INTRODUCTION TO LISTS IN PYTHON

What is Sequence in Python

- A “Sequence” is a Group of Items with a Deterministic Ordering, i.e., the Order in which the Items are put into the “Sequence” is the same Order in which the Items are retrieved from the “Sequence”.
- Python has three types of Sequences -
  - String
  - List
  - Tuple

What is List in Python

- Python does not have Array, instead, it has “List”. Python “List” is a Container, like an “Array”, that holds an Ordered Sequence of Items. The Items can be anything - String, Number, or, Data of any available Type.
- A “List” can be “Homogeneous”, or, “Heterogeneous”. It means either only one type of Data, or, multiple types of Data in combination, can be stored in a “List”.
- Every Item rest at some position in the “List”, referred to as by “Index”. The “Index” can be used to locate a particular Item. The First Index begins at “0”, next is “1”, and, so forth.

Create List in Python

- There are multiple ways to form a “List” in Python.
- ***Subscript Operator*** - The “Square Brackets”, i.e., “[]” represent the Subscript Operator in Python. Since, it does not require a Symbol Lookup, or, a Function Call, this is the fastest way to create a “List” in Python.
  A “List” is created by placing all the Items inside the Subscript Operator, i.e., “[]”, separated by Commas, i.e., “,”. The “List” can take any number of Items, and, each Item may belong to a different Data Type, i.e., Integer, Float, String etc.

```
# Create an Empty List
empty_list = []

# Create a Homogeneous List of Strings
str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]

# Create a Homogeneous List of Integers
int_list = [23, 12, 44, 9, 19, 5, 39]
```

```
# Create a Heterogeneous List
het_list = [1, 2, 5.6, 9.6, 'a', 'rty', "wer"]
```

A “List” can also have another “List” as an Item. This is called a “Nested List”.

```
# Create a Homogeneous Nested List of Strings
nest_str_list = ["Name", "Address", "City", ['Kolkata', 'Mumbai', ['Beadon St.', "VIP Road"], 'Pune'], "State", "Country"]

# Create a Homogeneous Nested List of Floats
nest_int_list = [5, 9, 49, [3, 6, 8, [11, 22, 67], 12, 15], 76, 34]

# Create a Heterogeneous Nested List
nest_het_list = [1.3, 4, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'ok']
```

- ***List Constructor*** - Python includes a built-in “list ()” Constructor. The “list ()” Constructor accepts either a “Sequence”, or, a “Tuple” as the Argument, and converts into a Python “List”.

```
# Create an Empty List
empty_list = list()

# Create a Homogeneous List of Strings
str_list = list(['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"])

# Create a Homogeneous List of Integers
int_list = list([23, 12, 44, 9, 19, 5, 39])

# Create a Heterogeneous List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
```

A Nested “Sequence” can also be passed as the input Argument to the “list ()” Constructor to create a “Nested List”.

```
# Create a Homogeneous Nested List of Strings
nest_str_list = list(["Name", "Address", "City", ['Kolkata', 'Mumbai', ['Beadon St.', "VIP Road"], 'Pune'], "State", "Country"])

# Create a Homogeneous Nested List of Floats
nest_int_list = list([5, 9, 49, [3, 6, 8, [11, 22, 67], 12, 15], 76, 34])

# Create a Heterogeneous Nested List
nest_het_list = list([1.3, 4, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'ok'])
```

- ***List Comprehension*** - List Comprehension is an elegant and concise way to create a new “List” from an existing “List” in Python. A List Comprehension has the following Syntax - newList = [expression (iter) for iter in oldList if filter (iter)]. It has Square Brackets “[]” grouping, an Expression followed by a “for–in” clause and zero, or, more “if” Statement. The result will always be a “List”.

```
# Create a List of Integers
int_list_comp = [num for num in range(7)]
print(int_list_comp)

# Create a List of the First Letters of Each Element from the Old List of Strings
listOfCountries = ['India', 'Japan', 'South Korea', "Thailand", "Hong Kong", "Vietnam"]
firstLetters_list = [country[0] for country in listOfCountries]
print(firstLetters_list)
```

Output -

```
[0, 1, 2, 3, 4, 5, 6]
['I', 'J', 'S', 'T', 'H', 'V']
```

List Comprehension allows an “if” Statement to only add Members to the New “List” from the Old “List”, which are fulfilling a specific condition.

```
# Create a List of Only Numeric Values from the Old Heterogeneous Nested List
het_list = [1, 2, 5.6, 9.6, 'a', 'rty', "wer"]
numeric_list = [num for num in het_list if str(num).isnumeric()]
print(numeric_list)

# Create a List of Only the Odd Months from the List of Months
months_list = ['Jan', "Feb", 'Mar', "Apr", 'May', "Jun", 'Jul', "Aug", 'Sep', "Oct", 'Nov', "Dec"]
enum_month_list = enumerate(months_list)
odd_months_list = [month for index, month in enum_month_list if (index % 2 == 0)]
print(odd_months_list)
```

Output -

```
[1, 2]
['Jan', 'Mar', 'May', 'Jul', 'Sep', 'Nov']
```

Access List Items in Python

- There are various ways in which the Items of a “List” can be accessed in Python.
- ***List Indexing*** - The “Index Operator”, i.e., “[]” can be used to access an Item from the “List”. In Python, the Index of a “List” starts at “0”. So, a “List”, having ten Items will have Indices from “0” to “9”.

```
# Access the Item Present in the Index "3"
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
print(het_list[3])
```

Output -

```
9.6
```

Any attempt to access an Item beyond the Indices Range of a “List” would raise an “IndexError” Exception.

```
# Can't Access Index Beyond the List Index Range
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
print(het_list[11])
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 75, in <module>
    print(het_list[11])
IndexError: list index out of range
```

The Value of Index must be an Integer. Using any other type of Value, like Float, or, String, will raise a “TypeError” Exception.

```
# Index Can't Be Any Values Other Than Integer
print(het_list[3.0])
print(het_list["3"])
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 74, in <module>
    print(het_list[3.0])
TypeError: list indices must be integers or slices, not float
```

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 75, in <module>
    print(het_list["3"])
TypeError: list indices must be integers or slices, not str
```

The Items in a “Nested List” can be accessed using Nested Indexing.

```
# Access the Nested Item "False" Present in the List
nest_het_list = list([1.3, 4, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'ok'])
print(nest_het_list[3][2][1])
```

Output -

```
False
```

- ***Negative Indexing*** - Python enables Reverse, or, Negative Indexing for the "Sequence" Data Type. So, for the Python "List" to Index in the opposite order, set the Index using the Minus "-" sign. Indexing the "List" with "-1" will return the Last Item of the "List", "-2" will return the Second Last Item, and, so on.

```
# Access the Third Last Item from the List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
print(het_list[-3])

# Access the Nested Item "True" Present in the List Using Negative Indexing
nest_het_list = list([1.3, 4, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'ok'])
print(nest_het_list[-3][-3][-2])
```

Output -

```
a
True
```

- ***List Slicing*** - Slicing in Python is a feature than enables accessing parts of Sequence. In "List Slicing", a "Sub-List" is created, which is essentially created from another "List". Slicing is used when only a certain Items of a "List" are required, and, not the complete "List". The Syntax of "List Slicing" is - list [start: end: step].
  - ***start*** - The Starting Index.
    If no Value is provided in "start", the "Sub-List" will be formed from the first Index of the "List".
  - ***end*** - The Ending Index, which is not inclusive in the "Sub-List".
    If no Value is provided in "end", the "Sub-List" will be formed till the last Index of the "List".
  - ***step*** - It is an optional argument that determines the increment between each Index for Slicing.
    Providing the Value "1" in "step" will display each Item of the" List".

