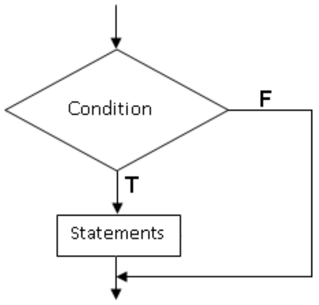
Conditional Statements

Conditional Statements are features of a programming language, which perform different computations or actions depending on whether the given condition evaluates to true or false. Conditional statements in python are of 3 types

- If statement
- If else statement
- If elif statement
- Nested if else

1. **If Statement or single alternate decision statement:** if Statement is used to run a statement conditionally i.e. if given condition is true then only the statement given in if block will be executed.
   Syntax:

```
if <condition>:
    <if statement block>
```



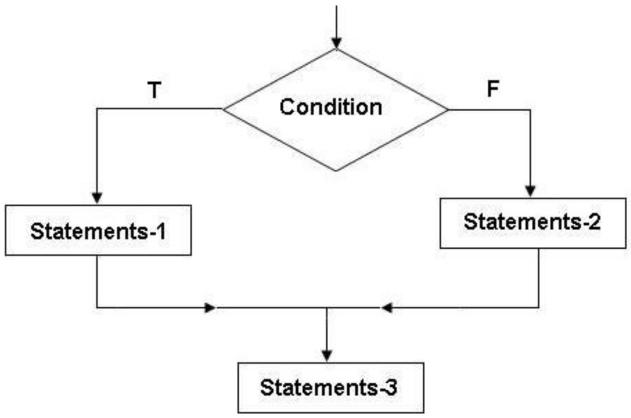
For **example** consider the code given below

```
if (percentage > 33):
        print ("Pass")
```

Explaination : In the above code if value of percentage is above 33 then only the message "Pass" will be printed.

2. **If else or two alternative decision statement:** Statement In the case of if else statement If given condition is true then the statement given in if block will be executed otherwise(else) the statements written in else block will be executed.
   Syntax:

```
If<condition> :
        <if statement block>
else:
        <else statement block>
```



For **example** consider the code given below

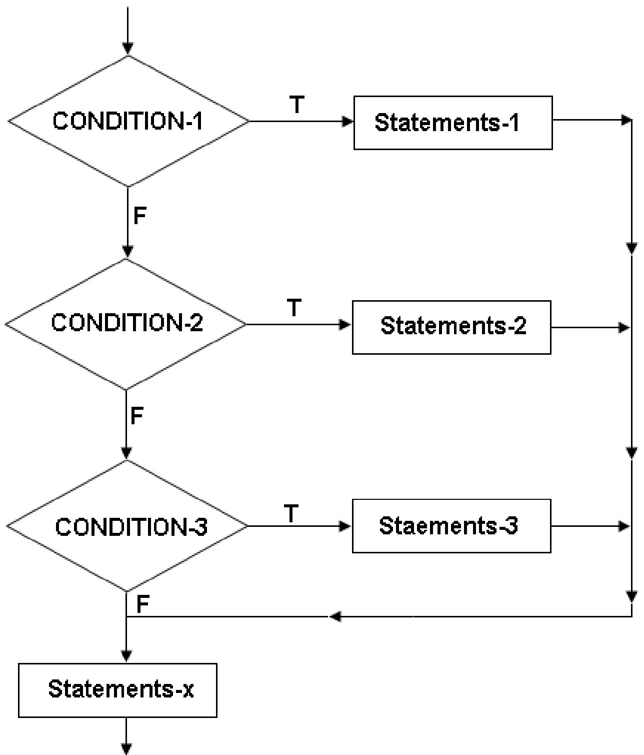
```
    if (percentage > 33):

            print ("Pass")

    else:

            print("Fail")
```

Explanation : In the above code if value of percentage is above 33 then only the message “Pass” will be printed otherwise it will print “Fail”.

3. If elif and elif ladder: Statements with if elif or If elif ladder are used for execution of statements based on several alternatives. Here we use one or more elif (short form of else if) clauses. Python evaluates each condition in turn and executes the statements corresponding to the first if that is true. If none of the expressions are true, and an else clause will be executed.

Syntax: -

```
    If< condition>:

            < statement (s)>

    elif <condition>:

    .       < statement(s) >

    .

    else:

            < statement (s)>
```



Explanation: In the above code if value of percentage is above 33 then only the message “Pass” will be printed otherwise it will print “Fail”.

Example:

```
If (percentage >90):
        Print("Outstanding")
elif (percentage >80):
        print ("Excellent")
elif (percentage >70):
        print ("Very Good")
elif (percentage >60):
        print ("Good")
 elif (percentage >33):
        print ("Pass")
else:
        print("Fail")
```

Explanation: In the above code

if value of percentage is above 90 then it will print “Outstanding”

if value of percentage is above 80 then it will print “Excellent”

if value of percentage is above 70 then it will print “Very Good”

if value of percentage is above 60 then it will print “Good”

if value of percentage is above 80 then it will print “Pass”

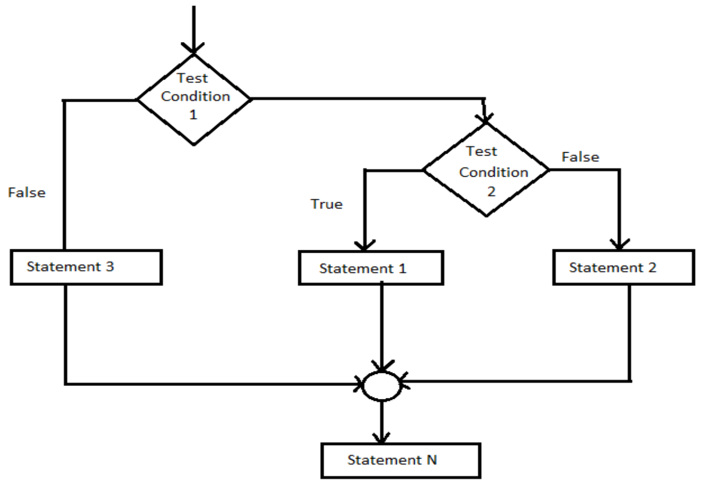
if no condition is true then it will print “Fail”

In above code only 1 condition can be true at a time if no condition is true then else statement will be executed.

4. Nested If else Statement A nested if is an if statement that is the target of another if statement. Nested if statement means an if statement within another if statement.

Syntax: -

```
if <condition1>:
            statement(s)
             if<condition2>:
                        statement(s)
             else:
else:
             if <condition3>:
                         statement(s)
             else
                         Statement(s)
```



Example:

```
if color =”red”:
            if item=”fruit”:
                        print(“ It is an Apple”)
            else :
                print("It may be Tomato or Rose")
else:
        if color="Yellow":
                print("It is a Banana")
         else:
                print("It may be corn or Marigold ")
```

OR

```
if color ="red":
        if item="fruit":
                print(" It is an Apple"
        else :
                print("It may be Tomato or Rose")
 elif color="Yellow":
         print("It is a Banana")
        else
               print("It may be corn or Marigold ")
```

Iterative Statements

Iteration statements or loop statements allow us to execute a block of statements repeatedly as long as the condition is true.

