List:

A list is similar to an array that consists of a group of elements or items. But, there is one major difference between an array and a list. An array can store only one type of elements whereas a list can store different types of elements. Hence, lists are more versatile and useful than an array.

To store student information of different types we can creat a list.

```
student = [10, 'Karthik', 'M', 50, 60, 70, 80, 90]
```

We can also create an empty list without any elements by simply writing empty square braces as

```
S = []    # this is a empty list
```

The elements in the list should be separated by a comma (,). To view the elements of a list as a whole, we can simply pass the list name to the print() function as

```
print(student)
```

The list appears as given below:

```
[10, 'Karthik', 'M', 50, 60, 70, 80, 90]
```

Indexing and slicing operations are commonly done on lists. Indexing represents accessing elements by their position numbers in the list.

The position numbers start from 0 onwards and are written inside square braces as student[0], student[1]... It means student[0] represents 0th element and soon. For ex, to print the student name, we can write:

```
print(student[1])
```

The name of the student appears as shown below:

```
Karthik
```

Slicing represents extracting a piece of the list by mentioning starting and ending position numbers. The general format of slicing is [start:stop:stepsize]. By default start will be 0, stop will be the last element and stepsize will be 1. For ex, student[0:3:1] represents a piece of list containing 0th and 2nd elements.

```
print(student[0:3:1])
```

The element are given below:

```
[10, 'Karthik', 'M']
```

we can also write the above statement as:

```
print(student[:3:])
```

The same elements appears as follows:

```
[10, 'Karthik', 'M']
```

Hence, since we did not mention the starting position, it will start at 0 and stepsize will be taken as 1. Suppose we do not mention anything in slicing, then the total list will be as:

Print (student[::])

It displays the output as:

[10, 'Karthik', 'M', 50, 60, 70, 80, 90].

Creating list using range() function:

We can use range() to generate a sequence of integers which can be stored in the list.

range (start, stop, stepsize)

If we do not mention the 'start', it it assumed as 0 and the stepsize is taken as 1. The range of numbers stops one element prior to 'stop'. For ex,

range (0, 10, 1)

This will generate numbers from 0 to 9 as: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Consider another ex:

range (4, 9, 2)

The preceding statement will generate numbers from 4th to 8th in steps of 2 i.e [4, 6, 8].

In fact, the range() does not return list of numbers. It returns only range class object that stores the data about start, stop and stepsize. For ex, if we write

```
Print (range (4, 9, 2))
```

The Preceding statement will display:

```
range (4, 9, 2)  # this is obj given by range().
```

This range object should be used in for loop to get the range of numbers desired by the programmer. For ex,

```
for i in range (4, 9, 2):
    Print (i)
```

The above statement will display 4 6 8. Hence, the range() can be used in for loop to display the numbers, or with list() function to creat a list. In the following, ex the range() function is used in the list() function to create a list:

```
lst = list (range (4, 9, 2))
Print (lst)
```

The list is given below:

```
[4, 6, 8]
```

If we are not using the list() function and using range() alone to concat a list,

```
lst = range(4, 9, 2)
print(lst)
```

The preceding statements will give the following o/p:

```
range(4, 9, 2)  # this is not a list, it is obj.
```

In this case, using a loop like for or while we can view the elements of the list. For ex,

```
for i in lst:
    print(i)
```

The above statement will display 4 6 8.

## Updating the Elements of a List:

Lists are mutable. It means we can modify the contents of a list. we can append, update or delete the elements of a list depending upon our requirements.

Appending an element means adding an element at the end of the list. To append a new element to the list, we should use the append() method. For ex,

```
lst = list (range(1,5))
print (lst)
```

The preceding statements will give the following output:

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

Now, consider the following statements:

```
lst.append(9)
print(lst)
```

The preceding statements will give the following o/p:

```
[1,2,3,4,9]
```

Update an element means changing the value of the element in the list. This can be done by accessing the specific element using indexing.

```
lst[1] = 8   # update 1st element of lst
print (lst)
```

The preceding statement will give:

[1, 8, 3, 4, 9]

Consider the following statements:

```
lst[1:3] = 10, 11  # update 1st and 2nd elements of lst
print(lst)
```

The preceding statements will give:

[1, 10, 11, 4, 9]

Deleting an element from the list can be done using 'del' statement. The del statement takes the position no. of the element to be deleted.

```
del lst[1]  # delete 1st element from lst
print(lst)
```

[1, 11, 4, 9]

We can also delete an element using remove() method. In this method, we should pass the element to be deleted.

```
lst.remove(11)  # delete 11 from list.
print(lst)
```

Now, the list appears like:

[1, 4, 9]

The elements of a list can be reverse using reverse() method, as:

```
list.reverse()
```

This will reverse the order of elements in the list and the reversed elements are available in the list.

Concatination of Two Lists:

We can use simply use '+' operator on two lists to join them. For ex,

```
x = [10, 20, 30, 40, 50]
y = [100, 200, 300]
Print (x+y)
```

The concatenated list appears like:

```
[10, 20, 30, 40, 50, 100, 200, 300]
```

Repetition of Lists:

We can repeat the elements of a list 'n' number of times by using '*' operator.

```
Print (x * 2)  # repeat the list x for 2 times
```

Now, the list appears as:

```
[10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
```

## Membership in List:

We can check if an element is a member of a list or not by using 'in' or 'not in' operator.

```
x = [10, 20, 30, 40, 50]
a = 20
Print (a in x)  # check if a is member of x
```

The Preceding statement will give:

True

If you write,

```
Print (a not in x)  # check if a is not a member of x.
```

Then, it will give

False.

## Aliasing and Cloning List:

Giving a new name to an existing list is called 'aliasing'. The new name is called 'alias name'. For ex,

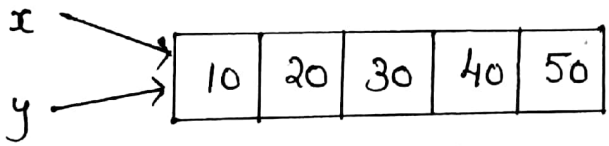
```
x = [10, 20, 30, 40, 50]
```

To Provide a new name to this list, we can simply use assignment operator as:

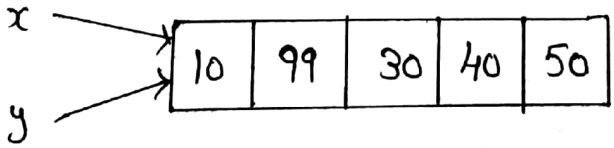
```
y = x
```

In this case, we are having only one list of element but with two different names 'x' and 'y'. Here, 'x' is the original name and 'y' is the alias name for the same list. Hence, any modifications done to 'x' will also modify 'y' and vice versa.

```
x = [10, 20, 30, 40, 50]
y = x        # x is aliased as y
Print(x)     will display [10, 20, 30, 40, 50]
Print(y)     will display [10, 20, 30, 40, 50]

x[1] = 99    # modify 1st element in x
Print(x)     will display [10, 99, 30, 40, 50]
Print(y)     will display [10, 99, 30, 40, 50]
```




Before Modification




After Modification

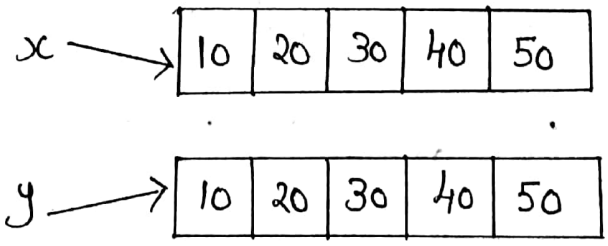
Hence, if the programmer wants two independent lists, he should not go for aliasing. On the other hand, he should use cloning or copying.

obtaining exact copy of a list is called cloning. To clone a list, we can take help of the slicing operation as:

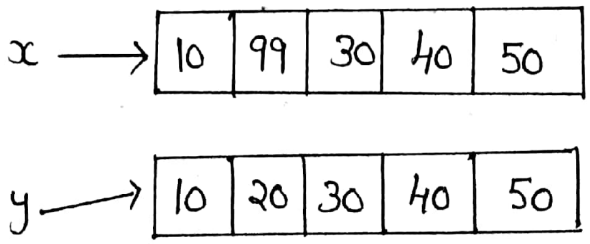
```
y = x[:]  # x is cloned as y
```

when we clone a list like this, a separate copy of all the elements is stored into 'y'. The lists 'x' and 'y' are independent lists.

```
x = [10, 20, 30, 40, 50]
y = x[:]
Print (x)   # [10, 20, 30, 40, 50]
Print (y)   # [10, 20, 30, 40, 50]
x[1] = 99
Print (x)   # [10, 99, 30, 40, 50]
Print (y)   # [10, 20, 30, 40, 50]
```




Before Modification




After Modification.

