UNIT – 2

A program's **control flow** is the order in which the program's code executes.

The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Python has *three* types of control structures:

- **Sequential** - default mode
- **Selection** - used for decisions and branching
- **Repetition** - used for looping, i.e., repeating a piece of code multiple times.

1. Sequential

Sequential statements are a set of statements whose execution process happens in a sequence.

The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

```
## This is a Sequential statement
 a=20
b=10
c=a-b
print("Subtraction is : ",c)
```

Example of sequential statement

2. Selection/Decision control statements

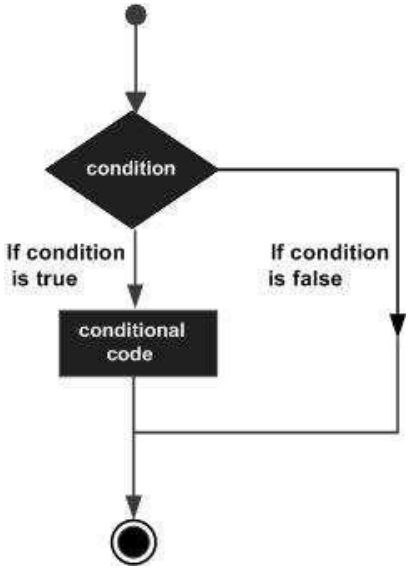
In Python, the selection statements are also known as *Decision control statements* or *branching statements*.

Decision Making:

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce True or False as outcome. You need to determine which action to take and which statements to execute if outcome is True or False otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages:



Python programming language assumes any non-zero and non-null values as True, and if it is either zero or null, then it is assumed as False value.

Statement	Description
if statements	**if statement** consists of a boolean expression followed by one or more statements.
if...else statements	**if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

The selection statement allows a program to test several conditions and execute instructions based on which condition is true.

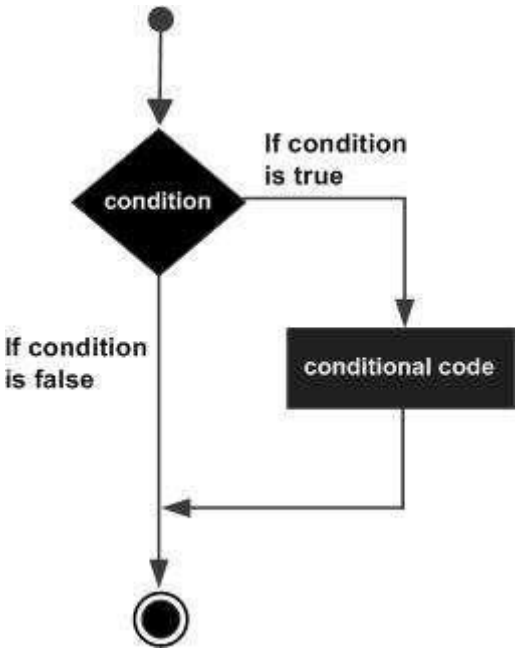
Some decision control statements are:

- `if`
- `if-else`
- `nested if`
- `if-elif-else`

The *if* Statement

It is similar to that of other languages. The **if** statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

`if` – It help us to run a particular code, but only when a certain condition is met or satisfied. A `if` only has one condition to check.



Syntax:

```
if condition:
    statements
```

First, the condition is tested. If the condition is True, then the statements given after colon (:) are executed. We can write one or more statements after colon (:).

Example:

```
a=10
b=15
if a < b:
print "B is big" print
    "B value is",b
```

Output:

B is big

B value is 15

The *if ... else* statement

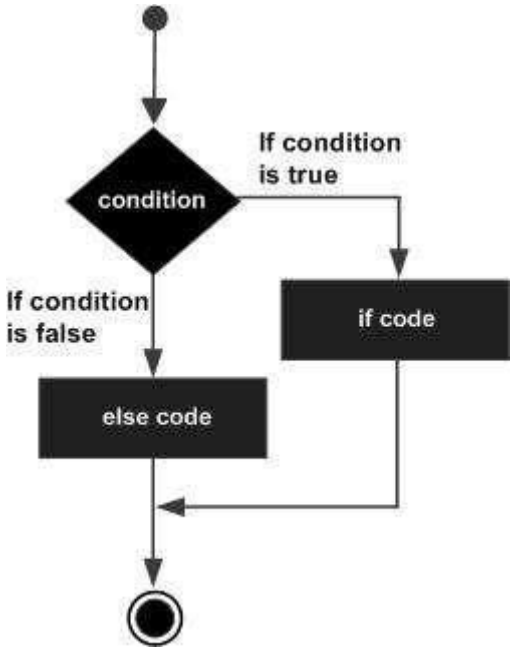
An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The *else* statement is an optional statement and there could be at most only one **else** statement following **if**.

`if-else` – The `if-else` statement evaluates the condition and will execute the body of `if` if the test condition is `True`, but if the condition is `False`, then the body of `else` is executed.

Syntax:

```
if condition:
    statement(s)
else:
statement(s)
```



Example:

```
a=48
b=34
if a < b:
print "B is big" print
   "B value is", b
else:
print "A is big" print
   "A value is", a
print "END"
```

Output:

A is big

A value is 48

END

Nested `if`: Nested `if` statements are an `if` statement inside another `if` statement.

Depiction of nested if statement

In the following code example, we can see first `if` condition checks `a` is greater than `b`. If yes, then we've another `if` condition that checks `a` is also greater than `c`. If yes, then `if` body will be executed.

```
a = 20
b = 10
c = 15
if a > b:
  if a > c:
    print("a value is big")
  else:
     print("c value is big")
elif b > c:
   print("b value is big")
else:
    print("c is big")
```

The *elif* Statement

The **elif** statement allows you to check multiple expressions for True and execute a block of code as soon as one of the conditions evaluates to True.

`if-elif-else`: The `if-elif-else` statement is used to conditionally execute a statement or a block of statements.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if**.

Syntax:

```
if condition1:
   statement(s)
elif condition2:
   statement(s)
 else:
statement(s)
```

Example:

```
a=20
b=10
c=30
if a >= b and a >= c:
   print "a is big"
elif b >= a and b >= c:
   print "b is big"
else:
print "c is big"
```

Output:

c is big

3. Repetition

A **repetition statement** is used to repeat a group(block) of programming instructions.

In Python, we generally have two loops/repetitive statements:

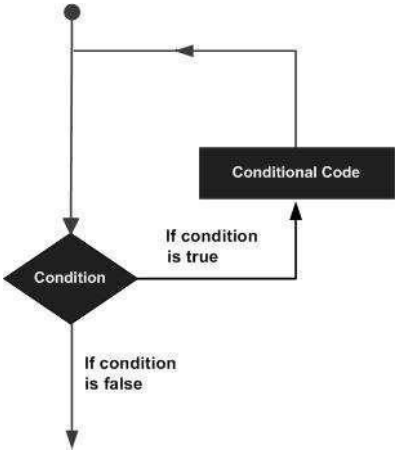
- `for` loop

- `while` loop

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement:



Python programming language provides following types of loops to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loops	You can use one or more loop inside any another while, for loop.