(Loops statements are used when we need to run same code again and again)

Type of Iteration Statements In Python 3

In Python Iteration (Loops) statements are of three types :-

1. While Loop
2. For Loop
3. Nested Loops

1. While Loop

In Python While Loop In Python is used to execute a block of statement till the given condition is true. And when the condition is false, the control will come out of the loop.

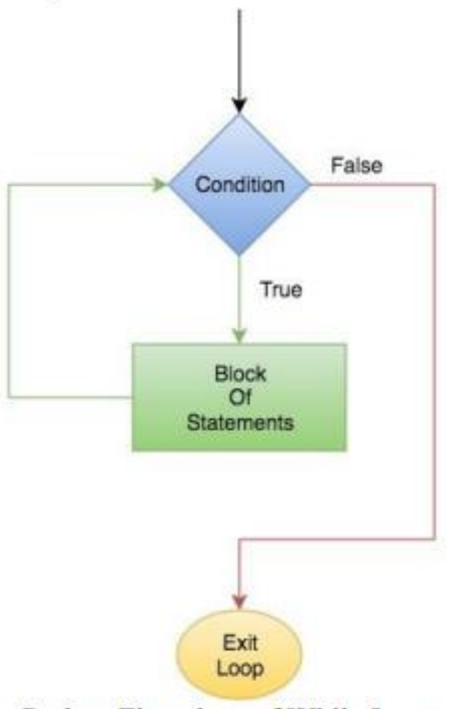
The condition is checked every time at the beginning of the loop.

While Loop Syntax

```
while (<condition>):
        statements
```

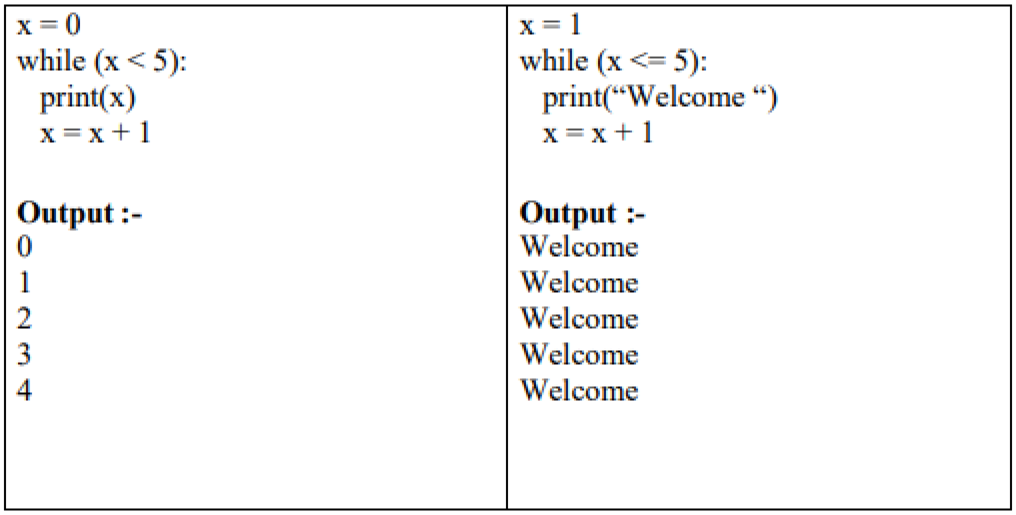
This type of loop can be used if it’s not in advance how many times that the loop will repeat (most powerful type of loop).

Flowchart of While Loop



Python Flowchart of While Loop

Examples Of While Loop



```
x = 0
while (x < 5):
   print(x)
   x = x + 1
```

Output :-

```
0
1
2
3
4
```

```
x = 1
while (x <= 5):
   print("Welcome ")
   x = x + 1
```

Output :-

```
Welcome
Welcome
Welcome
Welcome
Welcome
```

While Loop With Else In Python

The else part is executed if the condition in the while loop becomes False.

Syntax of While Loop With Else

```
while (condition):
        loop statements
 else:
        else statements
```

Example of While Loop With Else

```
x = 1
while (x < 5):
        print('inside while loop value of x is ',x)
        x = x + 1
        else: print('inside else value of x is , x)
```

Output :-

```
 inside while loop value of x is 1

 inside while loop value of x is 2

 inside while loop value of x is 3

 inside while loop value of x is 4

inside else value of x is 5
```

Infinite While Loop in Python: A Infinite loop is a loop in which condition always remain True.

Example of Infinite While Loop

```
 x = 1
while (x == 1):
            print(‘hello‘)
```

Output :- hello

```
 hello
hello
—
–
—
```

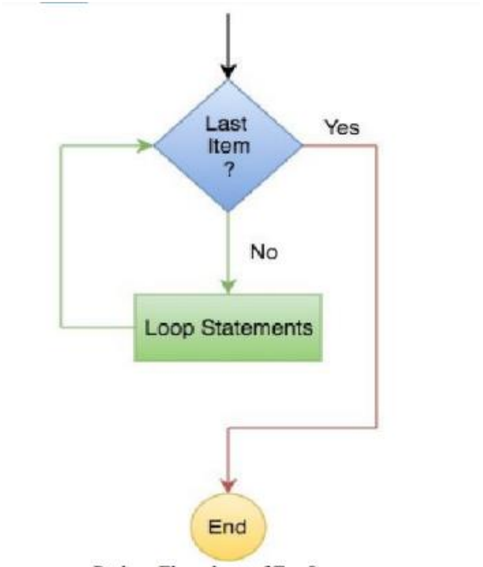
2. For Loop in Python

For loop in Python is used to iterate over items of any sequence, such as a list or a string.

