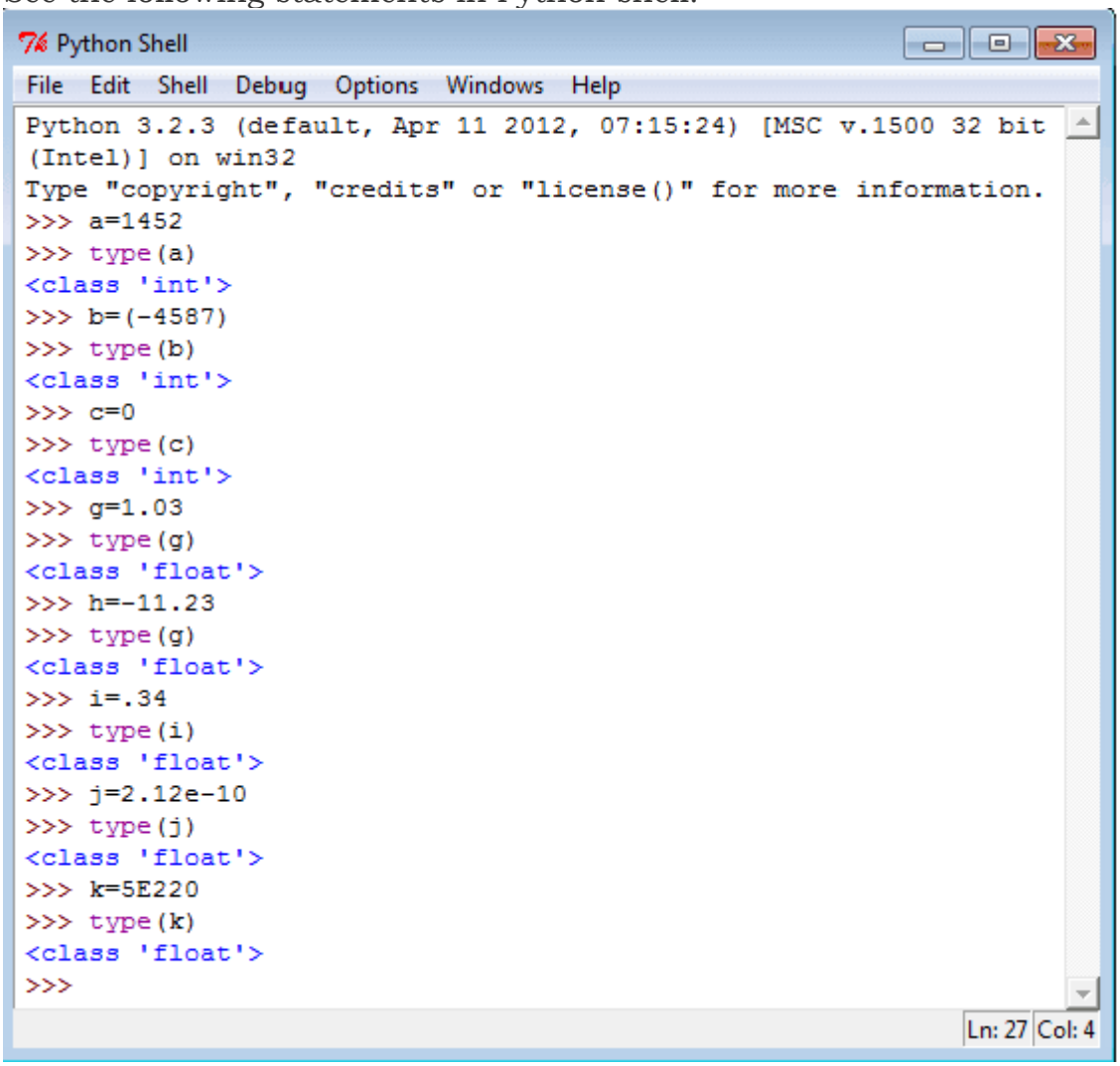
Data types:

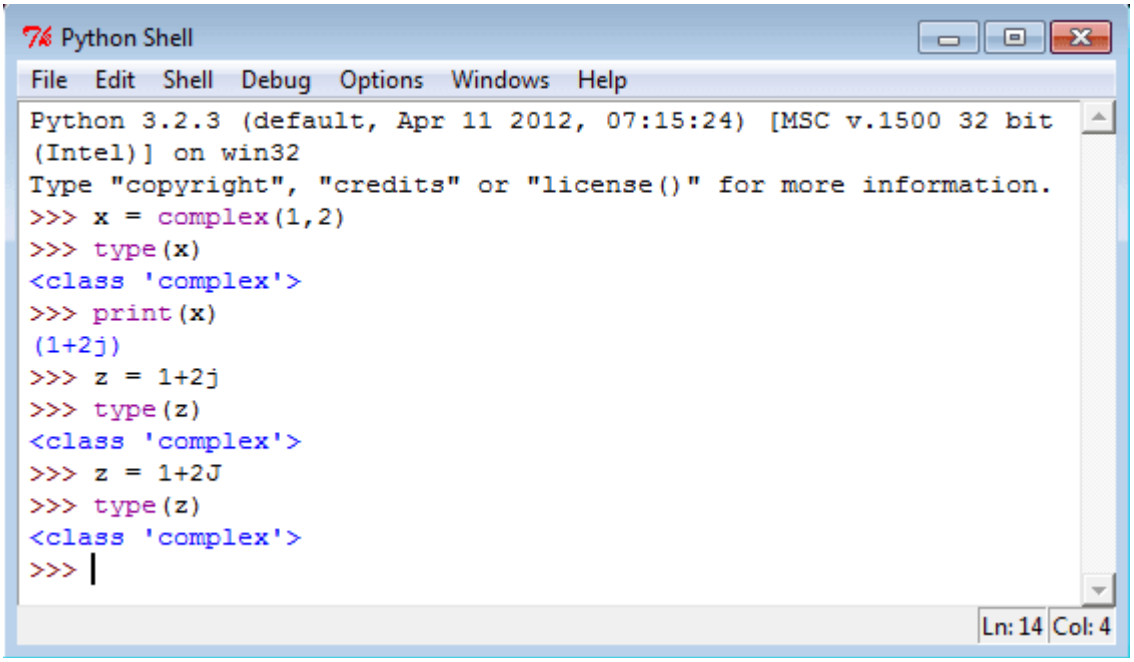
Numbers:

Numbers are created by numeric literals. Numeric objects are immutable, which means when an object is created its value cannot be changed.

Python has three distinct numeric types: **integers, floating point numbers, and complex numbers.** Integers represent negative and positive integers without fractional parts whereas floating point numbers represents negative and positive numbers with fractional parts. In addition, Booleans are a subtype of plain integers. See the following statements in Python shell.

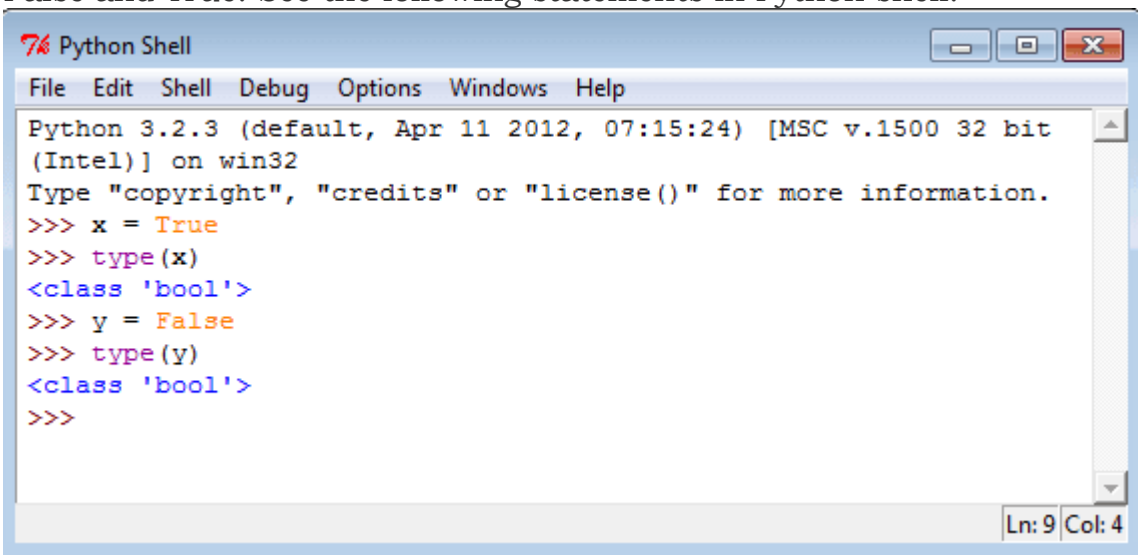


Mathematically, a complex number (generally used in engineering) is a number of the form A+Bi where i is the imaginary number. Complex numbers have a real and imaginary part. Python supports complex numbers either by specifying the number in (real + imagJ) or (real + imagj) form or using a built-in method complex(x, y). See the following statements in Python Shell.



