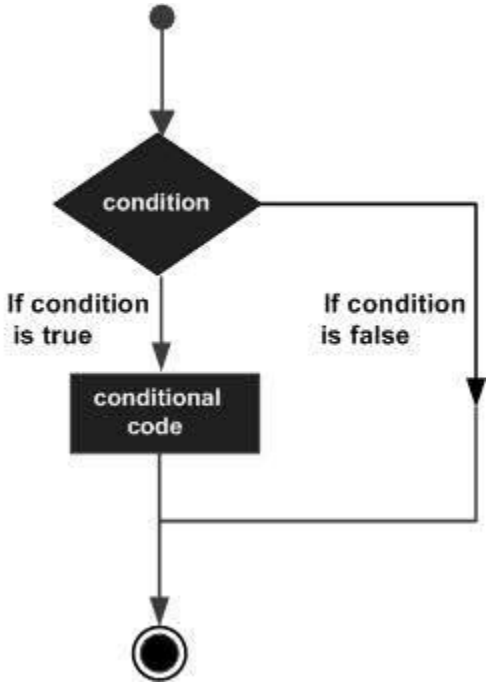
Unit 2

Conditionals

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

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. We 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.

Python programming language provides following types of decision making statements.

Sr.No.	Statement & Description
1	if statements An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statements An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
3.	elif elif is an abbreviation of else if
4	nested if statements We can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

Let us go through each decision making briefly −

1. **IF Statement**

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a **one-line if** clause –

```
var = 100
if ( var == 100 ) : print "Value of expression is 100"
```

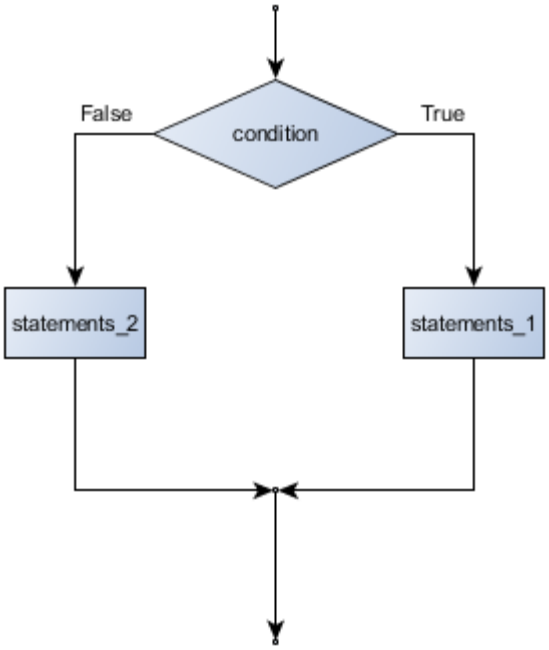
When the above code is executed, it produces the following result −

Value of expression is 100

Here it is printing because conditions is met and get true.

2. **The if else statement**

It is frequently the case that you want one thing to happen when a condition it true, and **something else** to happen when it is false. For that we have the if else statement.



Example :

```
if food == 'spam':
   print('Ummmm, my favorite!')
else:
   print("No, I won't have it. I want spam!")
```

Here, the first print statement will execute if food is equal to 'spam', and the print statement indented under the else clause will get executed when it is not.