For Loop Syntax

```
for val in sequence:
            statements
```

Flowchart of For Loop

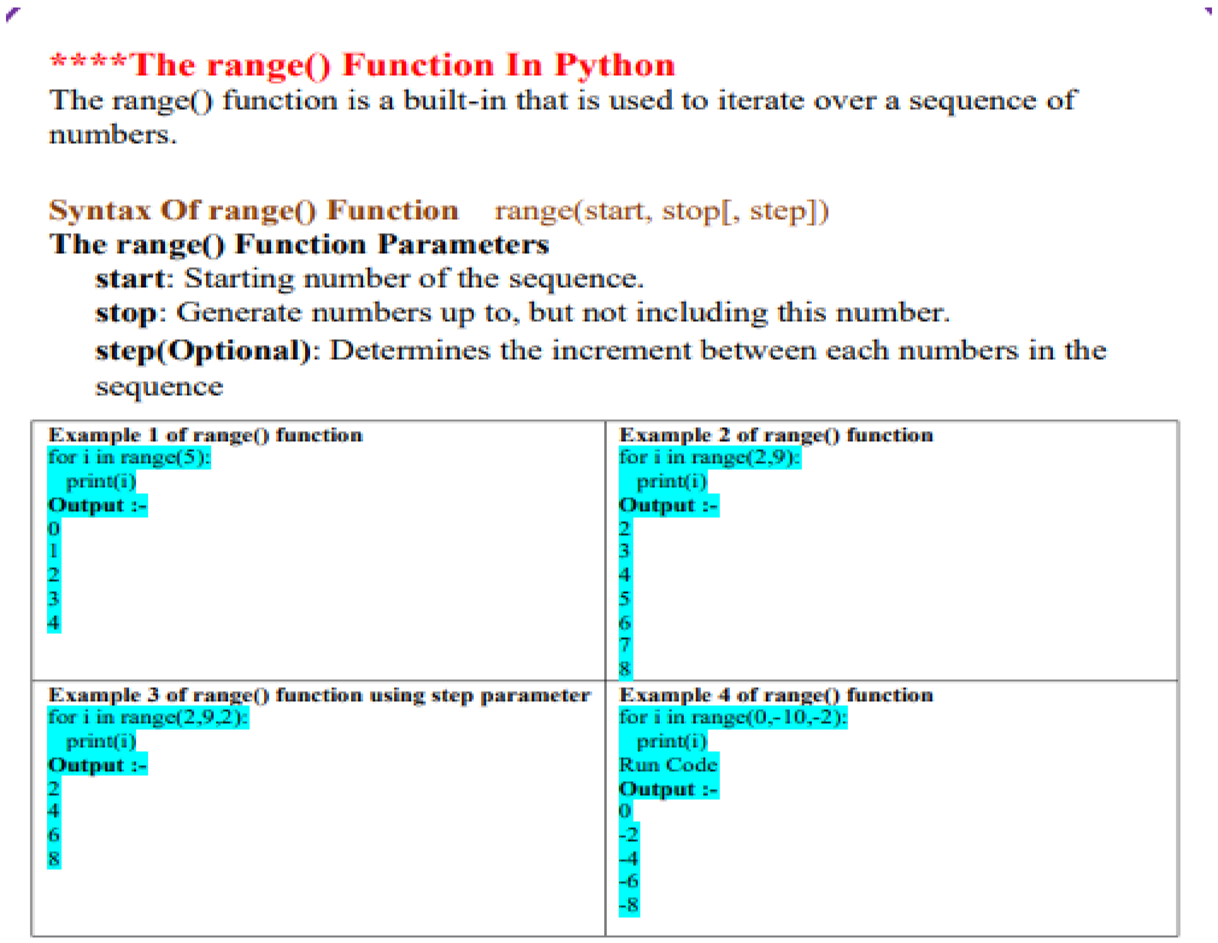


Python Flowchart of For Loop

ExampleS of For Loop

for i in range(1,5): print(i) Output :- 1 2 3 4	for i in [1,2,3,4] : print(“WELCOME ”) Output :- WELCOME WELCOME WELCOME WELCOME

Kiranmaie p
IT CBIT



****The range() Function In Python

The range() function is a built-in that is used to iterate over a sequence of numbers.

Syntax Of range() Function range(start, stop[, step])

The range() Function Parameters

- **start**: Starting number of the sequence.
- **stop**: Generate numbers up to, but not including this number.
- **step(Optional)**: Determines the increment between each numbers in the sequence

Example 1 of range() function for i in range(5): print(i) **Output :-** 0 1 2 3 4	**Example 2 of range() function** for i in range(2,9): print(i) **Output :-** 2 3 4 5 6 7 8
Example 3 of range() function using step parameter for i in range(2,9,2): print(i) **Output :-** 2 4 6 8	**Example 4 of range() function** for i in range(0,-10,-2): print(i) Run Code **Output :-** 0 -2 -4 -6 -8

For Loop with Else in Python

The else is an optional block that can be used with for loop. The else block with for loop executed only if for loops terminates normally. This means that the loop did not encounter any break.

Example 1 of For Loop with Else

```
list=[2,3,4,6,7]

for i in range(0,len(list)):

        if(list[i]==4):

                print(_list has 4‘)

         else:

               print(_list does not have 4‘)

                Output :- List has 4
```

list does not have 5

Example 2 of For Loop with Else

```
for i in range(0,len(list)):
        if(list[i]==5):
                print(_5 is there in the list‘) break
        else:
                print(_list does not have 5‘)
```

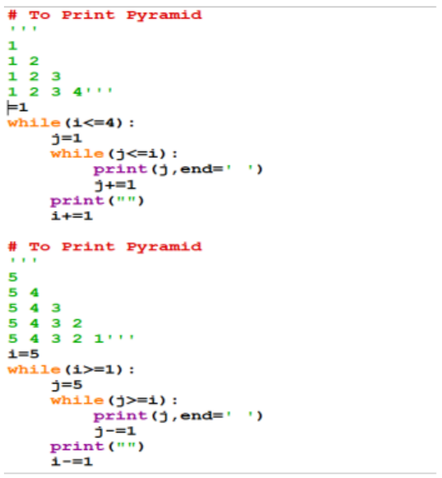
Output :- list does not have 5

NESTED loops in Python: The placing of one loop inside the body of another loop is called nesting. When you "nest" two loops, the outer loop takes control of the number of complete repetitions of the inner loop

Nested while loop Syntax:

```
Initialization
while(condition):
        initialization of inner loop
                while(conition):
                        ------
                        ------
                        Update expression of inner loop
Update expression of outer loop
```

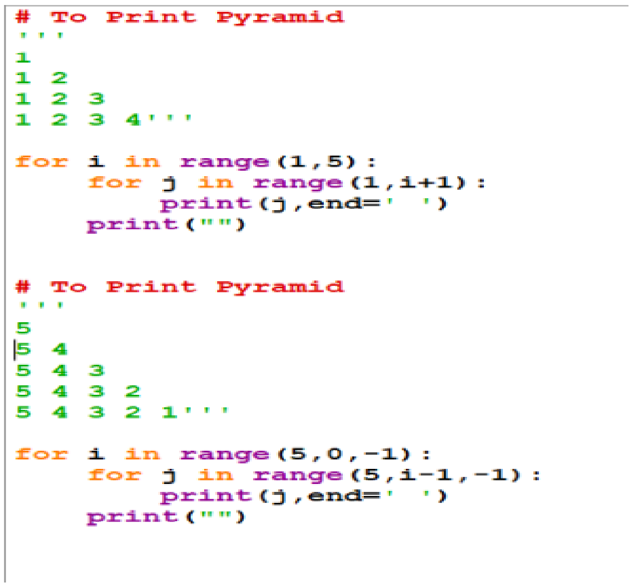
Examples



```
# To Print Pyramid
'''
1
1 2
1 2 3
1 2 3 4'''
i=1
while(i<=4):
    j=1
    while(j<=i):
        print(j,end='  ')
        j+=1
    print("")
    i+=1

# To Print Pyramid
'''
5
5 4
5 4 3
5 4 3 2
5 4 3 2 1'''
i=5
while(i>=1):
    j=5
    while(j>=i):
        print(j,end='  ')
        j-=1
    print("")
    i-=1
```

Nested for loop Syntax

```
for iterating_var in sequence:
    for iterating_var in sequence:
        statements(s)

    statements(s)
```



```
# To Print Pyramid
'''
1
1 2
1 2 3
1 2 3 4'''

for i in range(1,5):
    for j in range(1,i+1):
        print(j,end=' ')
    print("")


# To Print Pyramid
'''
5
5 4
5 4 3
5 4 3 2
5 4 3 2 1'''

for i in range(5,0,-1):
    for j in range(5,i-1,-1):
        print(j,end=' ')
    print("")
```

Nested While loop:

A nested while loop is a loop inside a loop. The inner loop will be executed completely for each iteration of the outer loop.