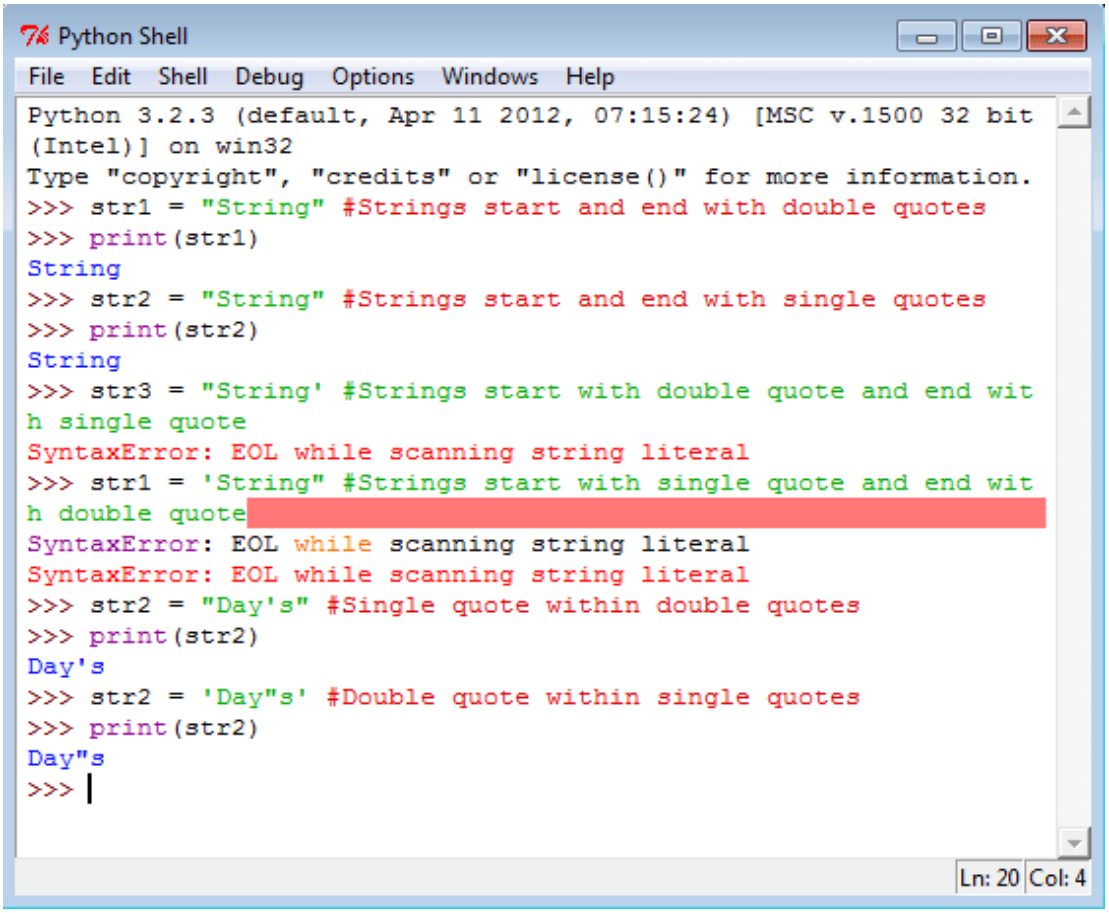
Boolean(bool):

The simplest build-in type in Python is the bool type, it represents the truth values False and True. See the following statements in Python shell.



Strings:

In Python, a string type object is a sequence (left-to- right order) of characters. Strings start and end with single or double quotes Python strings are immutable. Single and double quoted strings are same and you can use a single quote within a string when it is surrounded by double quote and vice versa. Declaring a string is simple, see the following statements.