The *while* Loop

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is True.

Syntax

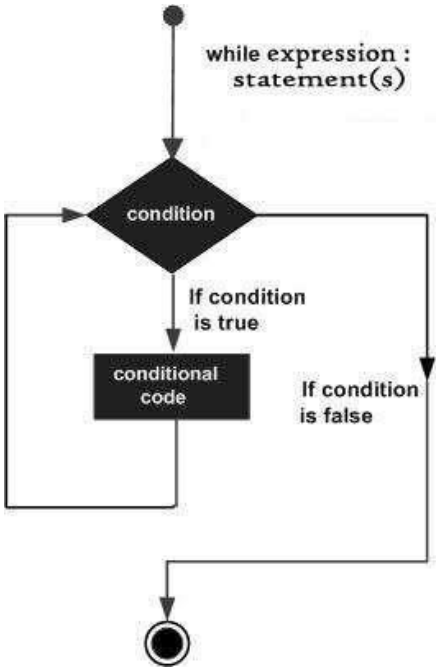
The syntax of a **while** loop in Python programming language is:

```
while expression:
statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements.

The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.



Example-1:

```
i=1
while i < 4:
print i i+=1
print "END"
```

Example-2:

```
i=1
while i < 4:
print i i+=1
print "END"
```

Output-1:

```
1
END
2
END
3
END
```

Output-2:

```
1
2
3
END
```

Q) Write a program to display factorial of a given number.

Program:

```
n=input("Enter the number:
") f=1
while n>0:
    f=f
    *n
    n=
print "Factorial is",f
    n-1
```

Output:

Enter the number: 5

Factorial is 120

The *for* loop:

The *for* loop is useful to iterate over the elements of a sequence. It means, the *for* loop can be used to execute a group of statements repeatedly depending upon the number of elements in the sequence. The *for* loop can work with sequence like string, list, tuple, range etc.

The syntax of the for loop is given below:

```
for var in
sequence:
statement (s)
```

The first element of the sequence is assigned to the variable written after „for" and then the statements are executed. Next, the second element of the sequence is assigned to the variable and then the statements are executed second time. In this way, for each element of the sequence, the statements are executed once. So, the *for* loop is executed as many times as there are number of elements in the sequence.

Example-1:

```
for i range(1,5):
print i
print "END"
```

Example-2:

```
for i range(1,5):
        pri
nt i print
"END"
```

```
1
END
2
END
3
END
```

Output-1:

```
1
2
3
END
```

Output-2:

Example-3:

```
name= "python"
for letter in name:
        print letter
```

Output-3:

```
p
y
t
h
o
n
```

Example-4:

```
for x in range(10,0,-1):
        print x,
```

Output-4:

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

Q) Write a program to display the factorial of given number.

Program:

```
n=input("Enter the number: ")
f=1
for i in range(1,n+1):
    f=f
    *i
print "Factorial is",f
```

Output:

Enter the number: 5

Factorial is 120

Nested Loop:

It is possible to write one loop inside another loop. For example, we can write a for loop inside a while loop or a for loop inside another for loop. Such loops are called “nested loops”.

Example-1:

```
for i in range(1,6):
    for j in
        range(1,6)
        : print j,
    print ""
```

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Example-2:

```
for i in range(1,6):
    for j in
        range(1,6)
        : print "*",
    print ""
```

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

Example-3:

```
for i in range(1,6):
    for j in range(1,6):
        if i==1 or j==1 or i==5 or
            j==5: print "*",
        else:
            print
    " ", print
    ""
```

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

Example-4:

```
for i in range(1,6):
    for j in
        range(1,6): if
        i==j:
            print "*",
        elif i==1 or j==1 or i==5 or
            j==5: print "*",
        else:
            print
    " ", print
    ""
```

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

Example-5:

```
for i in range(1,6):
    for j in
        range(1,6): if
        i==j:
            print "$",
        elif i==1 or j==1 or i==5 or
            j==5: print "*",
        else:
            print
    " ", print
    ""
```

```
$ * * * *
* $     *
*   $   *
*     $ *
* * * * $
```

Example-6:

```
for i in range(1,6):
    for j in range(1,4):
        if i==1 or j==1 or
            i==5: print "*",
        else:
            print
    " ", print
    ""
```

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

Example-7:

```
for i in range(1,6):
    for j in
        range(1,4): if
        i==2 and
        j==1:
            print "*",
        elif i==4 and
            j==3: print
            "*",
        elif i==1 or i==3 or
            i==5: print "*",
        else:
            print
    " ", print
    ""
```

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

Example-8:

```
for i in range(1,6):
    for j in range(1,4):
        if i==1 or j==1 or i==3 or
            i==5: print "*",
        else:
            print
    " ", print
    ""
```

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

Example-9:

```
for i in range(1,6):
    for c in
        range(i,6):
        print "",
    for j in
            range(1,i+1)
            : print "*",
    print ""
```

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

Example-10:

```
for i in range(1,6):
    for j in
        range(1,i+1):
        print j,
    print ""
```

```
    - -
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Example-11:

```
a=1
for i in range(1,5):
    for j in
        range(1,i+1):
        print a,
        a=a
        +1
    print
    ""
```

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

1) Write a program for print given number is prime number or not using for loop. Program:

```
n=input("Enter the n
value") count=0
for i in range(2,n):
    if n%i==0:
        count=count
        +1 break
if count==0:
    print "Prime
Number" else:
    print "Not Prime Number"
```

Output:

```
Enter n value: 17
 Prime Number
```

2) Write a program print Fibonacci series and sum the even numbers. Fibonacci series is 1,2,3,5,8,13,21,34,55

```
n=input("Enter n value ")
f0=1
f1=2
sum=f1
print f0,f1,
for i in range(1,n-1):
    f2=f0+f1
print f2,
    f0=f1
    f1=f2
    if f2%2==0:
        sum+=f2
print "\nThe sum of even Fibonacci numbers is", sum
```

Output:

Enter n value 10

1 2 3 5 8 13 21 34 55 89

The sum of even fibonacci numbers is 44

3) Write a program to print n prime numbers and display the sum of prime numbers.

Program:

```
n=input("Enter the range: ")
sum=0
for num in range(1,n+1):
    for i in
        range(2,num): if
        (num % i) == 0:
            break
    else:
        print
        num, sum
print "\nSum of prime numbers
is",sum num
```

Output:

Enter the range: 21

1 2 3 5 7 11 13 17 19

Sum of prime numbers is 78

4) Using a for loop, write a program that prints out the decimal equivalents of 1/2, 1/3, 1/4, . . . ,1/10

Program:

```
for i in range(1,11):
    print "Decimal Equivalent of 1/",i,"is",1/float(i)
```

Output:

Decimal Equivalent of 1/ 1 is 1.0

Decimal Equivalent of 1/ 2 is 0.5

Decimal Equivalent of 1/ 3 is 0.333333333333

Decimal Equivalent of 1/ 4 is 0.25

Decimal Equivalent of 1/ 5 is 0.2

Decimal Equivalent of 1/ 6 is 0.166666666667 Decimal Equivalent of 1/ 7 is 0.142857142857 Decimal Equivalent of 1/ 8 is 0.125

Decimal Equivalent of 1/ 9 is 0.111111111111

Decimal Equivalent of 1/ 10 is 0.1

5) Write a program that takes input from the user until the user enters -1. After display the sum of numbers.