Syntax:

```
while condition1:
   while condition2:
      # statement(s)
   # statement(s)
```

Example 1

Here is a simple example that shows how a nested while loop works in Python.

```
i = 1
while i <= 3:
    print("Outer Loop: ", i, "time ------------------")
    j = 1
    while j <= 2:
        print("Inner Loop:", j)
        j += 1
    i += 1
```

Output:

```
Outer Loop:  1 time ------------------

Inner Loop: 1

Inner Loop: 2

Outer Loop:  2 time ------------------

Inner Loop: 1

Inner Loop: 2

Outer Loop:  3 time ------------------

Inner Loop: 1

Inner Loop: 2
```

Example 2

Let's say we want to print the table up to 5. We can use a nested while loop to do this.

```
i = 1
# outer loop
while i <= 5:
    j = 1
    # inner loop
    while j <= 10:
        print(i * j, end=" ")
        j += 1
    print()
    i += 1
```

Output:

```
1 2 3 4 5 6 7 8 9 10

2 4 6 8 10 12 14 16 18 20

3 6 9 12 15 18 21 24 27 30

4 8 12 16 20 24 28 32 36 40

5 10 15 20 25 30 35 40 45 50
```

Example 3

Using nested while loop to print elements of a list of lists.

```python
# nested while loop to print elements of a list of lists
list1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

i = 0
# outer loop
while i < len(list1):
    j = 0
    # inner loop
    while j < len(list1[i]):
        print(list1[i][j], end=" ")
        j += 1
    print()
    i += 1
```

```
Output:

1 2 3

4 5 6

7 8 9
```

The outer loop runs for the number of elements in the list and for each iteration of the outer loop, the inner loop runs for the number of elements in the inner list.

Example 4

Printing a pyramid pattern using nested while loop.

```
size = 5
i = 0
# outer loop
while i < size:
    j = 0
    # inner loop to print spaces
    while j < size - i - 1:
        print(' ', end='')
        j += 1
    k = 0
    # inner loop to print stars
    while k < 2 * i + 1:
        print('*', end='')
        k += 1
    print()
    i += 1
```

Output:

```
    *

   ***

  *****

 *******

*********
```

Break and Continue in Python loops

break statement:

When a break statement executes inside a loop, control flow “breaks” out of the loop immediately:

Example:

```
i = 0

while i < 7:

print(i)

if i == 4:

print("Breaking from loop")

break

i += 1
```

The loop conditional will not be evaluated after the break statement is executed. Note that break statements are only allowed *inside python loops*, syntactically. A break statement inside a function cannot be used to terminate python loops that called that function. Executing the following prints every digit until number 4 when the break statement is met and the loop stops:
01234

Breaking from loop break statements can also be used inside for python loops, the other looping construct provided by Python:

Example 2:

```
for i in (0, 1, 2, 3, 4):

print(i)

if i == 2:

break
```

Executing this loop now prints: 012 Note that 3 and 4 are not printed since the loop has ended.
If a loop has an else clause, it does not execute when the loop is terminated through a break statement.

continue statement:

A continue statement will skip to the next iteration of the loop bypassing the rest of the current block but continuing the loop. As with break, continue can only appear inside python loops**:**

Example:

```
for i in (0, 1, 2, 3, 4, 5):

if i == 2 or i == 4:

continue

print(i)
```

output

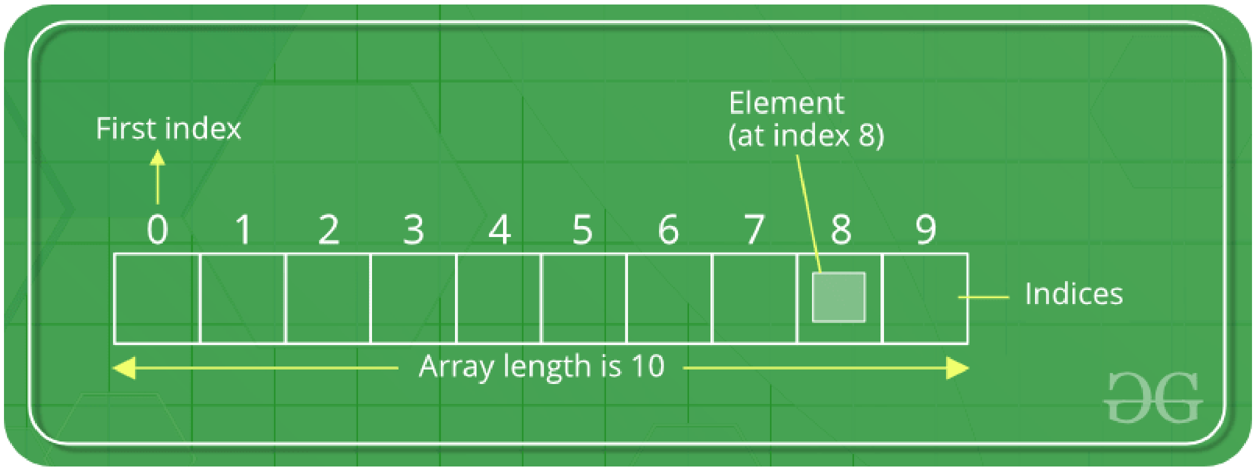
```
0

1

3

5
```

Note that 2 and 4 aren't printed, this is because continue goes to the next iteration instead of continuing on to.