## Methods to Process List:

The function len() is useful to find the no. of elements in a list. we can use this function as:

```
n = len(list)
```

| Method | Example | Description |
|---|---|---|
| index() | list.index(x) | Returns the first occurrence of x in the list. |
| append() | list.append(x) | Appends x at the end of the list. |
| insert() | list.insert(i, x) | Insert x into the list in the position specified by i. |
| copy() | list.copy() | copies all the list ele into a new list and returns it. |
| extend() | list.extend(list1) | Appends list1 to list. |
| count() | list.count(x) | Returns no. of occurences of x in list. |
| remove() | list.remove(x) | Removes x from the list. |
| pop() | list.pop() | Removes the ending ele. from list. |
| sort() | list.sort() | Sort the elements of list in accending order. |
| reverse() | list.reverse() | Reverses the seq. of elements. |
| clear() | list.clear() | Deletes all the elements from list. |

Ex: A python program to understand list Processing Methods.

Sol:

```
num = [10, 20, 30, 40, 50]
n = len(num)
Print ("No. of elements in list", n)
num.append(60)
Print ("num after appending 60:", num)
num.insert(0, 5)
Print ("list after inserting 5 at 0th Position:", num)
num1 = num.copy()
Print ("newly created list:", num1)
num.extend(num1)
Print ("list1 after appending", list2:", num)
n = num.count(50)
Print ("No. of times 50 found in list1", n)
num.remove(50)
Print ("No. o List after removing 50:", num)
num.POP()
Print ("list after removing ending element", num)
num.sort()
Print ("list after Sorting", num)
```

```
num.reverse()
print("list after reversing:", num)
num.clear()
print("list after removing all elements:", num)
```

O/P: No. of element in list : 5

list after appending 60 : [10, 20, 30, 40, 50, 60]

list after inserting 5 at 0th pos : [5, 10, 20, 30, 40, 50, 60]

Newly created list : [5, 10, 20, 30, 40, 50, 60]

list1 after appending list2: [5, 10, 20, 30, 40, 50, 60, 5, 10, 20, 30, 40, 50, 60]

No. of times 50 found in the list1 : 2

Nist after removing 50: [5, 10, 20, 30, 40, 60, 5, 10, 20, 30, 40, 50, 60]

list after removing ending element : [5, 10, 20, 30, 40, 60, 5, 10, 20, 30, 40, 50]

Nist after Sorting : [5, 5, 10, 10, 20, 20, 30, 30, 40, 40, 50, 60]

list after Reversing : [60, 50, 40, 40, 30, 30, 20, 20, 10, 10, 5, 5]

list after removing all element : [ ]

## Finding Biggest and Smallest Element in a list:

Python Provides min() and max() functions, which return the biggest and smallest elements from a list.

```
n1 = max(x)   # n1 gives the biggest element
n2 = min(x)   # n2 gives the smallest element.
```

## Sorting the List Elements:

Python Provides the sort() method to sort the elements of a list.

```
x.sort()
```

This will sort the list 'x' into ascending order. If we want to sort the elements of the list into descending order, then we can mention 'reverse = True' into sort() method as:

```
x.sort(reverse = True)
```

This will sort the list 'x' into descending order.

## Number of occurences of an Element in the list:

Python Provides count() method that returns the no. of times a particular element is repeated in the list. For ex,

```
n = x.count(y)
```

will return the no. of times 'y' is found in the list 'x'. This method returns 0 if the element is not found in the list.

## Finding Common Elements in Two Lists:

Let's take the two groups of students as two lists. First of all, we should convert the lists into sets, using set() function, as: set(list). Then we should find the common elements in the two sets using intersection() method as:

```
set1.intersection(set2)
```

This method returns a set that contains common or repeated elements in the two sets.

Ex: A Python program to find common elements in two lists.

Sol:

```
list1 = ['Vinay', 'Krishna', 'Karthik']
list2 = ['Govind', 'Kishore', 'Vinay', 'Karthik']
# Convert them into sets
s1 = set(list1)
s2 = set(list2)
# Find intersection of two sets
s3 = s1.intersection(s2)
# Convert the resultant set into list.
list3 = list(s3)
Print(list3)
```

o/p: ['Vinay', 'Karthik']

Nested Lists:

A list within another list is called a nested list. For ex, we have two lists 'a' and 'b' as:

```
a = [80,90]
b = [10,20,30,a]
```

observe that the list 'a' is inserted as an element in the list 'b' and hence 'a' is called a nested list.

```
Print(b)
```

The elements of b appears as:

```
[10,20,30,[80,90]]
```

The last element [80,90] represents a nested list. So, 'b' has 4 elements and they are:

```
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = [80,90]
```

So, b[3] represents the nested list and if we want to display its elements separately, we can use a for loop as:

```
for x in b[3]:
    Print(x)
```

## List Comprehensions:

It represent creation of new list from an iterable object (like a list, set, tuple, dictionary or range) that satisfy a given condition. List comprehensions contain very compact code usually a single statement that performs the task.

Ex: A program to create a list with squares of integers from 1 to 100.

Sol:

```
squares = [ ]
for x in range(1, 11):
    squares.append(x**2)
```

The preceding code will create 'squares' list with the elements as shown below:

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

The previous code can be rewritten in a compact way as:

```
squares = [x**2 for x in range(1,11)]
```

This is called list comprehension. From this, we can understand that a list comprehension consists of square brackets containing an expression (i.e. x**2). After the expression, a for loop and then zero or more if statements can be written. Consider the following syntax:

[exp for item1 in iterable1 if statement1
for item2 in iterable2 if statement2 ...]

Here, 'iterable' represents a list, set, tuple, dictionary or range object. The result of a list comprehension is a new list that contains elements as a result of executing the expression according to the for loops and if statements.

Ex: Suppose, we want to get squares of integers from 1 to 10 and take only the even numbers from the result, we can write a list comprehension as:

```
even_sqr = [x**2 for x in range(1, 11) if x%2==0]
```

If we display the list 'even_sqr', it will display as:

```
[4, 16, 36, 64, 100]
```

The same list comprehension can be written without using the if statement as:

```
even_sqr = [x**2 for x in range(2, 11, 2)]
```

- list are not sequencial memory

- l = [ ]

  l.append(3)

- l = [None] * 6

  [N, N, N, N, N, N]

  l[0] = 8

  l[2] = 6

  [8, N, 6, N, N, N]

  l.append(6)

  [8, N, 6, N, N, 8]

dynamic : l = [input() for i in range(6)]

for i in range(n)

l.append(int(input()) // l.append(input())

generic form

other way: (dynamically)

```
l = [[],[],[]]
for i in range(3):
    for j in range(3):
        l[i].append(int(input()))
```

```
> l = [[],[],[]]
> l[0].append(5)
> l
> [[5],[],[]]
> l[0].append(3)
> l
> [[5,3],[],[]]
> l[0][1]
> 3
> l[1].append(8)
> l
> [[5,3],[8],[]]
> l[1].append(9)
> l
> [[5,3],[8,9],[]]
> ~~l=[[None for j in 3] for i in 3]~~
> l = [[None for j in range(3)] for i in range(3)]
> l
> ~~l[0][0]~~ [[N,N,N],[N,N,N],[N,N,N]]
> l[0][0] = 3
> l
```

we can use ⓪

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

# Here we can use column for referencing

| | | |
|---|---|---|
| 98845 | SVEC_14121A1549 | KONGARA BHAVANA |
| 98847 | SVEC_14121A1551 | KOTTE MANIKANTA |
| 98848 | SVEC_14121A1552 | KRISHNAM HARIKA |
| 98849 | SVEC_14121A1553 | KRISHNAM SRAVANI |
| 98850 | SVEC_14121A1554 | KUMMARI VINAY KUMAR |
| 98851 | SVEC_14121A1555 | M MAHESH KUMAR |
| 98853 | SVEC_14121A1557 | MADANAPALLI MALLESH |
| 98855 | SVEC_14121A1559 | MADIRAJU ANUSHA |
| 98856 | SVEC_14121A1560 | MAJJARI TEJASWI |

```
> [[3, N, N], [N, N, N], [N, N, N]]
> lc = [[int(input()) for j in range(3)] for i in range(3)]
> 1
  2
  3
  .
  9
> l
> [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
> l = []
> x = []
> x.append(3)
> x.append(4)
> l.append(x)
> l
> [[3, 4]]
> x = []
> x.append(6)
> x.append(7)
> l
> [[3, 4] [6, 7]]
```

## Tuples:

A tuple is a python sequence which stores a group of elements or items. Tuples are similar to lists but the main difference is tuples are immutables whereas lists are mutable. Since tuples are immutable, once we create a tuple we cannot modify its elements. Hence we cannot perform operations like append(), insert(), remove(), pop() and clear() on tuples. Tuples are generally used to store data which should not be modified and retrieve that data on demand.

