Python

Logical operators & conditional statements

Logical operators Definition

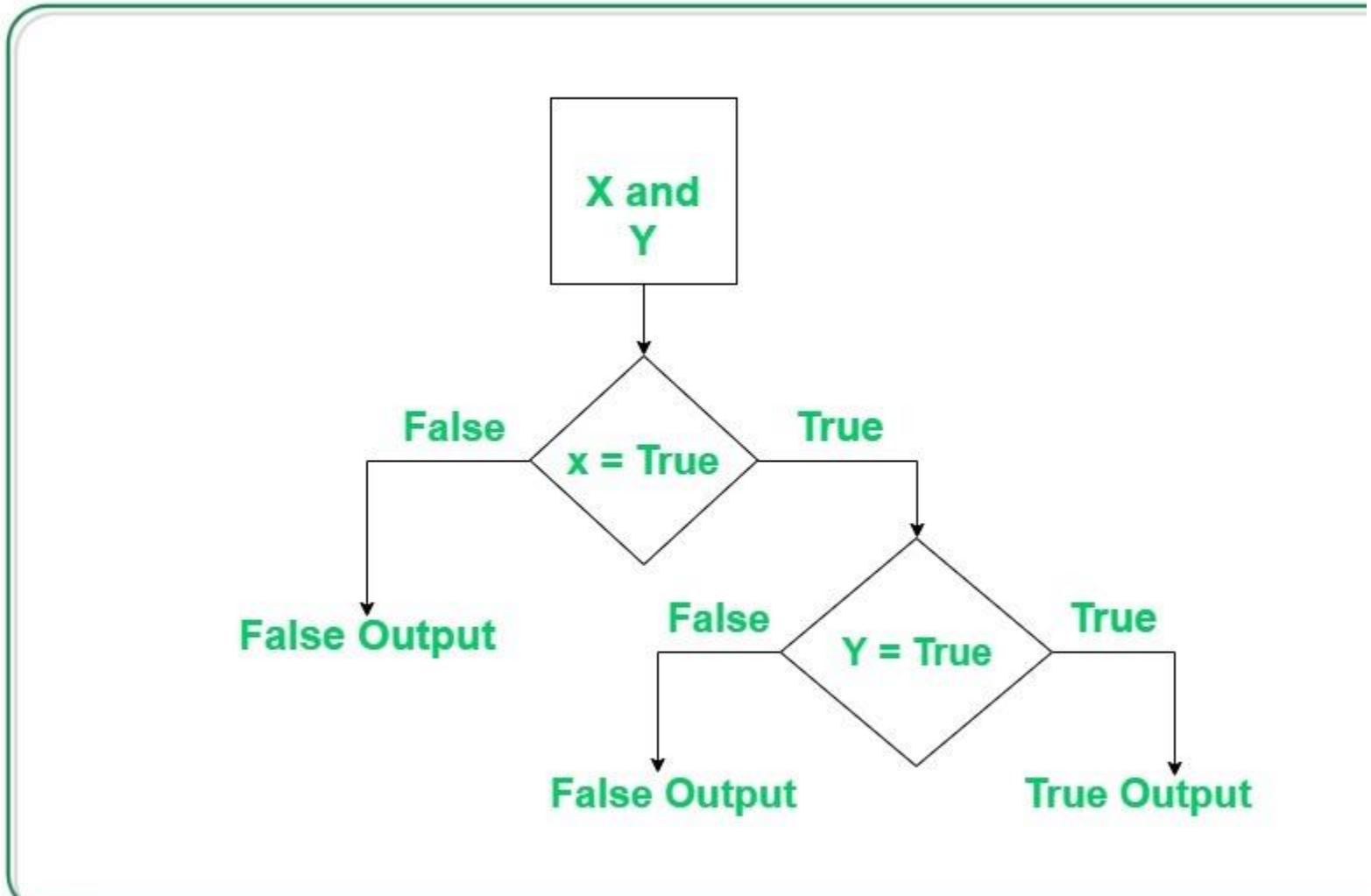
- There are three logical operators that are used to compare values. They evaluate expressions down to Boolean values, returning either True or False . These operators are **and , or , and not** and are defined in the table below

