Here is an example of string in python and how to print it.

```
# Assigning string to a variable
a = "This is a string"
print (a)
```

For declaring a string, we assign a variable to the string. The indexing of elements in a string starts from 0.

Strings can be assigned variable names

```
a = "My dog's name is"
b = "Bingo"
```

Strings can be concatenated using the "+" operator:

```
c = a + " " + b
print(c)
```

Output :

```
"My dog's name is Bingo"
```

In forming the string c, we concatenated *three* strings, a, b, and a *string literal*, in this case a space " ", which is needed to provide a space to separate string a from b.

String Operations

Create list of strings

To create a list of strings, first use square brackets [and] to create a list. Then place the list items inside the brackets separated by commas. Remember that strings must be surrounded by quotes. Also remember to use = to store the list in a variable.

Example :

```
colors = ["red", "blue", "green"]
It is also allowed to put each string on a separate line:
animals = [
    "deer",
    "beaver",
    "cow"
]
```

Print list of strings :

To print a whole list of strings in one line, you can simply call the built-in **print** function, giving the list as an argument:

```
colors = ["red", "blue", "green"]
print(colors)
```

Output :

```
['red', 'blue', 'green']
```

Add strings to list

When you already have a list of strings and you want to add another string to it, you can use the append method:

```
colors = ["red", "blue", "green"]
colors.append("purple")
print(colors)
```

Output :

```
['red', 'blue', 'green', 'purple']
```

You can also create a new list with only one string in it and add it to the current list:

```
colors = colors + ["silver"]
```

Output :

```
['red', 'blue', 'green', 'purple', 'silver']
```

Print list as string

If you want to convert a list to a string, you can use the built-in function repr to make a string representation of the list:

```
products = ["shelf", "drawer"]
products_as_string = repr(products)
print(products_as_string)
```

Output :

```
['shelf', 'drawer']
```

Concatenate lists of strings

You can use the + operator to concatenate two lists of strings. For example:

```
colors1 = ["red", "blue"]
colors2 = ["purple", "silver"]
concatenated = colors1 + colors2
print(concatenated)
```

Output :

```
['red', 'blue', 'purple', 'silver']
```

Check if string is in list

You can use the **in** keyword to check if a list contains a string. This gives you a boolean value: either **True** or **False**. You can store this value somewhere, or use it directly in an **if** statement:

```
colors = ["pink", "cyan"]
if "pink" in colors:
    print("yes!")
has_cyan = "cyan" in colors
print(has_cyan)
```

Output :

```
yes!
True
```

Sort list of strings

To sort a list of strings, you can use the sort method:

```
numbers = ["one", "two", "three", "four"]
numbers.sort()
print(numbers)
```

Output :

```
['four', 'one', 'three', 'two']
```

You can also use the sorted() built-in function:

```
numbers = ["one", "two", "three", "four"]
numbers = sorted(numbers)
print(numbers)
```

Output :

```
['four', 'one', 'three', 'two']
```

Join list of strings

To join a list of strings by another string, you need to call the join method on the string, giving your list as an argument. For example, if we have this list:

```
colors = ["red", "blue", "green"]
print(", ".join(colors))
print("|".join(colors))
```

Output :

```
red, blue, green
red|blue|green
```

Traversing a String

You can use a for loop, range in Python, slicing operator, and a few more methods to traverse the characters in a string.

Using for loop to traverse a string

It is the most prominent and straightforward technique to iterate strings.

Example :

```
string1 = "Data"
for char in string1:
    print(char)
```

Output :

```
D
a
t
a
```

Using range() to traverse a string

Another quite simple way to traverse the string is by using Python range function. This method lets us access string elements using the index.

Example :

```
string1 = "Data"
for ch in range(len(string1)):
    print(string1[ch])
```

Output :

```
D
a
t
a
```

Using Slice operator to traverse strings partially

You can traverse a string as a substring by using the Python slice operator ([]). It cuts off a substring from the original string and thus allows to traverse over it partially.

The [] operator has the following syntax:

string [starting index : ending index : step value]

To use this method, provide the starting and ending indices along with a step value and then traverse the string. Below is the example code that iterates over the first six letters of a string.

Example :

```
String1 = "Python Data Science"
for char in string1[0 : 6 : 1]:
    print(char)
```

Output :

```
P
y
t
h
o
n
```

You can take the slice operator usage further by using it to iterate over a string but leaving every alternate character. Check out the below example:

Example :

```
string1 = "Python_Data_Science"
for char in string1[ :  : 2]:
    print(char)
```

Output :

```
P
t
o
_
a
a
S
i
n
e
```

Traverse string backward using slice operator

If you pass a -ve step value and skipping the starting as well as ending indices, then you can iterate in the backward direction. Go through the given code sample.