```python
# Create a Sub-List from the First to Third Last Element
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
print(het_list[:-3])

# Create a Sub-List from the Third to the Last Element
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
print(het_list[2:])

# Create a Sub-List from of Every Alternate Item from the List
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
print(nest_het_list[::2])

# Create a Sub-List from the First to the Third Last Element in Reverse
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
print(het_list[-3:-7:-1])
```

```python
# Create a Sub-List from of Every Alternate Item from the List in Reverse
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], "New", 'U', 'Tia', 'ok'])
print(nest_het_list[::-2])
```

Output -

```
[1, 2, 5.6, 9.6]
[5.6, 9.6, 'a', 'rty', 'wer']
[1.3, 89.0, [2, 7.6, [True, False], 'yes', 'N'], 'Tia']
['a', 9.6, 5.6, 2]
['ok', 'U', [2, 7.6, [True, False], 'yes', 'N'], 89.0, 1.3]
```

Change the Values of the List Items in Python

- "Lists" are Mutable in Python, meaning the Values at each Index of a "List" can be changed, unlike "String", or "Tuple".
- ***Assignment Operator*** - The "Assignment Operator", i.e., "=" can be used to change an Item, or a Range of Items in a "List".

```
# Change Value of the Third Item to "India"
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list[2] = 'India'
print(het_list)

# Change Value of the Second Last Item to 100
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list[-2] = 100
print(het_list)

# Change Value of the First Inner Most Item to 'True'
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
nest_het_list[4][2][0] = "True"
print(nest_het_list)
```

```
# Change Value of the Last Inner Most Item to 'False' Using Negative Indexing
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
nest_het_list[-4][-3][-1] = "False"
print(nest_het_list)
```

```
# Change Values from the Second to the Fifth Items to [6, 'Japan', 'z', 21.6]
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list[1:4] = [6, 'Japan', 'z', 21.6]
print(het_list)

# Change Values from the Second Last to the Fifth Last Items to [6, 'Japan', 'z', 21.6]
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list[-2:-5] = [6, 'Japan', 'z', 21.6]
print(het_list)

# Change Values of the Innermost List Items to ['False', 'True']
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
nest_het_list[4][2] = ['False', 'True']
print(nest_het_list)
```

```
# Change Values of the Innermost List Items to ['True', 'False'] Using Negative Indexing
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
nest_het_list[-4][-3] = ['True', 'False']
print(nest_het_list)
```

Output -

```
[1, 2, 'India', 9.6, 'a', 'rty', 'wer']
[1, 2, 5.6, 9.6, 'a', 100, 'wer']
[1.3, 4, 89.0, 'Ian', [2, 7.6, ['True', False], 'yes', 'N'], 'U', 'Tia', 'ok']
[1.3, 4, 89.0, 'Ian', [2, 7.6, [True, 'False'], 'yes', 'N'], 'U', 'Tia', 'ok']
```

```
[1, 6, 'Japan', 'z', 21.6, 'a', 'rty', 'wer']
[1, 2, 5.6, 9.6, 'a', 6, 'Japan', 'z', 21.6, 'rty', 'wer']
[1.3, 4, 89.0, 'Ian', [2, 7.6, ['False', 'True'], 'yes', 'N'], 'U', 'Tia', 'ok']
[1.3, 4, 89.0, 'Ian', [2, 7.6, ['True', 'False'], 'yes', 'N'], 'U', 'Tia', 'ok']
```

Any attempt to change an Item beyond the Indices Range of a “List” would raise an “IndexEror” Exception.

```
# Changing Value of the Tenth Item, which does Not Exist, to 23.7, Raises Exception
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list[9] = 23.7
print(het_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 160, in <module>
    het_list[9] = 23.7
IndexError: list assignment index out of range
```

Add List Items in Python

- There are various ways in which Items can be added to a “List” in Python.
- ***append () Method*** - To add an Item to the end of the “List”, the “append ()” method can be used. Since, the “append ()” method does not return any Value, there is no reason to assign the “append ()” method Expression to any Variable. This method modifies the original “List”. This method returns “None”. The Syntax of the “append ()” method is - list.append (item). The “append ()” method takes a Single Argument -
  - ***item*** - An Item to be added at the end of the “List”. The Item can be Number, String, Dictionary, another “List” etc.

```
# 1. Add a Single List Item Using "append ()" Method
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list.append(86)
print(het_list)
```

```
# Adding Another List at the end of the List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
new_het_list = [43.6, 22, 'e', 'tqw', "ace"]
het_list.append(new_het_list)
print(het_list)
```

Output -

```
[1, 2, 5.6, 9.6, 'a', 'rty', 'wer', 86]
```

```
[1, 2, 5.6, 9.6, 'a', 'rty', 'wer', [43.6, 22, 'e', 'tqw', 'ace']]
```

- ***extend () Method*** - The "extend ()" method can be used to add all the Items of an Iterable, like List, Tuple, String etc., to the end of the "List" on which the "extend ()" method is called. Since, the "extend ()" method does not return any Value, there is no reason to assign the "extend ()" method Expression to any Variable. This method modifies the original "List". This method returns "None". The Syntax of the "extend ()" method is - list.extend (iterable). The "extend ()" method takes a Single Argument -
  - ***iterable*** - The "extend ()" method takes an Iterable, like "List", "Tuple", "String" etc. as Argument.

```
# 2. Add Multiple List Item Using "extend ()" Method
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
new_het_list = [43.6, 22, 'e', 'tqw', "ace"]
het_list.extend(new_het_list)
print(het_list)
```

Output -

```
[1, 2, 5.6, 9.6, 'a', 'rty', 'wer', 43.6, 22, 'e', 'tqw', 'ace']
```

- ***List Concatenation*** - The "Addition Operator", i.e., "+" can be used to append all the Items of an Iterable, like "List", "Tuple", "String" etc. at the end of a "List". This is called as "List Concatenation".

```
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
new_het_list = [43.6, 22, 'e', 'tqw', "ace"]
het_list = het_list + new_het_list
print(het_list)
```

Output -

```
[1, 2, 5.6, 9.6, 'a', 'rty', 'wer', 43.6, 22, 'e', 'tqw', 'ace']
```

- ***Multiplication Operator*** - The "Multiplication Operator", i.e., "*" can be used to repeat an Item of a "List", a specified number of times, to create a "List" of Repeated Items.

```
mul_op_list = ['Hi', 'Hello', 'Bye'] * 5
print(mul_op_list)
```

Output -

```
['Hi', 'Hello', 'Bye', 'Hi', 'Hello', 'Bye', 'Hi', 'Hello', 'Bye', 'Hi', 'Hello', 'Bye', 'Hi', 'Hello', 'Bye']
```

  - ***Creating Multi-Dimensional List*** - The "Multiplication Operator", i.e., "*" can also be used to repeat an entire "List" to create a "Multi-Dimensional List".

```
two_dim_list = [['Hi'] * 5] * 4
print(two_dim_list)

two_dim_list = [['Hi', 'Hello', 'Bye']] * 3
print(two_dim_list)
```

Output -

```
[['Hi', 'Hi', 'Hi', 'Hi', 'Hi'], ['Hi', 'Hi', 'Hi', 'Hi', 'Hi'], ['Hi', 'Hi', 'Hi', 'Hi', 'Hi'], ['Hi', 'Hi', 'Hi', 'Hi', 'Hi']]
[['Hi', 'Hello', 'Bye'], ['Hi', 'Hello', 'Bye'], ['Hi', 'Hello', 'Bye']]
```

But in this way, Python will only create the References as Sub-Lists, instead of, creating separate "List Objects". Trying to change the Value of the Third Item in the First Row, will also get the Values changed in the Third Item in other Rows.