## Creating Tuples:

We can create a tuple by writing elements separated by commas inside parentheses(). The elements can be of same datatype or different types. For ex, to create an empty tuple, we can simply write empty parantheis, as:

```
tup1 = ()  # empty tuple
```

If we want to create a tuple with only one element, we can mention that element in parantheis and after that comma is needed, as

```
tup2 = (10,)
```

Here is a tuple with different types of elements:

```
tup3 = (10, 20, 30.5, "India")
```

we can create a tuple with only one of type of elements also, like the following:

```
tup4 = (10, 20, 30)
```

If we donot mention any brackets and write the elements separating them by commas, then they are taken by default as tuple.

```
tup5 = 1, 2, 3, 4, 5
```

If we donot use braces it will become a tuple not a list, or any other data type.

It is also possible to create a tuple from a list.

```
list = [1, 2, 3]
tpl = tuple(list)
Print(tpl)
```

The tuple is shown below as:

```
(1, 2, 3)
```

Another way to create a tuple is by using range() that returns a sequence. To create a tuple 'tpl' that contains numbers from 4 to 8 in sequence 2, we can use the range() along with tuple() function as:

```
tpl = tuple(range(4,9,2))
Print(tpl)
```

o/p : (4, 6, 8)

## Accessing the Tuple Elements:

Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list.

```
tup = (50, 60, 70, 80, 90, 100)
```

Indexing represents the position number of the element in the tuple. Now, tup[0] represents 0th element, tup[1] represents the 1st element and so on.

```
Print (tup[0])

50
```

Now, if you write:

```
Print (tup[5])

100
```

Similarly, negative indexing is also possible. For ex, tup[-1] indicates the last ele and tup[-2] indicates the second last element and so on. Consider the following statement.

```
Print (tup[-1])
```

The Preceding statement will give:

```
100
```

Slicing represents extracting a piece or part of the tuple. Slicing is done in the format : [start : stop : stepsize]. Here 'start' represents the Position of the starting element and 'stop' represents the Position of the ending element and 'stepsize' indicates the

increamentation. If the tuple contains 'n' element, the default values will be 0 for 'start' and n-1 for 'stop' and 1 for 'stepsize'.

```
Print (tup[:3)
```

The elements of tuple appear:

(50, 60, 70, 80, 90, 100)

For ex, to extract the elements from 1st to 4th, we write,

```
Print (tup[1:4])
```

The element of tuple appears as:

(60, 70, 80)

Similarly, to extract every other element, i.e. alternate elements, we can write:

```
Print (tup[::2])
```

The following output appears:

(50, 70, 90)

Negative values can be given in the slicing. If the 'step size' is negative, the elements are extracted in reverse order.

```
Print (tup[::-2])
```

The elements appears in the reverse order:

(100, 80, 60)

Note: Here start and stops assume default value.

## Basic operations on Tuples:

The basic operations : finding length, concatenation, repetition, membership.

To find length of a tuple, we can use len() function. This function returns the no. of elements in the tuple.

```
student = (10, 'Karthik', 50, 60, 70, 80, 90)
len (student)
```

The preceding statement will give the following output.

```
7
```

We can concatenate or join two tuples and store the result in a new tuple.

```
fees = (25000.00,) * 4
print (fees)
```

It will give the following output:

```
(25000.0 , 25000.0, 25000.0, 25000.0)
```

Now, we can concatenate the 'student' and 'fees' tuples together to form a new tuple 'student1' as:

```
student1 = student + fees
print (student1)
```

It will give the following output

```
(10, 'Karthik', 50, 60, 70, 80, 90, 25000.0, 25000.0, 25000.0, 25000.0)
```

Searching whether an element is a member of the tuple or not can be done using 'in' and 'not in' operators.

```
name = 'Karthik'
name in student1
```

it returns, True

Suppose if you write:

```
name not in student1
```

it returns, False

The repetition operator repeat the tuple elements.

```
tp1 = (10, 11, 12)
tp11 = tp1 * 3
Print(tp11)
```

it returns, (10, 11, 12, 10, 11, 12, 10, 11, 12)

## Functions to Process Tuples:

| | Function | Example | Description |
|---|---|---|---|
| 1. | len() | len(tp1) | Returns the no of elements in tuple |
| 2. | min() | min(tp1) | Returns the smallest ele in tuple. |
| 3. | max() | max(tp1) | Returns the biggest ele in tuple. |
| 4. | Count() | tp1.count(x) | Returns how many times the ele 'x' is found in tp1. |
| 5. | index() | tp1.index(x) | Returns the first occurence of ele 'x' in tp1. Raises ValueError if x is not found. |

| | | |
|---|---|---|
| 6. sorted() | sorted(tpl) | Sorts the elements of the tuple into ascending order. sorted(tpl, reverse=True) will sort in reverse order. |

Dynamic Initialization of Tuples:

```
num = eval(input("Enter elements in():"))
```

The eval() function is useful to evaluate whether the typed elements are a list or a tuple depending upon the format of brackets given while typing the elements.

Ex: 1. A Python Program to accept elements in the form of a tuple and display their sum and average.

Sol:

```
num = eval(input("Enter elements in():"))
Sum = 0
n = len(num)  # no. of elements in the tuple
for i in range(n):
    Sum = Sum + num[i]
Print("Sum of numbers", Sum)
Print("Average of numbers", Sum/n).
```

O/P: Enter elements in(): (1, 2, 3, 4, 5)

Sum of numbers : 15

Average of numbers: 3

2. A python Program to find the first occurrence of an element in a tuple.

Sol:

```
str = input ('enter elements separated by commas:').split(',')
lst = [int(num) for num in str]
# convert strings into integers and store into a list.
tup = tuple (lst)  # convert list into tuple.
Print ('The tuple is:', tup)
ele = int (input ('enter an element to search'))
if (Pos = tup.index(ele)):
    Print ('Element Position no:', Pos+1)
else
    Print ('Element not found in tuple).
```

O/P: Enter the elements separated by commas: 10,20,30,20,40

The tuple is : (10,20,30,20,40)

Enter an element to search : 20

Element Position no: 2

## Sorting Nested Tuples:

To sort a tuple, we can use sorted() function. This function sorts by default into ascending order. For ex,

```
Print(sorted(emp))
```

will sort the tuple 'emp' in ascending order of the 0th element in the nested tuples, i.e identification number. If we want to sort the tuple based on employee name, which is the 1st element in the nested tuples, we can use a lambda expr as:

```
Print(sorted(emp, key = lambda x: x[1])) # Sort on name
```

Here, key indicates the key for the sorted() that tells on which element sorting should be done. The lambda function: lambda x: x[1] indicates that x[1] should be taken as the key that is nothing but 1st element. If we want to sort the tuple based on salary, we can use the lambda function as: lambda x: x[2]

Ex: 1. A Python Program to sort a tuple with nested tuples.

Sol:
```
emp = ((10, "Karthik", 20000.35), (20, "Kishore", 18000.75),
                                  (30, "Parvanthi", 32000.39))

Print(sorted(emp))
Print(sorted(emp, reverse = True))
Print(sorted(emp, key = lambda x: x[1]))
Print(sorted(emp, key = lambda x: x[2]))
```

## Nested Tuples:

A tuple inserted inside another tuple is called nested tuple. For ex,

tup = (50, 60, 70, 80, 90, (200, 201))

The nested tuple with the elements (200, 201) is treated as an element along with other elements in the tuple 'tup'. To retrieve the nested tuple, we can access it as an ordinary element as tup[5] as its index is 5.

Print ('Nested tuple = ', tup[5])

The preceding statement will give the following o/p.

Nested tuple = (200, 201)

Every nested tuple can represent a specific data record. For ex, to store 3 employee data in 'emp' tuple as:

emp = ((10, "Karthik", 20000.35), (20, "Kishore", 18000.75),

(30, "Pravanthi", 32000.39))

Here, 'emp' is the tuple name. It contains 3 nested tuples each of which represents the data of an employee.