LOGICAL OPERATORS

OPERATOR	DESCRIPTION	SYNTAX	EXAMPLE
and	Logical AND: True if both the operands are true	x and y	x < 5 and x < 10
or	Logical OR: True if either of the operands is true	x or y	x < 5 or x < 4
not	Logical NOT: True if operand is false	not x	not(x < 5 and x < 10)

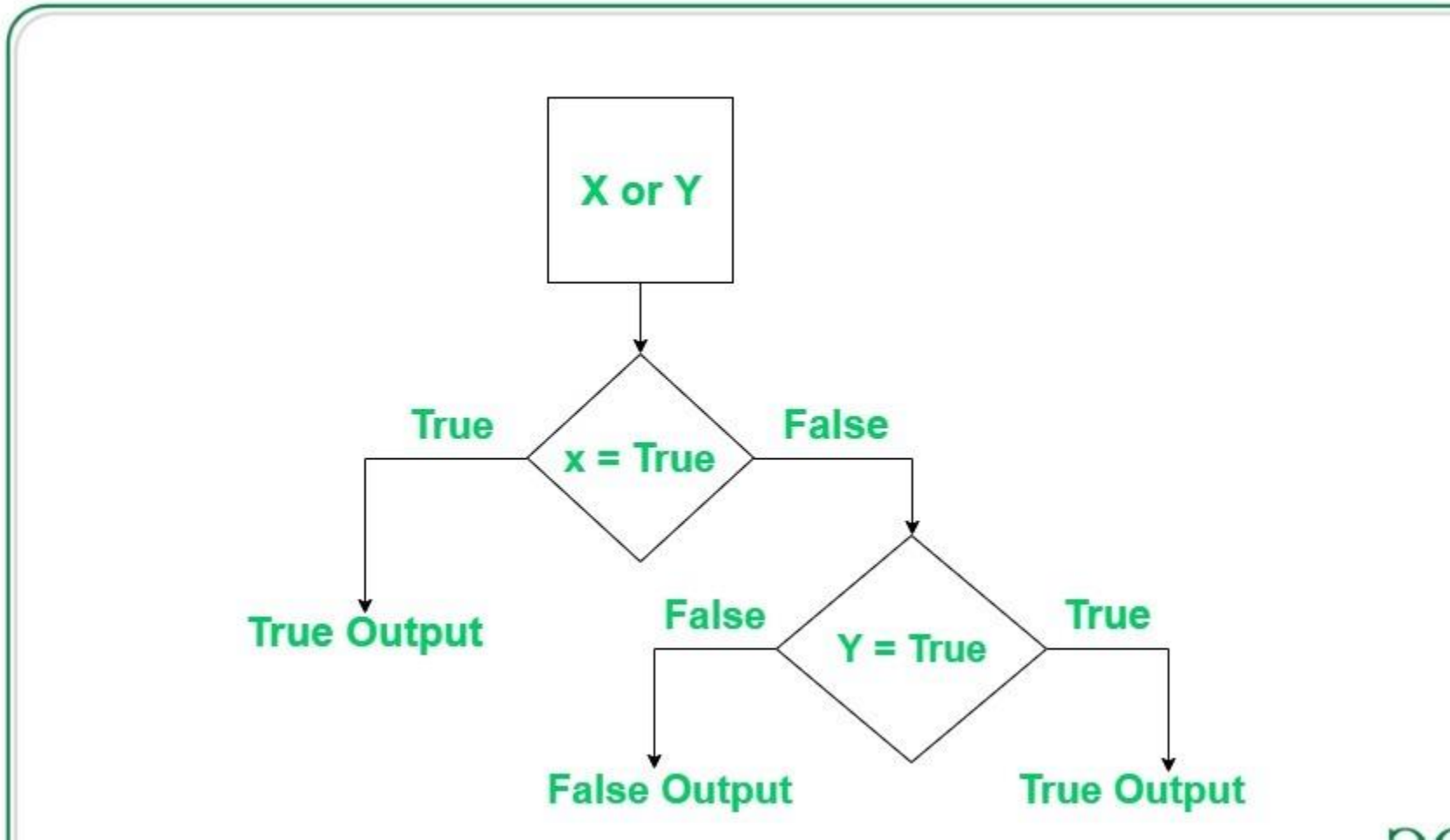
Logical AND operator

- Logical operator returns True if both the operands are True else it returns False