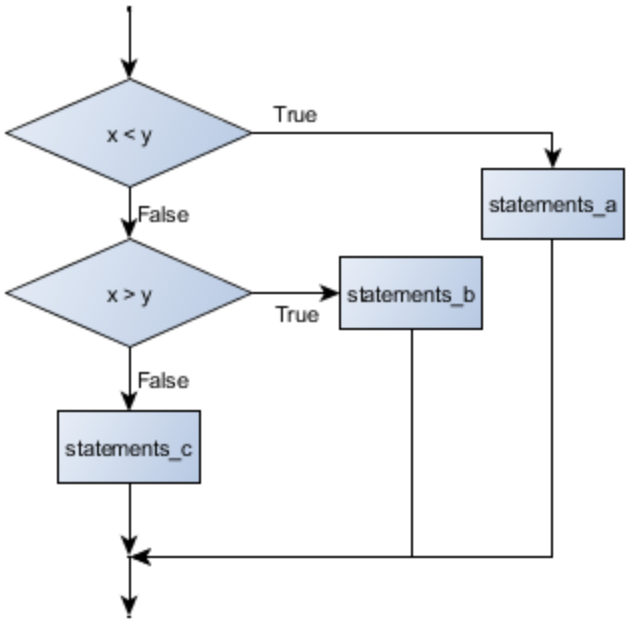
3. elif statements

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

SYNTAX

```
if x < y:
   STATEMENTS_A
elif x > y:
   STATEMENTS_B
else:
   STATEMENTS_C
```

Flowchart of this chained conditional



elif is an abbreviation of else if. Again, exactly one branch will be executed. There is no limit of the number of elif statements but only a single (and optional) final else statement is allowed and it must be the last branch in the statement:

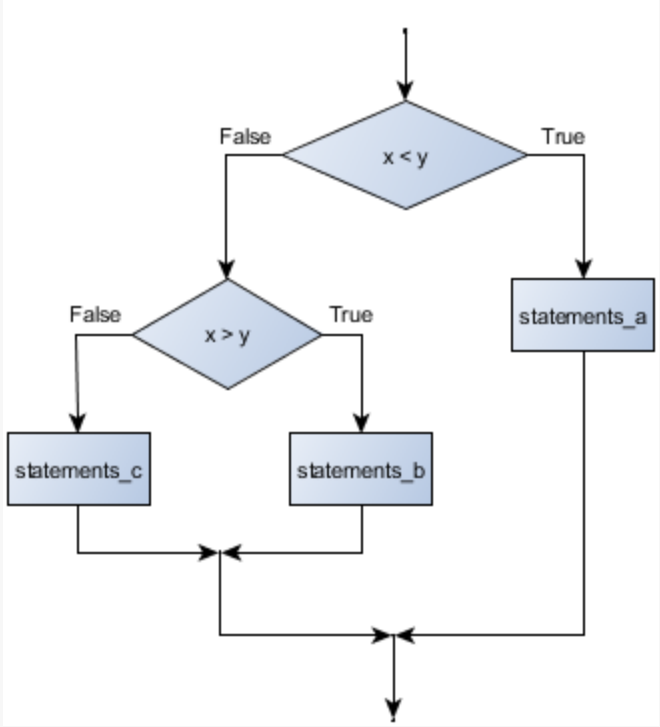
```
if choice == 'a':
    print("You chose 'a'.")
elif choice == 'b':
    print("You chose 'b'.")
elif choice == 'c':
    print("You chose 'c'.")
else:
    print("Invalid choice.")
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

4. Nested conditionals

One condition can also be **nested** within another. It is the same theme of composability.

Flowchart and Syntax:




```
if x < y:
    STATEMENTS_A
else:
    if x > y:
        STATEMENTS_B
    else:
        STATEMENTS_C
```

The outer conditional contains two branches. The second branch contains another if statement, which has two branches of its own. Those two branches could contain conditional statements as well.

Indentation of the statements makes the structure apparent.

Example:

```
i = 10
if (i == 10):
    #  First if statement
    if (i < 15):
        print ("i is smaller than 15")
    # Nested - if statement
    # Will only be executed if statement above
    # it is true
    if (i < 12):
        print ("i is smaller than 12 too")
    else:
        print ("i is greater than 15")
```

Output:

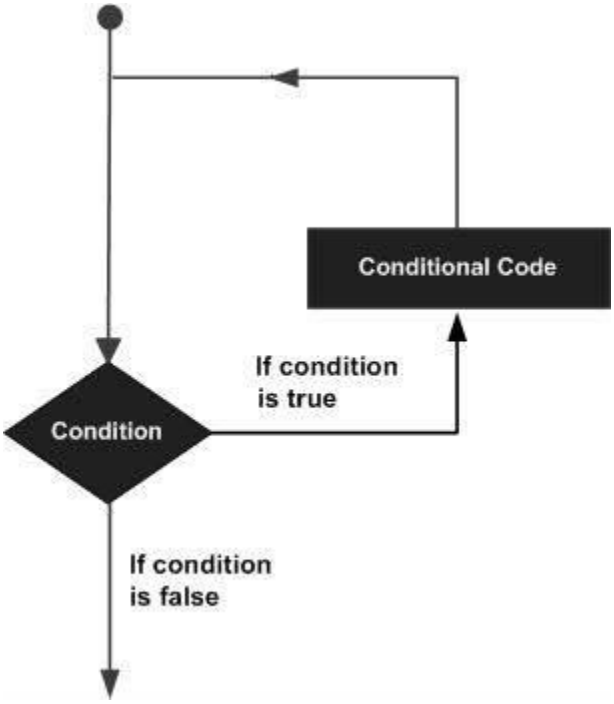
```
i is smaller than 15
i is smaller than 12 too
```

Loops

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.

Sr.No.	Loop Type & Description
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.
3	nested loops You can use one or more loop inside any another while, for or do..while loop.

While Loop

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 beforehand, the number of times to iterate.

Syntax of while Loop

```
while test_expression:

    Body of while
```

In 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. No brackets are used for opening and closing of loop body. 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.

Flowchart of while Loop

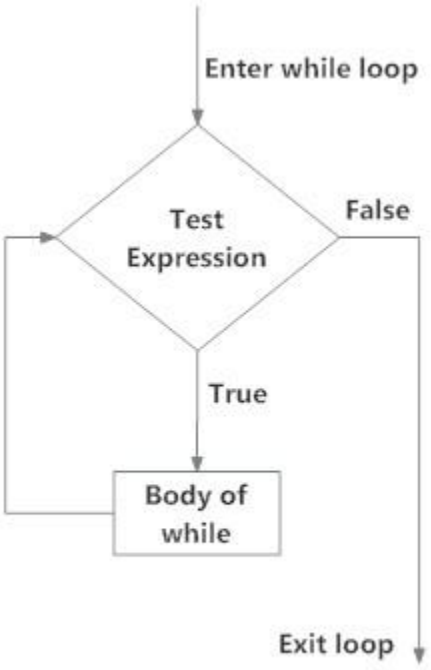


Fig: operation of while loop

Example:

Program to add natural

```
n = 10
# initialize sum and counter
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i+1    # update counter
# print the sum
print("The sum is", sum)
```

When you run the program, the output will be:

```
Enter n: 10
The sum is 55
```

In the above program, the test expression will be `True` as long as our counter variable `i` is less than or equal to `n` (10 in our program).

We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop).

Finally the result is displayed.

FOR LOOP

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.
Syntax of for Loop

```
for val in sequence:

        Body of for
```

Here, `val` is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Flowchart of for Loop

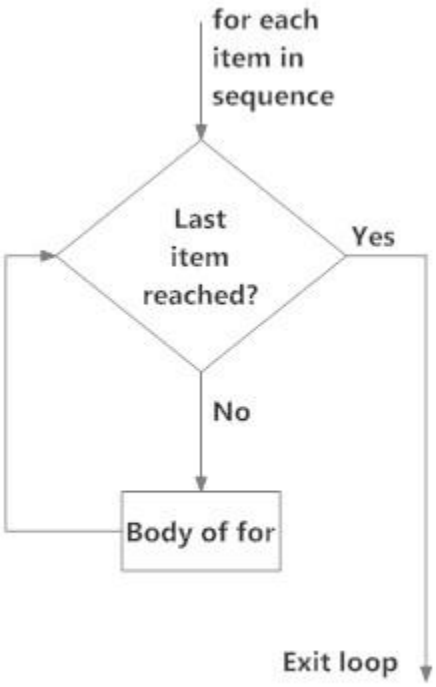


Fig: operation of for loop

Example

```
# Program to find the sum of all numbers stored in a list

# List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:

        sum = sum+val

print("The sum is", sum)
```

when you run the program, the output will be:

```
The sum is 48
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	break statement Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	pass statement The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

In Python, break and continue statements can alter the flow of a normal loop.
Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.
The break and continue statements are used in these cases.