Example :

```
string_to_iterate = "Learning"
for char in string_to_iterate[ :  : -1]:
    print(char)
```

Output :

```
g
n
i
n
r
a
e
L
```

Using indexing to iterate strings backward

Slice operator first generates a reversed string, and then we use the for loop to traverse it. Instead of doing it, we can use the indexing to iterate strings backward.

Example :

```
string1 = "Learning"
ch = len(string1) - 1
while ch>= 0:
   print(string1[ch])
ch -= 1
```

Output :

```
g
n
i
n
r
a
e
L
```

5.2 Strings Methods and Built-in Functions

Python String Functions

Let's take a look at the various string functions in Python.

1) String Replace

This method is used to replace the string, which accepts two arguments.

Example :

```
lang = "Hello Selenium"
print(lang.replace("Selenium", "Python"))
```

Output :

```
Hello Python
```

2) String Reverse

This method is used to reverse a given string

Example :

```
lang = "Python"
print(''.join(reversed(lang)))
```

Output :

```
nohtyP
```

3) String Join

This method returns the string concatenated with the elements of iterable.

Example :

```
s1 = "ABC"
s2 = "123"
print(s1.join(s2))
```

Output :

```
1ABC2ABC3
```

4) String Split

This method is used to split the string based on the user arguments

Example :

```
text = "Welcome to Python"
print(text.split())
```

Output :

```
['Welcome', 'to', 'Python']
```

5) String Length

This method returns the length of the String.

Example :

```
text = "Python"
print(len(text))
```

Output :

```
6
```

6) String Compare

This method is used to compare two strings.

Example :

```
s1 = "Python"
s2 = "Python"
if(s1 == s2):
    print("Both strings are equal")
```

Output :

```
Both strings are equal
```

7) String Lowercase

This method is used to convert the uppercase to lowercase.

Example :

```
text = "PYTHON"
print(text.lower())
```

Output :

```
python
```

5.3 Introduction to List and its Operations

A **list** is a sequential collection of Python data values, where each value is identified by an index. The values that make up a list are called its **elements**. Lists are similar to strings, except that the elements of a list can have different types.Lists are one of the most powerful tools in python.

There are several ways to create a new list. The simplest is to enclose the elements in square brackets ([and]).

```
[10, 20, 30, 40]
["hello", "how", "you"]
```

The first example is a list of four integers. The second is a list of three strings. As we said above, the elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list.

Types of lists

Below are some examples of homogeneous and heterogeneous lists in python:

Homogenous Lists:

```
list=['a','b','c']
list=['dog','cat','horse']
list=[1,2,3,4,5]
```

Heterogeneous Lists:

```
list=[1,'dog',2.2,'horse']
```

Accessing an Item from the List

```
list=[1,'dog',2.2,'horse']
print(list[1])
```

Output :

```
dog
```

As mentioned earlier, that indexing starts from 0, so when index [1] is passed, it gives the result as "dog". Similarly, if we pass the index, say [2], it will give the output 2.2

Lists are mutable, i.e., they can be altered once declared.

Example : Modifying lists

```
L = [1, "a" , "string" , 1+2]
print L
```

```
L.append(6)
print L
L.pop()
print L
print L[1]
```

Output :

```
[1, 'a', 'string', 3]
[1, 'a', 'string', 3, 6]
[1, 'a', 'string', 3]
a
```

The last element of this array is L[3], because L has 4 elements. The last element can also be accessed as L[-1], no matter how many elements L has, and the next-to-last element of the list is L[-2], etc.

Example :

```
L = [1, "a" , "string" , 1+2]
print(L)
print(L[-1])
print (L[-2])
```

Output :

```
[1, 'a', 'string', 3]
3
string
```

Individual elements of lists can be changed. For example:

```
L = [1, "a" , "string" , 1+2]
print(L)
L[0]=L[0]+2
L[3=3.14159
print(L)
```

Output :

```
[1, 'a', 'string', 3]
[3, 'a', 'string', 3.14159]
```

Here we see that 2 was added to the previous value of L[0] and the value 3 was replaced by the floating point number 3.14159.

You can also add lists.

Example :

```
L = [0, 1, 1, 2, 3, 5, 8, 13]
print(L)
print(L+L)
```

Output :

```
[0, 1, 1, 2, 3, 5, 8, 13]
[0, 1, 1, 2, 3, 5, 8, 13, 0, 1, 1, 2, 3, 5, 8, 13]
```

Adding lists concatenates them, just as the "+" operator concatenates strings.