Arrays:

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

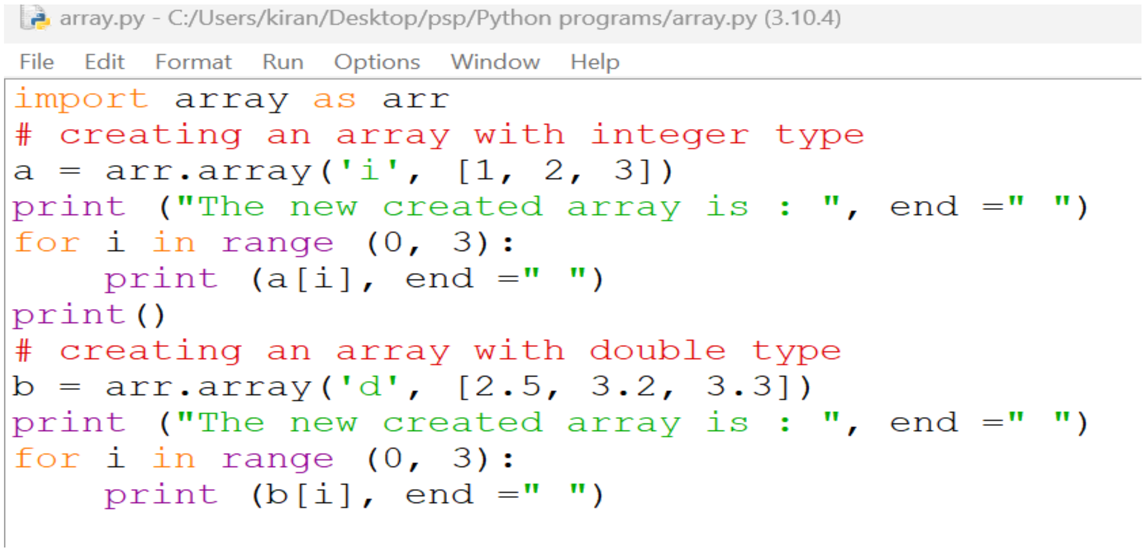
Array can be handled in Python by a module named **array**. They can be useful when we have to manipulate only a specific data type values. A user can treat lists as arrays. However, user cannot constraint the type of elements stored in a list. If you create arrays using the **array** module, all elements of the array must be of the same type.



1D arrays

a. Creating a Array

i. Array in Python can be created by importing array module. **Array (data_type, value_list)** is used to create an array with data type and value list specified in its arguments.

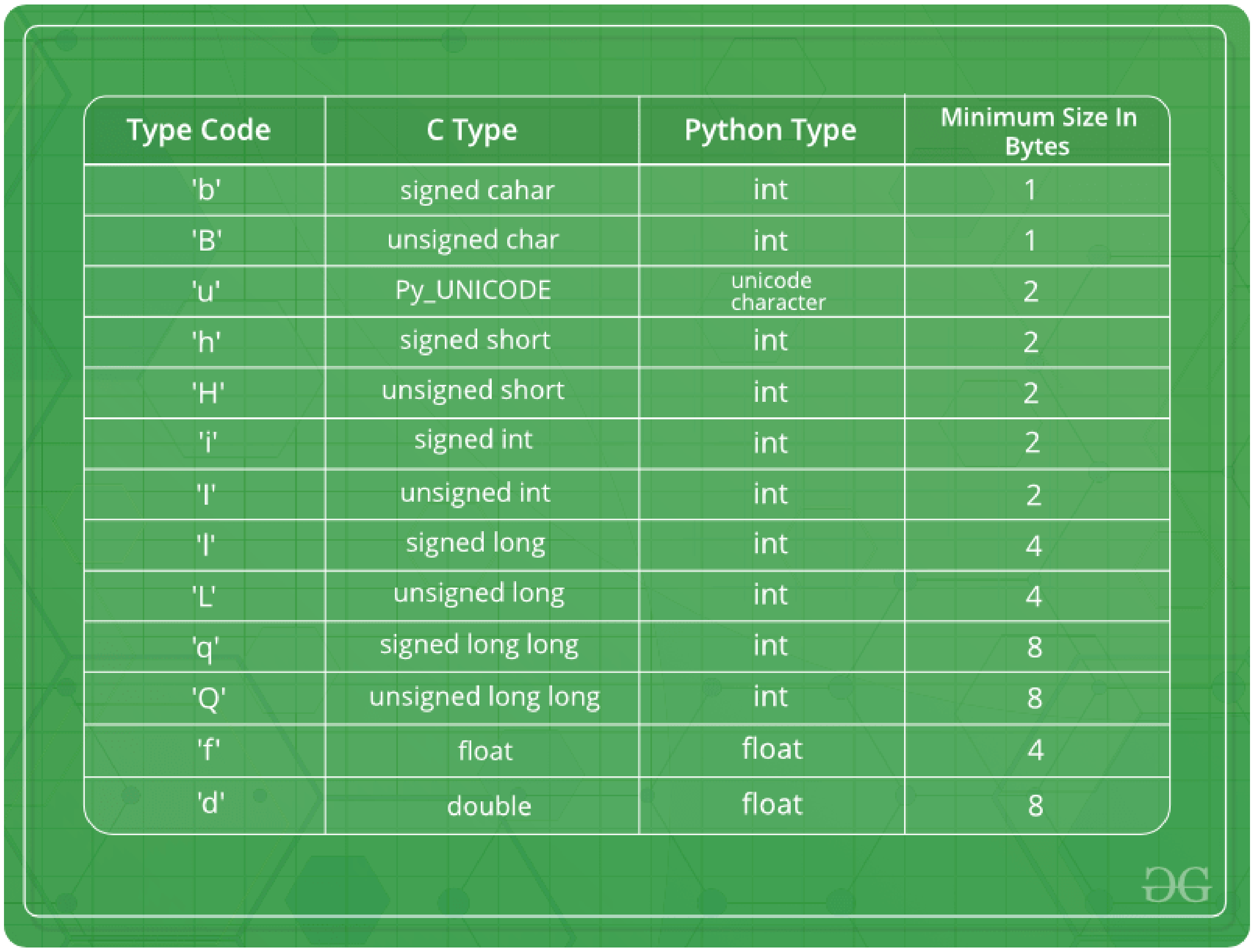

array.py - C:/Users/kiran/Desktop/psp/Python programs/array.py (3.10.4)
File Edit Format Run Options Window Help

```
import array as arr
# creating an array with integer type
a = arr.array('i', [1, 2, 3])
print ("The new created array is : ", end =" ")
for i in range (0, 3):
    print (a[i], end =" ")
print()
# creating an array with double type
b = arr.array('d', [2.5, 3.2, 3.3])
print ("The new created array is : ", end =" ")
for i in range (0, 3):
    print (b[i], end =" ")
```

Output:

```
The new created array is:  1 2 3

The new created array is:  2.5 3.2 3.3
```

Some of the data types are mentioned below which will help in creating an array of different data types.



Type Code	C Type	Python Type	Minimum Size In Bytes
'b'	signed cahar	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

b. Adding Elements to a Array

Elements can be added to the Array by using built-in insert() function. Insert is used to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. append() is also used to add the value mentioned in its arguments at the end of the array.