Every employee's identification no, name and salary are stored as a nested tuples.

o/p: [(10, "Karthik", 20000.35), (20, "Kishore", 18000.75),
(30, "Pravanthi", 32000.39)]

[(30, "Pravanthi", 32000.39), (20, "Kishore", 18000.75),
(10, "Karthik", 20000.35)]

[(10, "Karthik", 20000.35), (20, "Kishore", 18000.75),
(30, "Pravanthi", 32000.39)]

[(20, "Kishore", 18000.75), (10, "Karthik", 20000.35)
(30, "Pravanthi", 32000.39)]

## Inserting Elements in a Tuple:

Since tuples are immutable, we cannot modify the elements of the tuple once it is created. Let's take 'x' as an existing tuple. Since 'x' cannot be modified, we have to create a new tuple 'y' with the newly inserted element. The following logic can be used:

1. First of all, copy the elements ~~from~~ 'x' of 'x' from 0th pos to Pos-2 position into 'y' as:

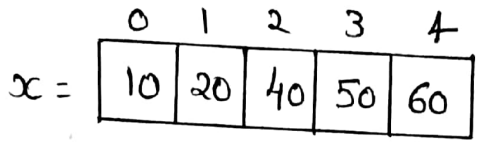
$$y = x[0:Pos-1]$$

2. Concatenate the new element to the new tuple 'y'.

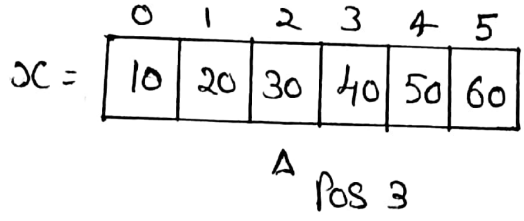
$$y = y + new$$

3 Concatenate the remaining elements (from pos-1 to till end) of x to the new tuple 'y'. The total tuple can be stored again with the old name 'x'.

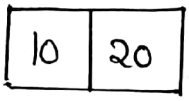
x = y + x[pos-1:]



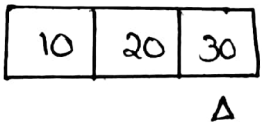
[insert at 3rd position]



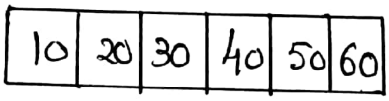
=> [Assumption]



copy x[0:2]



concatenate new element.



concatenate x[2:]

Ex: 1. A Pytho Program to insert a new element into a tuple of elements at a specified position.

Sol: name1 = ('Visnu', 'Anupama', 'Lakshmi', 'Bheeshma')

```
Print(name1)
lst = [input("Enter a new name")]
new = tuple(lst)
pos = int(input("Enter position no:"))
name11 = name1[0:pos-1]
```

```
name11 = name11 + neco
name1 = name11 + name1[pos-1:]
Print (name1)
```

O/P : ('vishnu','Anupama','Lakshmi', 'Bheethma')

Enter a new name : Ganesh

Enter a Position no: 2

('vishnu', 'Ganesh', 'Anupama','Lakshmi', 'Bheethma')

## Modifying Elements of a Tuple:

It is not possible to modify or update an element of a tuple since tuples are immutable. If we want to modify the element of a tuple, we have to create a new tuple with a new value in the position of the modified element. Let's take 'x' is the existing tuple and 'y' is the new tuple.

1 First of all, copy the elements of 'x' from 0th position to pos-2 position into 'y' as:

```
y = x[0:pos-1]
```

2. Concatenate the new element to the new tuple 'y'. Thus the new element is stored in the position of the element being modified.

```
y = y+new
```

3. Now concatenate the remaining elements from 'x' by eliminating the element which is at 'pos-1'. It means we should concatenate elements from 'pos' till the end. The total tuple can be assigned again to the old name 'x'.

```
x = y+x[pos:]
```

x =

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 | 60 |

△ modify element at 3rd position.

y =

| | |
|---|---|
| 10 | 20 |

copy x[0:2]

new = | 88 |

y = | 10 | 20 | 88 | concatenate y = y + new

y = | 10 | 20 | 88 | 40 | 50 | 60 | concatenate x[3:]

Ex: 1. A Python Program to modify or replace an existing element of a tuple with a new element.

Sol:
```
num = (10,20,30,40,50)
Print (num)
lst = [int (input ("Enter a new element"))]
new = tuple (lst)
Pos = int (input ("Enter Position no:"))
num1 = num [0:Pos-1]
num1 = num1 + new
num = num1 + num [Pos:]
Print (num)
```

O/P:
```
(10, 20, 30, 40, 50)
Enter a new element : 88
Enter Position no : 3
(10, 20, 88, 40, 50)
```

## Deleting Element from a Tuple:

The simplest way to delete an element from a particular position in the tuple is to copy all the elements into a new tuple except the element which is to be deleted. Let's assume that the user enters the position of the element to be deleted as 'pos'. The corresponding position will be 'pos-1' in the tuple as the elements start from 0th position. Now, the logic will be:

1. First of all, copy the elements from 'x' from 0th position to pos-2 position into 'y' as:

$$y = x[0 : pos-1]$$

2. Now concatenate the remaining elements from 'x' by eliminating the element which is at 'pos-1'. It means we should concatenate elements from 'pos' till the end. The total tuple can be assigned to the old name 'x'.

$$x = y + x[pos:]$$

Ex: 1. A program to delete an element from a particular position in the tuple.

Sol:
```
num = (10, 20, 30, 40, 50)
Print(num)
pos = int(input("Enter position no"))
num1 = num[0 : pos-1]
```

```
num = num1 + num[pos:]
Print(num)
```

o/p: (10, 20, 30, 40, 50)

Enter Position no: 3

(10, 20, 40, 50)

## Sets:

Sets is another data structure supported by Python. Basically sets are same as lists but with a difference that sets are lists with no duplicate entries. Technically, a set is a mutable and an unordered collection of items. This means that we can easily add or remove items for it.

## Creating a set:

A set is created by using built-in function set().

```
S = { }
type(S)
```

Here, the S is not set it is dictionary (dict).

```
S = set()
type(S)
```

Here, the S will be a set. That, means even though it is curly braces ({}) are using to create a set it will not be treated as set, by default it will be dictionary. If we want to create set we have to use set() function.

A set can have any number of items and they may be of different data types. It is also possible to convert list in to set by using set() function.

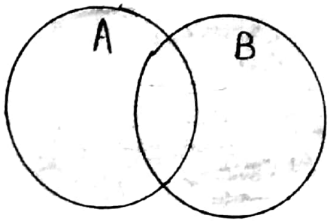
Ex: 1. Program to convert a list of value into a set.

Sol: S = Set([1, 2, "a", "b", "Hello", 4.56])

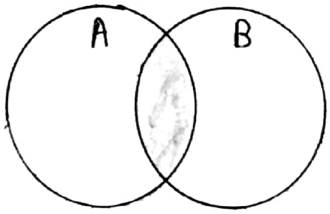
Print(S)

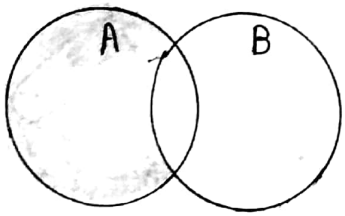
O/P: {1, 2, 'a', 'b', 'Hello', 4.56}

Union of two Seh includes all elements from both the Seh

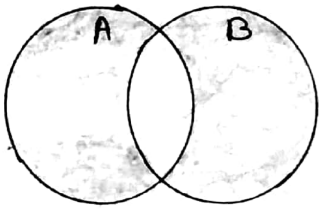
1. 


Intersection of two Seh includes that elements are common to both Seh.

2. 


3. 


Difference of A and B (A-B) includes all elements that are in A but not in B

4. 


Symmetric difference of A and B includes all elements that are in A and B but not the one that are common to A and B.