```
two_dim_list = [['Hi', 'Hello', 'Bye']] * 3
print(two_dim_list)

two_dim_list[0][2] = 'What'
print(two_dim_list)
```

Output -

```
[['Hi', 'Hello', 'Bye'], ['Hi', 'Hello', 'Bye'], ['Hi', 'Hello', 'Bye']]
[['Hi', 'Hello', 'What'], ['Hi', 'Hello', 'What'], ['Hi', 'Hello', 'What']]
```

- ***insert () Method*** - It is possible to Insert an Item at the specified Index in a "List", by using the "insert ()" method. Since, the "insert ()" method does not return any Value, there is no reason to assign the "insert ()" method Expression to any Variable. This

method updates the original “List”. This method returns “None”. The Syntax of the “insert ()” method is - list.insert (index, element). The “insert ()” method takes two Arguments -

- ***index*** - The Index where the Item “element” needs to be inserted in the “List”.
- ***element*** - The Item to be inserted in the “List”. All the Items present in the “List”, after the Item “element” to insert, will shift to the Right.

```
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list.insert(2, 65)
print(het_list)
```

Output -

```
[1, 2, 65, 5.6, 9.6, 'a', 'rty', 'wer']
```

It is also possible to insert an Iterable, like List, Tuple, String etc., at the specified Index in a “List” on which the “insert ()” method is called.

```
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
new_het_list = [6, 'Japan', 'z', 21.6]
het_list.insert(3, new_het_list)
print(het_list)
```

Output -

```
[1, 2, 5.6, [6, 'Japan', 'z', 21.6], 9.6, 'a', 'rty', 'wer']
```

- ***List Slicing Syntax*** - It is possible to Insert Multiple Items at a specified Index in a “List”, by using the Slice Assignment. The Multiple Items need to be squeezed into an Empty Slice of a “List”.

```
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
new_het_list = [43.6, 22, 'e', 'tqw', "ace"]
het_list[2:2] = new_het_list
print(het_list)
```

Output -

```
[1, 2, 43.6, 22, 'e', 'tqw', 'ace', 5.6, 9.6, 'a', 'rty', 'wer']
```

Deleting List Items in Python

- There are various ways in which Items can be deleted, or removed from a “List” in Python.
- ***del Operator*** - It is possible to delete, or, remove One, or, more Items from a “List” using the “del” Keyword.

```
# Delete the Fourth Item from the List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
del het_list[3]
print(het_list)

# Delete the Third Last Item from the List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
del het_list[-3]
print(het_list)
```

```
# Delete from the Second to Fifth Items from the List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
del het_list[1:5]
print(het_list)
```

Output -

```
[1, 2, 5.6, 'a', 'rty', 'wer']
[1, 2, 5.6, 9.6, 'rty', 'wer']

[1, 'rty', 'wer']

Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 233, in <module>
    print(het_list)
NameError: name 'het_list' is not defined
```

It is also possible to delete the entire “List” Object using the “del ()” Keyword.

```
# Delete an Entire List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
del het_list
print(het_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 233, in <module>
    print(het_list)
NameError: name 'het_list' is not defined
```

- ***remove () Method*** - The "remove ()" method removes the First Matching Item, which is passed as an Argument, from the "List". Since, the "remove ()" method does not return any Value, there is no reason to assign the "remove ()" method Expression to any Variable. This method modifies the original "List". This method returns "None". The Syntax of the "remove ()" method is - list.remove (element). The "remove ()" method takes a Single Argument -
  - ***element*** - The "remove ()" method takes a Single Item "element" as an Argument, and, removes it from the "List".

```
# Delete the Item, having Value "rty" from the List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list.remove('rty')
print(het_list)
```

Output -

```
[1, 2, 5.6, 9.6, 'a', 'wer']
```

If the Item to remove does not exist in the "List", the "remove ()" method raises a "ValueError" Exception.

```
# Deleting the Value, which does Not Exist, from the List, Raises Exception
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list.remove('abc')
print(het_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 243, in <module>
    het_list.remove('abc')
ValueError: list.remove(x): x not in list
```

If a "List" contains Duplicate Items, the "remove ()" method only removes the First Matching Occurrence.

```
# Delete the Item, having Value "rty" from the List having Duplicate Item "rty"
het_list = list([1, 2, 5.6, "rty", 9.6, 'a', 'rty', "wer", 2.6, 'rty', 12])
het_list.remove('rty')
print(het_list)
```

Output -

```
[1, 2, 5.6, 9.6, 'a', 'rty', 'wer', 2.6, 'rty', 12]
```

- ***pop*** - The "pop ()" method removes the Item at the Index, provided as an Argument, from the "List", and returns the removed Item. The Syntax of the "pop ()" method is - list.pop (index). The "pop ()" method takes a Single Argument -
  - ***index*** - This is an Optional Argument. The Index at which the Item to remove is passed as an Argument.

```
# Delete the Fourth Item from the List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
deleted_item = het_list.pop(3)
print('Deleted Item : ', deleted_item)
print('Updated List : ', het_list)

# Delete the Third Last Item from the List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
deleted_item = het_list.pop(-3)
print('Deleted Item : ', deleted_item)
print('Updated List : ', het_list)
```

Output -

```
Deleted Item :  9.6
Updated List :  [1, 2, 5.6, 'a', 'rty', 'wer']
Deleted Item :  a
Updated List :  [1, 2, 5.6, 9.6, 'rty', 'wer']
```

If no Argument is passed, the Default Index "-1" is passed as an Argument to the "pop ()" method implicitly. Hence, if no Index is provided, the "pop ()" method removes and returns the Last Item in the "List". This helps to implement "List" as "Stack", i.e., "FIFO" Data Structure - "First In Last Out".

```
# Delete the Last Item from the List
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
deleted_item = het_list.pop()
print('Deleted Item : ', deleted_item)
print('Updated List : ', het_list)
```

Output -

```
Deleted Item :  wer
Updated List :  [1, 2, 5.6, 9.6, 'a', 'rty']
```

If the Index passed to the “pop ()” method is not in Range, the “pop ()” method raises an “IndexError” Exception.

```
# Deleting the Item, at an Index that does Not Exist in the List, Raises Exception
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
deleted_item = het_list.pop(10)
print('Deleted Item : ', deleted_item)
print('Updated List : ', het_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 287, in <module>
    deleted_item = het_list.pop(10)
IndexError: pop index out of range
```

- ***clear*** - The “clear ()” method removes all Items from a “List”. Since, the “clear ()” method does not return any Value, there is no reason to assign the “clear ()” method Expression to any Variable. The “clear ()” method only empties the “List” on which the method is called. This method returns “None”.

```
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list.clear()
print(het_list)
```

Output -

```
[]
```

- ***Slicing*** - Items in a “List” can also be deleted, or, removed by assigning an “Empty List” to a Slice of its Items.

```
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
het_list[2:5] = []
print(het_list)
```

Output -

```
[1, 2, 'rty', 'wer']
```

Searching Items in List in Python

- There are various ways in which an Item can be searched in a "List" in Python.
- ***in Operator*** - It is possible to check if an Item in a "List" is present using the "in" Keyword. The "in" Keyword returns True if the Item is present in a "List". Otherwise, returns False.

```
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
print(2 in het_list)
print('z' in het_list)
```

Output -

```
True
False
```

- ***not in Operator*** - It is possible to check if an Item in a "List" is not present using the "not in" Keyword. The "not in" Keyword returns True if the Item is not present in a "List". Otherwise, returns False.

```
het_list = list([1, 2, 5.6, 9.6, 'a', 'rty', "wer"])
print('z' not in het_list)
print(2 not in het_list)
```

Output -

```
True
False
```

- ***index () Method*** - The "index ()" method returns the Index of the First Occurrence of the matching specified Item in the "List". The Syntax of the "index ()" method is - list.index (element, start, end). The "index ()" method takes a maximum of three Arguments -
  - ***element*** - The Item to be searched.

- ***start*** - This is an Optional Argument. “start” represents the Index in the “List” from where the search needs to be started.
- ***end*** - This is an Optional Argument. “end” represents the Index in the “List” up to where the search needs to be performed.