---

break statement

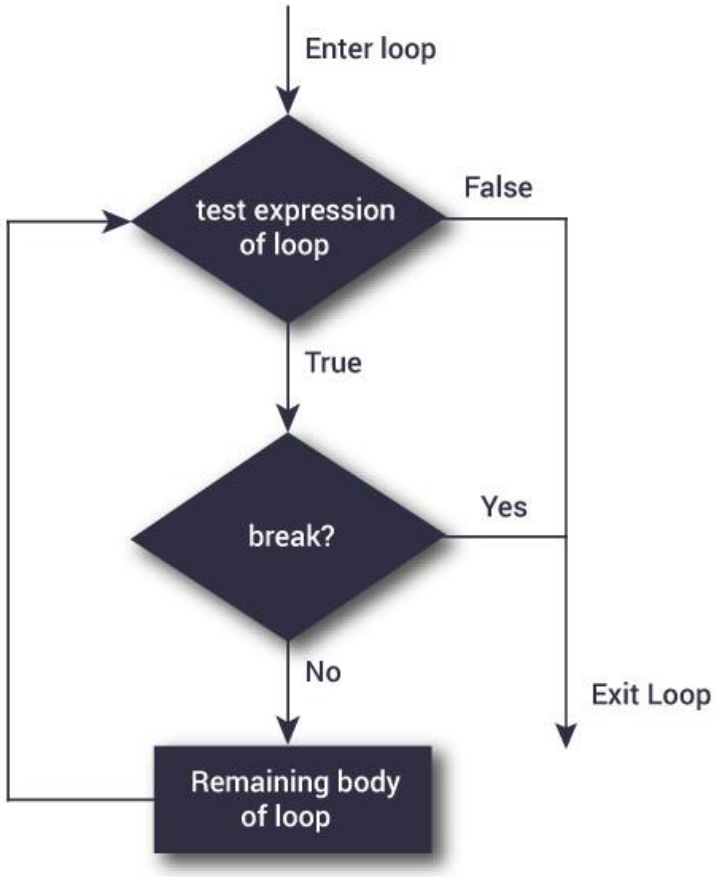
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

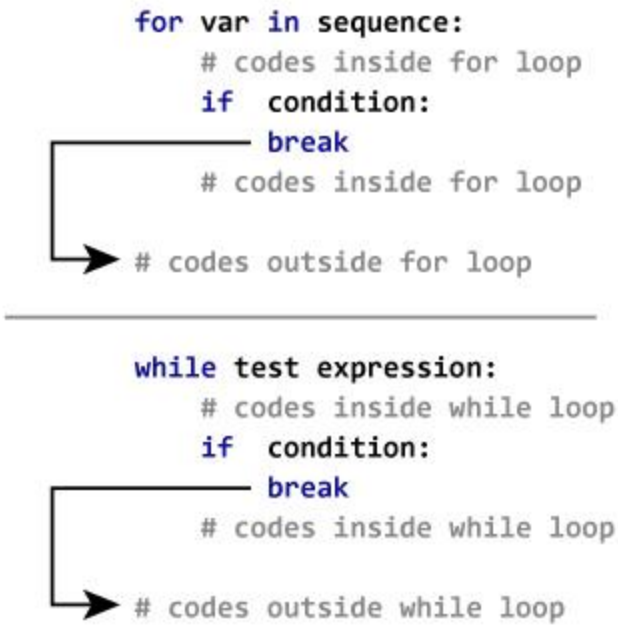
Syntax of break

```
break
```

Flowchart of break



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



Example:

```
# Use of break statement inside loop

for val in "string":
    if val == "i":
        break
    print(val)
print("The end")
```