## Set Operations:

| S.No. | Operation | Description | Code & output |
|---|---|---|---|
| 1. | S.update(t) | Adds elements of set t in the set S provided that all duplicates are avoided. | S = set([1,2,3,4])<br>t = set([6,7,8])<br>S.update(t)<br>print(S)<br>o/p: {1,2,3,4,6,7,8} |
| 2. | S.add(x) | Adds element x to the set s provided that all duplicates are avoided. | S = set([1,2,3,4,5])<br>S.add(6)<br>print(S)<br>o/p: {1,2,3,4,5,6} |
| 3. | S.remove(x) | Removes element x from set S. Returns KeyError if x is not present. | S = set{[1,2,3,4,5]]<br>S.remove(3)<br>print(S)<br>o/p: {1,2,4,5} |
| 4. | S.discard(x) | Same as remove() but does not give an error if x is not present in set. | S = set([1,2,3,4,5])<br>S.discard(3)<br>print(S)<br>o/p: {1,2,4,5} |
| 5. | S.pop() | Removes last element in the set and raise KeyError if set is empty | S = set([1,2,3,4,5])<br>S.pop()<br>print(S)<br>o/p: {2,3,4,5} |

| S.No. | Operation | Description | Code & output |
|---|---|---|---|
| 6. | S.clear() | Removes all elements from the set. | S = set([1,2,3,4]) S.clear() Print(S) o/p : set() #empty set |
| 7. | len(S) | Returns the length of set | S = set([1,2,3,4,5]) Print (len(S)) o/p : 5 |
| 8. | x in S | Returns True if x is present in set s and False otherwise. | S = set([1,2,3,4,5]) Print (3 in S) o/p : True. |
| 9. | x not in S | Returns True, if x is not present in set s and False otherwise. | S = set([1,2,3,4,5]) Print (6 not in S) o/p : True |
| 10. | S.issubset(t) or S<=t | Returns True if every element in set s is present in set t and otherwise False. | S = set([1,2,3,4,5]) t = set([1,2,3,4,5,6]) Print(S<=t) o/p : True. |
| 11. | S.issuperset(t) or S>=t | Returns True if every element in t is present in set s and False otherwise. | S = set([1,2,3,4,5]) t = set([1,2,3,4,5,6]) Print(S>=t) o/p : False. |

| S.No | operation | Description | Code & output |
|---|---|---|---|
| 12. | S.union(t) or S\|t | Returns a set s that has elements from both sets s and t. | S = set([1,2,3])<br>t = set([4,5,6])<br>Print(S\|t)<br>o/p : {1,2,3,4,5,6} |
| 13. | S.intersection(t) or S&t | Returns a new set that has element which are common to both sets S&t | S = Set([1,2,3])<br>t = set([2,3,4])<br>Print(S&t)<br>o/p : {2,3} |
| 14. | S.difference(t) or S-t | Returns a new set that has elements in set s but not in t. | S = Set([1,2,5])<br>t = set([4,5,6])<br>Print(S-t)<br>o/p : {1,2} |
| 15. | S.symmetric_difference(t) or S^t | Returns a new set with elements either in s or in t but not both. | S = Set([1,2,3,4,5])<br>t = set([1,2,5,6,7])<br>Print(S^t)<br>o/p : {3,4,6,7} |
| 16. | S.copy() | Returns a copy of set S | S = set([1,2,3,4])<br>t = set([4,5,6])<br>Print(S.copy())<br>o/p : {1,2,3,4} |

| S No | operation | Description | Code & output |
|---|---|---|---|
| 17. | s.isdisjoint(t) | Returns True if two sets have null intersection | s=set([1,2,3])<br>t=set([4,5,6])<br>Print(s.isdisjoint(t))<br>o/p: True |
| 18. | all(s) | Returns True if all elements in the set are True & False otherwise | s=set([1,2,3])<br>t=set([4,5,6])<br>Print(all(s))<br>o/p: True |
| 19. | any(s) | Returns True if any of the element in the Set is True. Returns False if the set is empty | s=set([0,1,2,3,4])<br>Print(any(s))<br>o/p: True. |
| 20. | max(s) | Returns the maximum value in the set | s=set([1,2,3,4,5])<br>Print(max(s))<br>o/p: 5 |
| 21. | min(s) | Returns the minimum value in the set. | s=set([1,2,3,4])<br>Print(min(s))<br>o/p: 1 |
| 22. | Sum(s) | Returns the sum of element in the set. | s=set([1,2,3])<br>Print(sum(s))<br>o/p: 6 |

| S.No. | Operation | Description | Code & output |
|---|---|---|---|
| 23. | Sorted(S) | Returns a new sorted list from elements in the set. It does not sort the set as sets are immutable. | S = set([6,7,2,1,])<br>Print (sorted(S))<br>o/p : {1,2,6,7} |
| 24. | S==t and S!=t | S==t returns True if the two sets are equal. S!=t returns True if the two set are not equal. | S = set([1,2,3])<br>t = set([4,5,6])<br>Print (S==t)<br>Print (S!=t)<br>o/p : False<br>True |

Dictionaries: It represents a group of elements arranged in the form of Key-value pairs.

```
dict1 = {'Name': 'chandra', 'Id': 200, 'Salary': 9080.50}

dict = { }  # empty dictionary
```

Ex: # Create & retrieve dict with employee details.

```
dict1 = {'Name': 'chandra', 'Id': 200, 'Salary': 9080.50}
Print ("Name = ", dict1['Name'])   # no indexing & slicing
Print ("Id no", dict1['Id'])
Print ("salary", dict1['Salary'])
```

Operations on Dictionaries:

```
n = len (dict1)
Print (n)  # o/p: 3.
```

- We can modify the existing value as dict1['Salary'] = 10500.
- We can also insert a new key-value pair as

```
dict1['Dept'] = 'Finance'.
```

{'Name': 'chandra', 'Dept': 'Finance', 'Id': 200, 'Salary': 10500.0}

- Here 'Dept' is not added at the last it was added somewhere, so we conclude that dictionary is a unordered (with) collection of elements.

- To delete we use del dict['Id'], Here id and corresponding value is going to delete
- To test whether a key is available in dictionary or not, we use 'in' and 'not in'

'Dept' in dict → True.

- 'Gender' in dict → False.
- 'Gender' not in dict → True.
- key should be unique, it means, duplicate keys are not allowed. If we enter same key again, the old key will be overwritten & only the new key will be available.

Ex: emp = {'Nag': 10, 'vishnu': 20, 'Nag': 30}

print (emp).

O/P: {'Nag': 30, 'vishnu': 20}

x.get (name, -1)
{'Laxmi': 22, 'Raju': 10}

1) Dictionary Methods:

| | Method | ex | Def |
|---|---|---|---|
| 1. | clear() | d.clear() | Removes all key-value pairs from dict 'd' |
| 2. | copy() | d1 = d.copy() | Copies all elements from dict d to d1 |
| 3. | fromkeys() | d.fromkeys(s, v) | create a dict with keys from sequence like lists, tuple and values all set to 'v'. |
| 4. | get() | d.get (k, v) | Returns the values associated with key 'k'. if key is not found it returns 'v' value |

| | | | |
|---|---|---|---|
| 5. | items() | d.items() | Return key-value pairs. |
| 6. | keys() | d.keys() | Return a sequence of keys from dict d. |
| 7. | values() | d.values() | " " of values " |
| 8. | update() | d.update(x) | Add all ele's from dict 'x' to d. |
| 9. | pop() | d.pop(k,v) | Remove k & ith value from d. If key is not found, then the value 'v' is returned. if key is not found and 'v' is not mentioned then 'KeyError' is raised |
| 10. | setdefault() | d.setdefault(k,v) | If key 'k' is found, ith value is returned. If key is not found, then the k, v pair is stored into the dictionary. |

Ex: Pgm to return keys, values and key-value pairs.

Sol:
```
dict = {'Name' : 'Karthik', 'Id' : 200, 'Salary' : 9080.50}
Print (dict)
Print ('Keys =', dict.keys())
Print ("Values =", dict.values())
Print ("Items =", dict.items())
```

o/p:
```
keys = dict_keys (['Name', 'Id', 'Salary'])
Values = dict_values (['chandra', 200, 9080.5])
Items = dict_items ([('Name', 'chandra'), ('Id', 200), ('Sal', 9080.5)])
```

2. Pgm to create a dict & find the sum of values

Sol:
```
dict = eval(input("enter ele in { }: "))
S = sum(dict.values())
Print("sum is", S)
```

O/P : enter ele in { } : {'A':10, 'B':20, 'C':35, 'Anil':50}

sum is 115.

3. Pgm to create a dict from key board & display elements

Sol:
```
x = { }
Print("How many ele")
n = int(input())
for i in range(n):
    Print('enter key")
    K = input()
    Print("enter value")
    V = int(input())
    x.update({K:V})
    Print("the dict is :", x)
```

O/P : How many ele: 2

enter key: Karthik
enter value: 10

enter key: Raju
Enter value: 20.