Slicing lists

You can access pieces of lists using the *slicing* feature of Python:

Example :

```
L = [10.0, 'girls & boys', (2+0j), 3.14159, 21]
print(L[1:4])
print(L[2:5])
print(L[2:])
print(L[:2])
print(L[:])
print(L[1:-1])
print(len(L))
```

Output :

```
['girls & boys', (2+0j), 3.14159]
[(2+0j), 3.14159, 21]
[(2+0j), 3.14159, 21]
[10.0, 'girls & boys']
[10.0, 'girls & boys', (2+0j), 3.14159, 21]
['girls & boys', (2+0j), 3.14159]
```

You access a subset of a list by specifying two indices separated by a colon ":". This is a powerful feature of lists that we will use often. If the left slice index is 0, you can leave it out; similarly, if the right slice index is the length of the list, you can leave it out also.You can get the length of a list using Python's len() function

Creating and modifying lists

Python has functions for creating and augmenting lists. The most useful is the range function, which can be used to create a uniformly spaced sequence of integers. The general form of the function is

```
range([start,] stop[, step])
```

where the arguments are all integers; those in square brackets are optional.

Example :

```
L=list(range(10))      # makes a list of 10 integers from 0 to 9
print(L)
L=list(range(3,10))    # makes a list of 10 integers from 3 to 12
print(L)
L=list(range(0,10,2))  # makes a list of 10 integers from 0 to 9 with increment 2
print(L)
```

Output :