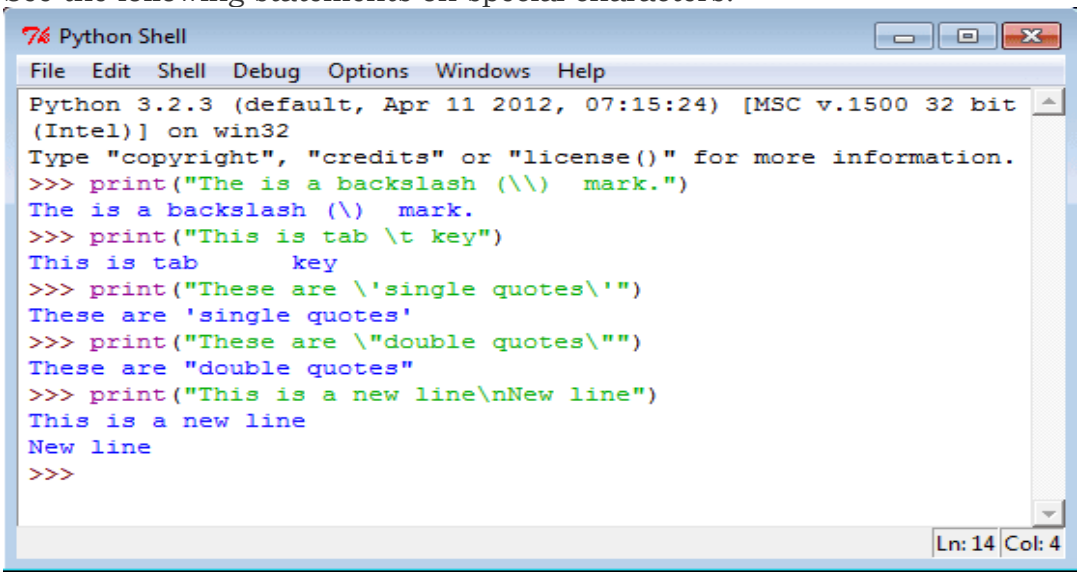


Special characters in strings:

The backslash (\) character is used to introduce a special character. See the following table.

Escape sequence	Meaning
\n	Newline
\t	Horizontal Tab
\\	Backslash
\'	Single Quote
\"	Double Quote

See the following statements on special characters.



What is an operator?

Simple answer can be given using expression *4 + 5 is equal to 9*. Here, 4 and 5 are called operands and + is called operator. Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (i.e., Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let's have a look on all operators one by one.

Python Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0
**	Exponent - Performs exponential (power) calculation on operators	a**b will give 10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

Python Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.

<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.

Python Assignment Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assigne value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	Floor Dividion and assigns a value, Performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows:

a = 0011 1100
b = 0000 1101
---
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

There are following Bitwise operators supported by Python language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(a | b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 1111

Python Logical Operators:

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(a or b) is true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not(a and b) is false.

Python Membership Operators:

In addition to the operators discussed previously, Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators explained below:

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here **in** results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here **not in** results in a 1 if x is not a member of sequence y.

Python Identity Operators:

Identity operators compare the memory locations of two objects. There are two Identity operators explained below:

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

if statement

The Python if statement is same as it is with other programming languages. It executes a set of statements conditionally, based on the value of a logical expression.

Here is the general form of a one way if statement.

Syntax:

```
if expression :
    statement_1
    statement_2
    ....
```

In the above case, expression specifies the conditions which are based on Boolean expression. When a Boolean expression is evaluated it produces either a value of true or false. If the expression evaluates true the same amount of indented statement(s) following if will be executed. This group of the statement(s) is called a block.

if .. else statement

In Python if .. else statement, if has two blocks, one following the expression and other following the else clause. Here is the syntax.

Syntax:

```
if expression :
    statement_1
    statement_2
    ....
     else :
   statement_3
   statement_4
   ....
```

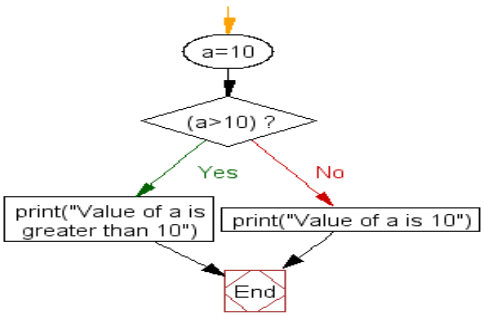
In the above case if the expression evaluates to true the same amount of indented statements(s) following if will be executed and if the expression evaluates to false the same amount of indented statements(s) following else will be executed. See the following example. The program will print the second print statement as the value of a is 10.

```
1. a=10
2. if(a>10):
3.     print("Value of a is greater than 10")
4. else :
5.     print("Value of a is 10")
```

Output :

```
Value of a is 10
```

Flowchart:



if .. elif .. else statement

Sometimes a situation arises when there are several conditions. To handle the situation Python allows adding any number of elif clause after an if and before an else clause. Here is the syntax.