Run

Powered by DataCamp

Output

```
s
t
r
The end
```

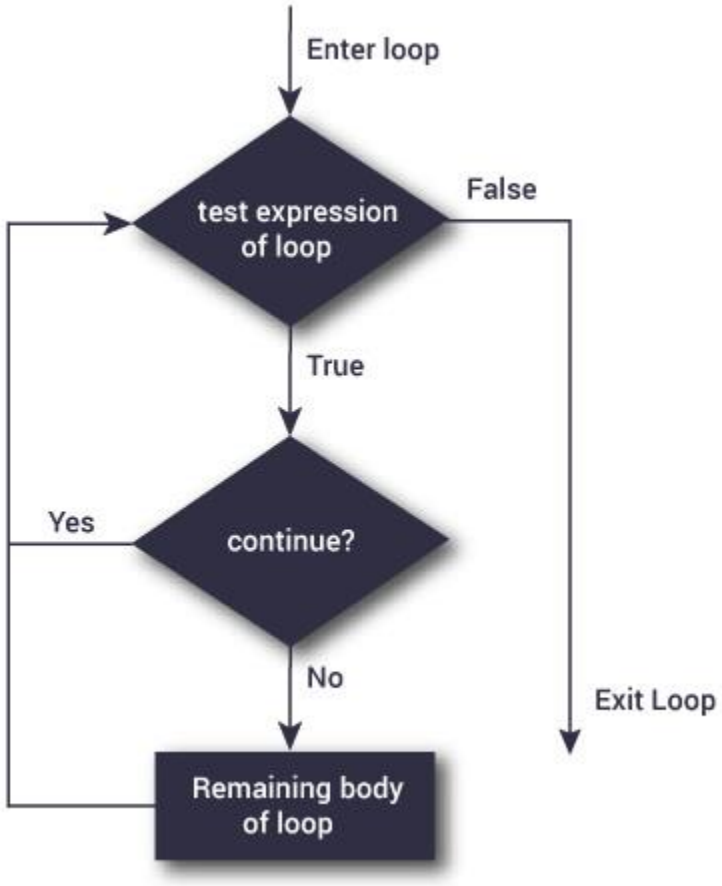
Continue Statement:

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

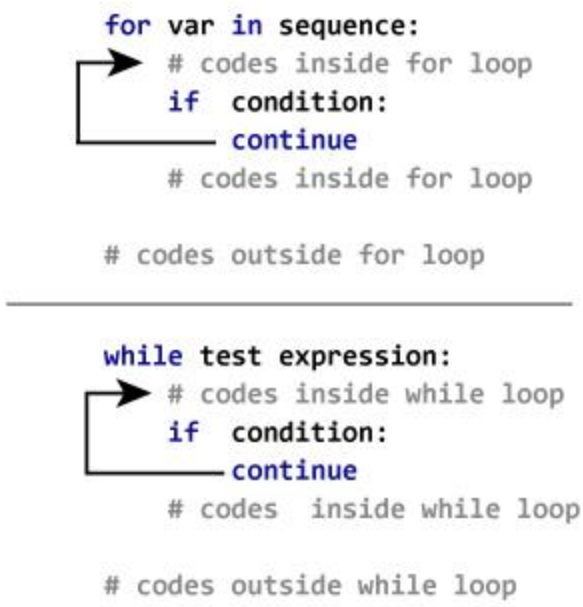
Syntax of Continue

```
continue
```

Flowchart of continue



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



Example:

```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)
print("The end")
```

Run

Output

```
s
t
r
n
g
The end
```

This program is same as the above example except the break statement has been replaced with continue.