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
```

5.4 List Methods and Built-in Functions

Method	Description	Example	Output
append(x)	Adds an item (x) to the end of the list. This is equivalent to a[len(a):] = [x].	a = ["bee", "moth"] print(a) a.append("ant") print(a)	['bee', 'moth'] ['bee', 'moth', 'ant']
extend(iterable)	Extends the list by appending all the items from the iterable. This allows you to join two lists together. This method is equivalent to a[len(a):] = iterable.	a = ["bee", "moth"] print(a) a.extend(["ant", "fly"]) print(a)	['bee', 'moth'] ['bee', 'moth', 'ant', 'fly']
insert(i, x)	Inserts an item at a given position. The first argument is the index of the element before which to insert. For example, a.insert(0, x) inserts at the front of the list.	a = ["bee", "moth"] print(a) a.insert(0, "ant") print(a) a.insert(2, "fly") print(a)	['bee', 'moth'] ['ant', 'bee', 'moth'] ['ant', 'bee', 'fly', 'moth']
remove(x)	Removes the first item from the list that has a value of x. Returns an error if there is no such item.	a = ["bee", "moth", "ant"] print(a) a.remove("moth") print(a)	['bee', 'moth', 'ant'] ['bee', 'ant']

pop([i])	Removes the item at the given position in the list, and returns it. If no index is specified, pop() removes and returns the last item in the list.	# Example 1: No index specified
		a = ["bee", "moth", "ant"]
		print(a)
		a.pop()
		print(a)
		# Example 2: Index specified
		a = ["bee", "moth", "ant"]
		print(a)
		a.pop(1)
		print(a)
			['bee', 'moth', 'ant']
			['bee', 'moth']
			['bee', 'moth', 'ant']
			['bee', 'ant']

clear()	Removes all items from the list. Equivalent to del a[:].	a = ["bee", "moth", "ant"] print(a)	['bee', 'moth', 'ant'] []

		a.clear() print(a)	
index(x[, start[, end]])	Returns the position of the first list item that has a value of x. Raises a ValueError if there is no such item. The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list.	a = ["bee", "ant", "moth", "ant"] print(a.index("ant")) print(a.index("ant", 2))	1 3
count(x)	Returns the number of times x appears in the list.	a = ["bee", "ant", "moth", "ant"] print(a.count("bee")) print(a.count("ant")) print(a.count(""))	1 2 0
sort(key=None, reverse=False)	Sorts the items of the list in place. The arguments can be used to customize the operation. *key* Specifies a function of one argument that is used to extract a comparison key from each list element. The default value is None (compares the elements directly). *reverse* Boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.	a = [3,6,5,2,4,1] a.sort() print(a) a = [3,6,5,2,4,1] a.sort(reverse=True) print(a) a = ["bee", "wasp", "moth", "ant"] a.sort() print(a) a = ["bee", "wasp", "butterfly"] a.sort(key=len) print(a) a = ["bee", "wasp", "butterfly"] a.sort(key=len, reverse=True) print(a)	[1, 2, 3, 4, 5, 6] [6, 5, 4, 3, 2, 1] ['ant', 'bee', 'moth', 'wasp'] ['bee', 'wasp', 'butterfly'] ['butterfly', 'wasp', 'bee']
reverse()	Reverses the elements of the list in place.	a = [3,6,5,2,4,1] a.reverse() print(a) a = ["bee", "wasp", "moth", "ant"] a.reverse() print(a)	[1, 4, 2, 5, 6, 3] ['ant', 'moth', 'wasp', 'bee']

copy()	Returns a shallow copy of the list. Equivalent to a[:].
	Use the copy() method when you need to update the copy without affecting the original list. If you don't use this method (eg, if you do something like list2 = list1), then any
		# WITHOUT copy()
		a = ["bee", "wasp", "moth"]
		b = a
		b.append("ant")
		print(a)
		print(b)
			['bee', 'wasp', 'moth', 'ant']
			['bee', 'wasp', 'moth', 'ant']
			['bee', 'wasp', 'moth']
			['bee', 'wasp', 'moth', 'ant']

	updates you do to list2 will also affect list1.
	The example at the side demonstrates this.
		# WITH copy()
		a = ["bee", "wasp", "moth"]
		b = a.copy()
		b.append("ant")
		print(a)
		print(b)	

Commonly Used Python List Functions

List Functions

The following Python functions can be used on lists.

Method	Description	Example	Output
len(s)	Returns the number of items in the list. The len() function can be used on any sequence (such as a string, bytes, tuple, list, or range) or collection (such as a dictionary, set, or frozen set).	a = ["bee", "moth", "ant"] print(len(a))	3
list([iterable])	The list() constructor returns a mutable sequence list of elements. The iterable argument is optional. You can provide any sequence or collection (such as a string, list, tuple, set, dictionary, etc). If no argument is supplied, an empty list is returned. Strictly speaking, list([iterable]) is actually a mutable sequence type.	print(list()) print(list([])) print(list(["bee", "moth", "ant"])) print(list([["bee", "moth"], ["ant"]])) a = "bee" print(list(a)) a = ("I", "am", "a", "tuple") print(list(a)) a = {"I", "am", "a", "set"} print(list(a))	[] [] ['bee', 'moth', 'ant'] [['bee', 'moth'], ['ant']] ['b', 'e', 'e'] ['I', 'am', 'a', 'tuple'] ['am', 'I', 'a', 'set']
max(iterable, *[, key, default]) or max(arg1, arg2, *args[, key])	Returns the largest item in an iterable (eg, list) or the largest of two or more arguments. The key argument specifies a one-argument ordering function like that used for sort(). The default argument specifies an object to return if the provided iterable is empty. If the iterable is empty and default is not provided, a ValueError is raised. If more than one item shares the maximum value, only the first one encountered is returned.	a = ["bee", "moth", "ant"] print(max(a)) a = ["bee", "moth", "wasp"] print(max(a)) a = [1, 2, 3, 4, 5] b = [1, 2, 3, 4] print(max(a, b))	moth wasp [1, 2, 3, 4, 5]

min(iterable, *[, key, default]) or min(arg1, arg2, *args[, key])	Returns the smallest item in an iterable (eg, list) or the smallest of two or more arguments. The key argument specifies a one-argument ordering function like that used for sort(). The default argument specifies an object to return if the provided iterable is empty. If the iterable is empty and default is not provided, a ValueError is raised. If more than one item shares the minimum value, only the first one encountered is returned.	a = ["bee", "moth", "wasp"] print(min(a)) a = ["bee", "moth", "ant"] print(min(a)) a = [1, 2, 3, 4, 5] b = [1, 2, 3, 4] print(min(a, b))	bee ant [1, 2, 3, 4]
range(stop) or range(start, stop[, step])	Represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops. It can be used along with list() to return a list of items between a given range. Strictly speaking, range() is actually a mutable sequence type.	print(list(range(10))) print(list(range(1,11))) print(list(range(51,56))) print(list(range(1,11,2)))	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] [51, 52, 53, 54, 55] [1, 3, 5, 7, 9]

1) len()

The Python list method **len()** returns the size(number of items) of the list by calling the list object's own length method. It takes in a list object as an argument and doesn't have a side effect on the list.

Syntax :

```
len(s)
```

Where s can be either a sequence or collection.

Example : Write a function that computes and returns the size/length of a list.