Syntax:

```
if expression1 :
        statement_1
        statement_2
        ....

    elif expression2 :
    statement_3
    statement_4
    ....
  elif expression3 :
    statement_5
    statement_6
    .....................
  else :
    statement_7
    statement_8
```

In the above case Python evaluates each expression (i.e. the condition) one by one and if a true condition is found the statement(s) block under that expression will be executed. If no true condition is found the statement(s) block under else will be executed. In the following example, we have applied if, series of elif and else to get the type of a variable.

```
1.  var1 = 1+2j
2.  if (type(var1) == int):
3.      print("Type of the variable is Integer")
4.  elif (type(var1) == float):
5.      print("Type of the variable is Float")
6.  elif (type(var1) == complex):
7.      print("Type of the variable is Complex")
8.  elif (type(var1) == bool):
9.      print("Type of the variable is Bool")
10.         elif (type(var1) == str):
11.             print("Type of the variable is String")
12.         elif (type(var1) == tuple):
13.             print("Type of the variable is Tuple")
14.         elif (type(var1) == dict):
15.             print("Type of the variable is Dictionaries")
16.         elif (type(var1) == list):
17.             print("Type of the variable is List")
18.         else:
19.             print("Type of the variable is Unknown")
```

Output :

```
Type of the variable is Complex
```

Nested if .. else statement

In general nested if-else statement is used when we want to check more than one conditions. Conditions are executed from top to bottom and check each condition whether it evaluates to true or not. If a true condition is found the statement(s) block associated with the condition executes otherwise it goes to next condition. Here is the syntax :

Syntax:

```
if expression1 :
   if expression2 :
    statement_3
    statement_4
  ....
 else :
   statement_5
   statement_6
  ....
else :
        statement_7
  statement_8
```

In the above syntax expression1 is checked first, if it evaluates to true then the program control goes to next if - else part otherwise it goes to the last else statement and executes statement_7, statement_8 etc.. Within the if - else if expression2 evaluates true then statement_3, statement_4 will execute otherwise statement_5, statement_6 will execute. See the following example.

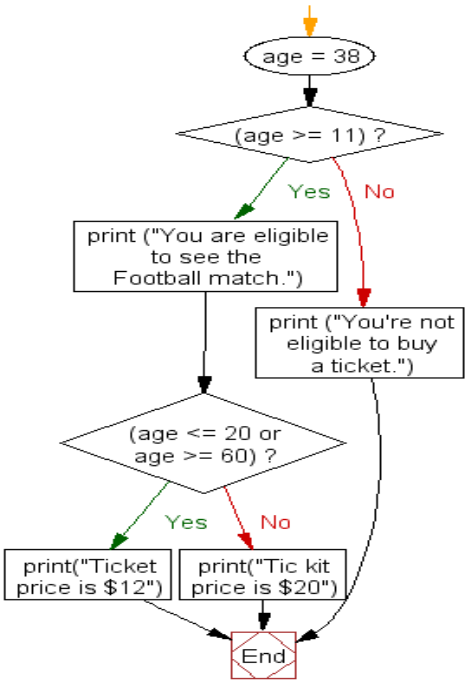
```
1. age = 38
2. if (age >= 11):
3.   print ("You are eligible to see the Football match.")
4.   if (age <= 20 or age >= 60):
5.       print("Ticket price is $12")
6.   else:
7.       print("Tic kit price is $20")
8. else:
9.     print ("You're not eligible to buy a ticket.")
```

Output :

```
You are eligible to see the Football match.
Tic kit price is $20
```

In the above example age is set to 38, therefore the first expression (age >= 11) evaluates to True and the associated print statement prints the string "You are eligible to see the Football match". There after program control goes to next if statement and the condition (38 is outside <=20 or >=60) is matched and prints "Tic kit price is $12".