The dic is : {'Karthik': 22, 'Raju': 10}

## Converting lists in to Dictinary:

countries = ["USA", "India", "Germany"]

cities = ["washington", "Delhi", "Berlin"]

- Two steps - first step is to create a 'zip' class obj by passing the two lists to zip()

z = zip(countries, cities)

- zip() is useful to convert the sequences into a zip class obj.
- The second step is to convert the zip obj into dict by using dict() fun.

d = dict(z).

Print(d)

=> {'India': 'Delhi', 'USA': 'washington', 'Germany': 'Berlin'}

## Passing Dictionaries to fun:

Ex: def. fun(dict):

for i, j in dict.items():

Print(i, '--', j)

d = {'a': 'APPLE', 'b': 'BOOK', 'c': 'COOK'}

fun(d)

o/p: b -- BOOK

a -- APPLE

c -- COOK.

## Ordered Dictionaries : Collections module -> orderedDict() method

```
from Collections import orderedDict
d = orderedDict()
```

ex: Pgm to create a dict that does not change the order of elem

Sol)

```
from Collections import orderedDict
d = orderedDict()
d[10] = 'A'
d[11] = 'B'
d[12] = 'C'
for i, j in d.items():
    Print(i, j)
```

o/p:
```
10  A
11  B
12  C.
```

## Converting strings to Dictionary

```
str = "Vijay=24, Rahul=23, Mani=18"
for x in str.split(','):
    y = x.split('=')
    lst.append(y).
d = dict(lst)
Print(d).
```

o/p: {'Vijay': '24', 'Rahul': '23', 'Mani': '18'}

## Using for loop with Dictionaries:

for loop is very convenient to retrieve the elements of a dictionary.

```
colour = {'r':'red', 'g':'green', 'b':'blue'}
```

i) 
```
for K in colour:
    Print (K)
```

In the above loop, 'K' stores each element of the colour dictionary. Here, 'K' assumes only keys and hence this loop displays only keys.

ii) If we want to retrieve values, then we can obtain them by passing the key to colour dictionary as colour[K].

```
for K in colour:
    Print (colour[K])
```

iii) If we want to retrieve both the keys and values, we can use the items() method in for loop as:

```
for K, V in colour.items():
    Print ("Key = %d It value = %d" % K, V)
```

Strings: A string represent a group of characters.

Creating strings:

```
S1 = 'Hello'
S2 = "Hello"
```

if you want represent a string that occupies several line we can use triple single or triple double quotes.

```
str = '''Hi welcome
bye'''
```

(or)

```
str = """Hi welcome
bye"""
```

It is also possible to display quotation marks to mark a sub - string in a string.

```
S1 = 'welcome to "college" for fun'
Print (S1)
```

o/p: welcome to "college" for fun.

(or)

```
S1 = "welcome to 'college' for fun"
Print (S1)
```

o/p: welcome to 'college' for fun.

- It is also possible to use escape characters inside the string.

```
S1 = "welcome to \n core Python \n learning"
Print (S1)
```

\a
\b
\n
\t
\v
\r enter
\\ single \.

o/p : welcome to corepython.
learning.

To nullify (no use) the effect of escape characters, we can create the string as a 'raw' string by adding 'r' before the string.

```
s1 = r"welcome to\t Python\n learning"
Print(s1)
```

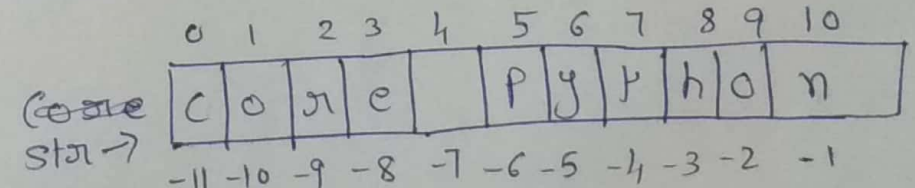
o/p : welcome to\t Python\n learning.

## Length of a string:

```
str = 'core python'
n = len(str)
Print (n)        o/p : 11
```

## Indexing in string:

Index represents the position no. Index is written using square braces [].




ex: Str = 'core Python'

```
n = len(str)
i = 0
while i<n:
    Print ( str[i], end = ' ')
    i = i+1
```

```
Print ( )
i = -1
while i >= -n :          # [-1] represents last element
    Print (Str[i], end = ' ')  in string.
    i = i-1.
```

Slicing the string:

```
          0    n-1   1
String [start : stop : stepsize]
```

ex: 1. Str = 'Core Python'

Str [0:9:1]

Core Python

2. Str [0:9:2]

Cr yh

3. Str [ : : ]

Core Python.

4. Str [2:4:1]

re

5. Str [ : : 2]

Cr yhn

6. Str [2: : ]

re Python.

7. Str [ :4: ]

Core

8. It is also possible to use reverse slicing.

8/ Str [-4:-1] from L to R

tho  Str[-4] to Str[-2]

9: Str [-6: : ]

Python

10/ Str [-1:-4:-1] retrieves from

noh  Str[-1] to Str[-3]

R to L

11. Str [-1: : -1]

R to L full

nohtyP eroC

Note: Forward i:e L to R step size is optional

backward i:e R to L step size is mandatory

Repeating the strings : The repetition is denoted by *.

Ex: 1. Str = 'Core Python'

Print (Str * 2)

o/p: Core python Core Python

S = Str[5:7:] * 3

Print(S)

o/p: PJPJPJ

Concatenation of Strings :

We can use '+' or join() method to attach a string.

Ex : S1 = 'Core'

S2 = 'Python'

S3 = S1+S2

Print (S3)

o/p: Core Python.

Checking Membership :

We can check if a string or character is a member of another string or not using 'in' or 'not in' operators.

S1 = 'This is core Python'

S2 = 'Core'

if S2 in S1:

Print (S2 + 'is found in S1')

else :

Print (S2 + 'is not found')

## Removing Spaces From a String:

A space is also considered as a character inside a string. Sometimes, the unnecessary spaces in a string will lead to wrong result.

ex: if 'Rakesh ' == 'Rakesh':
        Print ('welcome')
    else:
        Print ('Name not found')

Hence, spaces should be removed from the string using rstrip(), lstrip() and strip() methods.

1. name = ' Rakesh Krishnan '
   Print (name.rstrip()) # remove space at right
   o/p: _Rakesh Krishnan

2. Print (name.lstrip()) # remove left space.
   o/p: Rakesh Krishnan_

3. Print (name.strip()) # remove spaces from both side.
   o/p: Rakesh Krishnan.

Note: The above methods do not remove spaces which are in the middle of the string.

## Finding Sub Strings:

The find(), rfind(), index() and rindex() methods are useful to locate sub strings in a string. These methods return the location of the first occurrence of the sub string in the main string. The find() and index() methods search for the sub string from the beginning of the main string. The rfind() and rindex() methods search for the sub string from right to left i.e in backward direction.

The find() method returns -1 if the sub string is not found in the main string. The index() method returns 'ValueError' exception if the sub string is not found.

mainstring.find(substring, beginning, ending)

Ex: 1. A Python program to find the first occurrence of substring in a given main string.

Sol:

```
str = input('Enter the main string')
sub = input('Enter the sub string')
n = str.find(sub, 0, len(str))
if n==-1:
    print("sub string not found")
else:
    print("sub string found at pos:", n+1)
```

o/p: Enter the main string: This is a Book

Enter the sub string: is

Sub string is found at Position: 3

2. A Program to find the first occurrence of sub string in a given string using index() method.

Sol:

```
str = input ('Enter the main string')
sub = input ('Enter the sub string')
try:
    n = str.index (sub, 0, len(str))
except ValueError:
    Print ('sub string not found')
else:
    Print ('sub string found at position', n+1)
```

o/p: same as above program.

Counting Substrings in a string:

The count() method is available to count the no. of occurrences of a sub string in a main string.

```
Stringname.count (substring, beg, end)
```

Ex: 1.

```
str = 'New Delhi'
n = str.count ('e', 0, 3)
Print (n)
```

o/p: 1

String name . Count (substring)

2. Str = 'New Delhi'

n = Str . Count ('Delhi')

Print (n)

o/p : 1

3. If we want to search 'e' in the main string starting from 0th character to the end of the string, we can write:

n = Str . count ('e', 0, len(str))

Print (n)

o/p : 2

Strings are Immutable:

An immutable object is an object whose content cannot be changed . on the other hand, a mutable object is an object whose content can be changed as and when required. In Python, numbers, strings and tuples are immutable . List, sets, dictionaries are mutable objects.

In the following code, we create a string 'str' where we stored 4 characters : 'abcd'. when we display 0th character, i.e Str[0], it will display 'a'. If we try to replace the 0th character with a new character 'x', then there will be an error called 'Type Error'.