Adding elements by insert()

```
#Adding elements to array
import array as arr
a = arr.array('i', [1, 2, 3])
print ("Array before insertion : ", end =" ")
for i in range (0, 3):
    print (a[i], end =" ")
print()
# insert() function
a.insert(1, 4)
print ("Array after insertion : ", end =" ")
for i in (a):
    print (i, end =" ")
print()
```

Adding elements by append()

```
b = arr.array('d', [2.5, 3.2, 3.3])
print ("Array before insertion : ", end =" ")
for i in range (0, 3):
    print (b[i], end =" ")
print()
# adding an element using append()
b.append(4.4)
print ("Array after insertion : ", end =" ")
for i in (b):
    print (i, end =" ")
print()
```

Output:

```
Array before insertion: 1 2 3

Array after insertion:  1 4 2 3

Array before insertion: 2.5 3.2 3.3

Array after insertion:  2.5 3.2 3.3 4.4
```

c. Accessing elements from the Array

In order to access the array items, refer to the index number. Use the index operator [ ] to access an item in a array. The index must be an integer.

```
#Accessing elements in array
import array as arr
a = arr.array('i', [1, 2, 3, 4, 5, 6])
print("Access element is: ", a[0])
print("Access element is: ", a[3])
b = arr.array('d', [2.5, 3.2, 3.3])
print("Access element is: ", b[1])
print("Access element is: ", b[2])
```

Output:

```
Access element is:  1

Access element is:  4

Access element is:  3.2

Access element is:  3.3
```

d. Removing Elements from the Array

Elements can be removed from the array by using built-in remove() function but an Error arises if element doesn't exist in the set. Remove() method only removes one element at a time, to remove range of elements, iterator is used. pop() function can also be used to remove and return an element from the array, but by default it removes only the last element of the array, to remove element from a specific position of the array, index of the element is passed as an argument to the pop() method.

Note – Remove method in List will only remove the first occurrence of the searched element.

Delete the elements by pop ()

```
#deleting elements from array
import array
arr = array.array('i', [1, 2, 3, 1, 5])
print ("The new created array is : ", end ="")
for i in range (0, 5):
    print (arr[i], end =" ")
print ("\r")
# using pop() to remove element at 2nd position
print ("The popped element is : ", end ="")
print (arr.pop(2))
```

Delete elements by remove ()

```
print ("The array after popping is : ", end ="")
for i in range (0, 4):
    print (arr[i], end =" ")
print("\r")
# using remove() to remove 1st occurrence of 1
arr.remove(1)
print ("The array after removing is : ", end ="")
for i in range (0, 3):
    print (arr[i], end =" ")
```

Output:

The new created array is: 1 2 3 1 5

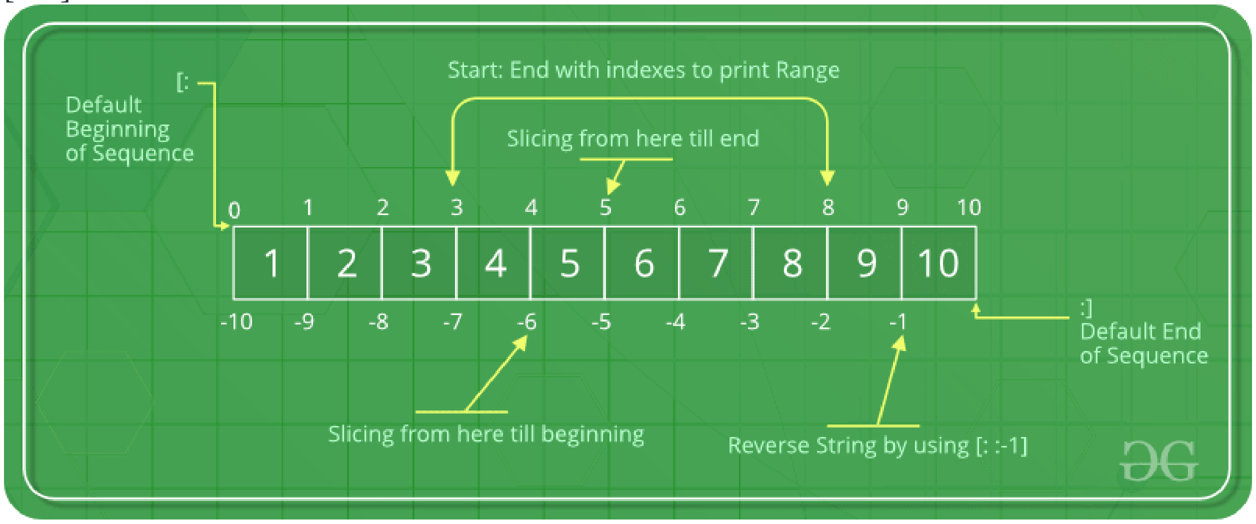
The popped element is: 3

The array after popping is: 1 2 1 5

The array after removing is: 2 1 5

e. Slicing of a Array

In Python array, there are multiple ways to print the whole array with all the elements, but to print a specific range of elements from the array, we use Slice operation. Slice operation is performed on array with the use of colon(:). To print elements from beginning to a range use [:Index], to print elements from end use [:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print whole List with the use of slicing operation, use [:]. Further, to print whole array in reverse order, use [::-1].


Start: End with indexes to print Range
[:
Default Beginning of Sequence
Slicing from here till end
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1
:]
Default End of Sequence
Slicing from here till beginning
Reverse String by using [: :-1]

```python
#slicing
import array as arr
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
a = arr.array('i', l)
print("Initial Array: ")
for i in (a):
    print(i, end =" ")
Sliced_array = a[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_array)
Sliced_array = a[5:]
print("\nElements sliced from 5th element till the end: ")
print(Sliced_array)
Sliced_array = a[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_array)
```

Output

```
Initial Array:

1 2 3 4 5 6 7 8 9 10

Slicing elements in a range 3-8:

array('i', [4, 5, 6, 7, 8])

Elements sliced from 5th element till the end:

array('i', [6, 7, 8, 9, 10])

Printing all elements using slice operation:

array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

f. Searching element in a Array

In order to search an element in the array we use a python in-built index() method. This function returns the index of the first occurrence of value mentioned in arguments.

```python
#searching of elements in array
import array
arr = array.array('i', [1, 2, 3, 1, 2, 5])
print ("The new created array is : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")
print ("\r")
# using index() to print index of 1st occurrence of 2
print ("The index of 1st occurrence of 2 is : ", end ="")
print (arr.index(2))
# using index() to print index of 1st occurrence of 1
print ("The index of 1st occurrence of 1 is : ", end ="")
print (arr.index(1))
```

Output:

The new created array is: 1 2 3 1 2 5

The index of 1st occurrence of 2 is: 1

The index of 1st occurrence of 1 is: 0

g. Updating Elements in a Array

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

```
#updating elements in array
import array
arr = array.array('i', [1, 2, 3, 1, 2, 5])
print ("Array before updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")
print ("\r")
# updating a element in a array
arr[2] = 6
print("Array after updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")
print()
# updating a element in a array
arr[4] = 8
print("Array after updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")
```

Output:

```
Array before updation: 1 2 3 1 2 5

Array after updation: 1 2 6 1 2 5

Array after updation: 1 2 6 1 8 5
```

We can also create 1D arrays in different ways

Creating 1d list Using Naive methods

