Control Structure: If statements, Loops, Break and Continue statements, Pass.

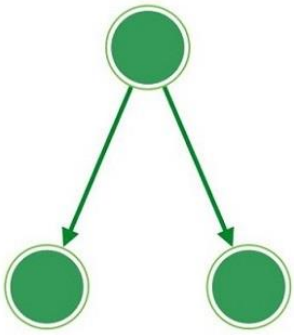
Python program control flow is regulated by various types of conditional statements, loops, and function calls. By default, the instructions in a computer program are executed in a sequential manner, from top to bottom, or from start to end. However, such sequentially executing programs can perform only simplistic tasks. We would like the program to have a decision-making ability, so that it performs different steps depending on different conditions.

Most programming languages including Python provide functionality to control the flow of execution of instructions. Normally, there are two type of control flow statements in any programming language and Python also supports them.

Decision-making Statements

Decision making statements are used in the Python programs to make them able to decide which of the alternative group of instructions to be executed, depending on value of a certain Boolean expression.

The following diagram illustrates how decision-making statements work −



The if Statement

Python provides if..elif..else control statements as a part of decision marking. Following is a simple example which makes use of if..elif..else. You can try to run this program using different marks and verify the result.

```
marks = 80
result = ""
if marks < 30:
   result = "Failed"
elif marks > 75:
   result = "Passed with distinction"
else:
   result = "Passed"

print(result)
```

This will produce following result:

Passed with distinction

The match Statement

Python supports Match-Case statement, which can also be used as a part of decision making. Following is a simple example which makes use of match statement.

```
def checkVowel(n):
  match n:
    case 'a': return "Vowel alphabet"
    case 'e': return "Vowel alphabet"
    case 'i': return "Vowel alphabet"
    case 'o': return "Vowel alphabet"
    case 'u': return "Vowel alphabet"
    case _: return "Simple alphabet"
print (checkVowel('a'))
print (checkVowel('m'))
print (checkVowel('o'))
```

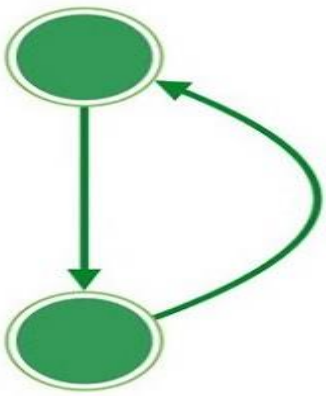
This will produce following result:

Vowel alphabet
Simple alphabet
Vowel alphabet

Loops or Iteration Statements

Most of the processes require a group of instructions to be repeatedly executed. In programming terminology, it is called a **loop**. Instead of the next step, if the flow is redirected towards any earlier step, it constitutes a loop.

The following diagram illustrates how the looping works −



If the control goes back unconditionally, it forms an infinite loop which is not desired as the rest of the code would never get executed.

In a conditional loop, the repeated iteration of block of statements goes on till a certain condition is met. Python supports a number of loops like for loop, while loop which we will study in next chapters.

The for Loop

Following is an example which makes use of For Loop to iterate through an array in Python:

```
words = ["one", "two", "three"]
for x in words:
  print(x)
```

This will produce following result:

one
two
three

The while Loop

Following is an example which makes use of While Loop to print first 5 numbers in Python:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

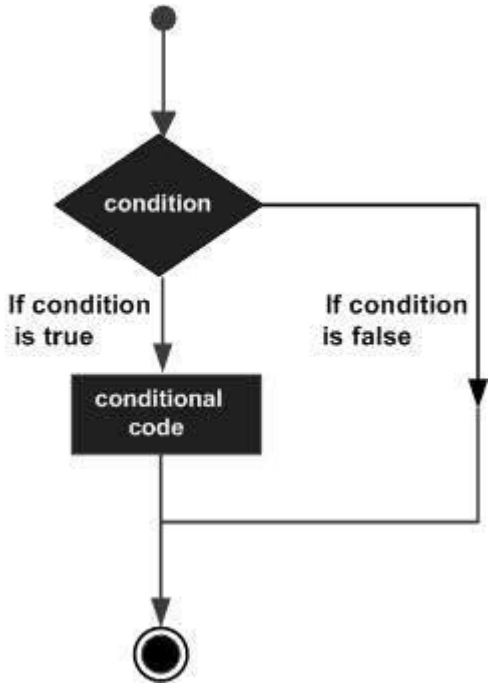
This will produce following result:

1
2
3
4
5

Python's decision making functionality is in its keywords − **if.. elif ...else**. The **if** keyword requires a Boolean expression, followed by colon (:) symbol. The colon (:) symbol starts an indented block. The statements with the same level of indentation are executed if the boolean expression in **if statement** is **True**. If the expression is not True (False), the interpreter bypasses the indented block and proceeds to execute statements at earlier indentation level.

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.

Python programming language provides following types of decision making statements. Click the following links to check their detail.

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

Let us go through each decision making briefly −

Single Statement Suites

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 −

```
#!/usr/bin/python

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

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

Value of expression is 100

Good bye!

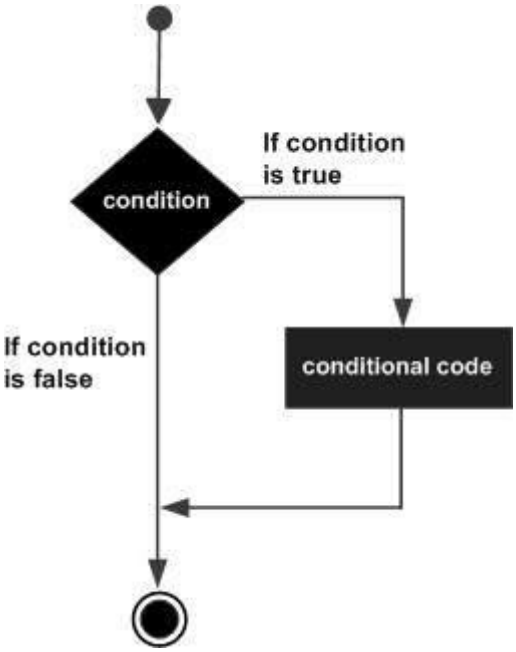
Python **if** statement is similar to that of other languages. The **if** statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax

```
if expression:
   statement(s)
```

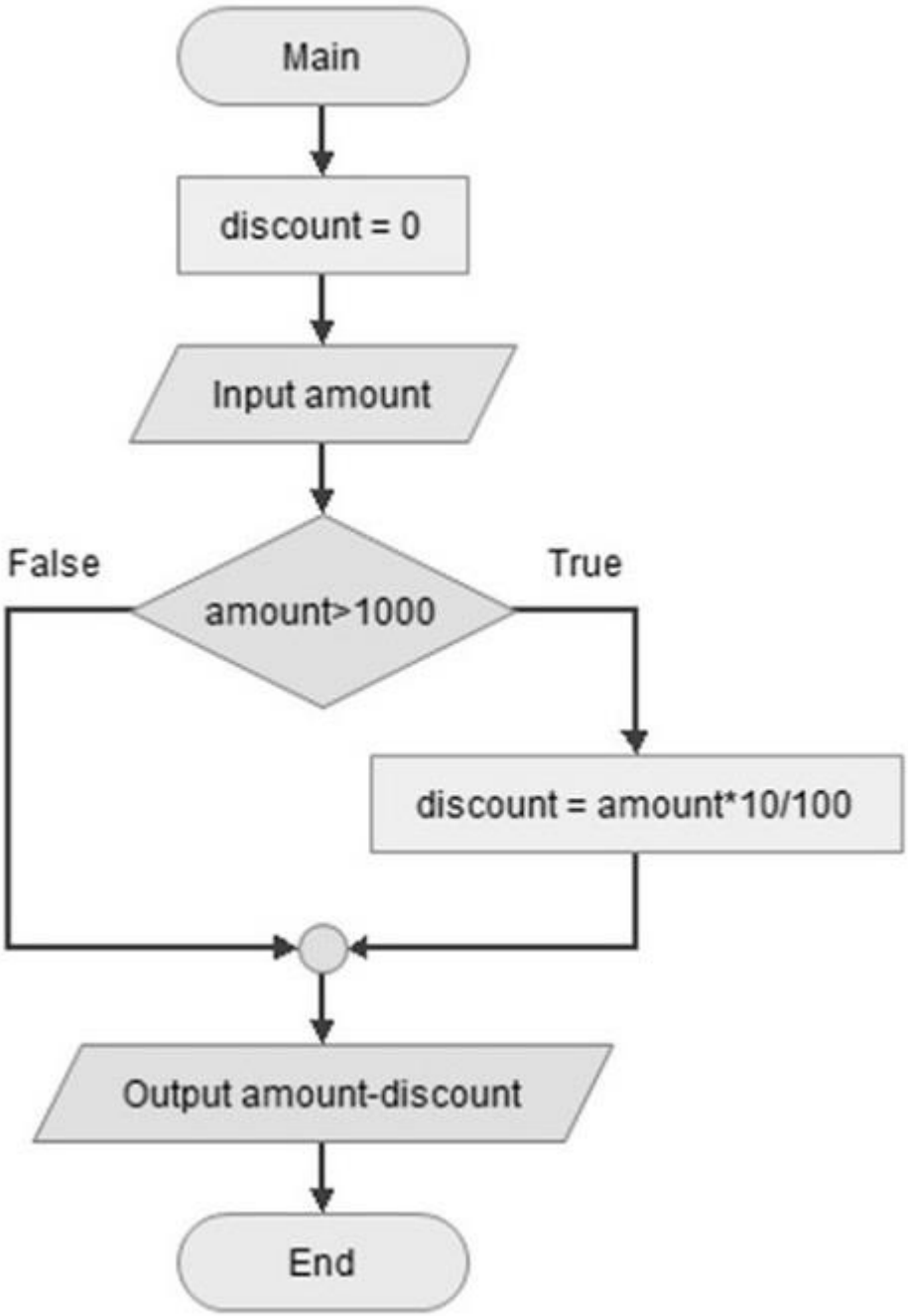
If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

Flow Diagram




Example

Let us consider an example of a customer entitiled to 10% discount if his purchase amount is > 1000; if not, then no discount is applicable. The following flowchart shows the whole decision making process.




In Python, we first set a discount variable to 0 and accept the amount as input from user.

Then comes the conditional statement if amount > 1000. Put **:** symbol that starts conditional block wherein discount applicable is calculated. Obviously, discount or not, next statement by default prints amount-discount. If applied, it will be subtracted, if not it is 0.