Strings are immutable for two reasons i.e Performance, Security.

Ex: str = 'abcd'

```
Print (str[0])
```

a

```
str[0] = 'x'
```

TypeError: 'str' obj does not support item assignment.

In this ex, we are creating two strings S1 and S2 as:

```
S1 = 'one'
S2 = 'Two'
S2 = S1    # store S1 into S2
Print (S2)
```

one

If you write,

```
Print (S1)   # display S1
```

one

It seems that the content of 'S2' is replaced by the content of 'S1' and hence 'S2' became mutable. But this is wrong.

S2 = S1

The name S2 will be adjusted to refer to the object that is referenced by S1. But, the original value of S2 that is 'two' is not altered. Since 'two' is not referenced, the gc deletes the object from memory. (gc - garabage collector)

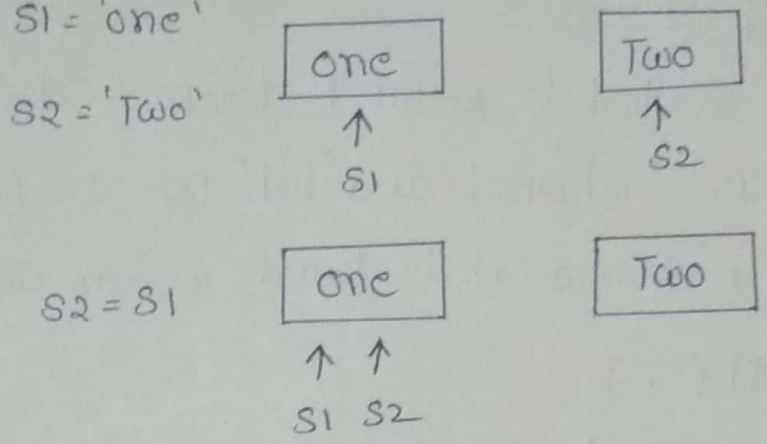



Fig: Immutability of string objects.

Replacing a string with another string:

The replace() method is useful to replace a sub string in a string with another sub string.

stringname.replace(old, new)

This will replace all the occurrences of 'old' sub string with 'new' sub string in the main string. For ex.

```
str = 'That is a beautiful Forest'
S1 = 'Forest'
S2 = 'Flower'
str1 = str.replace(S1, S2)
Print(str)
```

The o/p of the preceding statement is as follows:

That is a beautiful Forest

```
Print(str1)
```

o/p is, That is a beautiful Flower.

## Splitting and Joining strings:

The split() method is useful to break a string into a pieces. These pieces are returned as a "list". For, ex to break the string 'str' where a comma (,) is found, we can write as:

```
str.split(',')
```

Ex: 1. A Python Program to accept and display a group of no:s

Sol:
```
str = input('Enter the no:s separated by commas')
lst = str.split(',')
for i in lst:
    print(i)
```

o/p: Enter the no:s separated by commas: 10, 20, 30, 40, 50

```
10
20
30
40
50
```

When a group of strings are given, it is possible to join them all and make a single string. For this purpose, we can use join() method, as:

```
separator.join(str)
```

Where, the 'separator' represents the character to be used b/w the strings in the o/p. 'str' represents a tuple or list of strings. In the following ex, we are taking a tuple 'str' that contains 3 strings as:

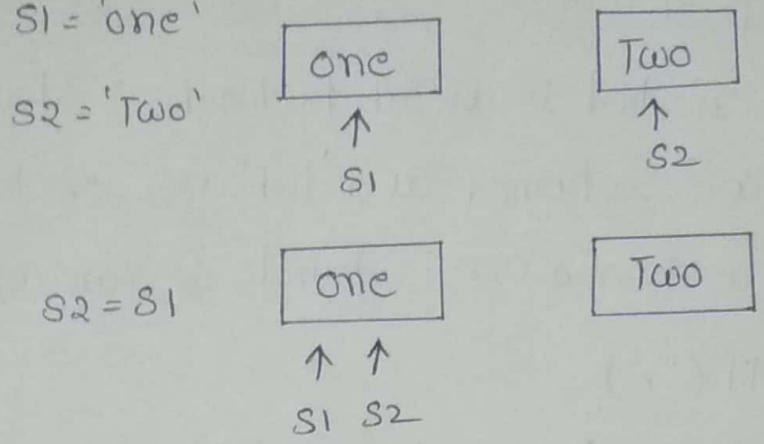



Fig: Immutability of string objects.

Replacing a string with another string:

The replace() method is useful to replace a sub string in a string with another sub string.

stringname.replace(old, new)

This will replace all the occurrences of 'old' sub string with 'new' sub string in the main string. For ex.

```
str = 'That is a beautiful Forest'
S1 = 'Forest'
S2 = 'Flower'
str1 = str.replace(S1,S2)
Print(str)
```

The o/p of the preceding statement is as follows:

That is a beautiful Forest

```
Print(str1)
```

o/p is, That is a beautiful Flower.

```
str = ('one', 'two', 'three')
```

We want to join the three strings and form a single string. Also, we want to use hyphen (-) b/w the three strings in the o/p. The join() method can be written as:

```
str1 = "-".join(str)
Print(str1)
```

o/p is, one-two-three

Here, we are taking a list comprising 3 strings and we are joining them using a colon (:) b/w them.

```
str = ['apple', 'lemon', 'orange']
sep = ':'
str1 = sep.join(str)
Print(str1)
```

o/p is, apple:lemon:orange

Changing case of a string:

Python offers 4 methods that are useful to change the case of a string. They are upper(), lower(), swapcase(), title().

1. upper(): This method is used to convert all the characters of a string into uppercase.
2. lower(): This method is used to convert all the characters of a string into lowercase.

3. The swapcase(): This method converts the capital letters into small letters and viceversa.

4. title(): This method converts the string such that each word in the string will start with a capital letter and remaining will be small letters.

Ex: 1. str = 'python is future'

print (str.upper())

output is, PYTHON IS FUTURE

2. str = 'PYTHON IS FUTURE'

print (str.lower())

output is, python is future.

3. str = 'python is future'

print (str.swapcase())

output is, ~~Python is fu~~ PYTHON IS FUTURE

4. str = 'PYTHON IS FUTURE'

print (str.title())

output is, Python Is Future.

## String Testing Methods:

There are several methods to test the nature of characters in a string.

| Method | Description |
|---|---|
| 1. isalnum() | - This method returns True if all characters in the string are alphanumeric (A to Z, a to z, 0 to 9) and there is atleast one character otherwise it returns False. |
| 2. isalpha() | - Returns True if the string has atleast one character and all characters are alphabetic (A to Z, a to z) otherwise, it return False. |
| 3. isdigit() | - Returns True if the string contains only numeric digit (0 to 9) and False otherwise. |
| 4. islower() | - Returns True if the string contains atleast one letter and all characters are in lower case, otherwise it returns False. |
| 5. isupper() | - Returns True if the string contains atleast one letter and all characters are in upper case. |
| 6. istitle() | - Returns True if the string contains initial letter capital and there is atleast one character otherwise returns False. |
| 7. isspace() | - Returns True if the string contains only spaces, otherwise it returns False. |

## Checking Starting and Ending of a String:

The startswith() method is useful to know whether a string is starting with a sub string or not.

```
Str.startswith(substring)
```

When the sub string is found in the main string 'Str', this method returns True. If the sub string is not found it returns False.

```
Str = "This is Python"
Print (Str.startswith('This'))
```

it will display, True

Similarly, to check the ending of a string, we can use endswith() method. It returns True if the string ends with the specified sub string, otherwise it returns False.

```
Str.endswith(substring)
Str = 'This is Python'
Print(Str.endswith('Python'))
```

it will display, True.

```
Print(Str.endswith('Java'))
```

Then it displays, False.

## String Testing Methods:

There are several methods to test the nature of characters in a string.

| Method | Description |
|---|---|
| 1. isalnum() | - This method returns True if all characters in the string are alphanumeric (A to Z, a to z, 0 to 9) and there is atleast one character otherwise it returns False. |
| 2. isalpha() | - Returns True if the string has atleast one character and all characters are alphabetic (A to Z, a to z) otherwise, it return False. |
| 3. isdigit() | - Returns True if the string contains only numeric digit (0 to 9) and False otherwise. |
| 4. islower() | - Returns True if the string contains atleast one letter and all characters are in lower case, otherwise it returns False. |
| 5. isupper() | - Returns True if the string contains atleast one letter and all characters are in upper case. |
| 6. istitle() | - Returns True if the string contains initial letter capital and there is atleast one character, otherwise returns False. |
| 7. isspace() | - Returns True if the string contains only spaces, otherwise it returns False. |

Ex: 1. str = 'Delhi999'

s = str.isalpha()

Print(s)

it returns False

2. str = 'Delhi'

Print(str.isalpha())

it return True.