Flowchart:



Use the and operator in an if statement

```
#create two boolean objects

x = False
y = True

#The validation will be True only if all the expressions generate a value True
if x and y:
    print('Both x and y are True')
else:
    print('x is False or y is False or both x and y are False')
```

Output :

```
x is False or y is False or both x and y are False
```

Use the in operator in an if statement

```
#create a string
s = 'jQuery'
#create a list
l = ['JavaScript', 'jQuery', 'ZinoUI']

# in operator is used to replace various expressions that use the or operator
if s in l:
    print(s + ' Tutorial')

#Alternate if statement with or operator

if s == 'JavaScript' or s == 'jQuery' or s == 'ZinoUI':
    print(s + ' Tutorial')
```

Output :

```
jQuery Tutorial
jQuery Tutorial
```

Write an if-else in a single line of code

```
#create a integer
n = 150
print(n)

#if n is greater than 500, n is multiplied by 7, otherwise n is divided by 7
result = n * 7 if n > 500 else n / 7
print(result)
```

Output :

```
150
21.428571428571427
```

Define a negative if

If a condition is true the not operator is used to reverse the logical state, then logical not operator will make it false.

```
#create a integer
x = 20
print(x)

#uses the not operator to reverse the result of the logical expression

if not x == 50:
    print('the value of x different from 50')
else:
    print('the value of x is equal to 50')
```

Output :

```
20
the value of x different from 50
```

for loop

Like most other languages, Python has for loops, but it differs a bit from other like C or Pascal. In Python for loop is used to iterate over the items of any sequence including the Python list, string, tuple etc. The for loop is also used to access elements from a container (for example list, string, tuple) using built-in function range().

Syntax

```
for variable_name in sequence :
   statement_1
   statement_2
   ....
```

Parameter

Name	Description
variable_name	It indicates target variable which will set a new value for each iteration of the loop.
sequence	A sequence of values that will be assigned to the target variable variable_name. Values are provided using a list or a string or from the built-in function range().
statement_1, statement_2	Block of program statements.

Example: Python for loop

```
>>> #The list has four elements, indices start at 0 and end at 3
>>> color_list = ["Red", "Blue", "Green", "Black"]
>>> for c in color_list:
        print(c)

Red
Blue
Green
Black
>>>
```

In the above example color_list is a sequence contains a list of various color names. When the for loop executed the first item (i.e. Red) is assigned to the variable c. After this, the print statement will execute and the process will continue until we rich the end of the list.

Python for loop and range() function

The range() function returns a list of consecutive integers. The function has one, two or three parameters where last two parameters are optional. It is widely used in for loops. Here is the syntax.

```
range(a)
range(a,b)
range(a,b,c)
```

range(a) : Generates a sequence of numbers from 0 to a, excluding a, incrementing by 1.

Syntax

```
for <variable> in range(<number>):
```

Example :

```
>>> for a in range(4):
    print(a)

0
1
2
3
>>>
```

range(a,b) : Generates a sequence of numbers from a to b excluding b, incrementing by 1.

Syntax

```
for "variable" in range("start_number", "end_number"):
```

Example :

```
>>> for a in range(2,7):
        print(a)

2
3
4
5
6
>>>
```

range(a,b,c) : Generates a sequence of numbers from a to b excluding b, incrementing by c.

Example :

```
>>> for a in range(2,19,5):
        print(a)<p></p>

 2
 7
 12
 17
>>>
```

Python for loop : Iterating over tuple, list, dictionary

Example : Iterating over tuple

The following example counts the number of even and odd numbers from a series of numbers.

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9) # Declaring the tuple
count_odd = 0
count_even = 0
for x in numbers:
        if x % 2:
            count_odd+=1
        else:
            count_even+=1
print("Number of even numbers :",count_even)
print("Number of odd numbers :",count_odd)
```

Output :

```
Number of even numbers : 4
Number of odd numbers : 5
```

In the above example a tuple named numbers is declared which holds the integers 1 to 9.

The best way to check if a given number is even or odd is to use the modulus operator (%).

The operator returns the remainder when dividing two numbers.

Modulus of 8 % 2 returns 0 as 8 is divided by 2, therefore 8 is even and modulus of 5 % 2 returns 1 therefore 5 is odd.

The for loop iterates through the tuple and we test modulus of x % 2 is true or not, for every item in the tuple and the process will continue until we rich the end of the tuple.

When it is true count_even increase by one otherwise count_odd is increased by one.

Finally, we print the number of even and odd numbers through print statements.

Example : Iterating over list

In the following example for loop iterates through the list "datalist" and prints each item and its corresponding Python type.

```
datalist = [1452, 11.23, 1+2j, True, 'w3resource', (0, -1), [5, 12],
{"class":'V', "section":'A'}]
for item in datalist:
   print ("Type of ",item, " is ", type(item))
```

Output :

```
Type of  1452  is  <class 'int'>
Type of  11.23  is  <class 'float'>
Type of  (1+2j)  is  <class 'complex'>
Type of  True  is  <class 'bool'>
Type of  w3resource  is  <class 'str'>
Type of  (0, -1)  is  <class 'tuple'>
Type of  [5, 12]  is  <class 'list'>
Type of  {'section': 'A', 'class': 'V'}  is  <class 'dict'>
```

Example : Iterating over dictionary

In the following example for loop iterates through the dictionary "color" through its keys and prints each key.

```
>>> color = {"c1": "Red", "c2": "Green", "c3": "Orange"}
>>> for key in color:
    print(key)

c2
c1
c3
>>>
```

Following for loop iterates through its values :

```
>>> color = {"c1": "Red", "c2": "Green", "c3": "Orange"}
>>> for value in color.values():
    print(value)

Green
Red
Orange
>>>
```

You can attach an optional else clause with for statement, in this case, syntax will be -

```
for variable_name in sequence :
   statement_1
   statement_2
   ....
else :
   statement_3
   statement_4
   ....
```

The else clause is only executed after completing the for loop. If a break statement executes in first program block and terminates the loop then the else clause does not execute.

While loop

Loops are used to repeatedly execute a block of program statements. The basic loop structure in Python is while loop. Here is the syntax.

Syntax :

```
while (expression) :
    statement_1
    statement_2
    ....
```

The while loop runs as long as the expression (condition) evaluates to True and execute the program block. The condition is checked every time at the beginning of the loop and the first time when the expression evaluates to False, the loop stops without executing any remaining statement(s). The following example prints the digits 0 to 4 as we set the condition x < 5.

```
1. x = 0;
2. while (x < 5):
3.     print(x)
4.     x += 1
```

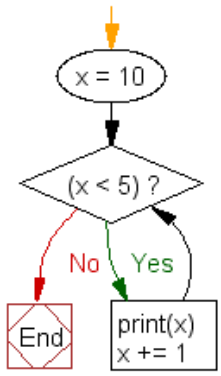
Output :

```
0
1
2
3
4
```

One thing we should remember that a while loop tests its condition before the body of the loop (block of program statements) is executed. If the initial test returns false, the body is not executed at all. For example the following code never prints out anything since before executing the condition evaluates to false.

```
1. x = 10;
2. while (x < 5):
3.     print(x)
4.     x += 1
```

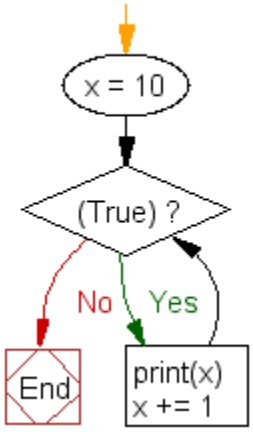
Flowchart :



The following while loop is an infinite loop, using True as the condition:

```
1. x = 10;
2. while (True):
3.     print(x)
4.     x += 1
```

Flowchart :



Python: while and else statement

There is a structural similarity between while and else statement. Both have a block of statement(s) which is only executed when the condition is true. The difference is that block belongs to if statement executes once whereas block belongs to while statement executes repeatedly.
You can attach an optional else clause with while statement, in this case, syntax will be -

```
while (expression) :
   statement_1
   statement_2
   ......
else :
   statement_3
   statement_4
   ......
```

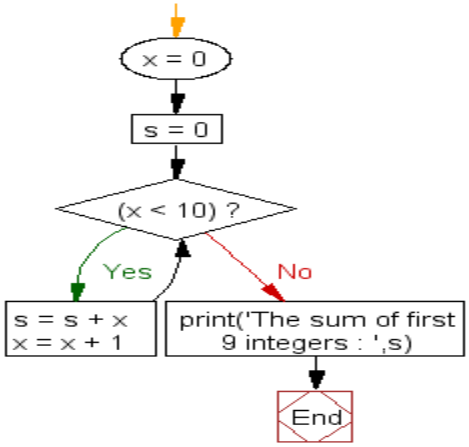
The while loop repeatedly tests the expression (condition) and, if it is true, executes the first block of program statements. The else clause is only executed when the condition is false it may be the first time it is tested and will not execute if the loop breaks, or if an exception is raised. If a break statement executes in first program block and terminates the loop then the else clause does not execute. In the following example, while loop calculates the sum of the integers from 0 to 9 and after completing the loop, else statement executes.