Program:

```
sum=0
while True:
    n=input("Enter the number: ") if n==-1:
        br
    eak
    else:
        sum+=n
print "The sum is",sum
```

Output:

Enter the number: 1

Enter the number: 5

Enter the number: 6

Enter the number: 7

Enter the number: 8

Enter the number: 1

Enter the number: 5

Enter the number: -1

The sum is 33

6) Write a program to display the following sequence.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Program:

```
ch='A'
for j in
    range(1,27):
    print ch,
    ch=chr(ord(ch)
    +1)
```

7) Write a program to display the following sequence. A

A B

A B C

A B C D

A B C D

E

Program:

```
for i in
   range(1,6):
      for j in
         range(1,i+1):
         print ch,
         ch=chr(ord(ch)
         +1)
      print ""
```

8) Write a program to display the following sequence. A

B C

D E

F

G H I J

K L M N O

Program:

```
ch='A'
for i in range(1,6):
   for j in
      range(1,i+1):
      print ch,
      ch=chr(ord(ch)
      +1)
   print ""
```

9) Write a program that takes input string user and display that string if string contains at least one Uppercase character, one Lowercase character and one digit.

Program:

```
pwd=input("Enter the
password:") u=False
l=Fal
se
d=Fa
lse
for i in range(0,len(pwd)):
    if
    pwd[i].isupper
    r(): u=True
  elif
    pwd[i].islower(
    ): l=True
  elif
    pwd[i].isdigit(
    ): d=True
if u==True and l==True and
  d==True: print
  pwd.center(20,"*")
else:
  print "Invalid Password"
```

Output-1:

Enter the password:"Mothi556"

******Mothi556******

Output-2:

Enter the password:"mothilal"

Invalid Password

10) Write a program to print sum of digits. Program:

```
n=input("Enter the number: ")
sum=0
while
n>0:
  sum
  +=r
  n=n/
  10
print "sum is",sum
```

Output:

Enter the number: 123456789

sum is 45

11) Write a program to print given number is Armstrong or not. Program:

```
n=input("Enter the number: ")
sum
=0
t=n
while n>0:
    r=n%10
    sum+=r*
    r*r
    n=n/10
if sum==t:
    print
"ARMSTRONG" else:
    print "NOT ARMSTRONG"
```

Output:

Enter the number: 153

ARMSTRONG

12) Write a program to take input string from the user and print that string after removing ovals.

Program:

```
st=input("Enter the
string:") st2=""
for i in st:
    if i not in
        "aeiouAEIOU":
        st2=st2+i
print st2
```

Output:

Enter the string:"Welcome to you"

Wlcm t y

Arrays:

An array is an object that stores a group of elements of same datatype.

- Arrays can store only one type of data. It means, we can store only integer type elements or only float type elements into an array. But we cannot store one integer, one float and one character type element into the same array.
- Arrays can increase or decrease their size dynamically. It means, we need not declare the size of the array. When the elements are added, it will increase its size and when the elements are removed, it will automatically decrease its size in memory.

Advantages:

- Arrays are similar to lists. The main difference is that arrays can store only one type of elements; whereas, lists can store different types of elements. When dealing with a huge number of elements, arrays use less memory than lists and they offer faster execution than lists.
- The size of the array is not fixed in python. Hence, we need not specify how many elements we are going to store into an array in the beginning.
- Arrays can grow or shrink in memory dynamically (during runtime).
- Arrays are useful to handle a collection of elements like a group of numbers or characters.
- Methods that are useful to process the elements of any array are available in „array" module.

Creating an array:

Syntax:

arrayname = array(type code, [elements])

The type code „i" represents integer type array where we can store integer numbers. If the type code is „f" then it represents float type array where we can store numbers with decimal point.

Type code	Description	Minimum size in bytes
„b"	Signed integer	1
„B"	Unsigned integer	1
„i"	Signed integer	2
„I"	Unsigned integer	2
„l"	Signed integer	4

„L“	Unsigned integer	4
„f“	Floating point	4
„d“	Double precision floating point	8
„u“	Unicode character	2

Example:

The type code character should be written in single quotes. After that the elements should be written in inside the square braces [ ] as

a = array („i“, [4,8,-7,1,2,5,9])

Importing the Array Module:

There are two ways to import the array module into our program.

The first way is to import the entire array module using import statement as,

```
import array
```

when we import the array module, we are able to get the „array" class of that module that helps us to create an array.

```
a = array.array('i', [4,8,-7,1,2,5,9] )
```

Here the first „array" represents the module name and the next „array" represents the class name for which the object is created. We should understand that we are creating our array as an object of array class.

The next way of importing the array module is to write:

```
from array import *
```

Observe the „*" symbol that represents „all". The meaning of this statement is this: import all (classes, objects, variables, etc) from the array module into our program. That means significantly importing the „array" class of „array" module. So, there is no need to mention the module name before our array name while creating it. We can create array as:

```
a = array('i', [4,8,-7,1,2,5,9] )
```

Example:

```
from array import *
arr = array(„i", [4,8,-7,1,2,5,9])
for i in arr:
    print i,
```

Output:

```
4 8 -7 1 2 5 9
```

Indexing and slicing of arrays:

An *index* represents the position number of an element in an array. For example, when we creating following integer type array:

```
a = array('i', [10,20,30,40,50] )
```

Python interpreter allocates 5 blocks of memory, each of 2 bytes size and stores the elements 10, 20, 30, 40 and 50 in these blocks.

10	20	30	40	50
a[0]	a[1]	a[2]	a[3]	a[4]

Example:

```
from array import *
a=array('i', [10,20,30,40,50,60,70])
print "length is",len(a)
print " 1st position character", a[1]
print "Characters from 2 to 4", a[2:5]
print "Characters from 2 to end", a[2:]
print "Characters from start to 4", a[:5]
print "Characters from start to end", a[:]
a[3]=45
a[4]=55
print "Characters from start to end after modifications ",a[:]
```

Output:

```
length is 7
1st position character 20
Characters from 2 to 4 array('i', [30, 40, 50])
Characters from 2 to end array('i', [30, 40, 50, 60, 70])
Characters from start to 4 array('i', [10, 20, 30, 40, 50])
Characters from start to end array('i', [10, 20, 30, 40, 50, 60, 70])
Characters from start to end after modifications array('i', [10, 20, 30, 45, 55, 60, 70])
```

Array Methods:

Method	Description
a.append(x)	Adds an element x at the end of the existing array a.
a.count(x)	Returns the number of occurrences of x in the array a.
a.extend(x)	Appends x at the end of the array a. „x‟ can be another array or iterable object.
a.fromfile(f,n)	Reads n items from from the file object f and appends at the end of the array a.
a.fromlist(*l*)	Appends items from the *l* to the end of the array. *l* can be any list or iterable object.
a.fromstring(s)	Appends items from string s to end of the array a.

a.index(x)	Returns the position number of the first occurrence of x in the array. Raises „ValueError‟ if not found.
a.pop(x)	Removes the item x from the array a and returns it.
a.pop( )	Removes last item from the array a
a.remove(x)	Removes the first occurrence of x in the array. Raises „ValueError‟ if not found.
a.reverse( )	Reverses the order of elements in the array a.
a.tofile(f)	Writes all elements to the file f.
a.tolist( )	Converts array „a‟ into a list.
a.tostring( )	Converts the array into a string.