Formatting the strings:

It means presenting the string in a clearly understandable manner. The format() method is used to format strings.

'format string with replacement fields'.format(values)

We should first understand the meaning of the first attribute i.e 'format string with replacement fields'. The replacement fields are denoted by curly braces {} that contains names or indexes. These names or indexes represent the order of the values.

Ex: id = 10

name = 'Karthik'

sal = 42500.66

str = '{}, {}, {}'.format(id, name, sal)

Print(str)

it displays. 10, Karthik, 42500.66

Suppose, we do not want to display commas after each value, rather we want to display hyphens (-).

```
str = '{}-{}-{}'.format(id, name, sal)
```

output will be, 10-Karthik-42500.66

we can also display message strings before every value, which can be done as:

```
str = 'Id={}\n Name={}\n Salary={}\n'.
                                format(id, name, sal)
print(str)
```

The output will be, Id=10
Name=Karthik
Salary=42500.66

we can also mention the order numbers in the replacement fields as 0, 1, 2, etc.

```
str = 'Id={0}\t Name={1}\t Salary={2}.
                                format(id, name, Sa
```

output is, Id=10 Name=Karthik Salary=42500.66

By changing the numbers in the replacement field we can also change the order of the values being displayed.

```
str = 'Id={2}\t Name={0}\t salary={1}.format(id, name
                                Salary)
print(str)
```

o/p: Id=42500.66 Name=10 Salary=Karthik.

We can also mention names in the replacement fields to represent the values. These names should refer to the values in the format() method as:

```
Str = 'Id = {one}, Name = {two}, Salary = {three}'.
                    format(one = id, two = name, three = Salary)
Print(Str)
```

The output will be, Id = 10, Name = Karthik, Salary = 42500.66

Formatting specification start with a colon (:) and after that, we can specify the format in the curly braces.

```
Str = 'Id = {:d}, Name = {:s}, Salary = {:10.2f}'. format(id, name,
                                                    Salary)
Print(Str)
```

output will be, ~~10~~, ~~Karthik~~,

Id = 10, Name = Karthik, Salary = 42500.66

It is possible to align the value in the replacement field. '<' represent align left, '>' represent align right, '^' (caret) represent align in the center.

Ex: 1. num = 5000

Print ('{:*>15d}'. format(num))

o/p is, **********5000

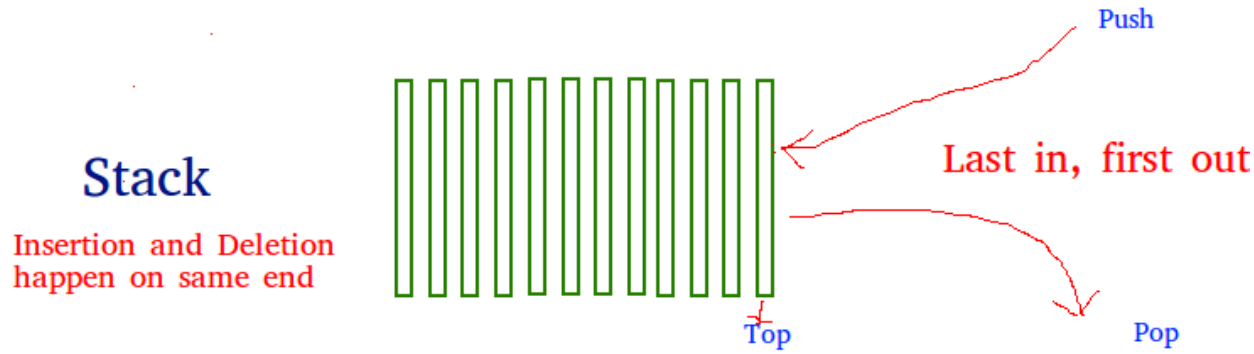
2. num = 5000

Print ('{:*^15d}', format(num))

o/p is, *****5000*****

# STACK IN PYTHON

A stack is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.




The functions associated with stack are:

- **empty()** – Returns whether the stack is empty
- **size()** – Returns the size of the stack
- **top()** – Returns a reference to the top most element of the stack
- **push(g)** – Adds the element 'g' at the top of the stack
- **pop()** – Deletes the top most element of the stack

**Implementation:** There are various ways from which a stack can be implemented in Python. This article covers the implementation of stack using data structures and modules from Python library.

Stack in Python can be implemented using following ways:

- list
- collections.deque
- queue.LifoQueue

**Implementation using list:** Python's buil-in data structure list can be used as a stack. Instead of push(), append() is used to add elements to the top of stack while pop() removes the element in LIFO order. Unfortunately, list has a few shortcomings. The biggest issue is that it can run into speed issue as it grows. The items in list are stored next to each other in memory, if the stack grows bigger than the block of memory that currently hold it, then Python needs to do some memory allocations. This can lead to some append() calls taking much longer than other ones.

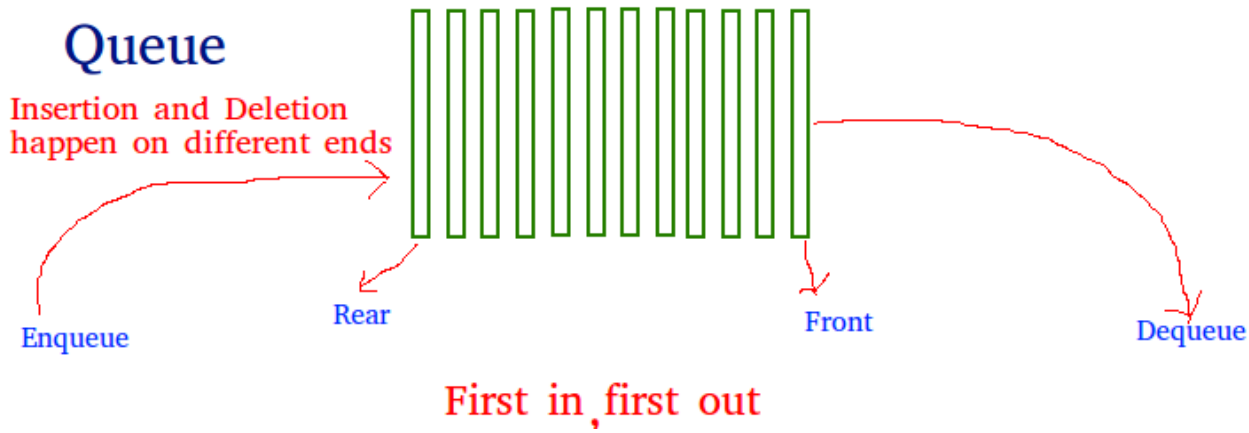
```
# Python program to
# demonstrate stack implementation
# using list
stack = []
# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)
# pop() fucntion to pop
# element from stack in
# LIFO order
print('\nElements poped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())
print('\nStack after elements are poped:')
print(stack)
# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

**Output:**

```
Initial stack

['a', 'b', 'c']

Elements poped from stack:

c

b

a

Stack after elements are poped:

[]
```

# QUEUE IN PYTHON

Like stack, queue is a linear data structure that stores items in First In First Out (FIFO) manner. With a queue the least recently added item is removed first. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.




Operations associated with queue are:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition
- **Front:** Get the front item from queue
- **Rear:** Get the last item from queue

**Implementation:** There are various ways to implement a queue in Python. This article covers the implementation of queue using data structures and modules from Python library.

Queue in Python can be implemented by the following ways:

- list
- collections.deque
- queue.Queue

**Implementation using list**

List is a Python's built-in data structure that can be used as a queue. Instead of enqueue() and dequeue(), append() and pop() function is used. However, lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all of the other elements by one, requiring O(n) time.

```
# Python program to
# demonstrate queue implementation
# using list
# Initializing a queue
queue = []
# Adding elements to the queue
queue.append('a')
queue.append('b')
queue.append('c')
print("Initial queue")
print(queue)
# Removing elements from the queue
print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))
print("\nQueue after removing elements")
print(queue)
# Uncommenting print(queue.pop(0))
# will raise and IndexError
# as the queue is now empty
```

**Output:**

```
Initial queue

['a', 'b', 'c']

Elements dequeued from queue

a
```

```
b

c

Queue after removing elements

[]
```