```
1. x = 0;
2. s = 0
3. while (x < 10):
4.     s = s + x
5.     x = x + 1
6. else :
7.     print('The sum of first 9 integers : ',s)
```

Output :

```
The sum of first 9 integers :  45
```

Flowchart :



Example: while loop with if-else and break statement

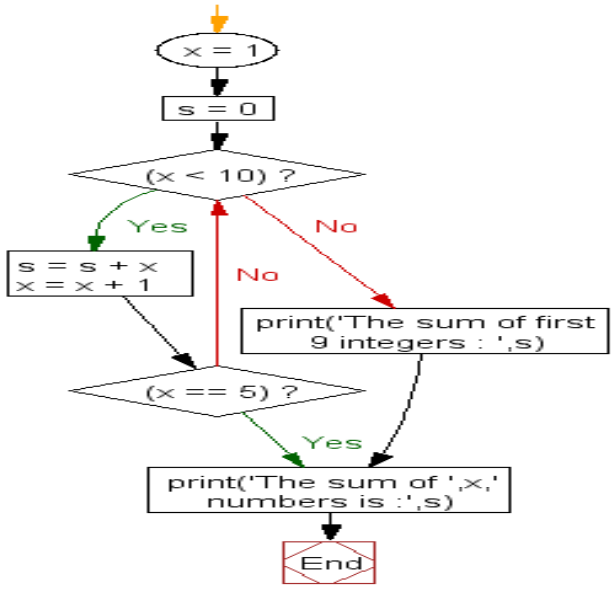
```
1. x = 1;
2. s = 0
3. while (x < 10):
4.     s = s + x
5.     x = x + 1
6.     if (x == 5):
7.         break
8. else :
9.     print('The sum of first 9 integers : ',s)
10.        print('The sum of ',x,' numbers is :',s)
```

Output :

```
The sum of  5  numbers is : 10
```

In the above example the loop is terminated when x becomes 5. Here we use break statement to terminate the while loop without completing it, therefore program control goes to outside the while - else structure and execute the next print statement.

Flowchart :



break statement:

The break statement is used to exit a for or a while loop. The purpose of this statement is to end the execution of the loop (for or while) immediately and the program control goes to the statement after the last statement of the loop. If there is an optional else statement in while or for loop it skips the optional clause also. Here is the syntax.

Syntax

```
while (expression1) :
   statement_1
   statement_2
   ......
   if expression2 :
   break

for variable_name in sequence :
   statement_1
   statement_2
   if expression3 :
     break
```

Example : break in for loop

In the following example for loop breaks when the count value is 5. The print statement after the for loop displays the sum of first 5 elements of the tuple numbers.

```
1. numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9) # Declaring the tuple
2. num_sum = 0
3. count = 0
4. for x in numbers:
5.     num_sum = num_sum + x
6.     count = count + 1
7.     if count == 5:
8.         break
9. print("Sum of first ",count,"integers is : ", num_sum)
```

Output: Sum of first 5 integers is : 15

Example : break in while loop

In the following example while loop breaks when the count value is 5. The print statement after the while loop displays the value of num_sum (i.e. 0+1+2+3+4).

```
1. num_sum = 0
2. count = 0
3. while(count<10):
4.     num_sum = num_sum + count
5.     count = count + 1
6.     if count== 5:
7.         break
8. print("Sum of first ",count,"integers is : ", num_sum)
```

Output: Sum of first 5 integers is : 10

continue statement

The continue statement is used in a while or for loop to take the control to the top of the loop without executing the rest statements inside the loop. Here is a simple example.

```
1. for x in range(6):
2.     if (x == 3 or x==6):
3.         continue
4.     print(x)
```

Output :

0
1
2
4
5

In the above example, the for loop prints all the numbers from 0 to 6 except 3 and 6 as the continue statement returns the control of the loop to the top

Python Pass statement

It is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

Syntax

```
pass
```

Example

```
#!/usr/bin/py
for letter in 'Python':
   if letter == 'h':
      pass
      print 'This is pass block'
   print 'Current Letter :', letter
print "Good bye!"
```

When the above code is executed, it produces following result −

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```