```
discount = 0
amount = 1200

# Check he amount value
if amount > 1000:
   discount = amount * 10 / 100

print("amount = ", amount - discount)
```

Here the amout is 1200, hence discount 120 is deducted. On executing the code, you will get the following **output** −

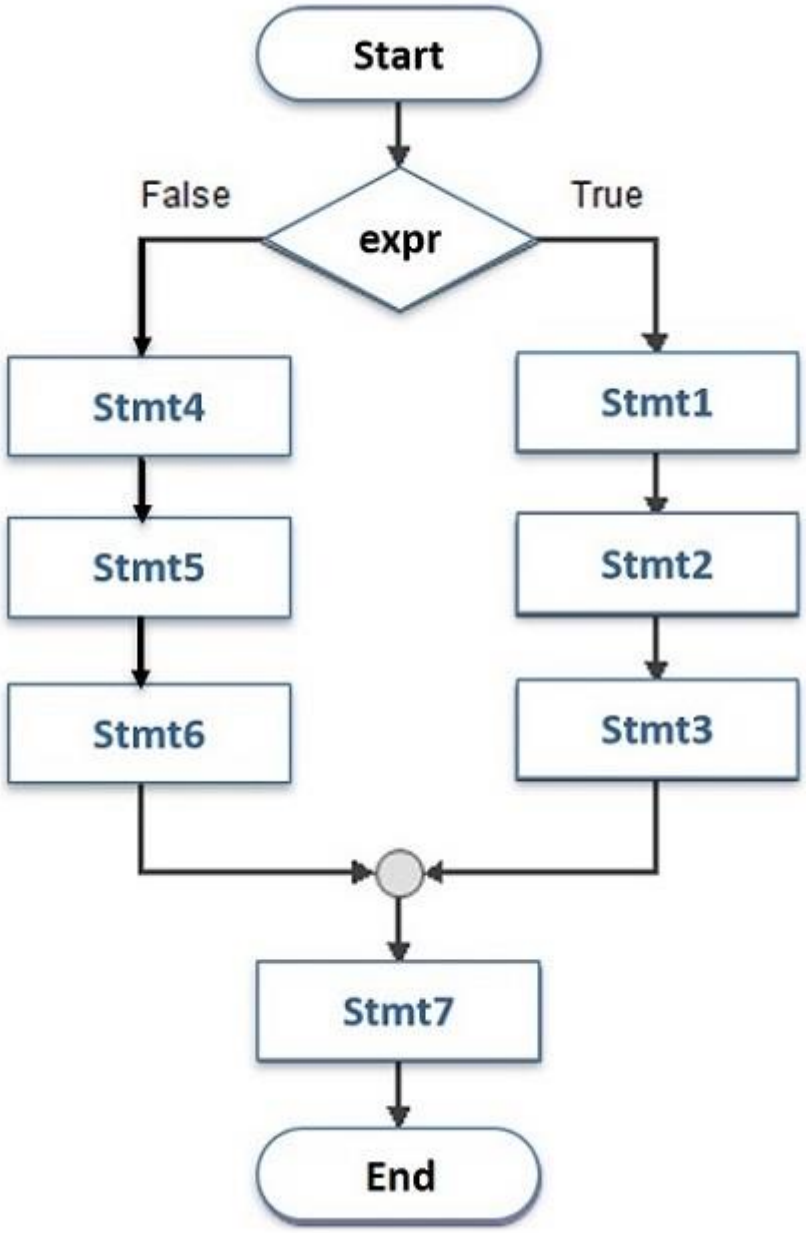
amount = 1080.0

Change the variable amount to 800, and run the code again. This time, no discount is applicable. And, you will get the following output −

amount = 800

Python − if-else Statement

Along with the **if** statement, **else** keyword can also be optionally used. It provides an alternate block of statements to be executed if the Boolean expression (in if statement) is not true. this flowchart shows how else block is used.




If the expr is True, block of stmt1,2,3 is executed then the default flow continues with stmt7. However, the If expr is False, block stmt4,5,6 runs then the default flow continues.

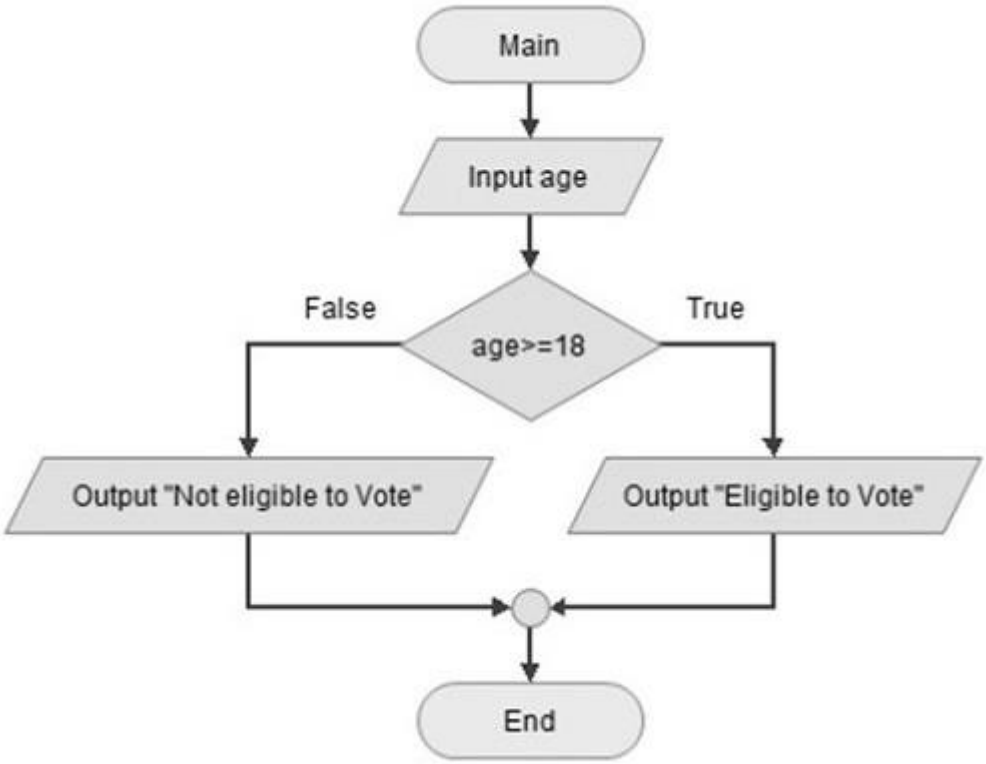
Syntax

Python implementation of the above flowchart is as follows −

```
if expr==True:
   stmt1
   stmt2
   stmt3
else:
   stmt4
   stmt5
   stmt6
Stmt7
```

Example

Let us understand the use of **else** clause with following example. The variable age can take different values. If the expression "age > 18" is true, message you are eligible to vote is displayed otherwise not eligible message should be displayed. Following flowchart illustrates this logic.




Its Python implementation is simple.

```
age=25
print ("age: ", age)
if age >=18:
   print ("eligible to vote")
else:
```

```
   print ("not eligible to vote")
```

To begin, set the integer variable "age" to 25.

Then use the **if** statement with "age>18" expression followed by ":" which starts a block; this will come in action if "age>=18" is true.

To provide **else** block, use **else:** the ensuing indented block containing message **not eligible** will be in action when "age>=18" is false.

On executing this code, you will get the following **ouput** −

```
age: 25
eligible to vote
```

To test the the **else** block, change the **age** to 12, and run the code again.

```
age: 12
not eligible to vote
```

Python − 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.

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

Syntax

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

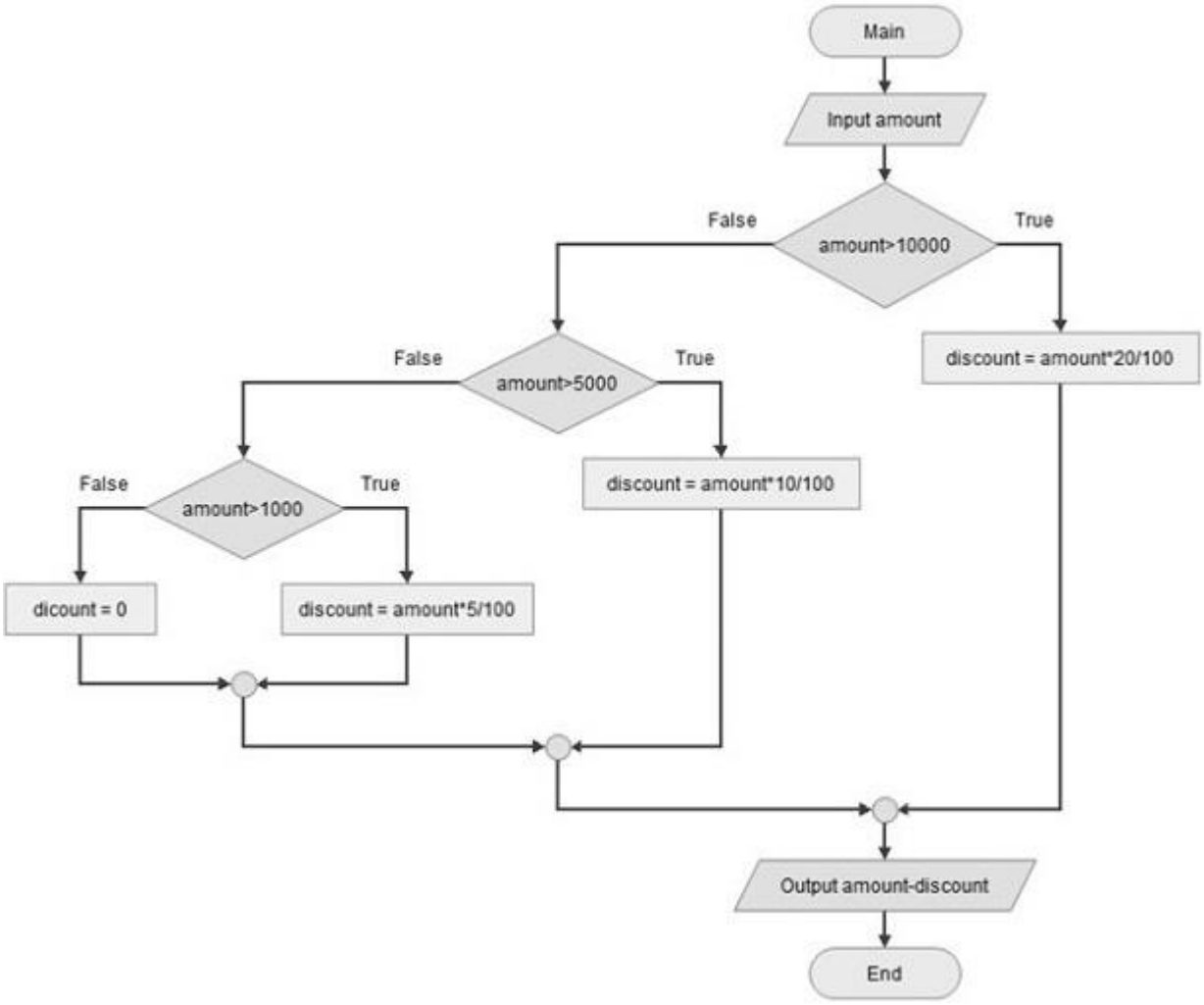
Example

Let us understand how elif works, with the help of following example.

The discount structure used in an earlier example is modified to different slabs of discount −

- 20% on amount exceeding 10000,
- 10% for amount between 5-10000,
- 5% if it is between 1 to 5000.
- no discount if amount<1000

The following flowchart illustrates these conditions −




Example

We can write a Python code for the above logic with **if-else** statements −