```
N = 5
ar = [0]*N
print(ar)
```

Output:
[0, 0, 0, 0, 0]

Creating a 1d list using List Comprehension

```
N = 5
arr = [0 for i in range(N)]
print(arr)
```

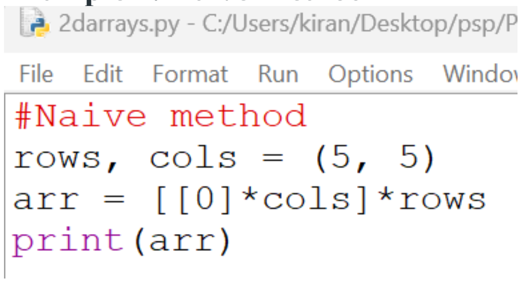
Output:
[0, 0, 0, 0, 0]

2D Arrays

Python provides many ways to create 2-dimensional lists/arrays. However one must know the differences between these ways because they can create complications in code that can be very difficult to trace out.

a. **Creating a 2-D list**

Example 1: Naive Method

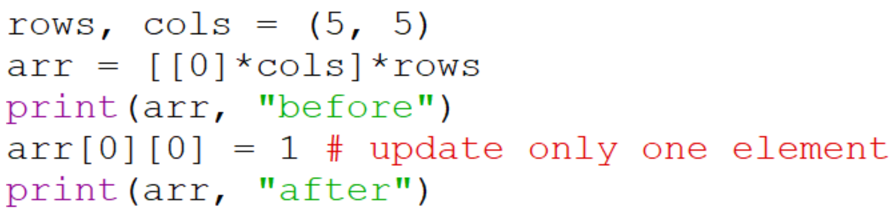

2darrays.py - C:/Users/kiran/Desktop/psp/P
File Edit Format Run Options Windo

```
#Naive method
rows, cols = (5, 5)
arr = [[0]*cols]*rows
print(arr)
```

Output
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

Explanation:
Here we are multiplying the number of columns and hence we are getting the 1-D list of size equal to the number of columns and then multiplying it with the number of rows which results in the creation of a 2-D list.

Something to be careful:
Using this method can sometimes cause unexpected behaviours. In this method, each row will be referencing the same column. This means, even if we update only one element of the array, it will update same column in our array.



```
rows, cols = (5, 5)
arr = [[0]*cols]*rows
print(arr, "before")
arr[0][0] = 1 # update only one element
print(arr, "after")
```

Output

```
([[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]], 'before')

([[1, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 0, 0, 0, 0]], 'after')
```

Example 2: Using List Comprehension

```
#using list comprehension
rows, cols = (5, 5)
arr = [[0 for i in range(cols)] for j in range(rows)]
print(arr)
```

Output

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Explanation:
Here we are basically using the concept of list comprehension and applying loop for a list inside a list and hence creating a 2-D list.

Example 3: Using empty list

```
#using empty list
arr=[]
rows, cols=5,5
for i in range(rows):
    col = []
    for j in range(cols):
        col.append(0)
    arr.append(col)
print(arr)
```

Output

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Explanation:
Here we are appending zeros as elements for number of columns times and then appending this 1-D list into the empty row list and hence creating the 2-D list.

Example:

```
rows, cols = (5, 5)
arr = [[0]*cols]*rows
arr[0][0] = 1
for row in arr:
    print(row)
arr = [[0 for i in range(cols)] for j in range(rows)]
arr[0][0] = 1
for row in arr:
    print(row)
```

Output

```
[1, 0, 0, 0, 0]

[1, 0, 0, 0, 0]

[1, 0, 0, 0, 0]

[1, 0, 0, 0, 0]

[1, 0, 0, 0, 0]

[1, 0, 0, 0, 0]

[0, 0, 0, 0, 0]

[0, 0, 0, 0, 0]

[0, 0, 0, 0, 0]

[0, 0, 0, 0, 0]
```

Some other ways to create 2D arrays

```
A = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
for i in A:
   for j in i:
      print(j,end = " ")
   print()
```

b. Creating 2D Arrays:

Student= [[72, 85, 87, 90, 69], [80, 87, 65, 89, 85], [96, 91, 70, 78, 97], [90, 93, 91, 90, 94], [57, 89, 82, 69, 60]]

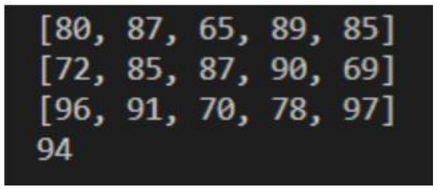
print(Student[1]) # print all elements of index 1

print(Student[0]) # print all elements of index 0

print(Student[2]) # print all elements of index 2

print(Student[3][4]) # it defines the 3rd index and 4 position of the data element.

Output:

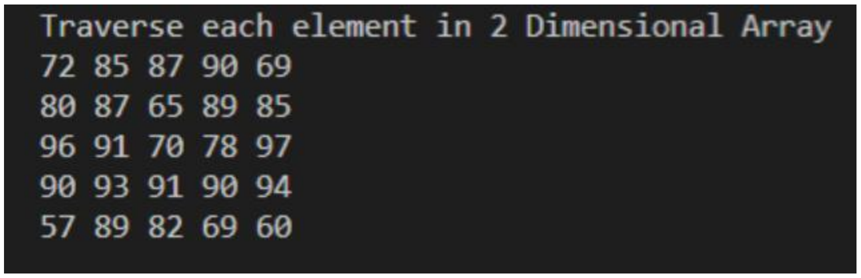


```
[80, 87, 65, 89, 85]
[72, 85, 87, 90, 69]
[96, 91, 70, 78, 97]
94
```

c. **Traversing the element in 2D (two dimensional)**

```
Student_dt = [ [72, 85, 87, 90, 69], [80, 87, 65, 89, 85], [96, 91, 70, 78, 97], [90, 93, 91, 90, 94], [ 57, 89, 82, 69, 60] ]
for x in Student_dt:  # outer loop
    for i in x:  # inner loop
        print(i, end = " ") # print the elements
    print()
```

Output:



```
Traverse each element in 2 Dimensional Array
72 85 87 90 69
80 87 65 89 85
96 91 70 78 97
90 93 91 90 94
57 89 82 69 60
```

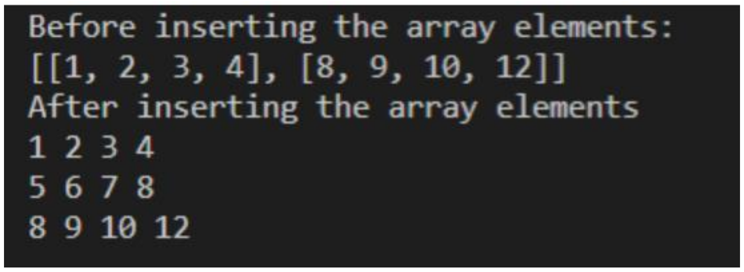
d. **Insert elements in a 2D (Two Dimensional) Array**

We can insert elements into a 2 D array using the **insert()** function that specifies the element' index number and location to be inserted.