```
het_list = list([1, 2, 5.6, "rty", 9.6, 'a', 'rty', "wer", 2.6, 'rty', 12])
result_index = het_list.index('rty')
print("Index : ", result_index)

het_list = list([1, 2, 5.6, "rty", 9.6, 'a', 'rty', "wer", 2.6, 'rty', 12])
result_index = het_list.index('rty', 4, 9)
print("Index : ", result_index)
```

Output -

```
Index :  3
Index :  6
```

If the Item to be searched is not found, the “index ()” method raises a “ValueError” Exception.

```
het_list = list([1, 2, 5.6, "rty", 9.6, 'a', 'rty', "wer", 2.6, 'rty', 12])
result_index = het_list.index('xyz')
print("Index : ", result_index)

het_list = list([1, 2, 5.6, "rty", 9.6, 'a', 'rty', "wer", 2.6, 'rty', 12])
result_index = het_list.index('zyx', 2, 9)
print("Index : ", result_index)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 317, in <module>
    result_index = het_list.index('xyz')
ValueError: 'xyz' is not in list
```

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 321, in <module>
    result_index = het_list.index('zyx', 2, 9)
ValueError: 'zyx' is not in list
```

- ***min () Function*** - Python “List” supports the “min ()” function to find the Item carrying the Minimum Value in the “List”.

```
int_list = [23, 12, 44, 9, 19, 5, 39]
print(min(int_list))

months_list = ['Jan', "Feb", 'Mar', "Apr", 'May', "Jun", 'Jul', "Aug", 'Sep', "Oct", 'Nov', "Dec"]
print(min(months_list))
```

Output -

```
5
Apr
```

The “min ()” function does not work on “Heterogeneous List”. In this case, “TypeError” Exception is raised.

```
het_list = list([1, 2, 5.6, "rty", 9.6, 'a', 'rty', "wer", 2.6, 'rty', 12])
print(min(het_list))
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 332, in <module>
    print(min(het_list))
TypeError: '<' not supported between instances of 'str' and 'int'
```

The “min ()” function also does not work on “Homogeneous Nested List”. In this case, “TypeError” Exception is raised.

```
nest_int_list = [5, 9, 49, [3, 6, 8, [11, 22, 67], 12, 15], 76, 34]
print(min(nest_int_list))

nest_str_list = ["Name", "Address", "City", ['Kolkata', 'Mumbai', ['Beadon St.', "VIP Road"], 'Pune'], "State", "Country"]
print(min(nest_str_list))
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 335, in <module>
    print(min(nest_int_list))
TypeError: '<' not supported between instances of 'list' and 'int'
```

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 338, in <module>
    print(min(nest_str_list))
TypeError: '<' not supported between instances of 'list' and 'str'
```

- ***Python min () Function*** - Python “min ()” Function returns the Smallest Item in an Iterable. It can also be used to find the Smallest Item between two, or more Parameters. The “min ()” Function has two Forms -

- ✓ ***min (Iterable, *iterables, key=..., default=...)*** - To find the smallest Item in an Iterable, this Syntax is used.
  - ***iterable*** - The "iterable" can be a Sequence (String, Tuple, List), or, a Collection (Dictionary, Set, Frozen Set).
  - ****iterables*** - This is an Optional Argument. "*iterables" represents any Number of Iterables, can be more than one.
  - ***key*** - This is an Optional Argument. "key" represents a Function where the Iterables are passed, and, Comparison is performed based on its Return Value.
  - ***default*** - This is an Optional Argument. "default" represents a Default Value, if the given Iterable is Empty.

  If an Empty Iterator is passed, the "min ()" Function raises a "ValueEror" Exception.

```
empty_list = []
min_value = min(empty_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 346, in <module>
    min_value = min(empty_list)
ValueError: min() arg is an empty sequence
```

To avoid this, the "default" Argument can be passed.

```
empty_list = []
min_value = min(empty_list, default=0)
print("Minimum Value : ", min_value)
```

Output -

```
Minimum Value :  0
```

If more than one Iterators are passed, the Iterator, having the Smallest Value in the First Index, i.e., in "0th Index", is returned.

```
int_list1 = [78, 1, 2, 82, 12]
int_list2 = [34, 2, 67, 2, 90, 12]
int_list3 = [44, 12, 1, 54, 93]
min_value = min(int_list1, int_list2, int_list3)
print("Minimum Value : ", min_value)
```

Output -

```
Minimum Value :  [34, 2, 67, 2, 90, 12]
```

- ✓ ***min (arg1, arg2, *args, key)*** - To find the smallest Item between two, or, more Arguments, this Syntax is used.
  - ***arg1*** - "arg1" represents an Object, which can be Numbers, Strings etc.
  - ***arg2*** - "arg2" represents an Object, which can be Numbers, Strings etc.
  - ****args*** - This is an Optional Argument. "*args" represents any Number of Objects, can be more than one.
  - ***key*** - This is an Optional Argument. "key" represents a Function where each Argument is passed, and, Comparison is performed based on its Return Value.

```
min_value = min(4, -5, 12, 5, -9, 9)
print("Minimum Value : ", min_value)
```

Output -

```
Minimum Value :  -9
```

- ***max () Function*** - Python "List" supports the "max ()" function to find the Item carrying the Maximum Value in the "List".

```
int_list = [23, 12, 44, 9, 19, 5, 39]
print(max(int_list))

months_list = ['Jan', "Feb", 'Mar', "Apr", 'May', "Jun", 'Jul', "Aug", 'Sep', "Oct", 'Nov', "Dec"]
print(max(months_list))
```

Output -

```
44
Sep
```

The “max ()” function does not work on “Heterogeneous List”. In this case, “TypeError” Exception is raised.

```
het_list = list([1, 2, 5.6, "rty", 9.6, 'a', 'rty', "wer", 2.6, 'rty', 12])
print(max(het_list))
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 348, in <module>
    print(max(het_list))
TypeError: '>' not supported between instances of 'str' and 'float'
```

The “max ()” function also does not work on “Homogeneous Nested List”. In this case, “TypeError” Exception is raised.

```
nest_int_list = [5, 9, 49, [3, 6, 8, [11, 22, 67], 12, 15], 76, 34]
print(max(nest_int_list))

nest_str_list = ["Name", "Address", "City", ['Kolkata', 'Mumbai', ['Beadon St.', "VIP Road"], 'Pune'], "State", "Country"]
print(max(nest_str_list))
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 351, in <module>
    print(max(nest_int_list))
TypeError: '>' not supported between instances of 'list' and 'int'
```

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 354, in <module>
    print(max(nest_str_list))
TypeError: '>' not supported between instances of 'list' and 'str'
```

- ***Python max () Function*** - Python “max ()” Function returns the Largest Item in an Iterable. It can also be used to find the Largest Item between two, or, more Parameters. The “max ()” Function has two Forms –
  - ✓ ***max (Iterable, *iterables, key=..., default=...)*** - To find the largest Item in an Iterable, this Syntax is used.
    - ***iterable*** - The “iterable” can be a Sequence (String, Tuple, List), or, a Collection (Dictionary, Set, Frozen Set).
    - ****iterables*** - This is an Optional Argument. “*iterables” represents any Number of Iterables, can be more than one.
    - ***key*** - This is an Optional Argument. “key” represents a Function where the Iterables are passed, and, Comparison is performed based on its Return Value.

- ***default*** - This is an Optional Argument. "default" represents a Default Value, if the given Iterable is Empty.