```
amount = 2500
print('Amount = ',amount)
if amount > 10000:
   discount = amount * 20 / 100
else:
   if amount > 5000:
      discount = amount * 10 / 100
   else:
      if amount > 1000:
         discount = amount * 5 / 100
      else:
         discount = 0

print('Payable amount = ',amount - discount)
```

Set **amount** to test all possible conditions: 800, 2500, 7500 and 15000. The **outputs** will vary accordingly −

Amount: 800
Payable amount = 800
Amount: 2500
Payable amount = 2375.0
Amount: 7500
Payable amount = 6750.0
Amount: 15000
Payable amount = 12000.0

While the code will work perfectly ok, if you look at the increasing level of indentation at each if and else statement, it will become difficult to manage if there are still more conditions.

The **elif** statement makes the code easy to read and comprehend.

**Elif** is short for **else if**. It allows the logic to be arranged in a cascade of **elif** statements after the first if statement. If the first **if** statement evaluates to false, subsequent elif statements are evaluated one by one and comes out of the cascade if any one is satisfied.

Last in the cascade is the **else** block which will come in picture when all preceding if/elif conditions fail.

```
amount = 2500
print('Amount = ',amount)
if amount > 10000:
   discount = amount * 20 / 100
elif amount > 5000:
   discount = amount * 10 / 100
elif amount > 1000:
   discount = amount * 5 / 100
else:
   discount=0

print('Payable amount = ',amount - discount)
```

Set **amount** to test all possible conditions: 800, 2500, 7500 and 15000. The **outputs** will vary accordingly −

Amount: 800
Payable amount = 800
Amount: 2500
Payable amount = 2375.0
Amount: 7500
Payable amount = 6750.0
Amount: 15000
Payable amount = 12000.0

Python supports **nested if statements** which means we can use a conditional **if** of **else...if** statement inside an existing **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

The syntax of the nested **if...elif...else** construct will be like this −

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)3
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Example

Now let's take a Python code to understand how it works −

```
num=8
print ("num = ",num)
if num%2==0:
   if num%3==0:
      print ("Divisible by 3 and 2")
   else:
      print ("divisible by 2 not divisible by 3")
else:
   if num%3==0:
      print ("divisible by 3 not divisible by 2")
   else:
      print ("not Divisible by 2 not divisible by 3")
```

When the above code is executed, it produces the following **output** −

num = 8
divisible by 2 not divisible by 3
num = 15
divisible by 3 not divisible by 2
num = 12

Divisible by 3 and 2

num = 5

not Divisible by 2 not divisible by 3

Before its 3.10 version, Python lacked a feature similar to switch-case in C or C++. In Python 3.10, a pattern matching technique called **match-case** has been introduced, which is similar to the **switch-case** construct available in C/C++/Java etc.

Python match-case Statement

A Python **match-case** statement takes an expression and compares its value to successive patterns given as one or more case blocks. The usage is more similar to pattern matching in languages like Rust or Haskell than a switch statement in C or C++. Only the first pattern that matches gets executed. It is also possible to extract components (sequence elements or object attributes) from the value into variables.

Syntax

The basic usage of **match-case** is to compare a variable against one or more values.

```
match variable_name:
  case 'pattern 1' : statement 1
  case 'pattern 2' : statement 2
  ...
  case 'pattern n' : statement n
```

Example

The following code has a function named weekday(). It receives an integer argument, matches it with all possible weekday number values, and returns the corresponding name of day.

```
def weekday(n):
  match n:
    case 0: return "Monday"
    case 1: return "Tuesday"
    case 2: return "Wednesday"
    case 3: return "Thursday"
    case 4: return "Friday"
    case 5: return "Saturday"
    case 6: return "Sunday"
    case _: return "Invalid day number"
print (weekday(3))
print (weekday(6))
print (weekday(7))
```

Output

On executing, this code will produce the following output −

Thursday

Sunday

Invalid day number

The last case statement in the function has "_" as the value to compare. It serves as the wildcard case, and will be executed if all other cases are not true.

Combined Cases

Sometimes, there may be a situation where for more thanone cases, a similar action has to be taken. For this, you can combine cases with the OR operator represented by "|" symbol.

Example

```
def access(user):
  match user:
    case "admin" | "manager": return "Full access"
    case "Guest": return "Limited access"
    case _: return "No access"
print (access("manager"))
print (access("Guest"))
print (access("Ravi"))
```

Output

The above code defines a function named access() and has one string argument, representing the name of the user. For admin or manager user, the system grants full access; for Guest, the access is limited; and for the rest, there's no access.

Full access

Limited access

No access

List as the Argument

Since Python can match the expression against any literal, you can use a list as a case value. Moreover, for variable number of items in the list, they can be parsed to a sequence with "*" operator.

Example

```
def greeting(details):
  match details:
    case [time, name]:
      return f'Good {time} {name}!'
    case [time, *names]:
      msg=''
      for name in names:
        msg+=f'Good {time} {name}!\n'
```