```
l1 = []                    # defined an empty list
l2 = [5,43,6,1]            # define a list of 4 elements
l3 = [[4,3],[0,1],[3]]     # define a list of 3 elements(lists
print("L1 len: ", len(l1))
print("L2 len: ", len(l2))
print("L3 len: ", len(l3))
```

Output :

```
L1 len:  0
L2 len:  4
L3 len:  3
```

2) list()

list() is actually a Python built-in class that creates a list out of an iterable passed as an argument. As it will be used a lot throughout this tutorial, we will take a quick look at what this class offers.

Syntax :

```
list([iterable])
```

The bracket tells us that the argument passed to it is optional.

The **list()** function is mostly used to:

- Convert other sequences or iterables to a list.
- Create an empty list – In this case, no argument is given to the function.

Example 2 : Convert tuple, dict to list, and create an empty list.

```
t = (4,3,5,0,1)        # define a tuple
s = 'hello world!'   # define a string
d = {'name':"Jitendra","age":40,"gender":"Male"}  # define a dict

# convert all sequences to list
t_list, s_list, d_list = list(t), list(s), list(d)
# create empty list
empty_list = list()

print("tuple_to_list: ", t_list)
print("string_to_list: ", s_list)
print("dict_to_list: ", d_list)
print("empty_list: ", empty_list)
```

Output :

```
tuple_to_list:  [4, 3, 5, 0, 1]
string_to_list:  ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']
dict_to_list:  ['name', 'age', 'gender']
empty_list:  []
```

Note : Converting a dictionary using list(dict) will extract all its keys and create a list. That is why we have the output ['name','age','gender'] above. If we want to create a list of a dictionary's values instead, we'll have to access the values with dict.values().

3) range()

The Python list function **range()** takes in some integers as arguments and generates a list of integers.

Syntax :

```
range([start,]stop[,step])
```

Where :

- **start**: Specifies where to start generating integers for the list.
- **stop**: Specifies where to stop generating integers for the list.
- **step**: Specifies the incrementation.

From the syntax above, start and step are both optional and they default to 0 and 1 respectively.

Example 3 : Create a sequence of numbers from 4 to 20, but increment by 2 and print it.

```
start = 4  # define our start number
end = 20   # define out end number
step = 2   # define out step number

print("Range of numbers:")

r = range(start, end, step)
# print items in the range object.
for item in r:
    print(item)
```

Output :

```
Range of numbers:
4
6
8
10
12
14
16
18
```

4) sum()

The Python **sum()** function adds all items in an iterable and returns the result.

Syntax :

```
sum(iterable[,start])
```

Where :

- The **iterable** contains items to be added from left to right.
- **start** is a number that will be added to the returned value.

The **iterable's** items and **start** should be numbers. If start is not defined, it defaults to zero(0).

Example 4 : Sum items from a list

```
>>> sum([9,3,2,5,1,-9])
11
```

Example 5 : Start with 9 and add all items from the list [9,3,2,5,1,-9].

```
>>>sum([9,3,2,5,1,-9], 9)
20
```

5) min()

The Python **min()** function returns the smallest item in a sequence.

Syntax :

```
min(iterable[,key, default])
```

Where :

- **iterable** here will be a list of items.
- **key** here specifies a function of one argument that is used to extract a comparison key from each list element.
- **default** here specifies a value that will be returned if the iterable is empty.

Example 6 : Find the smallest number in the list [4,3,9,10,33,90].

```
>>> numbers = [4,3,9,10,33,90]
>>> min(numbers)
3
```

6) max()

The Python **max()** function returns the highest item in a sequence.

Syntax :

```
max(iterable[,key, default])
```

Where :

- **iterable** here will be a list of items.
- **key** here specifies a function of one argument that is used to extract a comparison key from each list element.
- **default** here specifies a value that will be returned if the iterable is empty.

Example 8 : Find the largest number in the list [4,3,9,10,33,90].

```
>>> numbers = [4,3,9,10,33,90]
>>> max(numbers)
90
```

7) sorted()

The Python **sorted()** method returns a new sorted list of items from an iterable.

Syntax :

```
sorted(iterable[,key,reverse])
```

Where :

- **iterable** here will be a list of items.
- **key** here specifies a function of one argument that is used to extract a comparison key from each list element.
- **reverse** is a bool that specifies if the sorting should be done in ascending(False) or descending(True) order. It defaults to False.

Example 9 : Sort the list [4,3,10,6,21,9,23] in descending order.

```
>>> numbers = [4,3,10,6,21,9,23]
>>>sorted(numbers, reverse=True)
[23, 21, 10, 9, 6, 4, 3]
```

8) reversed()

The Python **reversed()** function returns a reverse iterator in which we can request the next value or iterate through until we hit the end.

Syntax :

```
reversed(iterator)
```

Example 11 : Find the reverse order of the list.