If an Empty Iterator is passed, the "max ()" Function raises a "ValueEror" Exception.

```
empty_list = []
max_value = max(empty_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 382, in <module>
    max_value = max(empty_list)
ValueError: max() arg is an empty sequence
```

To avoid this, the "default" Argument can be passed.

```
empty_list = []
max_value = max(empty_list, default=0)
print("Maximum Value : ", max_value)
```

Output -

```
Maximum Value :  0
```

If more than one Iterators are passed, the Iterator, having the Smallest Value in the First Index, i.e., in "0th Index", is returned.

```
int_list1 = [44, 12, 1, 54, 93]
int_list2 = [34, 2, 67, 2, 90, 12]
int_list3 = [78, 1, 2, 82, 12]
max_value = max(int_list1, int_list2, int_list3)
print("Maximum Value : ", max_value)
```

Output -

```
Maximum Value :  [78, 1, 2, 82, 12]
```

✓ ***max (arg1, arg2, *args, key)*** - To find the largest Item between two, or, more Arguments, this Syntax is used.

- ***arg1*** - "arg1" represents an Object, which can be Numbers, Strings etc.
- ***arg2*** - "arg2" represents an Object, which can be Numbers, Strings etc.

- ***args*** - This is an Optional Argument. "*args" represents any Number of Objects, can be more than one.
- ***key*** - This is an Optional Argument. "key" represents a Function where each Argument is passed, and, Comparison is performed based on its Return Value.

```
max_value = max(4, -5, 12, 5, -9, 9)
print("Maximum Value : ", max_value)
```

Output -

```
Maximum Value :  12
```

- ***count () Method*** - The "count ()" method returns the Number of Times, the specified Item appears in the "List". The Syntax of the "count ()" method is - list.count (element). The "count ()" method takes a Single Argument -
  - ***element*** - The Item to be counted.

```
het_list = list([1, 2, 5.6, "rty", 9.6, 'a', 'rty', "wer", 2.6, 'rty', 12])
count_of_item = het_list.count('rty')
print("No. of Occurrences : ", count_of_item)
```

Output -

```
No. of Occurrences :  3
```

If the specified Item does not appear in the "List", the "count ()" method returns "0".

```
het_list = list([1, 2, 5.6, "rty", 9.6, 'a', 'rty', "wer", 2.6, 'rty', 12])
count_of_item = het_list.count(10)
print("No. of Occurrences : ", count_of_item)
```

Output -

```
No. of Occurrences :  0
```

If the specified Item appears in any of the "Inner List", present inside the "List" on which the "count ()" method is called, the method returns only the Number of Times the specified Item is present outside any of the "Inner List", present inside the "List" on which the "count ()" method is called.

```
nest_int_list = [5, 9, 49, [3, 9, 8, [11, 9, 67], 12, 99], 9, 34]
count_of_item = nest_int_list.count(9)
print("No. of Occurrences : ", count_of_item)
```

Output -

```
No. of Occurrences :  2
```

It is also possible to find the Number of Times the specified “List”, passed as Argument to the “count ()” method, is present inside the “List” as Item (Not Nested Item).

```
nest_list_list = [['Hi', 'Hello', 'Bye'], ['Hi', 'Hello', 'Bye'], ['Hi', 'Hello', 'Bye']]
count_of_item = nest_list_list.count(['Hi', 'Hello', 'Bye'])
print("No. of Occurrences : ", count_of_item)
```

Output -

```
No. of Occurrences :  3
```

If the specified “List”, passed as Argument to the “count ()” method exists as “Inner List” also inside the “List” on which the “count ()” method is called, the method returns only the Number of Times the specified “List” is present outside any of the “Inner List”, present inside the “List” on which the “count ()” method is called.

```
nest_list_list = [['Hi', 'Bye'], 7.9, 2, ['Hi', ['Hi', 'Bye'], 'Why', 'Oindrila'], 8.4, True]
count_of_item = nest_list_list.count(["Hi", "Bye"])
print("No. of Occurrences : ", count_of_item)
```

Output -

```
No. of Occurrences :  1
```

- ***all () Function*** - The “all ()” Function returns True, if all Items in the given Iterable, passed as Argument, are True, or, Non-Zero. If any Item in the Iterable is False, or, Zero “0”, the “all ()” Function returns False. The Syntax of the “all ()” Function is - all (iterable). The “all ()” Function takes a Single Argument -
  - ***iterable*** - The “all ()” Function takes a Single Item “iterable” as an Argument, which can be any Iterable, like a Sequence (String, Tuple, List), or, a Collection (Set, Dictionary, Frozen Set), or, any other Iterator.

```
int_list = [4, 6, 8, 9, 0, 6]
print("All List Items are : ", all(int_list))

int_list = [4, 6, 8, 9, -4, 6]
print("All List Items are : ", all(int_list))
```

```
het_list = [4, 6.8, 't', True, -4, 'yes']
print("All List Items are : ", all(het_list))

het_list = [4, 6.8, 't', True, -4, 'yes', False, 78, 'no']
print("All List Items are : ", all(het_list))
```

Output -

```
All List Items are :  False
All List Items are :  True
```

```
All List Items are :  True
All List Items are :  False
```

The “all ()” Function cannot check if the Nested Items of a “Nested List” are True, or, Non-Zero.

```
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, 8, False], "yes", 'N'], 'U', 'Tia', 'ok'])
print("All List Items are : ", all(nest_het_list))
```

Output -

```
All List Items are :  True
```

The “all ()” Function returns True for an “Empty List”.

```
empty_list = []
print("All List Items are : ", all(empty_list))
```

Output -

```
All List Items are :  True
```

- ***any () Function*** - The “any ()” Function returns True, if at least one Item of the given Iterable, passed as Argument, is True, or, Non-Zero. If all Items in the Iterable are False, or, Zero “0”, the “any ()” Function returns False. The Syntax of the “any ()” Function is - any (iterable). The “any ()” Function takes a Single Argument -
  - ***iterable*** - The “any ()” Function takes a Single Item “iterable” as an Argument, which can be any Iterable, like a Sequence (String, Tuple, List), or, a Collection (Set, Dictionary, Frozen Set), or, any other Iterator.

```
int_list = [0, 0, 0, 0, 0, 0]
print("All List Items are : ", any(int_list))

int_list = [0, 0, 0, 10, 0, 0]
print("All List Items are : ", any(int_list))
```

```
het_list = [False, False, False, False, False, False]
print("All List Items are : ", any(het_list))

het_list = [False, False, False, -5, False, False]
print("All List Items are : ", any(het_list))

het_list = [False, False, False, True, False, False]
print("All List Items are : ", any(het_list))
```

Output -

```
All List Items are :  False
All List Items are :  True
```

```
All List Items are :  False
All List Items are :  True
All List Items are :  True
```

The “any ()” Function cannot check if the Nested Items of a “Nested List” are True, or, Non-Zero.

```
nest_het_list = list([False, False, False, False, [False, False, [False, False, False], False, False], False, False])
print("All List Items are : ", any(nest_het_list))
```

Output -

```
All List Items are :  True
```

The “any ()” Function returns False for an “Empty List”.

```
empty_list = []
print("All List Items are : ", any(empty_list))
```

Output -

```
All List Items are :  False
```

Sorting a List in Python

- There are two ways in which the Items in a “List” can be sorted in Python.
- ***sort () Method*** - The “sort ()” method sorts the Items of the “List”, on which the method is called, in a specific Ascending, or, Descending Order. Since, the “sort ()” method does not return any Value, there is no reason to assign the “sort ()” method Expression to any Variable. This method updates the original “List”. This method returns “None”. The Syntax of the “sort ()” method is - list.sort (key=..., reverse=...). By default, the “sort ()” method does not require any extra Arguments. However, it takes two Optional Arguments -
  - ***key*** - This is an Optional Argument. “key” represents the Function that serves as a “Key” for the Sort Comparison. Default is None, if no Value is provided.
  - ***reverse*** - This is an Optional Argument. If True is provided, the “Sorted List” is reversed, i.e., sorted in Descending Order. Default is False, if no Value is provided.

```
# Sort the List Using "sort ()" Method in Ascending Order
int_list = [23, 12, 44, 9, 19, 5, 39]
int_list.sort()
print(int_list)

str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]
str_list.sort()
print(str_list)
```