```
        return msg

print (greeting(["Morning", "Ravi"]))
print (greeting(["Afternoon","Guest"]))
print (greeting(["Evening", "Kajal", "Praveen", "Lata"]))
```

Output

On executing, this code will produce the following output −

Good Morning Ravi!
Good Afternoon Guest!
Good Evening Kajal!
Good Evening Praveen!
Good Evening Lata!

Using "if" in "Case" Clause

Normally Python matches an expression against literal cases. However, it allows you to include if statement in the case clause for conditional computation of match variable.

In the following example, the function argument is a list of amount and duration, and the intereset is to be calculated for amount less than or more than 10000. The condition is included in the **case** clause.

Example

```
def intr(details):
  match details:
    case [amt, duration] if amt<10000:
      return amt*10*duration/100
    case [amt, duration] if amt>=10000:
      return amt*15*duration/100
print ("Interest = ", intr([5000,5]))
print ("Interest = ", intr([15000,3]))
```

Output

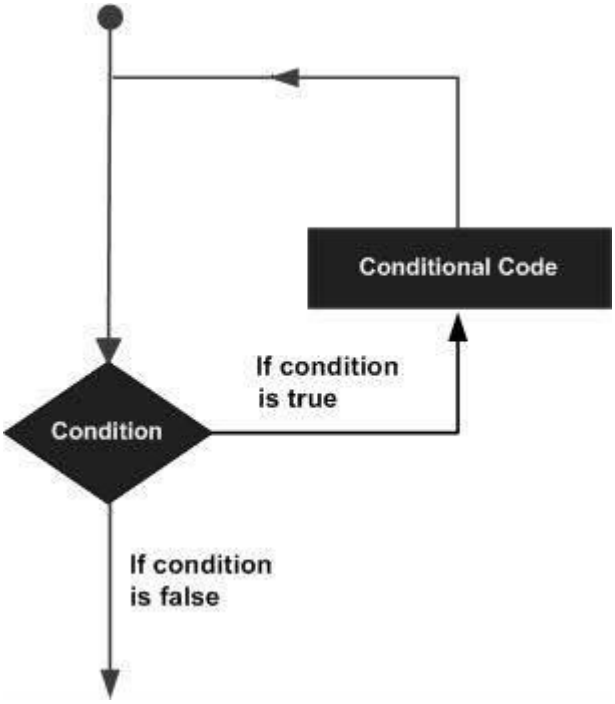
On executing, this code will produce the following output −

Interest = 2500.0
Interest = 6750.0

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**<br>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**<br>Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | **nested loops**<br>You can use one or more loop inside any another while, for or do..while loop. |

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. Click the following links to check their detail.

Let us go through the loop control statements briefly

| Sr.No. | Control Statement & Description |
|---|---|
| 1 | **break statement**<br>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.

**pass statement**

3 The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

The **for** loop in Python has the ability to iterate over the items of any sequence, such as a list, tuple or a string.

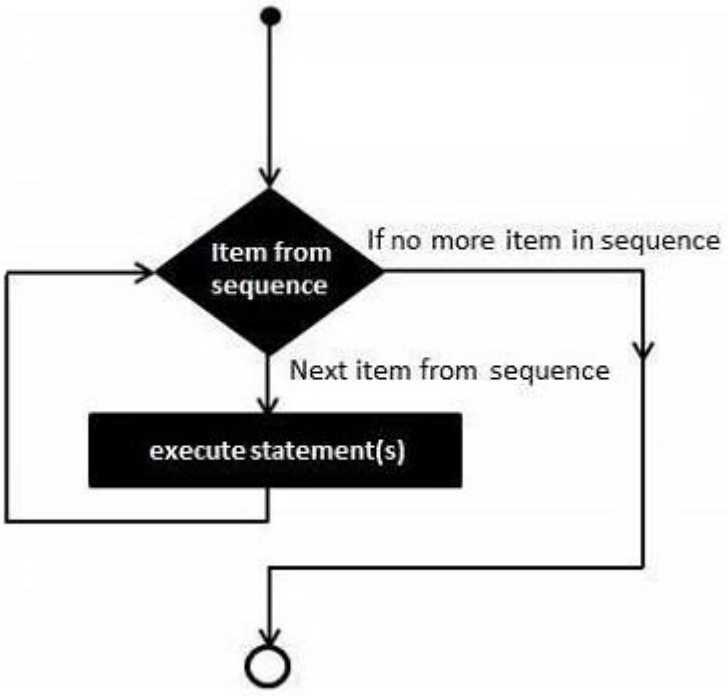
Syntax

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

If a sequence contains an expression list, it is evaluated first. Then, the first item (at 0th index) in the sequence is assigned to the iterating variable iterating_var.

Next, the statements block is executed. Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.

The following flow diagram illustrates the working of **for** loop −




Since the loop is executed for each member element in a sequence, there is no need for explicit verification of Boolean expression controlling the loop (as in **while** loop).

The sequence objects such as list, tuple or string are called **iterables**, as the for loop iterates through the collection. Any iterator object can be iterated by the **for** loop.

The view objects returned by items(), keys() and values() methods of dictionary are also iterables, hence we can run a **for** loop with these methods.

Python's built-in **range()** function returns an iterator object that streams a sequence of numbers. We can run a for loop with range as well.

Using "for" with a String

A string is a sequence of Unicode letters, each having a positional index. The following example compares each character and displays if it is not a vowel ('a', 'e', 'I', 'o' or 'u')

Example

```
zen = '''
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
'''
for char in zen:
   if char not in 'aeiou':
      print (char, end='')
```

Output

On executing, this code will produce the following output −

Btfl s bttr thn gly.
Explct s bttr thn mplct.
Smpl s bttr thn cmplx.
Cmplx s bttr thn cmplctd.

Using "for" with a Tuple

Python's tuple object is also an indexed sequence, and hence we can traverse its items with a **for** loop.

Example

In the following example, the **for** loop traverses a tuple containing integers and returns the total of all numbers.

```
numbers = (34,54,67,21,78,97,45,44,80,19)
total = 0
for num in numbers:
   total+=num
print ("Total =", total)
```

Output

On executing, this code will produce the following output −

Total = 539

Using "for" with a List

Python's list object is also an indexed sequence, and hence we can traverse its items with a **for** loop.

Example

In the following example, the for loop traverses a list containing integers and prints only those which are divisible by 2.

```
numbers = [34,54,67,21,78,97,45,44,80,19]
total = 0
for num in numbers:
  if num%2 == 0:
    print (num)
```

Output

On executing, this code will produce the following output −

34
54
78
44
80

Using "for" with a Range Object

Python's buil-in range() function returns a range object. Python's range object is an iterator which generates an integer with each iteration. The object contains integrrs from start to stop, separated by step parameter.

Syntax

The range() function has the following syntax −

```
range(start, stop, step)
```

Parameters

- **Start** − Starting value of the range. Optional. Default is 0
- **Stop** − The range goes upto stop-1
- **Step** − Integers in the range increment by the step value. Option, default is 1.

Return Value

The range() function returns a range object. It can be parsed to a list sequence.

Example

```
numbers = range(5)
'''
start is 0 by default,
```

```
step is 1 by default,
range generated from 0 to 4
'''
print (list(numbers))
# step is 1 by default, range generated from 10 to 19
numbers = range(10,20)
print (list(numbers))
# range generated from 1 to 10 increment by step of 2
numbers = range(1, 10, 2)
print (list(numbers))
```

Output

On executing, this code will produce the following output −

```
[0, 1, 2, 3, 4]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[1, 3, 5, 7, 9]
```

Example

Once we obtain the range, we can use the **for** loop with it.