1) Write a program to perform stack operations using array.

Program:

```
import sys
from array
import *
a=array('i',[])
while True:
  print "\n1.PUSH 2.POP 3.DISPLAY 4.EXIT"
  ch=input("Enter Your Choice:
  ") if ch==1:
     ele=input("Enter
     element: ") a.append(ele)
     print
  "Inserted" elif
  ch==2:
     if len(a)==0:
        print "\t STACK IS
     EMPTY" else:
        print "Deleted element is",
  a.pop( ) elif ch==3:
     if len(a)==0:
        print "\t STACK IS
     EMPTY" else:
        print "\tThe Elements in Stack
        is", for i in a:
           prin
  t i, elif
  ch==4:
     sys.ex
  it() else:
     print "\tINVALID CHOICE"
```

Output:

```
1.PUSH 2.POP 3.DISPLAY 4.EXIT
Enter Your Choice: 1
Enter element: 15
Inserted
1.PUSH 2.POP 3.DISPLAY 4.EXIT
Enter Your Choice: 1
Enter element: 18
Inserted
1.PUSH 2.POP 3.DISPLAY 4.EXIT
Enter Your Choice: 3
        The Elements in Stack is 15 18
1.PUSH 2.POP 3.DISPLAY 4.EXIT
Enter Your Choice: 2
Deleted element is 18
```

2) Write a program to perform queue operations using array. Program:

```
import sys
from array
import *
a=array('i',[])
while True:
    print "\n1.INSERT 2.DELETE 3.DISPLAY 4.EXIT"
    ch=input("Enter Your Choice:
    ") if ch==1:
        ele=input("Enter
        element: ") a.append(ele)
    elif ch==2:
        if len(a)==0:
            print "\t QUEUE IS
        EMPTY" else:
            print "Deleted element is",
            a[0] a.remove(a[0])
    elif ch==3:
        if len(a)==0:
            print "\t QUEUE IS
        EMPTY" else:
            print "\tThe Elements in Queue
            is", for i in a:
                prin
    t i, elif
    ch==4:
        sys.ex
    it() else:
        print "\tINVALID CHOICE"
```

Output:

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter Your Choice: 1

Enter element: 12

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter Your Choice: 1

Enter element: 13

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter Your Choice: 1

Enter element: 14

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter Your Choice: 3

The Elements in Queue is 12 13 14

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter Your Choice: 2

Deleted element is 12

Python break and continue

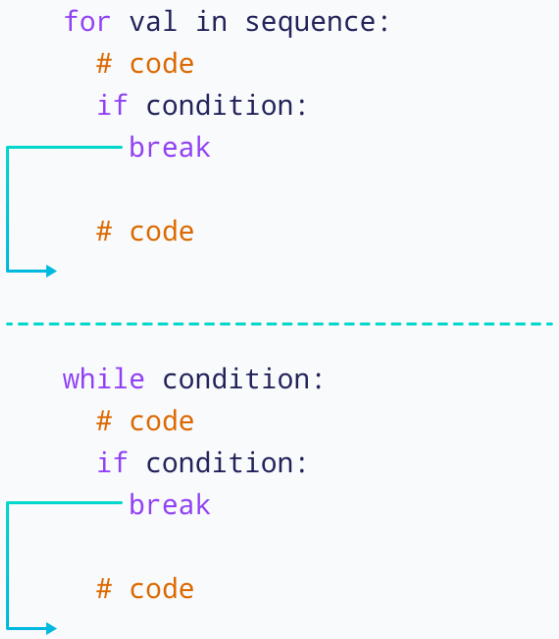
Python break Statement

The break statement is used to terminate the loop immediately when it is encountered.

The syntax of the break statement is:

break

Working of Python break Statement



Working of the break statement

The working of break statement in for loop and while loop is shown above.

Python break Statement with for Loop

We can use the break statement with the for loop to terminate the loop when a certain condition is met. For example,

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Run Code

Output

```
0
1
2
```

In the above example, we have used the for loop to print the value of i. Notice the use of the break statement,

```
if i == 3:
    break
```

Here, when i is equal to 3, the break statement terminates the loop. Hence, the output doesn't include values after 2.

Python break Statement with while Loop

We can also terminate the while loop using the break statement. For example,

```
# program to find first 5 multiples of 6
i = 1
while i <= 10:
    print('6 * ',(i), '=',6 * i)
    if i >= 5:
        break
     i = i + 1
```

Run Code

Output

```
6 *  1 = 6
6 *  2 = 12
6 *  3 = 18
6 *  4 = 24
```

6 * 5 = 30

In the above example, we have used the while loop to find the first 5 multiples of 6. Here notice the line,

```
if i >= 5:
   break
```

This means when i is greater than or equal to 5, the while loop is terminated.

Python continue Statement

The continue statement is used to skip the current iteration of the loop and the control flow of the program goes to the next iteration.

The syntax of the continue statement is:

```
Continue
```

Working of Python continue Statement

```
for val in sequence:
    # code
    if condition:
        continue

    # code
-----------------------------------
while condition:
    # code
    if condition:
        continue

    # code
```

How continue statement works in python

The working of the continue statement in for and while loop is shown above.

Python continue Statement with for Loop

We can use the continue statement with the for loop to skip the current iteration of the loop. Then the control of the program jumps to the next iteration. For example,

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

Run Code

Output

```
0
1
2
4
```

In the above example, we have used the for loop to print the value of i. Notice the use of the continue statement,

```
if i == 3:
    continue
```

Here, when i is equal to **3**, the continue statement is executed. Hence, the value **3** is not printed to the output.

Python continue Statement with while Loop

In Python, we can also skip the current iteration of the while loop using the continue statement.

For example,

```
# program to print odd numbers from 1 to 10
num = 0
while num < 10:
   num += 1
    if (num % 2) == 0:
      continue
   print(num)
```

Run Code

Output

```
1
3
5
7
9
```

In the above example, we have used the while loop to print the odd numbers between 1 to 10.

Notice the line,

```
if (num % 2) == 0:
   continue
```

Here, when the number is even, the continue statement skips the current iteration and starts the next iteration.

Try and Except in Python

The *try except* statement can handle exceptions. Exceptions may happen when you run a program. Exceptions are errors that happen during execution of the program. Python won't tell you about errors like syntax errors (grammar faults), instead it will abruptly stop.

An abrupt exit is bad for both the end user and developer.

Instead of an emergency halt, you can use a try except statement to properly deal with the problem. An emergency halt will happen if you do not properly handle exceptions.

Related course: Complete Python Programming Course & Exercises

What are exceptions in Python?

Python has built-in exceptions which can output an error. If an error occurs while running the program, it's called an exception.

If an exception occurs, the type of exception is shown. Exceptions needs to be dealt with or the program will crash. To handle exceptions, the try-catch block is used.

Some exceptions you may have seen before

are FileNotFoundError, ZeroDivisionError or ImportError but there are many more.

All exceptions in Python inherit from the class BaseException. If you open the Python interactive shell and type the following statement it will list all built-in exceptions:

```
>>> dir(builtins)
```

The idea of the try-except clause is to handle exceptions (errors at runtime). The syntax of the try-except block is:

```
try:
   <do something>
except Exception:
   <handle the error>
```

The idea of the *try-except block* is this:

- **try**: the code with the exception(s) to catch. If an exception is raised, it jumps straight into the except block.
- **except**: this code is only executed *if an exception occured* in the try block. The except block is required with a try block, even if it contains only the pass statement.

It may be combined with the **else** and **finally** keywords.

- **else**: Code in the else block is only executed if no exceptions were raised in the try block.
- **finally**: The code in the finally block is always executed, regardless of if a an exception was raised or not.

Catching Exceptions in Python

The try-except block can handle exceptions. This prevents abrupt exits of the program on error. In the example below we purposely raise an exception.

```
try:
   1 / 0
except ZeroDivisionError:
   print('Divided by zero')

print('Should reach here')
```

After the except block, the program continues. Without a try-except block, the last line wouldn't be reached as the program would crash.

```
$ python3 example.py

Divided by zero
Should reach here
```

In the above example we catch the specific exception ZeroDivisionError. You can handle any exception like this:

```
try:
    open("fantasy.txt")
except:
    print('Something went wrong')

print('Should reach here')
```

You can write different logic for each type of exception that happens:

```
try:
    # your code here
except FileNotFoundError:
    # handle exception
except IsADirectoryError:
    # handle exception
except:
    # all other types of exceptions

print('Should reach here')
```

Related course: Complete Python Programming Course & Exercises

try-except

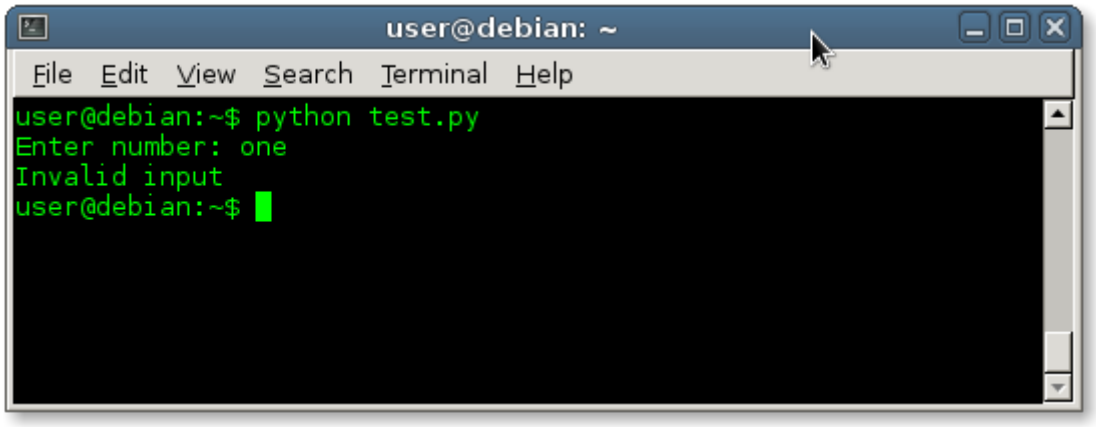
Lets take do a real world example of the try-except block.

The program asks for numeric user input. Instead the user types characters in the input box. The program normally would crash. But with a try-except block it can be handled properly.

The *try except* statement prevents the program from crashing and properly deals with it.

```
try:
    x = input("Enter number: ")
    x = x + 1
    print(x)
except:
    print("Invalid input")
```

Entering invalid input, makes the program continue normally:



The try except statement can be extended with the finally keyword, this will be executed if no exception is thrown:

```
finally:
    print("Valid input.")
```

The program continues execution if no exception has been thrown.

There are different kinds of exceptions: ZeroDivisionError, NameError, TypeError and so on.

Sometimes modules define their own exceptions.

The try-except block works for function calls too:

```
def fail():
    1 / 0


try:
    fail()
except:
    print('Exception occured')
```

```
print('Program continues')
```

This outputs:

```
$ python3 example.py

Exception occured
Program continues
```

If you are a beginner, then I highly recommend this book.

try finally

A try-except block can have the finally clause (optionally). The finally clause is always executed. So the general idea is:

```
try:
   <do something>
except Exception:
   <handle the error>
finally:
   <cleanup>
```

For instance: if you open a file you'll want to close it, you can do so in the finally clause.

```
try:
   f = open("test.txt")
except:
   print('Could not open file')
finally:
   f.close()

print('Program continue')
```

try else

The else clause is executed if and only if no exception is raised. This is different from the finally clause that's always executed.

```
try:
    x = 1
except:
    print('Failed to set x')
else:
    print('No exception occured')
finally:
    print('We always do this')
```

Output:

```
No exception occured
We always do this
```

You can catch many types of exceptions this way, where *the else clause* is executed only if no exception happens.

```
try:
    lunch()
except SyntaxError:
    print('Fix your syntax')
except TypeError:
    print('Oh no! A TypeError has occured')
except ValueError:
    print('A ValueError occured!')
except ZeroDivisionError:
    print('Did by zero?')
else:
    print('No exception')
finally:
    print('Ok then')
```

Raise Exception

Exceptions are raised when an error occurs. But in Python you can also force an exception to occur with the keyword raise.

Any type of exception can be raised:

```
>>> raise MemoryError("Out of memory")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError: Out of memory
```

```
>>> raise ValueError("Wrong value")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Wrong value
>>>
```

Related course: Complete Python Programming Course & Exercises

Built-in exceptions

A list of Python's Built-in Exceptions is shown below. This list shows the Exception and why it is thrown (raised).

Exception	Cause of Error
AssertionError	if assert statement fails.
AttributeError	if attribute assignment or reference fails.
EOFError	if the input() functions hits end-of-file condition.
FloatingPointError	if a floating point operation fails.
GeneratorExit	Raise if a generator's close() method is called.
ImportError	if the imported module is not found.
IndexError	if index of a sequence is out of range.
KeyError	if a key is not found in a dictionary.

KeyboardInterrupt	if the user hits interrupt key (Ctrl+c or delete).
MemoryError	if an operation runs out of memory.
NameError	if a variable is not found in local or global scope.
NotImplementedError	by abstract methods.
OSError	if system operation causes system related error.
OverflowError	if result of an arithmetic operation is too large to be represented.
ReferenceError	if a weak reference proxy is used to access a garbage collected referent.
RuntimeError	if an error does not fall under any other category.
StopIteration	by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError	by parser if syntax error is encountered.
IndentationError	if there is incorrect indentation.
TabError	if indentation consists of inconsistent tabs and spaces.
SystemError	if interpreter detects internal error.
SystemExit	by sys.exit() function.
TypeError	if a function or operation is applied to an object of incorrect type.
UnboundLocalError	if a reference is made to a local variable in a function or method, but no value has been bound to that variable.