Output -

```
[5, 9, 12, 19, 23, 39, 44]
['Debojyoti', 'Dola', 'Kasturi', 'Oindrila', 'Premanshu', 'Rama', 'Soumyajyoti']
```

Attempting to Sort a “Homogeneous Nested List” using the “sort ()” method called on that “Homogeneous Nested List” will raise a “TypeError” Exception.

```
nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
nest_int_list.sort()
print(nest_int_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 395, in <module>
    nest_int_list.sort()
TypeError: '<' not supported between instances of 'list' and 'int'
```

Attempting to Sort a “Heterogeneous List” using the “sort ()” method called on that “Nested List” will raise a “TypeError” Exception.

```
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
het_list.sort()
print(het_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 400, in <module>
    het_list.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

Sort List in Descending Order - If “reverse=True” is provided as the Argument to the “sort ()” method, the “List” is sorted in Descending Order.

```
# Sort the List Using "sort ()" Method in Descending Order
int_list = [23, 12, 44, 9, 19, 5, 39]
int_list.sort(reverse=True)
print(int_list)

str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]
str_list.sort(reverse=True)
print(str_list)
```

Output -

```
[44, 39, 23, 19, 12, 9, 5]
['Soumyajyoti', 'Rama', 'Premanshu', 'Oindrila', 'Kasturi', 'Dola', 'Debojyoti']
```

Sort with Custom Function Using Key - To implement different Sorting Logic to sort a “List”, the “sort ()” method accepts an Optional Argument “key”.
Example - Since the “List”, i.e., “str_list” is of Type String, Python by default sorts it using the Alphabetical Order. To sort the “List” based on the Length of Each Item, from

Lowest Count to Highest, the Python's built-in Function "len ()" can be provided as the Optional Argument "key".

- ***len () Function*** - The "len ()" Function returns the Number of Items, i.e., the Length in an Object. The Syntax of the "len ()" Function is - len (item). The "len ()" Function takes a Single Argument –
  - ✓ ***item*** - The "item" can be a Sequence (String, Tuple, List), or, a Collection (Dictionary, Set, Frozen Set).

```
str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]
str_list.sort(key=len)
print(str_list)
```

Output -

```
['Rama', 'Dola', 'Kasturi', 'Oindrila', 'Premanshu', 'Debojyoti', 'Soumyajyoti']
```

Failing to pass an Argument, or, passing an Invalid Argument will raise a "TypeError" Exception.

```
# Not giving Argument to "len ()" Function raises "TypeError" Exception
str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]
length = len()
```

```
# Giving Invalid Argument to "len ()" Function raises "TypeError" Exception
age = 78
length = len(age)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 463, in <module>
    length = len()
TypeError: len() takes exactly one argument (0 given)
```

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 467, in <module>
    length = len(age)
TypeError: object of type 'int' has no len()
```

It is also possible to sort the "List", i.e., "str_list" based on the Length of Each Item, from Highest Count to Lowest using by provided the "len ()" Function as the Optional Argument "key" to the "sort ()" method.

```
str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]
str_list.sort(key=len, reverse=True)
print(str_list)
```

Output -

```
['Soumyajyoti', 'Premanshu', 'Debojyoti', 'Oindrila', 'Kasturi', 'Rama', 'Dola']
```

- ***sorted () Function*** - The "sorted ()" Function sorts the Items of a given Iterable in a specific Order (either Ascending or Descending) and returns the "Sorted Iterable" as a "List". The Syntax of the "sorted ()" Function is - sorted (iterable, key=..., reverse=...). The "sorted ()" Function takes a maximum of three Arguments -
  - ***iterable*** - "iterable" can be a Sequence (String, Tuple, List), or, a Collection (Set, Dictionary, Frozen Set), or, any other Iterator.
  - ***key*** - This is an Optional Argument. "key" represents the Function that serves as a "Key" for the Sort Comparison. Default is None, if no Value is provided.
  - ***reverse*** - This is an Optional Argument. If True is provided, the "Sorted List" is reversed, i.e., sorted in Descending Order. Default is False, if no Value is provided.

```
int_list = [23, 12, 44, 9, 19, 5, 39]
sorted_list = sorted(int_list)
print(sorted_list)

str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]
sorted_list = sorted(str_list)
print(sorted_list)
```

Output -

```
[5, 9, 12, 19, 23, 39, 44]
['Debojyoti', 'Dola', 'Kasturi', 'Oindrila', 'Premanshu', 'Rama', 'Soumyajyoti']
```

Attempting to Sort a "Homogeneous Nested List" using the "sorted ()" Function will raise a "TypeError" Exception.

```
nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
sorted_list = sorted(nest_int_list)
print(sorted_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 434, in <module>
    sorted_list = sorted(nest_int_list)
TypeError: '<' not supported between instances of 'list' and 'int'
```

Attempting to Sort a “Heterogeneous List” using the “sorted ()” Function will raise a “TypeError” Exception.

```
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
sorted_list = sorted(het_list)
print(sorted_list)
```

Output -

```
Traceback (most recent call last):
  File "D:\Tutorials\Python Tutorial\Python_Project_Folder\Sequence In Python\WorkingWithListFile.py", line 439, in <module>
    sorted_list = sorted(het_list)
TypeError: '<' not supported between instances of 'str' and 'int'
```

Sort List in Descending Order - If “reverse=True” is provided as the Argument to the “sorted ()” Function, the “List” is sorted in Descending Order.

```
int_list = [23, 12, 44, 9, 19, 5, 39]
sorted_list = sorted(int_list, reverse=True)
print(sorted_list)

str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]
sorted_list = sorted(str_list, reverse=True)
print(sorted_list)
```

Output -

```
[44, 39, 23, 19, 12, 9, 5]
['Soumyajyoti', 'Rama', 'Premanshu', 'Oindrila', 'Kasturi', 'Dola', 'Debojyoti']
```

Sort with Custom Function Using Key - To implement different Sorting Logic to sort a “List”, the “sorted ()” Function accepts an Optional Argument “key”.
Example - Since the “List”, i.e., “str_list” is of Type String, Python by default sorts it using the Alphabetical Order. To sort the “List” based on the Length of Each Item, from Lowest Count to Highest, the Python’s built-in Function “len ()” can be provided as the Optional Argument “key”. The “len ()” Function is used to count the length of the Argument passed to this Function.

```
str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]
sorted_list = sorted(str_list, key=len)
print(sorted_list)
```

Output -

```
['Rama', 'Dola', 'Kasturi', 'Oindrila', 'Premanshu', 'Debojyoti', 'Soumyajyoti']
```

It is also possible to sort the "List", i.e., "str_list" based on the Length of Each Item, from Highest Count to Lowest using by provided the "len ()" Function as the Optional Argument "key" to the "sorted ()" Function.

```
str_list = ['Oindrila', 'Kasturi', 'Rama', "Premanshu", "Soumyajyoti", 'Dola', "Debojyoti"]
sorted_list = sorted(str_list, key=len, reverse=True)
print(sorted_list)
```

Output -

```
['Soumyajyoti', 'Premanshu', 'Debojyoti', 'Oindrila', 'Kasturi', 'Rama', 'Dola']
```

- ***Difference Between "sort ()" Method and "sorted ()" Function*** - The simplest difference between "sort ()" Method and "sorted ()" Function is that –
  - The "sort ()" Method changes the "List" directly, and, doesn't return any Value.
  - While, the "sorted ()" Function doesn't change the "List" and returns the "Sorted List".