```
for num in range(5):
 print (num, end=' ')
print()
for num in range(10,20):
 print (num, end=' ')
print()
for num in range(1, 10, 2):
 print (num, end=' ')
```

Output

On executing, this code will produce the following output −

```
0 1 2 3 4
10 11 12 13 14 15 16 17 18 19
1 3 5 7 9
```

Factorial of a Number

Factorial is a product of all numbers from 1 to that number say n. It can also be defined as product of 1, 2, up to n.

```
Factorial of a number n! = 1 * 2 * . . . . . * n
```

We use the range() function to get the sequence of numbers from 1 to n-1 and perform cumumulative multplication to get the factorial value.

```
fact=1
N = 5
for x in range(1, N+1):
   fact=fact*x
print ("factorial of {} is {}".format(N, fact))
```

Output

On executing, this code will produce the following output −

factorial of 5 is 120

In the above program, change the value of N to obtain factorial value of different numbers.

Using "for" Loop with Sequence Index

To iterate over a sequence, we can obtain the list of indices using the range() function

```
Indices = range(len(sequence))
```

We can then form a **for** loop as follows:

```
numbers = [34,54,67,21,78]
indices = range(len(numbers))
for index in indices:
   print ("index:",index, "number:",numbers[index])
```

On executing, this code will produce the following **output** −

index: 0 number: 34
index: 1 number: 54
index: 2 number: 67
index: 3 number: 21
index: 4 number: 78

Using "for" with Dictionaries

Unlike a list, tuple or a string, dictionary data type in Python is not a sequence, as the items do not have a positional index. However, traversing a dictionary is still possible with different techniques.

Running a simple **for** loop over the dictionary object traverses the keys used in it.

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers:
   print (x)
```

On executing, this code will produce the following **output** −

10
20
30
40

Once we are able to get the key, its associated value can be easily accessed either by using square brackets operator or with the **get()** method. Take a look at the following example −

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers:
   print (x,":",numbers[x])
```

It will produce the following **output** −

10 : Ten
20 : Twenty
30 : Thirty
40 : Forty

The items(), keys() and values() methods of dict class return the view objects dict_items, dict_keys and dict_values respectively. These objects are iterators, and hence we can run a for loop over them.

The dict_items object is a list of key-value tuples over which a for loop can be run as follows −

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers.items():
   print (x)
```

It will produce the following **output** −

(10, 'Ten')
(20, 'Twenty')
(30, 'Thirty')
(40, 'Forty')

Here, "x" is the tuple element from the dict_items iterator. We can further unpack this tuple in two different variables. Check the following code −

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x,y in numbers.items():
   print (x,":", y)
```

It will produce the following **output** −

10 : Ten
20 : Twenty

30 : Thirty
40 : Forty

Similarly, the collection of keys in dict_keys object can be iterated over. Take a look at the following example −

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers.keys():
   print (x, ":", numbers[x])
```

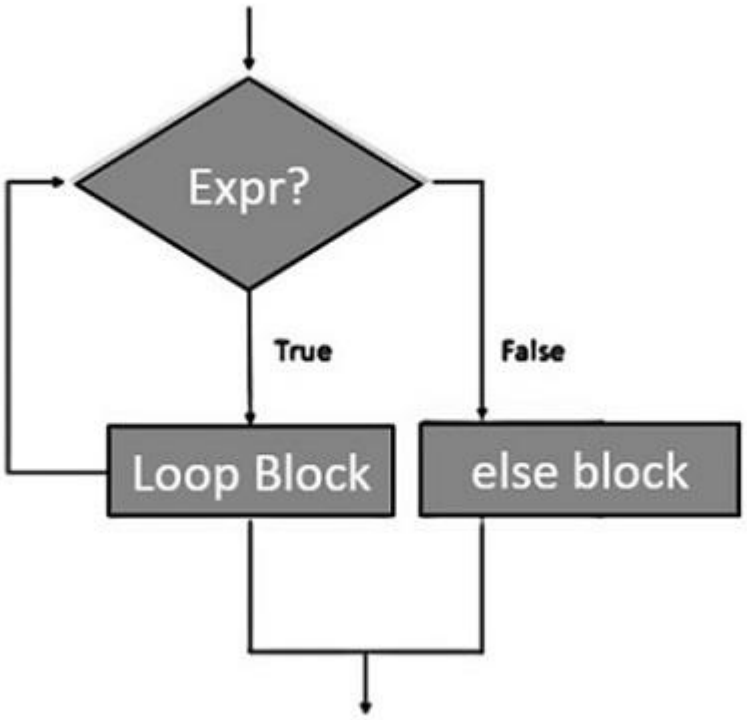
It will produce the same **output** −

10 : Ten
20 : Twenty
30 : Thirty
40 : Forty

Python for-else Loops

Python supports having an "else" statement associated with a "for" loop statement. If the "else" statement is used with a "for" loop, the "else" statement is executed when the sequence is exhausted before the control shifts to the main line of execution.

The following flow diagram illustrates how to use **else** statement with **for** loop −




Example

The following example illustrates the combination of an else statement with a for statement. Till the count is less than 5, the iteration count is printed. As it becomes 5, the print statement in else block is executed, before the control is passed to the next statement in the main program.

```
for count in range(6):
   print ("Iteration no. {}".format(count))
else:
   print ("for loop over. Now in else block")
print ("End of for loop")
```

On executing, this code will produce the following **output** −

Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
for loop over. Now in else block
End of for loop

Nested Loops in Python

Python programming language allows the use of one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

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

The syntax for a nested **while** loop statement in Python programming language is as follows −

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

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

Example

The following program uses a nested-for loop to display multiplication tables from 1-10.

```
for i in range(1,11):
   for j in range(1,11):
      k=i*j
      print ("{:3d}".format(k), end=' ')
   print()
```

The print() function inner loop has end=' ' which appends a space instead of default newline. Hence, the numbers will appear in one row.

The last print() will be executed at the end of inner **for** loop.

When the above code is executed, it produces the following **output** −

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Python while Loops

Normally, flow of execution of steps in a computer program goe from start to end. However, instead of the next step, if the flow is redirected towards any earlier step, it constitutes a loop.