```
from array import * # import all package related to the array.
arr1 = [[1, 2, 3, 4], [8, 9, 10, 12]]  # initialize the array elements.
print("Before inserting the array elements: ")
print(arr1)
arr1.insert(1, [5, 6, 7, 8])
print("After inserting the array elements ")
for i in arr1: # Outer loop
    for j in i: # inner loop
```

```
            print(j, end = " ") # print inserted elements.
        print()
```

Output:



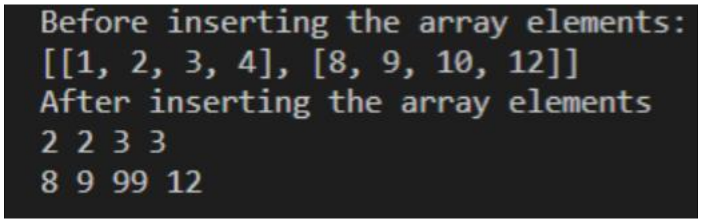
```
Before inserting the array elements:
[[1, 2, 3, 4], [8, 9, 10, 12]]
After inserting the array elements
1 2 3 4
5 6 7 8
8 9 10 12
```

e. **Update elements in a 2 -D (Two Dimensional) Array**

In a 2D array, the existing value of the array can be updated with a new value. In this method, we can change the value as well as the entire index of the array. Let's understand with an example of a 2D array, as shown below.

```
from array import * # import all package related to the array.
arr1 = [[1, 2, 3, 4], [8, 9, 10, 12]]
print("Before inserting the array elements: ")
print(arr1)
arr1[0] = [2, 2, 3, 3]
arr1[1][2] = 99
print("After inserting the array elements ")
for i in arr1: # Outer loop
    for j in i: # inner loop
        print(j, end = " ") # print inserted elements.
    print()
```

Output:



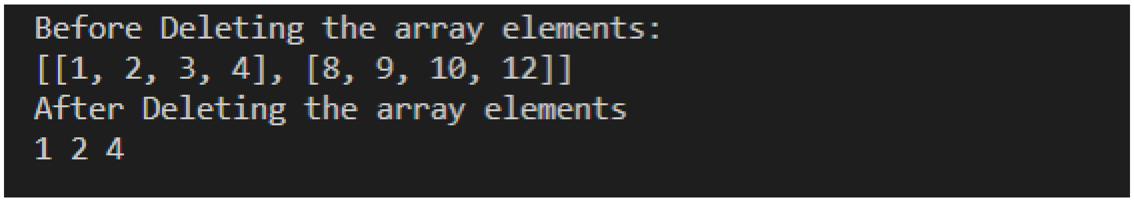
```
Before inserting the array elements:
[[1, 2, 3, 4], [8, 9, 10, 12]]
After inserting the array elements
2 2 3 3
8 9 99 12
```

f. Delete values from a 2D (two Dimensional) array in Python

In a 2- D array, we can remove the particular element or entire index of the array using del() function in Python. Let's understand an example to delete an element.

```
from array import * # import all package related to the array.
arr1 = [[1, 2, 3, 4], [8, 9, 10, 12]]
print("Before Deleting the array elements: ")
print(arr1) # print the arr1 elements.
del(arr1[0][2]) # delete the particular element of the array.
del(arr1[1]) # delete the index 1 of the 2-D array.
print("After Deleting the array elements ")
for i in arr1: # Outer loop
    for j in i: # inner loop
        print(j, end = " ") # print inserted elements.
    print()
```

Output:



```
Before Deleting the array elements:
[[1, 2, 3, 4], [8, 9, 10, 12]]
After Deleting the array elements
1 2 4
```

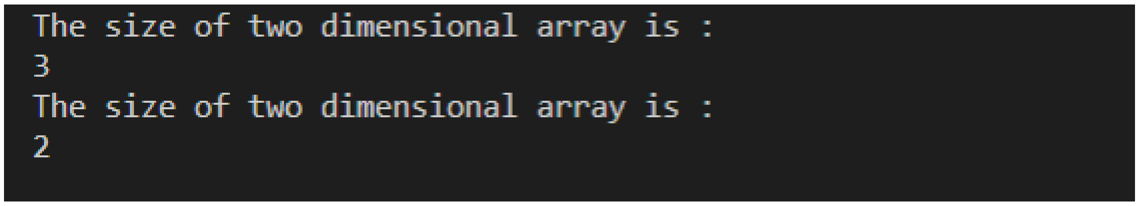
g. Size of a 2D array

A len() function is used to get the length of a two-dimensional array. In other words, we can say that a len() function determines the total index available in 2-dimensional arrays.

Let's understand the len() function to get the size of a 2-dimensional array in Python.

```
array_size = [[1, 3, 2],[2,5,7,9], [2,4,5,6]] # It has 3 index
print("The size of two dimensional array is : ")
print(len(array_size)) # it returns 3
array_def = [[1, 3, 2], [2, 4, 5, 6]] # It has 2 index
print("The size of two dimensional array is : ")
print(len(array_def)) # it returns 2
```

Output:



```
The size of two dimensional array is :
3
The size of two dimensional array is :
2
```

Write a program to print the sum of the arrays in Python.

Method-1:

```
import numpy as np
#define 2 different arrays
arr1 = np.array([1,2,3,4])
arr2 = np.array([1,2,3,4])
res = arr1 + arr2
res
```

```
Output:
array([2, 4, 6, 8])
```

Method-2:

```
import numpy as np
arr1 = np.array([1,2,3,4])
arr2 = np.array([1,2,3,4])
np.add(arr1,arr2)
```

```
Output:
array([2, 4, 6, 8])
```

Method-3

```
# Program to add two matrices using nested loop

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

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


result = [[0,0,0],
        [0,0,0],
        [0,0,0]]

# iterate through rows
for i in range(len(X)):
# iterate through columns
    for j in range(len(X[0])):
        result[i][j] = X[i][j] + Y[i][j]

for r in result:
    print(r)
```

Output

```
[10, 10, 10]

[10, 10, 10]

[10, 10, 10]
```

Method-4:

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

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

result = [[X[i][j] + Y[i][j]  for j in range
(len(X[0]))] for i in range(len(X))]

for r in result:
    print(r)
```

Output

```
[10, 10, 10]

[10, 10, 10]

[10, 10, 10]
```

Method-5:

```
import numpy as np

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

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

result = np.array(X) + np.array(Y)

print(result)
```

Output:

```
[[10 10 10]

[10 10 10]

[10 10 10]]
```

Method-6:

```
from sympy import Matrix
 X = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
 Y = [[9, 8, 7],
     [6, 5, 4],
     [3, 2, 1]]
 # Create Matrix objects from the lists
matrix_x = Matrix(X)
matrix_y = Matrix(Y)
 result = matrix_x + matrix_y
 # Print the result
print(result)
```

This will print the following output:

```
Matrix([
[10, 10, 10],
[10, 10, 10],
[10, 10, 10]])
```