Iterating Through a List in Python

- There are various ways in which each Items in a "List" can be iterated in Python.
- ***for-in Loop*** - Using a "for-in" Loop it is possible to iterate through each Item in a "List".

```
int_list = [23, 12, 44, 9, 19, 5, 39]
for int_item in int_list:
    print(int_item)

nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
for nest_int_item in nest_int_list:
    print(nest_int_item)
```

```
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
for het_list_item in het_list:
    print(het_list_item)

nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
for nest_het_list_item in nest_het_list:
    print(nest_het_list_item)
```

Output -

```
23
12
44
9
19
5
39
```

```
5
49
19
[8, 9, 1, [11, 9, 67], 99, 12]
34
9
```

```
2
1
9.6
rty
5.2
a
rty
wer
2.6
rty
12
```

```
1.3
4
89.0
Ian
[2, 7.6, [True, False], 'yes', 'N']
U
Tia
ok
```

- ***enumerate () Function*** - The "enumerate ()" Function adds Counter to an Iterable and returns it. The returned Object is an "Enumerate Object", which an be Converted to "List" using the "list ()" Constructor. The Syntax of the "enumerate ()" Function is - enumerate (iterable, start). The "enumerate ()" Function takes a maximum of two Arguments -
  - ***iterable*** - "iterable" can be a Sequence (String, Tuple, List), or, a Collection (Set, Dictionary, Frozen Set), or, any other Iterator.
  - ***start*** - This is an Optional Argument. "start" represents the Number from where the "numerate ()" Function starts Counting.
    If no Value is provided as "start", then the "enumerate ()" Function considers "0" as "start" Argument.

```
int_list = [23, 12, 44, 9, 19, 5, 39]
enum_int_list = enumerate(int_list)
for enum_item in enum_int_list:
    print(enum_item)
```

```
nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
enum_nest_int_list = enumerate(nest_int_list)
for enum_item in enum_nest_int_list:
    print(enum_item)
```

```
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
enum_het_list = enumerate(het_list)
for enum_item in enum_het_list:
    print(enum_item)
```

```
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
enum_nest_het_list = enumerate(nest_het_list)
for enum_item in enum_nest_het_list:
    print(enum_item)
```

```
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
enum_nest_het_list = enumerate(nest_het_list, start=100)
for enum_item in enum_nest_het_list:
    print(enum_item)
```

Output -

```
(0, 23)
(1, 12)
(2, 44)
(3, 9)
(4, 19)
(5, 5)
(6, 39)
```

```
(0, 5)
(1, 49)
(2, 19)
(3, [8, 9, 1, [11, 9, 67], 99, 12])
(4, 34)
(5, 9)
```

```
(0, 2)
(1, 1)
(2, 9.6)
(3, 'rty')
(4, 5.2)
(5, 'a')
(6, 'rty')
(7, 'wer')
(8, 2.6)
(9, 'rty')
```

```
(0, 1.3)
(1, 4)
(2, 89.0)
(3, 'Ian')
(4, [2, 7.6, [True, False], 'yes', 'N'])
(5, 'U')
(6, 'Tia')
(7, 'ok')
```

```
(100, 1.3)
(101, 4)
(102, 89.0)
(103, 'Ian')
(104, [2, 7.6, [True, False], 'yes', 'N'])
(105, 'U')
(106, 'Tia')
(107, 'ok')
```

While Iterating over an Enumerate Object, it is possible to display -

- The Counter of each Item separately by using any Variable to express the Counter, e.g., “index”.
- The Value of each Item separately by using any Variable to express the Item, e.g., “enum_item”.

```
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
enum_nest_het_list = enumerate(nest_het_list, start=100)
for index, enum_item in enum_nest_het_list:
    print(index, enum_item)
```

Output -

```
100 1.3
101 4
102 89.0
103 Ian
104 [2, 7.6, [True, False], 'yes', 'N']
105 U
106 Tia
107 ok
```

- ***reverse () Method*** - The “reverse ()” Method reverses the Items of the “List” on which the method is called. Since, the “reverse ()” method does not return any Value, there is no reason to assign the “reverse ()” method Expression to any Variable. This method updates the existing “List”. This method returns “None”.

```
int_list = [23, 12, 44, 9, 19, 5, 39]
int_list.reverse()
print(int_list)

nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
nest_int_list.reverse()
print(nest_int_list)
```

```
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
het_list.reverse()
print(het_list)
```

```
nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
nest_het_list.reverse()
print(nest_het_list)
```

Output -

```
[39, 5, 19, 9, 44, 12, 23]
[9, 34, [8, 9, 1, [11, 9, 67], 99, 12], 19, 49, 5]
```

```
[12, 'rty', 2.6, 'wer', 'rty', 'a', 5.2, 'rty', 9.6, 1, 2]
```

```
['ok', 'Tia', 'U', [2, 7.6, [True, False], 'yes', 'N'], 'Ian', 89.0, 4, 1.3]
```

- ***reversed () Function*** - The “reversed ()” Function returns the Reversed Iterator of the given Sequences (String, Tuple, List), passed as Argument. The Syntax of the “reversed ()” Function is - reversed (sequence). The “reversed ()” Function takes a Single Argument -
  - ***sequence*** - “sequence” represents a Sequence (String, Tuple, List) to be reversed.

```
int_list = [23, 12, 44, 9, 19, 5, 39]
rev_int_list = reversed(int_list)
print(list(rev_int_list))

nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
rev_nest_int_list = reversed(nest_int_list)
print(list(rev_nest_int_list))

het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
rev_het_list = reversed(het_list)
print(list(rev_het_list))

nest_het_list = list([1.3, 4, 89.0, 'Ian', [2, 7.6, [True, False], "yes", 'N'], 'U', 'Tia', 'ok'])
rev_nest_het_list = reversed(nest_het_list)
print(list(rev_nest_het_list))
```

Output -

```
[39, 5, 19, 9, 44, 12, 23]
[9, 34, [8, 9, 1, [11, 9, 67], 99, 12], 19, 49, 5]
[12, 'rty', 2.6, 'wer', 'rty', 'a', 5.2, 'rty', 9.6, 1, 2]
['ok', 'Tia', 'U', [2, 7.6, [True, False], 'yes', 'N'], 'Ian', 89.0, 4, 1.3]
```

- ***Slicing Operator*** - The "Slicing Operator" can be used to display Items of a "List" between two Indices, if specified, both in Forward and Reverse Order. If no Values for "start" and "end" Indices are provided, then all the Items of the "List" are displayed.

```
# Display All The Items of a List
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
print(het_list[:])

# Display from the Second to Fourth Items from of the List
nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
print(nest_int_list[1:5])
```

Output -

```
[2, 1, 9.6, 'rty', 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
[49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34]
```

If no Values for "start" and "end" Indices are provided, and, "-1" is provided as the Value for "step", then all the Items of the "List" are displayed in Reversed Order.

```
# Display All The Items of a List in Reverse
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
print(het_list[::-1])
```

Output -

```
[12, 'rty', 2.6, 'wer', 'rty', 'a', 5.2, 'rty', 9.6, 1, 2]
```

It is also possible to display Items of a "List" between two Indices, if specified, in Reverse Order, by providing Values for "start" and "end" Indices in Reverse Indexing, and, providing "-1" as the Value for "step".

```
# Display from the Second Last to Fourth Last Items from of the List
nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
print(nest_int_list[-2:-5:-1])
```