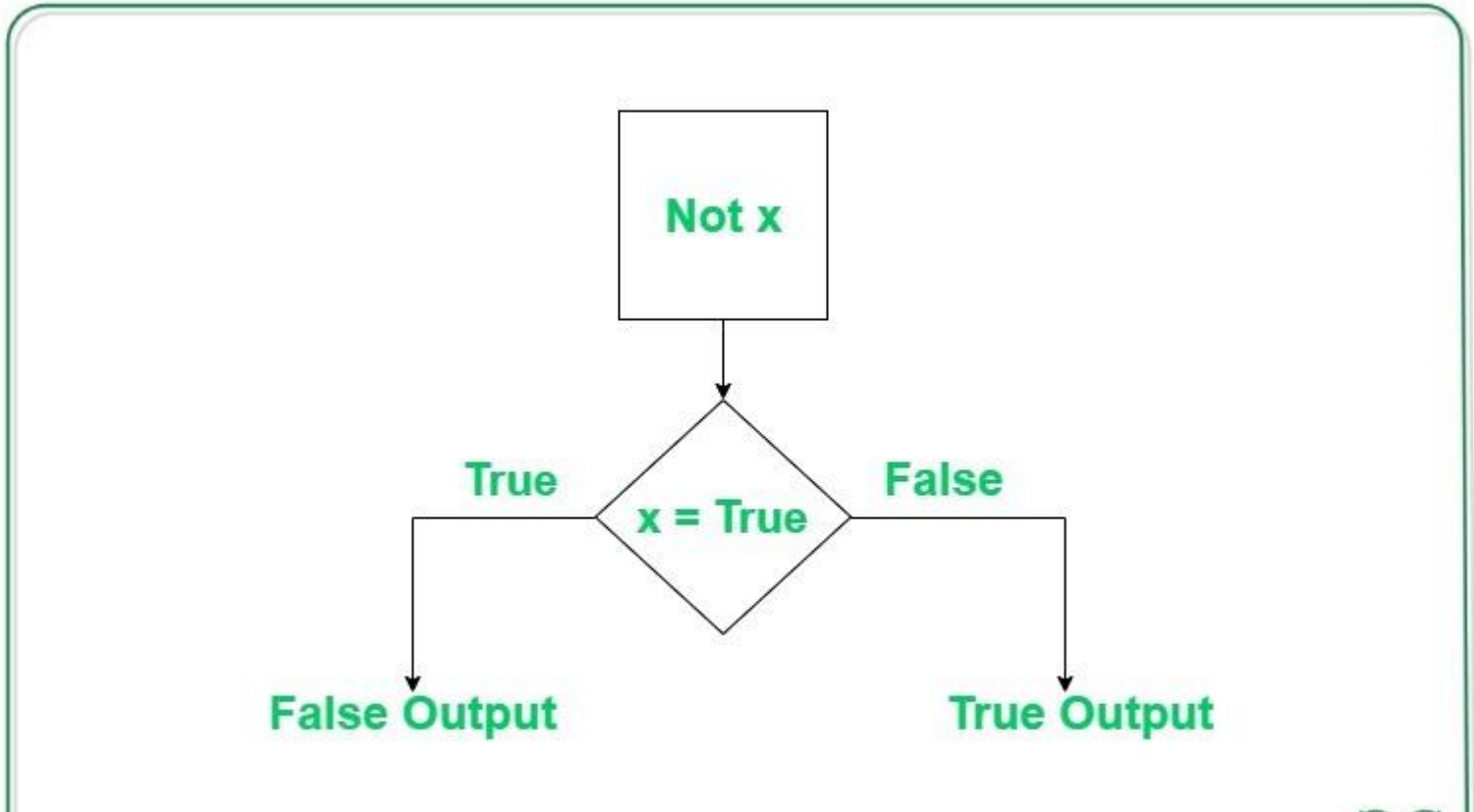
Logical or operator

Logical or operator returns True if either of the operands is True.



Logical not operator

- Logical not operator work with the single boolean value. If the boolean value is True it returns False and vice-versa.



Python Conditions and If statements

- Python supports the usual logical conditions from mathematics:
- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- These conditions can be used in several ways, most commonly in "if statements" and loops.
- An "if statement" is written by using the if keyword.

EXAMPLE OF IF STATEMENT

```
a = 33

b = 200

if b > a:

  print("b is greater than a") (THIS STATEMENT  INDENTATION which is
```

used to represent it belongs to if statement or if condition)

Elif

- The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Example

- ```
  a = 33
  b = 33
  if b > a:
    print("b is greater than a")
  elif a == b:
    print("a and b are equal")
  ```
  ```

Else

- The else keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

And

- The and keyword is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, AND if c is greater than a:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

Or - operator

- The or keyword is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, OR if a is greater than c:

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

Nested If

- You can have if statements inside if statements, this is called nested if statements.

- Example

```
x = 41
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

The pass Statement

- if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

- Example

```
a = 33
b = 200

if b > a:
  pass
```

Python Booleans

- Booleans represent one of two values: True or False.

Boolean Values

- In programming you often need to know if an expression is True or False.

You can evaluate any expression in Python, and get one of two answers, True or False.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

Example

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

Example

Print a message based on whether the condition is True or False:

```
a = 200
b = 33

if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Evaluate Values and Variables

The bool() function allows you to evaluate any value, and give you True or False in return,

Example

Evaluate a string and a number:

```
print(bool("Hello"))
print(bool(15))
```

Evaluate two variables:

```
x = "Hello"
y = 15

print(bool(x))
print(bool(y))
```

Most Values are True

Almost any value is evaluated to True if it has some sort of content.

Any string is True, except empty strings.

Any number is True, except 0.

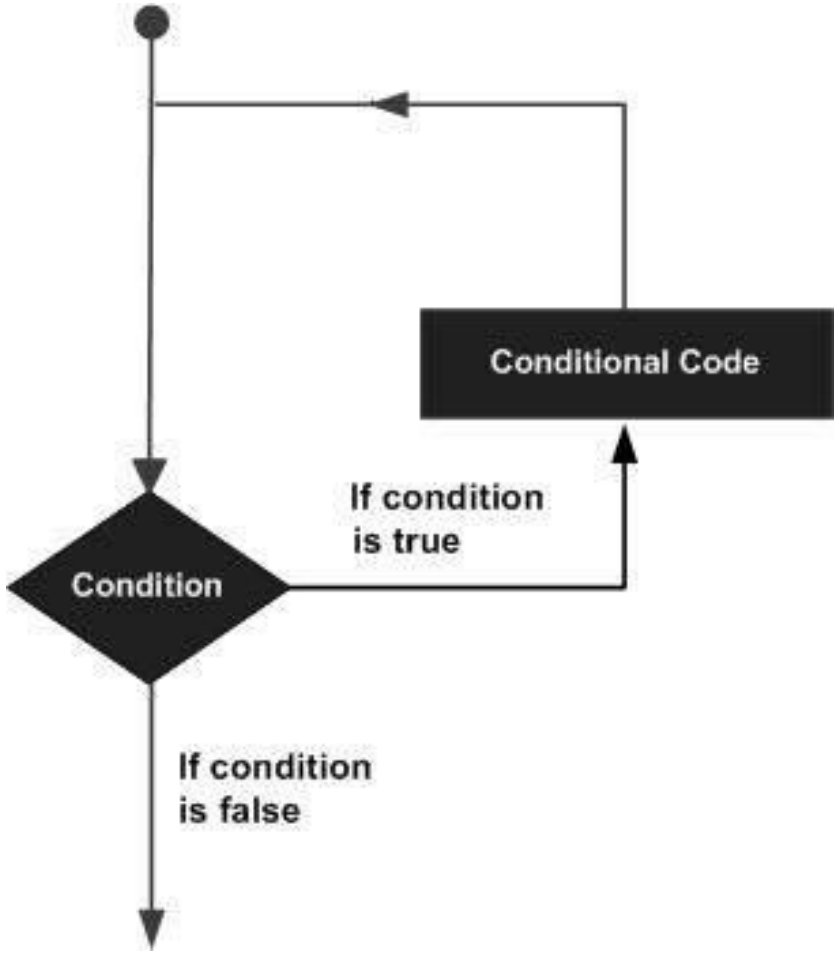
Loops

Loops are used in programming to repeat a specific block of code

Loops in python

- 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



Types of loops

- Python programming language provides following types of loops to handle looping requirements.
- **Entry check loop**