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

Syntax

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

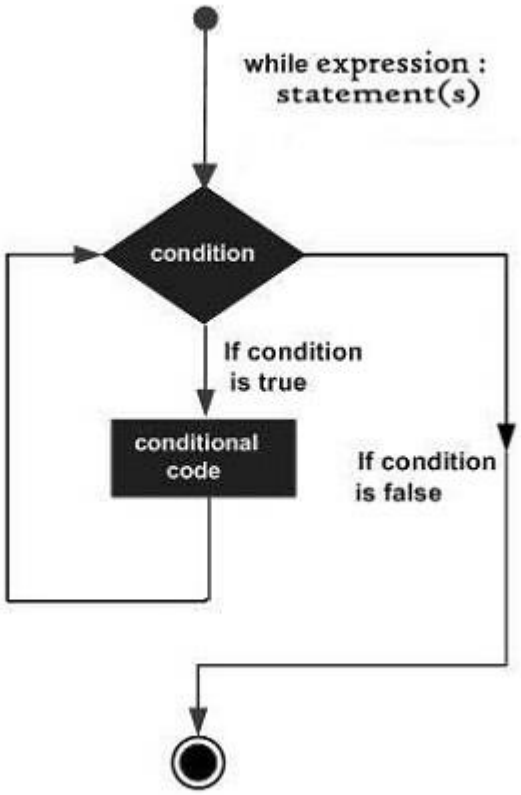
```
while expression:
   statement(s)
```

The **while** keyword is followed by a boolean expression, and then by colon symbol, to start an indented block of statements. Here, statement(s) may be a single statement or a block of statements with uniform indent. The condition may be any expression, and true is any non-zero value. The loop iterates while the boolean expression is true.

As soon as the expression becomes false, the program control passes to the line immediately following the loop.

If it fails to turn false, the loop continues to run, and doesn't stop unless forcefully stopped. Such a loop is called infinite loop, which is undesired in a computer program.

The following flow diagram illustrates the **while** loop −




Example 1

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.

```
count=0
while count<5:
   count+=1
   print ("Iteration no. {}".format(count))

print ("End of while loop")
```

We initialize count variable to 0, and the loop runs till "count<5". In each iteration, count is incremented and checked. If it's not 5 next repetion takes place. Inside the looping block, instantenous value of count is printed. When the **while** condition becomes false, the loop terminates, and next statement is executed, here it is End of **while** loop message.

Output

On executing, this code will produce the following output −

Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
End of while loop

Example 2

Here is another example of using the **while** loop. For each iteration, the program asks for user input and keeps repeating till the user inputs a non-numeric string. The isnumeric() function that returns true if input is an integer, false otherwise.

```
var='0'
while var.isnumeric()==True:
   var=input('enter a number..')
   if var.isnumeric()==True:
      print ("Your input", var)
print ("End of while loop")
```

Output

On executing, this code will produce the following output −

```
enter a number..10
Your input 10
enter a number..100
Your input 100
enter a number..543
Your input 543
enter a number..qwer
End of while loop
```

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must be cautious when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

Example 3

Let's take an example to understand how the infinite loop works in Python −

```
var = 1
while var == 1 : # This constructs an infinite loop
   num = int(input("Enter a number :"))
   print ("You entered: ", num)
print ("Good bye!")
```

Output

On executing, this code will produce the following output −

*The above example goes in an infinite loop and you need to use CTRL+C to exit the program.*

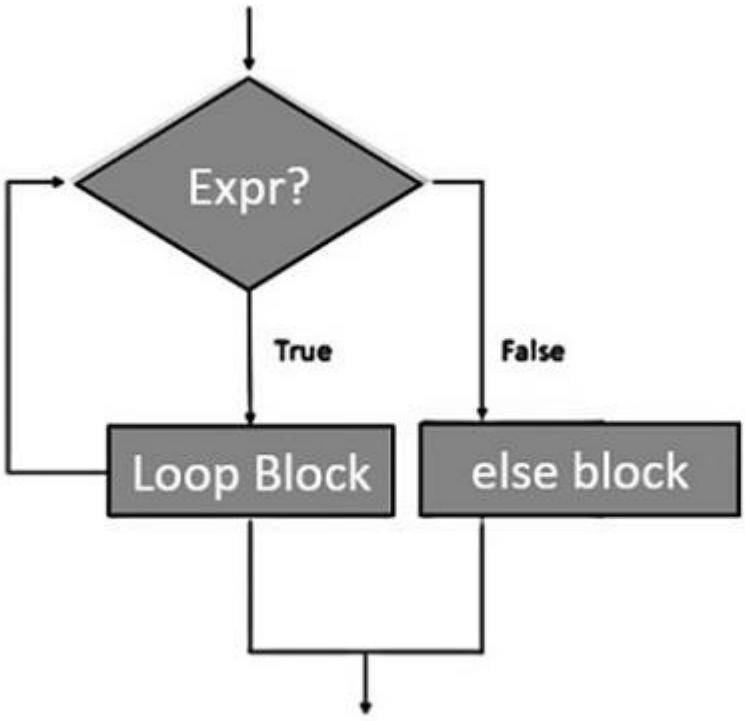
```
Enter a number :20
You entered: 20
Enter a number :29
You entered: 29
Enter a number :3
You entered: 3
Enter a number :11
You entered: 11
Enter a number :22
You entered: 22
Enter a number :Traceback (most recent call last):
  File "examples\test.py", line 5, in
    num = int(input("Enter a number :"))
Key board Interrupt
```

The while-else Loop

Python supports having an **else** statement associated with a **while** loop statement.

If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false before the control shifts to the main line of execution.

The following flow diagram shows how to use **else** with **while** statement −




Example

The following example illustrates the combination of an else statement with a while statement. Till the count is less than 5, the iteration count is printed. As it becomes 5, the print statement in else block is executed, before the control is passed to the next statement in the main program.

```
count=0
while count<5:
   count+=1
   print ("Iteration no. {}".format(count))
else:
   print ("While loop over. Now in else block")
print ("End of while loop")
```

Output

On executing, this code will produce the following **output** −

Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
While loop over. Now in else block
End of while loop

Python break Statement

Python break statement is used to terminate the current loop and resumes execution at the next statement, just like the traditional break statement in C.

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

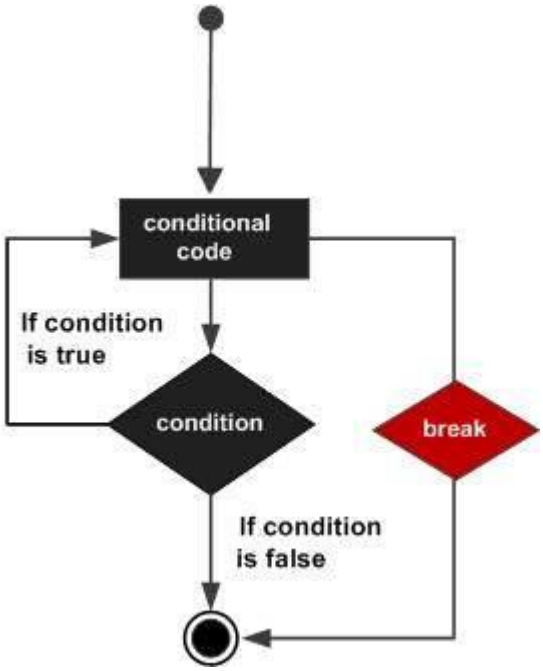
If you are using nested loops in Python, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in Python is as follows −

```
break
```

Flow Diagram




Example

```
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
   print ('Current variable value :', var)
   var = var -1
   if var == 5:
      break

print ("Good bye!")
```

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

Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!