```
>>> numbers = [4,3,10,6,21,-9,23]
>>> list(reversed(numbers))
[23, -9, 21, 6, 10, 3, 4]
```

Note :

We should note the following

- As **reversed()** returns a generator expression, we can use **list()** to create the list of items.
- The Python **reversed()** function is similar to the list method **reverse()**. However, the latter reverses the list in-place.
- Using slicing(a[::-1]), we can reverse a list similar to the **reversed()** function.

9) enumerate()

The Python **enumerate()** function returns an enumerate object in which we can request the next value or iterate through until we hit the end.

Syntax :

```
enumerate(sequence, start=0)
```

Each next item of the returned object is a tuple (count, item) where the count starts from 0 as default, and the item is gotten from iterating through the iterator.

Example 12 : Enumerate the list of names ["eyong","kevin","enow","ayamba","derick"] with the count starting from 3 and returns a list of tuples such as (count, item).

```
>>> names = ["eyong","kevin","enow","ayamba","derick"]
>>>list(enumerate(names, 3))
[(3, 'eyong'), (4, 'kevin'), (5, 'enow'), (6, 'ayamba'), (7, 'derick')]
```

10) zip()

The Python **zip()** function returns an iterator that contains an aggregate of each item of the iterables.

Syntax :

```
zip(*iterables)
```

Where the * indicates that the **zip()** function can take any number of iterables.

Example 13 : Add the i-th item of each list.

```
l1 = [4,6,1,9]
l2 = [9.0,2.7]
result = [] # define an empty list to hold the result

# aggregate each item of the lists
# for each iteration, item1 and item2 comes from l1 and l2 respectively
for item1, item2 in zip(l1, l2):
result.append(item1 + item2) # add and append.
print("RESULT: ", result)
```

Output :

```
RESULT:  [13, 6, 3, 16]
```

Note: *It* is important to note that this resulting iterator stops when the shortest iterable argument is exhausted.

Example :

```
l1 = [3,4,7]
l2 = [0,1]
result = [] # define an empty list to hold the result

# aggregate each item of the lists
# for each iteration, item1 and item2 comes from l1 and l2 respectively
```

```
for item1, item2 in zip(l1, l2):
result.append(item1 + item2) # add and append.
print("RESULT: ", result)
```

Output :

```
RESULT:  [3, 5]
```

The result above didn't include 7 from l1. This is because l2 is 1 item shorter than l2.

11) map()

The Python **map()** function maps a function to each item of iterables and returns an iterator.

Syntax :

```
map(function, iterable,...)
```

This function is mostly used when we want to apply a function on each item of iterables but we don't want to use the traditional **for loop**.

Example 14 : Add **2** to each item of the list

```
l = [6,4,8,9,2,3,6]
result = [] # create empty list to hold result
# iterate over the list
for item in l:
result.append(item+2) # add 2 and append
print("MAP: ", result)
```

Output :

```
MAP:  [8, 6, 10, 11, 4, 5, 8]
```

Note: The **map()** function can take any number of iterables given that the function argument has an equivalent number of arguments to handle each item from each iterable. Like **zip()**, the iterator stops when the shortest iterable argument is exhausted.

12) filter()

The Python **filter()** method constructs an iterator from the items of iterables that satisfy a certain condition

Syntax :

```
filter(function, iterable)
```

The function argument sets the condition that needs to be satisfied by the items of the iterable. Items that do not satisfy the condition are removed.

Example 15 : Filter out the names with length smaller than 4 from the list

```
["john","petter","job","paul","mat"].
```

Example :

```
names = ["john","petter","job","paul","mat"]
result=list(filter(lambda name: len(name) >=4, names))
print("MAP: ", result)
```

Output :

```
MAP: ['john', 'petter', 'paul']
```

13) iter()

The Python **iter()** function converts an iterable into an iterator in which we can request the next value or iterate through until we hit the end.

Syntax :

```
iter(object[,sentinel])
```

Where :

- **object** can be represented differently based on the presence of **sentinel**. It should be an iterable or sequence if a sentinel is not provided or a callable object otherwise.
- **sentinel** specifies a value that will determine the end of the sequence.

Example 16 : Convert the list ['a','b','c','d','e'] into an iterator and use **next()** to print each value.

```
l1 = ['a','b','c','d','e'] # create our list of letters
iter_list = iter(l1) # convert list to iterator
print(next(iter_list))    # access the next item
print(next(iter_list))
print(next(iter_list))
print(next(iter_list))
```

Output :

```
a
b
c
d
```

14) all()

The Python **all()** function returns True if all the elements of an iterable are true, or if the iterable is empty.

Syntax

```
all(iterable)
```

Note :

- In Python, **False**; **empty list**([]), **strings**(''), **dict**({}); **zero**(0), **None**, etc are all false.
- Since the Python **all()** function takes in an iterable argument, if an empty list is passed as an argument, then it will return True. However, if a list of an empty list is passed, then it will return False.

Example 18 : Check if all items of a list are true.