1 while loop

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

2 for loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

- **Exit Check Loop**

do…while loop

3 nested loops

You can use one or more loop inside any another while, for or do..while loop.

What is while loop in Python?

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

- We generally use this loop when we don't know the number of times to iterate beforehand.

- Syntax of while Loop in Python
- while test_expression:
- Body of while

Working of while loop

- In the while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

- In Python, the body of the while loop is determined through indentation.

- The body starts with indentation and the first unindented line marks the end.

- Python interprets any non-zero value as True. None and 0 are interpreted as False

Flow chart of while loop

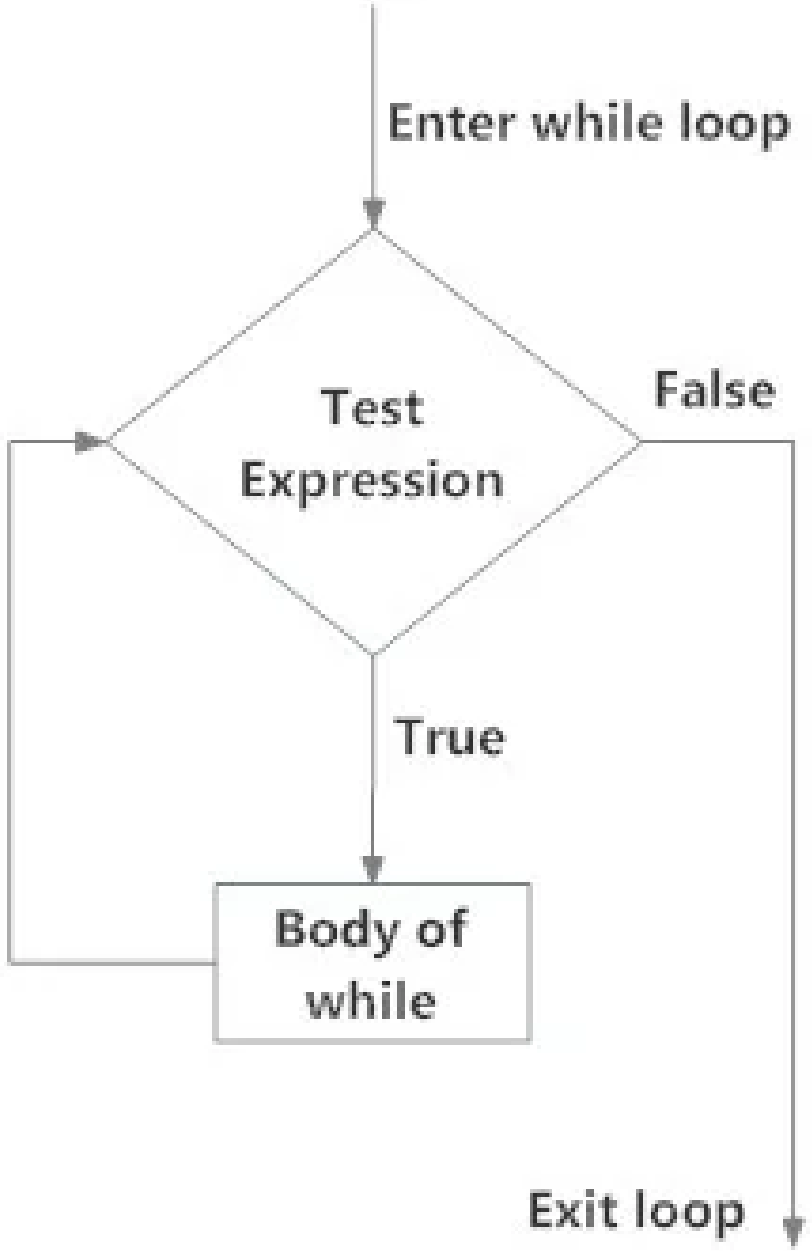


Fig: operation of while loop

Python For Loops

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

For loop flowchart

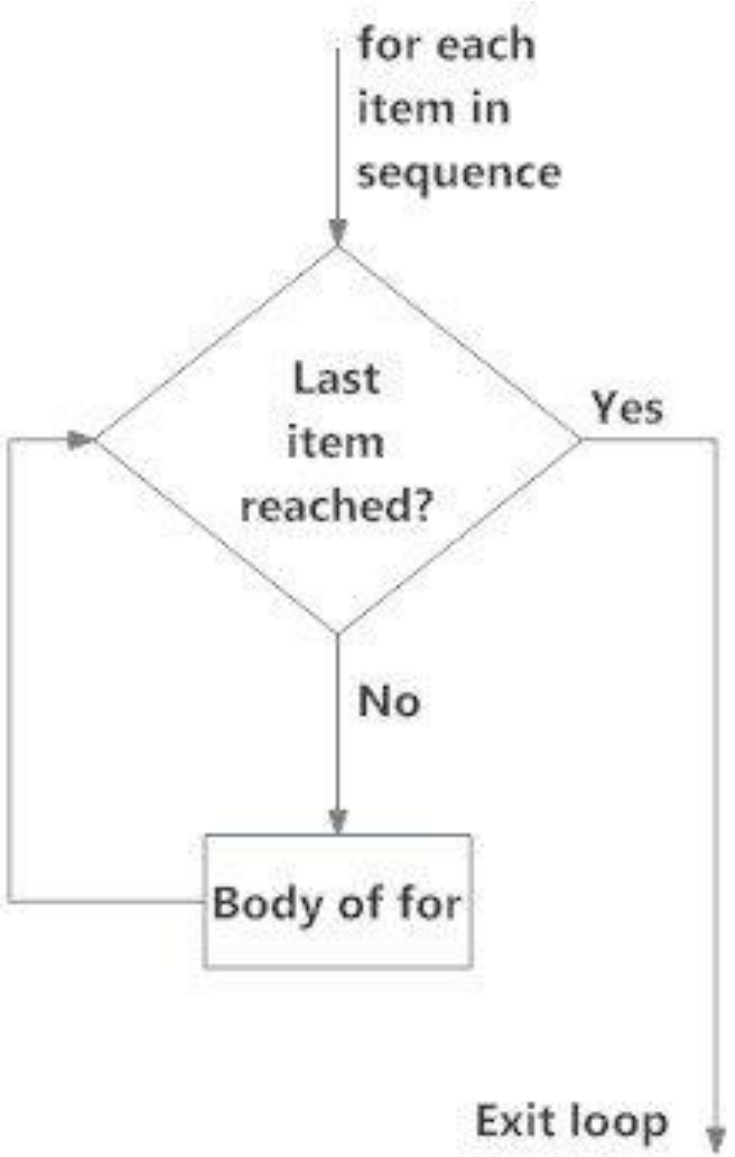


Fig: operation of for loop

Examples

