 125]
```

For example if you had a list of strings, you can easily create a new list with the length of each string in the list.

```
org_list = ["Hello", "world", "freecodecamp"]
fin_list = list(map(len, org_list))
print(fin_list) # [5, 5, 12]
```

Note: The map function takes n number of values as input and it returns n values as a output

III.14 Filter method in python

- Python's built-in filter() function is a powerful too; for performing data filtering procedure on sequences like lists, tuples, and strings. The filter() function is utilized to apply a function to each element of an iterable (like a list or tuple) and return another iterable containing just the elements for which the function brings True back. Along these lines, filter() permits us to filter out elements from a grouping based on some condition. The first argument can be None if the function is not available and returns only elements that are True.

Syntax: filter (function, iterable)

Parameters

function: It is a function. If set to None returns only elements that are True.

Iterable: Any iterable sequence like list, tuple, and string.

Example 1: This simple example returns values higher than 5 using filter function

```
# Python filter() function example
def filterdata(x):
  if x>5:
    return x
# Calling function
result = filter(filterdata,(1,2,6))
# Displaying result
```

print(list(result))

using lambda function in filter : Instead of writing a separate function to fin the a number greater than 5, we can use a lambda expression in its place. Here's how you'd do that:

result = filter(lambda x:x>5,(1,2,6))

Displaying result

print(list(result))

Note: The filter function takes n number of values as input and it returns less than or equal to n values as a output

III.15 Reduce method in python

In Python, reduce() is a built-in function that applies a given function to the elements of an iterable, reducing them to a single value.

The syntax for reduce() is as follows:

functools.reduce(function, iterable [, initializer])

- The function argument is a function that takes two arguments and returns a single value. The first argument is the accumulated value, and the second argument is the current value from the iterable.
- The iterable argument is the sequence of values to be reduced.
- The optional initializer argument is used to provide an initial value for the accumulated result. If no initializer is specified, the first element of the iterable is used as the initial value.

Example that demonstrates how to use reduce () to find the sum of a list of numbers:

```
# Examples to understand the reduce() function
from functools import reduce
 # Function that returns the sum of two numbers
def add(a, b):
   return a + b
```

```
# Our Iterable
num_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# add function is passed as the first argument, and num_list is passed as the second argument
sum = reduce(add, num_list)
print(f"Sum of the integers of num_list : {sum}")
# Passing 10 as an initial value
sum = reduce(add, num_list, 10)
print(f"Sum of the integers of num_list with initial value 10 : {sum}")
```

Output:

Sum of the integers of num_list : 55

Sum of the integers of num_list with initial value 10 : 65

Let us use the lambda function as the first argument of the reduce() function

```
# Importing reduce function from functools
from functools import reduce
 # Creating a list
my_list = [1, 2, 3, 4, 5]
# Calculating the product of the numbers in my_list
# using reduce and lambda functions together
product = reduce(lambda x, y: x * y, my_list)
# Printing output
print(f"Product = {product}")
```

Output : Product = 120

Differences between map,reduce and filter in python

Key characteristics	Map	Filter	Reduce
Syntax	map(function, iterable object)	filter(function, iterable object)	reduce(function, iterable object)
Effect of the function on the iterable	Applies the function evenly to all the items in the iterable object	Function is a boolean condition that rejects all the items in the iterable object that are not true	Breaks down the entire process of applying the function into pair-wise operations
Input to output variables	N to N	N to M where N>=M	N to 1
Use Case	Transformation/Mapping	Splitting the data	Single output operations
Example	List of the square of all numbers in a list	All even numbers from a list	Product of all the items in the list

III.15 Comprehensions in python

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, sets, dictionaries, etc.) using previously defined sequences. Python supports the following 4 types of comprehension:

1. List Comprehensions
2. Dictionary Comprehensions
3. Set Comprehensions
4. Generator Comprehensions

1. **List Comprehensions:** List Comprehensions provide an elegant way to create new lists. The following is the basic structure of list comprehension:

Syntax: output_list = [output_exp for var in input_list if (var satisfies this condition)]

Note that list comprehension may or may not contain an if condition. List comprehensions can contain multiple

Example 1: Generating an Even list WITHOUT using List comprehensions

Suppose we want to create an output list that contains only the even numbers which are present in the input list

Example:

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
output_list = []
for var in input_list:
    if var % 2 == 0:
        output_list.append(var)
print("Output List using for loop:", output_list)
```

Output: Output List using for loop: [2, 4, 4, 6]

Example 2: Generating Even list using List comprehensions

Here we use the list comprehensions in Python. It creates a new list named list_using_comp by iterating through each element var in the input_list. Elements are included in the new list only if they satisfy the condition, which checks if the element is even. As a result, the output list will contain all even numbers.

Example:

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
list_using_comp = [var for var in input_list if var % 2 == 0]
print("Output List using list comprehensions:",list_using_comp)
```

output: Output List using list comprehensions: [2, 4, 4, 6]

2. **Dictionary Comprehensions:** Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions. The basic structure of a dictionary comprehension looks like below.

```
output_dict = {key:value for (key, value) in iterable if (key, value satisfy this condition)}
```

Example 1: Generating odd number with their cube values without using dictionary comprehension

```
input_list = [1, 2, 3, 4, 5, 6, 7]
output_dict = {}
for var in input_list:
 if var % 2 != 0:
        output_dict[var] = var**3
print("Output Dictionary using for loop:",output_dict )
```

Example 2: Generating odd number with their cube values with using dictionary comprehension

```
input_list = [1,2,3,4,5,6,7]
dict_using_comp = {var:var ** 3 for var in input_list if var % 2 != 0}
print("Output Dictionary using dictionary comprehensions:",dict_using_comp)
```

Output:

Output Dictionary using dictionary comprehensions: {1: 1, 3: 27, 5: 125, 7: 343}

3. **Set Comprehensions:** Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }

Example 1 : Checking Even number Without using set comprehension

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
output_set = set()
for var in input_list:
 if var % 2 == 0:
        output_set.add(var)
print("Output Set using for loop:", output_set)
```

Output:

Output Set using for loop: {2, 4, 6}

Example 2: Checking Even number using set comprehension

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
set_using_comp = {var for var in input_list if var % 2 == 0}
print("Output Set using set comprehensions:",set_using_comp)
```

Output:

Output Set using set comprehensions: {2, 4, 6}

4. **Generator Comprehensions:** Generator Comprehensions are very similar to list comprehensions. One difference between them is that generator comprehensions use circular brackets whereas list comprehensions use square brackets.

   The major difference between them is that generators don't allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient

   Example:

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
output_gen = (var for var in input_list if var % 2 == 0)
print("Output values using generator comprehensions:", end = ' ')
for var in output_gen:
 print(var, end = ' ')
```

   Output:

   Output values using generator comprehensions: 2 4 4 6

All the best