```
l = [3,'hello',0, -2]  # note that a negative number is not false
print(all(l))
```

Output :

```
False
```

In the example above, the result is False as element 0 in the list is not true.

15) any()

The Python **any()** function returns True if at least one item of the iterable is true. Unlike **all()**, it will return False if the iterable is empty.

Syntax :

```
any(iterable)
```

Example 19 : Check if at least one item of the list ['hi',[4,9],-4,True] is true.

```
l1 = [3,'hello',0, -2]
l2 = ['',[],{},False,0,None] # all is false
print(any(l1))
print(any(l2))
```

Output :

```
True
False
```

Nested and Copying Lists

Nested list is a list having another list inside it.

Create a Nested List

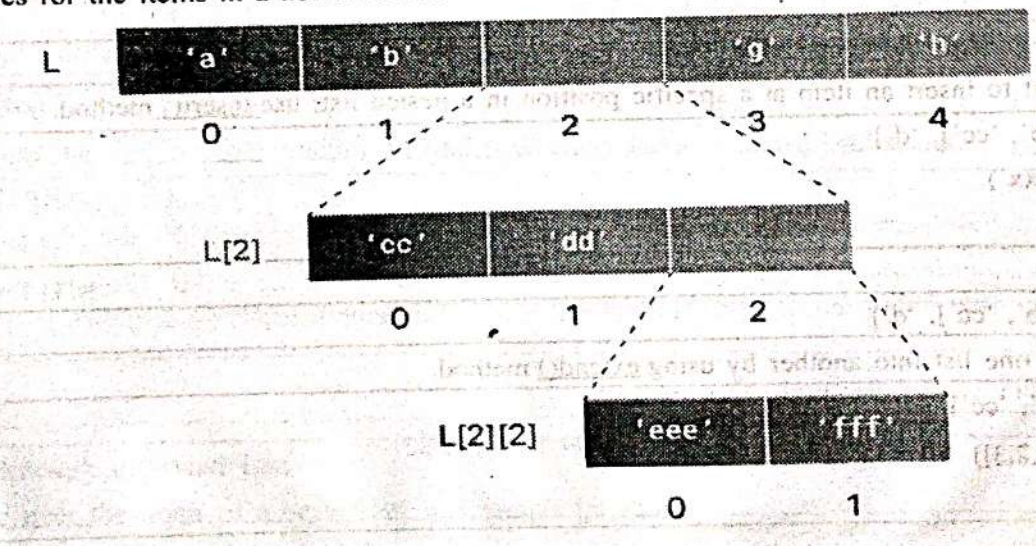
A nested list is created by placing a comma-separated sequence of sublists.

```
L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']
```

Access Nested List Items by Index

You can access individual items in a nested list using multiple indexes.

The indexes for the items in a nested list are illustrated as below:



```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']

print(L[2])
```

Output :

```
['cc', 'dd', ['eee', 'fff']]

print(L[2][2])
```

Output :

```
['eee', 'fff']

print(L[2][2][0])
```

Output :

```
eee
```

Change Nested List Item Value

You can change the value of a specific item in a nested list by referring to its index number.

```
L = ['a', ['bb', 'cc'], 'd']
L[1][1] = 0
print(L)
```

Output :

```
['a', ['bb', 0], 'd']
```

Add items to a Nested list

To add new values to the end of the nested list, use append() method.

```
L = ['a', ['bb', 'cc'], 'd']
L[1].append('xx')
print(L)
```

Output :

```
['a', ['bb', 'cc', 'xx'], 'd']
```

When you want to insert an item at a specific position in a nested list, use insert() method.

```
L = ['a', ['bb', 'cc'], 'd']
L[1].insert(0,'xx')
print(L)
```

Output :

```
['a', ['xx', 'bb', 'cc'], 'd']
```

You can merge one list into another by using extend() method.

```
L = ['a', ['bb', 'cc'], 'd']
L[1].extend([1,2,3])
print(L)
```

Output :

```
['a', ['bb', 'cc', 1, 2, 3], 'd']
```

Remove items from a Nested List

If you know the index of the item you want, you can use pop() method. It modifies the list and returns the removed item.

```
L = ['a', ['bb', 'cc', 'dd'], 'e']
x = L[1].pop(1)
print(L)
# removed item
print(x)
```

Output :

```
['a', ['bb', 'dd'], 'e']
cc
```

If you don't need the removed value, use the del statement.