```
for x in "banana":
  print(x)
```

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

Python Lists

- The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

- Creating a list is as simple as putting different comma-separated values between square brackets. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

- Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

List

- Lists are used to store multiple items in a single variable.
- Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

- Example:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

Accessing Values in Lists

- To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

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

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

- You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

```
list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

- Note − append() method is discussed in subsequent section.

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

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

- To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

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

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Note − remove() method is discussed in subsequent section.

Basic List Operations

- Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.
- In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

The break Statement

- With the break statement we can stop the loop before it has looped through all the items:

The continue Statement

- With the continue statement we can stop the current iteration of the loop, and continue with the next:

The range() Function

- To loop through a set of code a specified number of times, we can use the range() function

- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Examples of range in for loop statements

- print(range(10))
- print(list(range(10)))
- print(list(range(2, 8)))
- print(list(range(2, 20, 3)))

for loop with else

- A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.
- The break keyword can be used to stop a for loop. In such cases, the else part is ignored.
- Hence, a for loop's else part runs if no break occurs.

Here is an example to illustrate this.

```
digits = [0, 1, 5]
for i in digits:
    print(i)
else:
    print("No items left.")
```

While loop with else

- The else part is executed if the condition in the while loop evaluates to False.
- The while loop can be terminated with a break statement. In such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

- Here is an example to illustrate this.
- '''Example to illustrate the use of else statement with the while loop'''

```
counter = 0
while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Pass statement

- 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
- Example of pass statement in for loop

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

Tuple

- A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

- Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

- tup1 = ('physics', 'chemistry', 1997, 2000);
- tup2 = (1, 2, 3, 4, 5);
- tup3 = "a", "b", "c", "d";

Advantages of Tuple over List

- Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Programming using tuples (Do in Colab)

Different types of tuples

```
# Empty tuple
my_tuple = ()
print(my_tuple)
```

Tuple having integers

```
my_tuple = (1, 2, 3)
print(my_tuple)
```

tuple with mixed datatypes

```
my_tuple = (1, "Hello", 3.4)
print(my_tuple)
```

nested tuple

```
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

Unpacking of Tuples

```
my_tuple = 3, 4.6, "dog"
print(my_tuple)

# tuple unpacking is also possible
a, b, c = my_tuple

print(a)      # 3
print(b)      # 4.6
print(c)      # dog
```

Method used in tuple

- my_tuple = ('a', 'p', 'p', 'l', 'e',)

- print(my_tuple.count('p')) # Output: 2
- print(my_tuple.index('l')) # Output: 3

Membership test in tuple

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)

# In operation
print('a' in my_tuple)
print('b' in my_tuple)

# Not in operation
print('g' not in my_tuple)
```

Tuples used in for loop

```
for name in ('John', 'Kate’,’manju’,’computer’):
    print("Hello", name)
```

Creating a Tuple

- A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

- A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

Access Tuple Elements

- There are various ways in which we can access the elements of a tuple.

Indexing

- We can use the index operator [] to access an item in a tuple, where the index starts from 0.

- So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an Index Error.

- The index must be an integer, so we cannot use float or other types. This will result in Type Error.

Set

- Sets are used to store multiple items in a single variable.
- Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A set is a collection which is *unordered*, *unchangeable*, and *unindexed*.
- Sets are written with curly brackets.

Dictionary

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and does not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values: