PYTHON PROGRAMMING - I

Chap - 2

Basics of Python Programming

By-

Prof. A. P. Chaudhari

(M.Sc. Computer Science, SET)

HOD,

Department of Computer Science

S.V.S's Dadasaheb Rawal College, Dondaicha

Chap – 2 Basics of Python Programming

- Python Identifiers, Variables and Keywords
- Putting Comments
- Expressions and Statements
- Standard Data Types – Basic, None, Boolean, Numbers
- Type Conversion Function
- Operators in Python
- Operator Precedence
- Accepting Input and Displaying Output

Flow Control Statements -

- Conditional Statements
- Looping Statements
- break, continue, pass Statements

Identifiers:

- A Python identifier is a name used to identify a variable, function, class, module or other object.
- An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits 0 to 9.
- Python does not allow punctuation characters such as @, $, and % within identifiers.
- Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Identifiers:

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Variables:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables:

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

Variables:

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively.

Variables:

Multiple Assignment:

Python allows you to assign a single value to several variables simultaneously. For example :− a = b = c = 1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location.

You can also assign multiple objects to multiple variables. For example − a, b, c = 1, 2, "john"

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Keywords:

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

And	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
Def	if	return
Del	import	try
Elif	in	while
Else	is	with
except	lambda	yield

Putting Comments:

A hash sign # that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
# First comment
print "Hello, Python!"    # second comment
```

This produces the following result −

```
Hello, Python!
```

Python does not have a syntax for multi-line comments. If you wish to add a multiline comment you could insert a # for each line, as like in following code:

```
# This is a comment.
# This is a comment, too.
# This is more comment
```

Expressions and Statements:

Expression:

Expression in any programing language is a combination of values, and operators. A value is also considered expression and same with a variable, so the followings are all legal expressions.

e.g.: If you type an expression in an interactive mode, the interpreter evaluates it and display the result:

1) >>> 5 * 4

   20

2) >>> 2 + 2

   4

3) >>> 'ABC‘ + 'abc'

   'ABCabc'

Expressions and Statements:

Statements:

A statement is a part of code that is executed by the Python interpreter. When you type a statement in interactive mode, the interpreter executes it and display the result, if there is one statement.

A script usually contains a sequence of statements. If there is more than one statement, the result appear at a time as the statement execute.

e.g.:

```
print 10
x = 20
print x
```

O/P:

```
10
20
```

Data Types:

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

1) Numbers
2) String
3) List
4) Tuple
5) Dictionary

Data Types:

1) Numbers:

Number data types store numeric values. Number objects are created when you assign a value to them. For example −

var1 = 1

var2 = 10

Python supports four different numerical types −

i. int (signed integers)

ii. long (long integers, they can also be represented in octal and hexadecimal)

iii. float (floating point real values)

iv. complex (complex numbers)

Data Types:

Here are some examples of numbers −

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

Data Types:

2) Strings:

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example −

str = ‘Good Morning'

print str # Prints complete string (Good Morning)

print str[0] # Prints first character of the string (G)

print str[5:8] # Prints characters starting from 5th to 8th (Morn)

print str[5:] # Prints string starting from 5th character (Morning)

print str * 2 # Prints string two times (Good MorningGood Morning)

print str + “ Ram" # Prints concatenated string (Good Morning Ram)

Data Types:

3) Lists:

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([ ]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

For example −

```
list = [ 'abcd', 135, 2.23, ‘shyam', 70.2 ]
tinylist = [123, ‘viraj']
```

Data Types:

list = ['abcd', 135, 2.23, ‘shyam', 70.2]

tinylist = [123, ‘viraj']

```
print list              # Prints complete list          [ 'abcd', 135, 2.23, ‘shyam', 70.2 ]
print list[0]           # Prints first element of the list                    abcd
print list[1:3]         # Prints elements starting from 2nd till 3rd     [135, 2.23]
print list[2:]          # Prints elements starting from 3rd element   [2.23, shyam ', 70.2]
print tinylist * 2      # Prints list two times                  [123, ' viraj ', 123, ' viraj ']
print list + tinylist   # Prints concatenated lists
                                        [ 'abcd', 135, 2.23, ‘shyam', 70.2, 123, ‘viraj']
```

Data Types:

4) Tuples:

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists.

For example −

tuple = ('abcd', 135, 2.23, ‘shyam', 70.2)

tinytuple = (123, ‘viraj')

Data Types:

```
tuple = ('abcd', 135, 2.23, ‘shyam', 70.2)
tinytuple = (123, ‘viraj')

print tuple                 # Prints complete list     ('abcd', 135, 2.23, ‘shyam', 70.2)
print tuple[0]              # Prints first element of the list          abcd
print tuple[1:3]            # Prints elements starting from 2nd till 3rd      (135, 2.23)
print tuple[2:]             # Prints elements starting from 3rd element
                                                                  (2.23, ‘shyam', 70.2)
print tinytuple * 2         # Prints list two times          (123, ‘viraj‘, 123, ‘viraj')
print tuple + tinytuple     # Prints concatenated lists
                                    ('abcd', 135, 2.23, ‘shyam', 70.2, 123, ‘viraj')
```

Data Types:

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

```
tuple = ('abcd', 135, 2.23, ‘shyam', 70.2)
list = ['abcd', 135, 2.23, ‘shyam', 70.2]

tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Data Types:

5) Dictionary:

Python's dictionaries are kind of hash table type. They work like associative arrays and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

For example −

```
dict = { }
dict ['one'] = "This is one"
dict [2]     = "This is two"


tinydict = {'name': 'john', 'code':6734, 'dept': 'sales'}
```

Data Types:

```
dict = { }
dict ['one'] = "This is one"
dict [2]     = "This is two"
tinydict = {'name': 'john', 'code':6734, 'dept': 'sales'}


print dict ['one']          # Prints value for 'one' key          This is one
print dict[2]               # Prints value for 2 key              This is two
print tinydict              # Prints complete dictionary
                                    {'dept': 'sales', 'code': 6734, 'name': 'john'}
print tinydict.keys()       # Prints all the keys          ['dept', 'code', 'name']
print tinydict.values()     # Prints all the values        ['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Type Conversion Function:

Sometimes, it may be needed to perform conversion between the built-in types. To convert between types, you can simply use the type name as a function. There are a number of built-in functions to perform the conversion from one data type to another. These functions return a new object that represents the converted value.

Function	Description
int (x)	Converts x into an integer.
long(x)	Converts x into long integer.
float(x)	Converts x into floating-point number.
complex(real,[imag])	Creates a complex number.
str(x)	Converts x into a string.
tuple(x)	Converts x into tuple.

Type Conversion Function:

Function	Description
list (x)	Converts x into a list.
dict(x)	Creates a dictionary. x must be a sequence of (key,value) tuples.
char (x)	Converts x into character.
hex(x)	Converts x into hexadecimal string.
oct(x)	Converts x into octal string.

Operators:

Operators are use to perform operations on the value of operands. Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Python language supports the following types of operators-

1) Arithmetic Operators
2) Comparison Relational Operators
3) Assignment Operators
4) Logical Operators
5) Bitwise Operators
6) Membership Operators
7) Identity Operators

Operators:

1) Arithmetic Operators:

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

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = 10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	a / b = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	a % b = 0
** Exponent	Performs exponential power calculation on operators	a**b =20 to the power 10
// Floor Division	The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero towards negative infinity −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Operators:

2) Comparison Operators :

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable **a holds 10** and variable **b holds 20**, then −

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

Operators:

3) Assignment Operators :

Assume variable **a holds 10** and variable **b holds 20**, then −

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	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	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	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	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	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential power calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Operators:

4) Logical Operator: The following table lists the logical operators −

Assume Boolean variables **A holds true** and variable **B holds false**, then -

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any one of the two operands are non-zero, then the condition becomes true.	(A || B) is true
! (logical 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.	!(A && B) is true

Operators:

5) **Bitwise Operator:** Python defines several bitwise operators, which can be applied to the integer types- long, int, short, char, and byte. Bitwise operator works on bits and performs 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

& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	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

Operators:

6) Membership Operator:

Python's membership operators test for membership in a sequence, such as strings, lists, etc.

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.

Operators:

7) Identity Operator:

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 idx equals idy.
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 idx is not equal to idy.

Displaying Output:

The simplest way to produce output is using the ***print* statement** where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows −

```
print "Python is really a great language,", "isn't it?“
```

This produces the following result on your standard screen −

```
Python is really a great language, isn't it?
```

Accepting Input:

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are −

1) raw_input()
2) input()

1) raw_input():

This function works in older version. It takes exactly what it typed from the keyboard, convert it into string and then return it to the variable in which we want to store it.

e.g: val = raw_input(“Enter any value:”)
print “Value is: “,val

o/p: Enter any value: 5 * 4
Value is: 5 * 4

Accepting Input:

2) input():

This function initially takes the input from the user and then evaluates the expression, it means Python automatically identifies whether user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by Python.

e.g: val = input(“Enter any value:”)

print “Value is: “,val

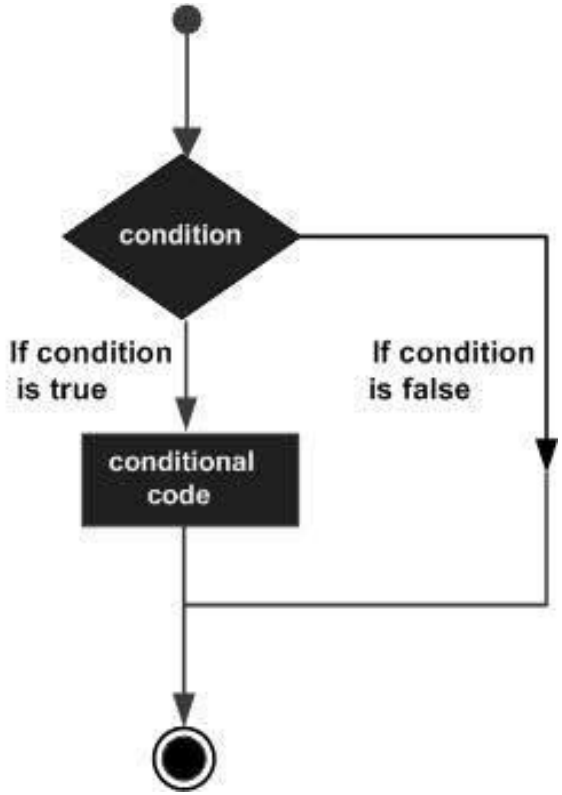
o/p: Enter any value: 5 * 4

Value is: 20

Conditional Statements:

Conditional Statement in Python performs different computations or actions depending on whether a specific condition evaluates to True or False. Conditional statement are handled by **if statements** in Python.

In Python, if statement is used for decision making. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.



Python programming language provides following types of conditional statements.

1) if statement

2) if … else statement

3) Nested if statement

Conditional Statements:

1) 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.

Syntax: **if condition:**

statement(s)

If the condition evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If condition expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

Ex 1:

```
no = 5
if no > 0:
        print "Positive Value"
```

o/p – Positive Value

Ex 2:

```
a = 10
b = 5
if  a > b:
        print "A greater than B"
```

o/p- A greater than B

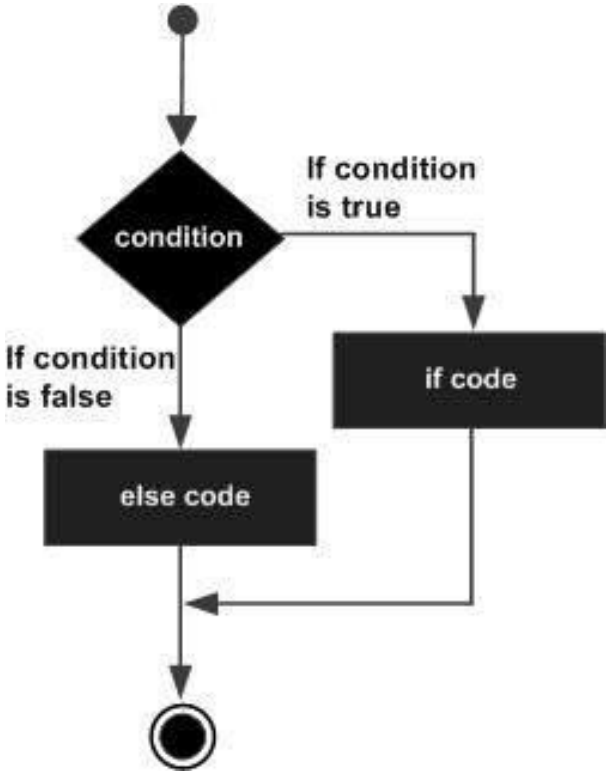
Conditional Statements:

2) 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 FALSE value.

Syntax:

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



Conditional Statements:

Example 1:

```
a = input("Enter first value:")
b = input("Enter second value:")
if a>b:
        print "A greater than B"
else:
        print "B greater than A"
```

o/p – Enter first value: 5
Enter second value: 10
B greater than A

Example 2:

```
a = input("Enter any number:")
if a%2==0:
        print "Even Number"
else:
        print "Odd Number"
```

o/p – Enter any number: 18
Even Number

Enter any number: 21
Odd Number

Conditional Statements:

3) Nested if statement:

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax:

```
if expression1:
        statement(s)
        if expression2:
                statement(s)
        elif expression3:
                statement(s)
        elif expression4:
                statement(s)
        else: statement(s)
else: statement(s)
```

Conditional Statements:

Example:

```
per = input("Enter Percentage:")
if per <=100:
        print "Percentage is less than 100"
        if per >= 70:
                print "Distinction Class"
        elif per >= 60:
                print "First Class"
        elif per >= 35:
                print "Second Class"
        else:
                print "Fail"
else:
        print "Not valid Percentage"
```

O/P:

```
Enter Percentage: 85
Distinction Class

Enter Percentage: 48
Second Class

Enter Percentage: 24
Fail
```

Looping Statements:

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.

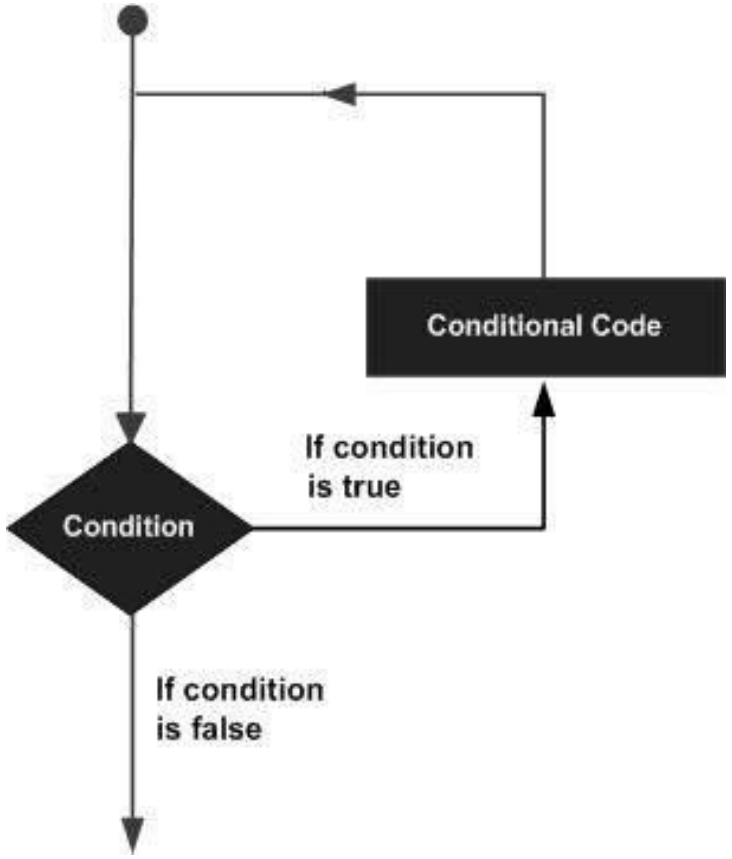
A loop statement allows us to execute a statement or group of statements multiple times.

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

1) while loop

2) for loop

3) nested loops



Looping Statements:

1) While loop:

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

Syntax:

```
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 is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Looping Statements:

e.g 1:

```
count = 1
while (count < 6):
        print 'The count is:', count
        count = count + 1
print "Good bye!"
```

O/P:

```
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
Good bye!
```

e.g 2:

```
no = input('Enter any number: ')
s = 0;
while no > 0:
        r = no % 10
        s = s + r
        no = no / 10
print 'Sum is: ',s
```

O/P:

```
Enter any number: 352
Sum is:  10
```

Looping Statements:

2) For loop:

The for **loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

Syntax:

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

e.g 1:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
        print(x)
```

O/P:

```
apple
banana
cherry
```

e.g 2:

```
str = "Python"
for i in str:
        print(i)
```

O/P:

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

Looping Statements:

e.g 3:

```
list = [1,2,3,4,5,6,7,8]
n = 5
for i in list:
    c = n*i
    print(c)
```

O/P:

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

e.g 4:

```
list = [10,30,23,43,65,12]
sum = 0
for i in list:
        sum = sum+i
print("The sum is: ",sum)
```

O/P:

```
The sum is: 183
```

Looping Statements:

For loop Using range() function:

The **range()** function is used to generate the sequence of the numbers. If we pass the range(10), it will generate the numbers from 0 to 9.

Syntax:

range(start, stop, step_size)

- The start represents the beginning of the iteration.
- The stop represents that the loop will iterate till stop-1. The **range(1,5)** will generate numbers 1 to 4 iterations. It is optional.
- The step size is used to skip the specific numbers from the iteration. By default, the step size is 1. It is optional.

Looping Statements:

e.g 1:

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

O/P: 1 2 3 4 5

e.g 2:

```
n = input("Enter the number ")
for i in range(1,11):
    c = n*i
print(c)
```

O/P: Enter the number 8

8 16 24 32 40 48 56 64 72 80

e.g 3:

```
n = input("Enter the number ")
for i in range(2,n,2):
    print(i)
```

O/P:

Enter the number 20

2 4 6 8 10 12 14 16 18

Looping Statements:

3) Nested loop:

Python programming language allows to use one loop inside another loop. The inner loop is executed n number of times for every iteration of the outer loop.

Syntax:

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

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Looping Statements:

e.g :

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print x, y
```

O/P:

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

Break Statements:

You might face a situation in which you need to exit a loop completely when an external condition is triggered or there may also be a situation when you want to skip a part of the loop and start next execution.

Python provides **break** and **continue** statements to handle such situations and to have good control on your loop.

Break Statement: The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.

The most common use for break is when some external condition is triggered requiring an exit from a loop. The **break** statement can be used in both *while* and *for* loops.

Break Statements:

e.g. 1:

```
for letter in 'Python':
      if letter == 'h':
            break
print  'Current Letter :', letter
```

O/P:

```
Current Letter : P
Current Letter : y
Current Letter : t
```

e.g. 2:

```
var = 1
while var < 10:
            print ‘Variable value :', var
            if var == 3:
                        break
            var = var + 1
print "Good bye!"
```

O/P:

```
Variable value : 1
Variable value : 2
Variable value : 3
Good bye!
```

Continue Statements:

The **continue** statement in Python returns the control to the beginning of the loop. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration.

The **continue** statement can be used in both *while* and *for* loops.

e.g.1:

```
i = 1
while(i < 10):
    if(i == 5):
        continue
    print(i)
    i = i+1
```

```
O/P:
1
2
3
4
```

Continue Statements:

e.g. 2:

```
str = 'Python'
for i in str:
   if i=='t':
      continue
   print i
```

O/P:

P

y

h

o

n

e.g. 3:

```
fruit = ['Apple', 'Banana', 'Mango', 'Papaya']
for i in fruit:
   if i=='Banana':
      continue
   print i
```

O/P:

Apple

Mango

Papaya