UnicodeError	if a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	if a Unicode-related error occurs during encoding.
UnicodeDecodeError	if a Unicode-related error occurs during decoding.
UnicodeTranslateError	if a Unicode-related error occurs during translating.
ValueError	if a function gets argument of correct type but improper value.
ZeroDivisionError	if second operand of division or modulo operation is zero.

User-defined Exceptions

Python has many standard types of exceptions, but they may not always serve your purpose.
Your program can have your own type of exceptions.
To create a user-defined exception, you have to create a class that inherits from Exception.

```
class LunchError(Exception):
   pass


raise LunchError("Programmer went to lunch")
```

You made a user-defined exception named LunchError in the above code. You can raise this new exception if an error occurs.
Outputs your custom error:

```
 $ python3 example.py
Traceback (most recent call last):
  File “example.py”, line 5, in
    raise LunchError(“Programmer went to lunch”)
main.LunchError: Programmer went to lunch
```

Your program can have many user-defined exceptions. The program below throws exceptions based on a new projects money:

```
class NoMoneyException(Exception):
   pass
```

```
class OutOfBudget(Exception):
    pass

balance = int(input("Enter a balance: "))
if balance < 1000:
    raise NoMoneyException
elif balance > 10000:
    raise OutOfBudget
```

Python String

A String is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once you have created a string, you cannot change it. Strings are used widely in many different applications, such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.

What is a String in Python?

Python does not have a character data type, a single character is simply a string with a length of 1.

Example:

"Geeksforgeeks" or 'Geeksforgeeks' or "a"

- Python3

```
print("A Computer Science portal for geeks")
print('A')
```

Output:

A Computer Science portal for geeks

A

Creating a String in Python

Strings in Python can be created using single quotes or double quotes or even triple quotes. Let us see how we can define a string in Python.

Example:

In this example, we will demonstrate different ways to create a Python String. We will create a string using single quotes (‘ ‘), double quotes (” “), and triple double quotes (“”” “”””). The triple quotes can be used to declare multiline strings in Python.

- Python3

```
# Python Program for
# Creation of String

# Creating a String
# with single Quotes
String1 = 'Welcome to the Geeks World'
print("String with the use of Single Quotes: ")
print(String1)

# Creating a String
# with double Quotes
String1 = "I'm a Geek"
print("\nString with the use of Double Quotes: ")
print(String1)

# Creating a String
# with triple Quotes
String1 = '''I'm a Geek and I live in a world of "Geeks"'''
print("\nString with the use of Triple Quotes: ")
print(String1)

# Creating String with triple
# Quotes allows multiple lines
String1 = '''Geeks
        For
        Life'''
print("\nCreating a multiline String: ")
print(String1)
```

Output:

String with the use of Single Quotes:
Welcome to the Geeks World
String with the use of Double Quotes:

I'm a Geek

String with the use of Triple Quotes:

I'm a Geek and I live in a world of "Geeks"

Creating a multiline String:

Geeks

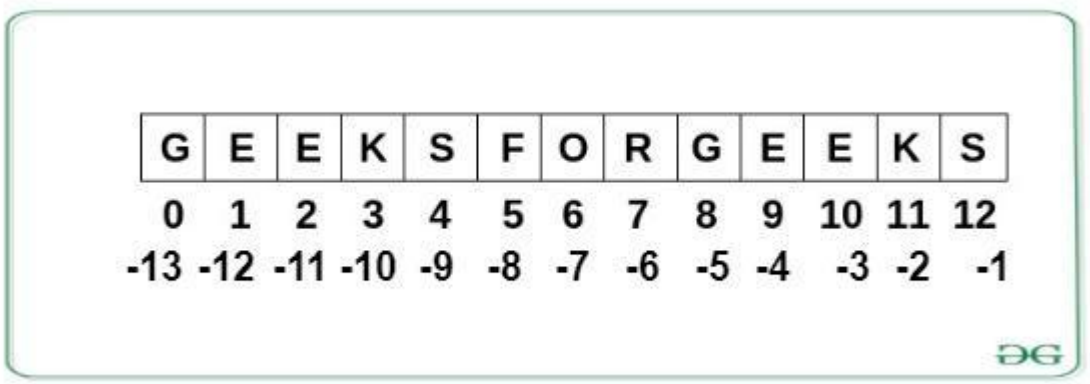
For

Life

Accessing characters in Python String

In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

While accessing an index out of the range will cause an **IndexError**. Only Integers are allowed to be passed as an index, float or other types that will cause a **TypeError**.



Python String indexing

Example:

In this example, we will define a string in Python and access its characters using positive and negative indexing. The 0th element will be the first character of the string whereas the -1th element is the last character of the string.

- Python3

```
# Python Program to Access
# characters of String

String1 = "GeeksForGeeks"
print("Initial String: ")
```

```
print(String1)

# Printing First character
print("\nFirst character of String is: ")
print(String1[0])

# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])
```

Output:

Initial String:
GeeksForGeeks
First character of String is:
G
Last cha racter of String is:
s

String Slicing

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

Example:

In this example, we will use the string-slicing method to extract a substring of the original string. The [3:12] indicates that the string slicing will start from the 3rd index of the string to the 12th index, (12th character not including). We can also use negative indexing in string slicing.

- Python3

```
# Python Program to
# demonstrate String slicing

# Creating a String
String1 = "GeeksForGeeks"
```

```
print("Initial String: ")
print(String1)

# Printing 3rd to 12th character
print("\nSlicing characters from 3-12: ")
print(String1[3:12])

# Printing characters between
# 3rd and 2nd last character
print("\nSlicing characters between " +
      "3rd and 2nd last character: ")
print(String1[3:-2])
```

Output:

Initial String:
GeeksForGeeks
Slicing characters from 3-12:
ksForGeek
Slicing characters between 3rd and 2nd last character:
ksForGee

Reversing a Python String

By accessing characters from a string, we can also reverse strings in Python. We can Reverse a string by using String slicing method.

Example:

In this example, we will reverse a string by accessing the index. We did not specify the first two parts of the slice indicating that we are considering the whole string, from the start index to the last index.

- Python3

```
#Program to reverse a string
gfg = "geeksforgeeks"
print(gfg[::-1])
```

Output:

skeegrofskeeg

Example:

We can also reverse a string by using built-in join and reversed functions, and passing the string as the parameter to the reversed() function.

- Python3

```
# Program to reverse a string

gfg = "geeksforgeeks"

# Reverse the string using reversed and join function
gfg = "".join(reversed(gfg))

print(gfg)
```

Output:

skeegrofskeeg

Deleting/Updating from a String

In Python, the Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is not supported. Although deletion of the entire String is possible with the use of a built-in del keyword. This is because Strings are immutable, hence elements of a String cannot be changed once assigned. Only new strings can be reassigned to the same name.

Updating a character

A character of a string can be updated in Python by first converting the string into a Python List and then updating the element in the list. As lists are mutable in nature, we can update the character and then convert the list back into the String.

Another method is using the string slicing method. Slice the string before the character you want to update, then add the new character and finally add the other part of the string again by string slicing.

Example:

In this example, we are using both the list and the string slicing method to update a character. We converted the String1 to a list, changes its value at a particular element, and then converted it back to a string using the Python string join() method.

In the string-slicing method, we sliced the string up to the character we want to update, concatenated the new character, and finally concatenate the remaining part of the string.

- Python3

```
# Python Program to Update
# character of a String

String1 = "Hello, I'm a Geek"
print("Initial String: ")
print(String1)

# Updating a character of the String
## As python strings are immutable, they don't support item updation directly
### there are following two ways
#1
list1 = list(String1)
list1[2] = 'p'
String2 = ''.join(list1)
print("\nUpdating character at 2nd Index: ")
print(String2)

#2
String3 = String1[0:2] + 'p' + String1[3:]
print(String3)
```

Output:

Initial String:
Hello, I'm a Geek
Updating character at 2nd Index:
Heplo, I'm a Geek
Heplo, I'm a Geek

Updating Entire String

As Python strings are immutable in nature, we cannot update the existing string. We can only assign a completely new value to the variable with the same name.

Example:

In this example, we first assign a value to ‘String1’ and then updated it by assigning a completely different value to it. We simply changed its reference.

- Python3

```
# Python Program to Update
# entire String

String1 = "Hello, I'm a Geek"
print("Initial String: ")
print(String1)

# Updating a String
String1 = "Welcome to the Geek World"
print("\nUpdated String: ")
print(String1)
```

Output:

Initial String:
Hello, I'm a Geek
Updated String:
Welcome to the Geek World

Deleting a character

Python strings are immutable, that means we cannot delete a character from it. When we try to delete thecharacter using the **del** keyword, it will generate an error.

- Python3

```
# Python Program to delete
# character of a String

String1 = "Hello, I'm a Geek"
print("Initial String: ")
print(String1)

print("Deleting character at 2nd Index: ")
del String1[2]
print(String1)
```

Output:

Initial String:

Hello, I'm a Geek

Deleting character at 2nd Index:

Traceback (most recent call last):

File "e:\GFG\Python codes\Codes\demo.py", line 9, in <module>

del String1[2]

TypeError: 'str' object doesn't support item deletion

But using slicing we can remove the character from the original string and store the result in a new string.

Example:

In this example, we will first slice the string up to the character that we want to delete and then concatenate the remaining string next from the deleted character.

- Python3

```
# Python Program to Delete
# characters from a String

String1 = "Hello, I'm a Geek"
print("Initial String: ")
print(String1)

# Deleting a character
# of the String
String2 = String1[0:2] + String1[3:]
print("\nDeleting character at 2nd Index: ")
print(String2)
```

Output:

Initial String:

Hello, I'm a Geek

Deleting character at 2nd Index:

Helo, I'm a Geek

Deleting Entire String

Deletion of the entire string is possible with the use of del keyword. Further, if we try to print the string, this will produce an error because the String is deleted and is unavailable to be printed.

- Python3

```
# Python Program to Delete
# entire String

String1 = "Hello, I'm a Geek"
print("Initial String: ")
print(String1)

# Deleting a String
# with the use of del
del String1
print("\nDeleting entire String: ")
print(String1)
```

Error:

```
Traceback (most recent call last):
File "/home/e4b8f2170f140da99d2fe57d9d8c6a94.py", line 12, in
print(String1)
NameError: name 'String1' is not defined
```

Escape Sequencing in Python

While printing Strings with single and double quotes in it causes **SyntaxError** because String already contains Single and Double Quotes and hence cannot be printed with the use of either of these. Hence, to print such a String either Triple Quotes are used or Escape sequences are used to print Strings.

Escape sequences start with a backslash and can be interpreted differently. If single quotes are used to represent a string, then all the single quotes present in the string must be escaped and the same is done for Double Quotes.

Example:

- Python3

```
# Python Program for
```

```
# Escape Sequencing
# of String

# Initial String
String1 = '''I'm a "Geek"'''
print("Initial String with use of Triple Quotes: ")
print(String1)

# Escaping Single Quote
String1 = 'I\'m a "Geek"'
print("\nEscaping Single Quote: ")
print(String1)

# Escaping Double Quotes
String1 = "I'm a \"Geek\""
print("\nEscaping Double Quotes: ")
print(String1)
  # Printing Paths with the
# use of Escape Sequences
String1 = "C:\\Python\\Geeks\\"
print("\nEscaping Backslashes: ")
print(String1)
  # Printing Paths with the
# use of Tab
String1 = "Hi\tGeeks"
print("\nTab: ")
print(String1)
  # Printing Paths with the
# use of New Line
String1 = "Python\nGeeks"
print("\nNew Line: ")
print(String1)
```

Output:

Initial String with use of Triple Quotes:

I'm a "Geek"

Escaping Single Quote:

I'm a "Geek"

Escaping Double Quotes:

I'm a "Geek"

Escaping Backslashes:

C:\Python\Geeks\

Tab:

Hi Geeks

New Line:

Python

Geeks

Example:

To ignore the escape sequences in a String, **r** or **R** is used, this implies that the string is a raw string and escape sequences inside it are to be ignored.

- Python3

```
# Printing hello in octal
String1 = "\110\145\154\154\157"
print("\nPrinting in Octal with the use of Escape Sequences: ")
print(String1)

# Using raw String to
# ignore Escape Sequences
String1 = r"This is \110\145\154\154\157"
print("\nPrinting Raw String in Octal Format: ")
print(String1)

# Printing Geeks in HEX
String1 = "This is \x47\x65\x65\x6b\x73 in \x48\x45\x58"
print("\nPrinting in HEX with the use of Escape Sequences: ")
print(String1)
```

```
# Using raw String to
# ignore Escape Sequences
String1 = r"This is \x47\x65\x65\x6b\x73 in \x48\x45\x58"
print("\nPrinting Raw String in HEX Format: ")
print(String1)
```

Output:

Printing in Octal with the use of Escape Sequences:
Hello
Printing Raw String in Octal Format:
This is \110\145\154\154\157
Printing in HEX with the use of Escape Sequences:
This is Geeks in HEX
Printing Raw String in HEX Format:
This is \x47\x65\x65\x6b\x73 in \x48\x45\x58

Formatting of Strings

Strings in Python can be formatted with the use of format() method which is a very versatile and powerful tool for formatting Strings. Format method in String contains curly braces {} as placeholders which can hold arguments according to position or keyword to specify the order.

Example 1:

In this example, we will declare a string which contains the curly braces {} that acts as a placeholders and provide them values to see how string declaration position matters.

- Python3

```
# Python Program for
# Formatting of Strings

# Default order
String1 = "{} {} {}".format('Geeks', 'For', 'Life')
print("Print String in default order: ")
print(String1)
```

```
# Positional Formatting
String1 = "{1} {0} {2}".format('Geeks', 'For', 'Life')
print("\nPrint String in Positional order: ")
print(String1)

# Keyword Formatting
String1 = "{l} {f} {g}".format(g='Geeks', f='For', l='Life')
print("\nPrint String in order of Keywords: ")
print(String1)
```

Output:

Print String in default order:
Geeks For Life
Print String in Positional order:
For Geeks Life
Print String in order of Keywords:
Life For Geeks

Example 2:

Integers such as Binary, hexadecimal, etc., and floats can be rounded or displayed in the exponent form with the use of format specifiers.

- Python3

```
# Formatting of Integers
String1 = "{0:b}".format(16)
print("\nBinary representation of 16 is ")
print(String1)

# Formatting of Floats
String1 = "{0:e}".format(165.6458)
print("\nExponent representation of 165.6458 is ")
print(String1)

# Rounding off Integers
```

```
String1 = "{0:.2f}".format(1/6)
print("\none-sixth is : ")
print(String1)
```

Output:

Binary representation of 16 is
10000
Exponent representation of 165.6458 is
1.656458e+02
one-sixth is :
0.17

Example 3:

A string can be left, right, or center aligned with the use of format specifiers, separated by a colon(:). The (<) indicates that the string should be aligned to the left, (>) indicates that the string should be aligned to the right and (^) indicates that the string should be aligned to the center. We can also specify the length in which it should be aligned. For example, (<10) means that the string should be aligned to the left within a field of width of 10 characters.

- Python3

```
# String alignment
String1 = "|{:<10}|{:^10}|{:>10}|".format('Geeks',
                                          'for',
                                          'Geeks')
print("\nLeft, center and right alignment with Formatting: ")
print(String1)

# To demonstrate aligning of spaces
String1 = "\n{0:^16} was founded in {1:<4}!".format("GeeksforGeeks",
                                                    2009)
print(String1)
```

Output:

Left, center and right alignment with Formatting:
|Geeks | for | Geeks|
GeeksforGeeks was founded in 2009 !

Example 4:

Old-style formatting was done without the use of the format method by using the **%** operator

- Python3

```
# Python Program for
# Old Style Formatting
# of Integers

Integer1 = 12.3456789
print("Formatting in 3.2f format: ")
print('The value of Integer1 is %3.2f' % Integer1)
print("\nFormatting in 3.4f format: ")
print('The value of Integer1 is %3.4f' % Integer1)
```

Output:

Formatting in 3.2f format:
The value of Integer1 is 12.35
Formatting in 3.4f format:
The value of Integer1 is 12.3457

Python String constants

Built-In Function	Description
string.ascii_letters	Concatenation of the ascii_lowercase and ascii_uppercase constants.
string.ascii_lowercase	Concatenation of lowercase letters
string.ascii_uppercase	Concatenation of uppercase letters
string.digits	Digit in strings
string.hexdigits	Hexadigit in strings

Built-In Function	Description
string.letters	concatenation of the strings lowercase and uppercase
string.lowercase	A string must contain lowercase letters.
string.octdigits	Octadigit in a string
string.punctuation	ASCII characters having punctuation characters.
string.printable	String of characters which are printable
String.endswith()	Returns True if a string ends with the given suffix otherwise returns False
String.startswith()	Returns True if a string starts with the given prefix otherwise returns False
String.isdigit()	Returns “True” if all characters in the string are digits, Otherwise, It returns “False”.
String.isalpha()	Returns “True” if all characters in the string are alphabets, Otherwise, It returns “False”.
string.isdecimal()	Returns true if all characters in a string are decimal.
str.format()	one of the string formatting methods in Python3, which allows multiple substitutions and value formatting.
String.index	Returns the position of the first occurrence of substring in a string
string.uppercase	A string must contain uppercase letters.

Built-In Function	Description
string.whitespace	A string containing all characters that are considered whitespace.
string.swapcase()	Method converts all uppercase characters to lowercase and vice versa of the given string, and returns it
replace()	returns a copy of the string where all occurrences of a substring is replaced with another substring.

Deprecated string functions

Built-In Function	Description
string.Isdecimal	Returns true if all characters in a string are decimal
String.Isalnum	Returns true if all the characters in a given string are alphanumeric.
string.Istitle	Returns True if the string is a title cased string
String.partition	splits the string at the first occurrence of the separator and returns a tuple.
String.Isidentifier	Check whether a string is a valid identifier or not.
String.len	Returns the length of the string.
String.rindex	Returns the highest index of the substring inside the string if substring is found.
String.Max	Returns the highest alphabetical character in a string.
String.min	Returns the minimum alphabetical character in a string.

Built-In Function	Description
String.splitlines	Returns a list of lines in the string.
string.capitalize	Return a word with its first character capitalized.
string.expandtabs	Expand tabs in a string replacing them by one or more spaces
string.find	Return the lowest indexing a sub string.
string.rfind	find the highest index.
string.count	Return the number of (non-overlapping) occurrences of substring sub in string
string.lower	Return a copy of s, but with upper case, letters converted to lower case.
string.split	Return a list of the words of the string, If the optional second argument sep is absent or None
string.rsplit()	Return a list of the words of the string s, scanning s from the end.
rpartition()	Method splits the given string into three parts
string.splitfields	Return a list of the words of the string when only used with two arguments.
string.join	Concatenate a list or tuple of words with intervening occurrences of sep.
string.strip()	It returns a copy of the string with both leading and trailing white spaces removed
string.lstrip	Return a copy of the string with leading white spaces removed.

Built-In Function	Description
string.rstrip	Return a copy of the string with trailing white spaces removed.
string.swapcase	Converts lower case letters to upper case and vice versa.
string.translate	Translate the characters using table
string.upper	lower case letters converted to upper case.
string.ljust	left-justify in a field of given width.
string.rjust	Right-justify in a field of given width.
string.center()	Center-justify in a field of given width.
string-zfill	Pad a numeric string on the left with zero digits until the given width is reached.
string.replace	Return a copy of string s with all occurrences of substring old replaced by new.
string.casefold()	Returns the string in lowercase which can be used for caseless comparisons.
string.encode	Encodes the string into any encoding supported by Python. The default encoding is utf-8.
string.maketrans	Returns a translation table usable for str.translate()

Advantages of String in Python:

- Strings are used at a larger scale i.e. for a wide areas of operations such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.

- Python has a rich set of string methods that allow you to manipulate and work with strings in a variety of ways. These methods make it easy to perform common tasks such as converting strings to uppercase or lowercase, replacing substrings, and splitting strings into lists.
- Strings are immutable, meaning that once you have created a string, you cannot change it. This can be beneficial in certain situations because it means that you can be confident that the value of a string will not change unexpectedly.
- Python has built-in support for strings, which means that you do not need to import any additional libraries or modules to work with strings. This makes it easy to get started with strings and reduces the complexity of your code.
- Python has a concise syntax for creating and manipulating strings, which makes it easy to write and read code that works with strings.