Output -

```
[34, [8, 9, 1, [11, 9, 67], 99, 12], 19]
```

Copying a List in Python

- There are various ways in which a "List" can be copied to another "List" in Python.
- ***Assignment Operator*** - A "List" can be copied using the "Assignment Operator", i.e., "=".

```
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
copy_het_list = het_list

print("Original List : ", het_list)
print("Copied List : ", copy_het_list)
```

Output -

```
Original List :  [2, 1, 9.6, 'rty', 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
Copied List :  [2, 1, 9.6, 'rty', 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
```

The problem with copying "Lists" in this way is that, if the "New List" is modified, the "Old List" will also be modified. It is because, the "New List" is Referencing, or, Pointing to the same "Old List" Object.

```
# Change the Fourth Item to 100 in the Copied List
copy_het_list[3] = 100

print("Original List : ", het_list)
print("Copied List : ", copy_het_list)
```

Output -

```
Original List :  [2, 1, 9.6, 100, 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
Copied List :  [2, 1, 9.6, 100, 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
```

- ***copy () Method*** - However, to keep the "Original List" unchanged when the "New List" is modified, the "copy ()" method is used on the "Original List". The "copy ()" method returns a Shallow Copy of the "List" on which the method is called, i.e., this method returns a "New List", and, does not modify the "Original List".

```
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
copy_het_list = het_list.copy()

print("Original List : ", het_list)
print("Copied List : ", copy_het_list)

# Change the Fourth Item to 100 in the Copied List
copy_het_list[3] = 100

print("Original List : ", het_list)
print("Copied List : ", copy_het_list)
```

Output -

```
Original List :  [2, 1, 9.6, 'rty', 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
Copied List :  [2, 1, 9.6, 'rty', 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
Original List :  [2, 1, 9.6, 'rty', 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
Copied List :  [2, 1, 9.6, 100, 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
```

However, if a "Nested List" is copied using the "copy ()" method, and, any changes are made to any Nested Item in the "Original Nested List", the same changes will reflect to the "New Nested List".

```
nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
copy_nest_int_list = nest_int_list.copy()

print("Original List : ", nest_int_list)
print("Copied List : ", copy_nest_int_list)

nest_int_list[3][3][1] = 109

print("Original List : ", nest_int_list)
print("Copied List : ", copy_nest_int_list)
```

Output -

```
Original List :  [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
Copied List :  [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
Original List :  [5, 49, 19, [8, 9, 1, [11, 109, 67], 99, 12], 34, 9]
Copied List :  [5, 49, 19, [8, 9, 1, [11, 109, 67], 99, 12], 34, 9]
```

Also, if a “Nested List” is copied using the “copy ()” method, and, any changes are made to any Nested Item in the “New Nested List”, the same changes will reflect to the “Original Nested List”.

```
nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
copy_nest_int_list = nest_int_list.copy()

print("Original List : ", nest_int_list)
print("Copied List : ", copy_nest_int_list)

copy_nest_int_list[3][3][0] = 221

print("Original List : ", nest_int_list)
print("Copied List : ", copy_nest_int_list)
```

Output -

```
Original List :  [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
Copied List :  [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
Original List :  [5, 49, 19, [8, 9, 1, [221, 9, 67], 99, 12], 34, 9]
Copied List :  [5, 49, 19, [8, 9, 1, [221, 9, 67], 99, 12], 34, 9]
```

- ***Slicing Operator*** - The "Slicing Operator" can be used to copy a "List" also. The "Slicing Operator" returns a Shallow Copy of the "List" on which the Operator is used.

```
het_list = list([2, 1, 9.6, "rty", 5.2, 'a', 'rty', "wer", 2.6, 'rty', 12])
copy_het_list = het_list[1:9]

print("Original List : ", het_list)
print("Copied List : ", copy_het_list)
```

Output -

```
Original List :  [2, 1, 9.6, 'rty', 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
Copied List :  [1, 9.6, 'rty', 5.2, 'a', 'rty', 'wer', 2.6]
```

In this type of copy, changes made to the "New List" does not reflect on the "Original List".

```
# Change the Fourth Item to 100 in the Copied List
copy_het_list[3] = 100

print("Original List : ", het_list)
print("Copied List : ", copy_het_list)
```

Output -

```
Original List :  [2, 1, 9.6, 'rty', 5.2, 'a', 'rty', 'wer', 2.6, 'rty', 12]
Copied List :  [1, 9.6, 'rty', 100, 'a', 'rty', 'wer', 2.6]
```

However, if a "Nested List" is copied using the "Slicing Operator", and, any changes are made to any Nested Item in the "Original Nested List", the same changes will reflect to the "New Nested List".

```
nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
copy_nest_int_list = nest_int_list[1:5]

print("Original List : ", nest_int_list)
print("Copied List : ", copy_nest_int_list)

nest_int_list[3][3][1] = 109

print("Original List : ", nest_int_list)
print("Copied List : ", copy_nest_int_list)
```

Output -

```
Original List :  [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
Copied List :  [49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34]
Original List :  [5, 49, 19, [8, 9, 1, [11, 109, 67], 99, 12], 34, 9]
Copied List :  [49, 19, [8, 9, 1, [11, 109, 67], 99, 12], 34]
```

Also, if a “Nested List” is copied using the “Slicing Operator”, and, any changes are made to any Nested Item in the “New Nested List”, the same changes will reflect to the “Original Nested List”.

```
nest_int_list = [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
copy_nest_int_list = nest_int_list[1:5]

print("Original List : ", nest_int_list)
print("Copied List : ", copy_nest_int_list)

copy_nest_int_list[2][3][0] = 221

print("Original List : ", nest_int_list)
print("Copied List : ", copy_nest_int_list)
```

Output -

```
Original List :  [5, 49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34, 9]
Copied List :  [49, 19, [8, 9, 1, [11, 9, 67], 99, 12], 34]
Original List :  [5, 49, 19, [8, 9, 1, [221, 9, 67], 99, 12], 34, 9]
Copied List :  [49, 19, [8, 9, 1, [221, 9, 67], 99, 12], 34]
```

Sum All the Items of a List in Python

- The "sum ()" Function adds the Value, provided in the "start" Argument, and the Items of given Iterable from Left to Right, and, returns the sum. The Syntax of the "sum ()" Function is - sum (iterable, start). The "sum ()" Function takes a maximum of two Arguments -
  - ***iterable*** - "iterable" can be a Sequence (String, Tuple, List), or, a Collection (Set, Dictionary, Frozen Set), or, any other Iterator.
    The important thing to remember is that the Items of the Iterable should be Numbers.
  - ***start*** – This is an Optional Argument. The Value of the "start" Argument is added to the sum of the Items of the Iterable.
    If no Value is provided in the "start" Argument, the Default Value is "0".

```
num_list = [10, 5, 8.6789, 2.6409, 56, 20, 12]
sum_of_list_items = sum(num_list)
print("Sum of List Items is : ", sum_of_list_items)

num_list = [10, 5, 8.6789, 2.6409, 56, 20, 12]
sum_of_list_items = sum(num_list, start=907.3456)
print("Sum of List Items is : ", sum_of_list_items)
```

Output -

```
Sum of List Items is :  114.3198
Sum of List Items is :  1021.6654
```

If it is needed to add Floating-Point Numbers with exact precision, the Function "math.fsum (iterable)" should be used instead.

```
num_list = [10.88889079070970970, 5.586585559595959, 8.6789, 2.6409, 56, 20, 12]
sum_of_list_items = math.fsum(num_list)
print("Sum of List Items is : ", sum_of_list_items)

num_list = [10.88889079070970970, 5.586585559595959, 8.6789, 2.6409, 56, 20, 12]
sum_of_list_items = sum(num_list, start=907.3456898454747477)
print("Sum of List Items is : ", sum_of_list_items)
```

Output -

```
Sum of List Items is :  115.79527635030567
Sum of List Items is :  1023.1409661957805
```