```
L = ['a', ['bb', 'cc', 'dd'], 'e']
del L[1][1]
print(L)
```

Output :

```
['a', ['bb', 'dd'], 'e']
```

If you're not sure where the item is in the list, use remove() method to delete it by value.

```
L = ['a', ['bb', 'cc', 'dd'], 'e']
L[1].remove('cc')
print(L)
```

Output :

```
['a', ['bb', 'dd'], 'e']
```

Find Nested List Length

You can use the built-in len() function to find how many items a nested sublist has.

```
L = ['a', ['bb', 'cc'], 'd']
print(len(L))
print(len(L[1]))
```

Output :

```
3
2
```

Iterate through a Nested List

To iterate over the items of a nested list, use simple for loop.

```
L = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
for list in L:
    for number in list:
        print(number, end=' ')
```

Output :

```
1 2 3 4 5 6 7 8 9
```

Various other ways to Create a Nested Listsin Python

There are many approaches to create a list of lists. Here, we are using the **append()** method and **list comprehension** technique to create a list of lists.

Create a list of lists using the append() method in Python

In this example, we are using an **append()** method that used to append a list into a list as an element. we created two lists and append them into another list using the **append()** method and print the list which is actually a list of lists.

```
# Take two lists
list1 = [1,2,3,4]
list2 = [5,6,7,8]
list3 = [] # Take an empty list
# make list of lists
list3.append(list1)
list3.append(list2)
print(list3)
```

Output :

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

Create a list of lists using the list initializer in Python

The list initializer syntax is used to create a list in Python. We can use this technique to create a list of lists by passing lists as elements into the list initializer. See the code and output.

```
# Take two lists
list1 = [1,2,3,4]
list2 = [5,6,7,8]
# make list of lists
list3 = [list1, list2]
# Display result
print(list3)
```

Output :

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

Create a list of lists using for-loop in Python

We can use for loop to create a list of lists in Python. We used the **append()** method inside the loop to add the element into the list to form a list of lists. See the code and output.

```
lists = []
# make list of lists
for i in range(2):
# append list
lists.append([])
for j in range(5):
lists[i].append(j)
# Display result
print(lists)
```

Output :

```
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

Create a list of lists using list comprehension in Python

If you are comfortable with list comprehension then use it to make a list of lists as we did in the below code example. See the code and output here.

```
# Take a list
list = ['Apple','Mango','Orange']
lists = []
# make list of lists
lists = [[val] for val in list]
# Display result
print(lists)
```

Output :

```
[['Apple'], ['Mango'], ['Orange']]
```

How to access elements from a list of lists in Python

We can access elements by using an index. The list index starts from 0 and end to n-1 where n is the length of the list. Here, we used **0** index to get the first element of the list.

```
# Take a list
list = ['Apple','Mango','Orange']
lists = []
# make list of lists
lists = [[val] for val in list]
# Display result
```

```
print(lists)
# Access Element
print(lists[0])
print(lists[2])
```

Output :

```
[['Apple'], ['Mango'], ['Orange']]
['Apple']
['Orange']
```

List as Arguments to Function

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example :

```
def my_function(food):
  for x in food:
    print(x)

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

my_function(fruits)
```

Output :

```
apple
banana
cherry
```

Solved Questions

Unit
5

Strings and Lists

Lists are part of core Python. Lists have a variety of uses. Like arrays, they are sometimes used to store data. However, lists do not have the specialized properties and tools that make arrays so powerful for scientific computing. So in general, we prefer arrays to lists for working with scientific data. For other tasks, lists work just fine and can even be preferable to arrays.

Strings are lists of keyboard characters as well as other characters not on your keyboard.

Dictionaries are like lists, but the elements of dictionaries are accessed in a different way than for lists. The elements of lists and arrays are numbered consecutively, and to access an element of a list or an array, you simply refer to the number corresponding to its position in the sequence. The elements of dictionaries are accessed by "keys", which can be either strings or (arbitrary) integers (in no particular order). Dictionaries are an important part of core Python.

5.1 Introduction to Strings

A string is a sequence of characters. It can be declared in python by using double-quotes. Strings are immutable, i.e., they cannot be changed.

A string that contains no characters, often referred to as the **empty string**, is still considered to be a string. It is simply a sequence of zero characters and is represented by '' or "" (two single or two double quotes with nothing in between).

A string in python is an ordered sequence of characters. The point to be noted here is that a list is an ordered sequence of object types and a string is an ordered sequence of characters. This is the main difference between the two.

Strings are created by enclosing a sequence of characters within a pair of single or double quotes. Examples of strings include "Marylyn", "omg", "good_bad".