Python break Statement

Python break statement is used to terminate the current loop and resumes execution at the next statement, just like the traditional break statement in C.

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

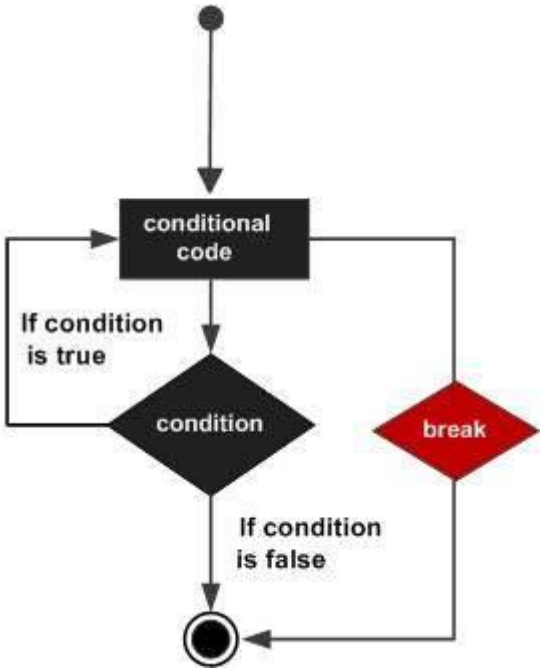
If you are using nested loops in Python, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in Python is as follows −

```
break
```

Flow Diagram




Example

```
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
   print ('Current variable value :', var)
   var = var -1
   if var == 5:
      break
```

```
print ("Good bye!")
```

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

Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!

Python Continue Statement

Python **continue** statement is used to skip the execution of the program block and returns the control to the beginning of the current loop to start the next iteration. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.

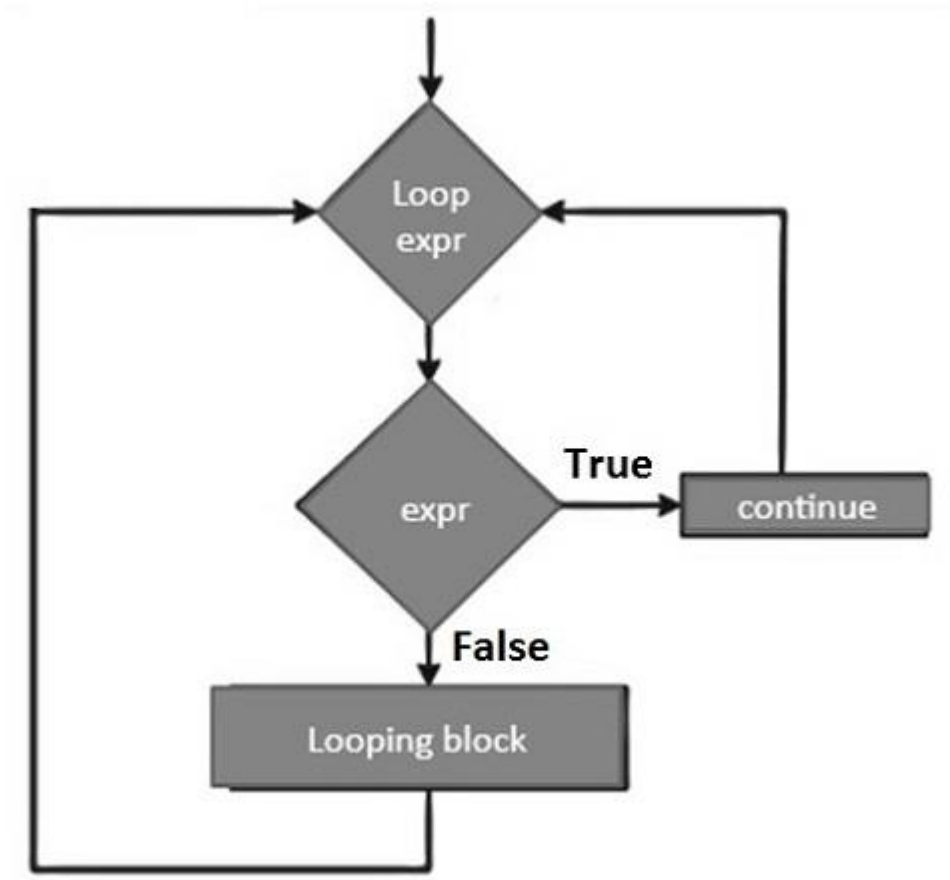
The continue statement can be used in both **while** and **for** loops.

Syntax

```
continue
```

Flow Diagram

The flow diagram of the **continue** statement looks like this −




The **continue** statement is just the opposite to that of **break**. It skips the remaining statements in the current loop and starts the next iteration.

Example 1

Now let's take an example to understand how the **continue** statement works in Python −

```
for letter in 'Python': # First Example
   if letter == 'h':
      continue
   print ('Current Letter :', letter)
var = 10 # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print ('Current variable value :', var)
print ("Good bye!")
```

When the above code is executed, it produces the following **output** −

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

```
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

Checking Prime Factors

Following code uses continue to find the prime factors of a given number. To find prime factors, we need to successively divide the given number starting with 2, increment the divisior and continue the same process till the input reduces to 1.

The algorithm for finding prime factors is as follows −

- Accept input from user (n)
- Set divisor (d) to 2
- Perform following till n>1
- Check if given number (n) is divisible by divisor (d).
- If n%d==0
    - a. Print d as a factor
    - Set new value of n as n/d
    - Repeat from 4
- If not
- Increment d by 1
- Repeat from 3

Given below is the Python code for the purpose −

```
num = 60
print ("Prime factors for: ", num)
d=2
while num > 1:
   if num%d==0:
      print (d)
      num=num/d
      continue
   d=d+1
```

On executing, this code will produce the following **output** −

Prime factors for: 60

2
2
3
5

Assign different value (say 75) to num in the above program and test the result for its prime factors.

Prime factors for: 75
3
5
5

Python pass Statement

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

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

Syntax

```
pass
```

Example

The following code shows how you can use the **pass** statement in Python −

```
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 the following **output** –

Current Letter : P

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

Dummpy Infinite Loop with Pass

This is simple enough to create an infinite loop using **pass** statement. For instance, if you want to code an infinite loop that does nothing each time through, do it with a pass:

```
while True: pass                # Type Ctrl-C to stop
```

Because the body of the loop is just an empty statement, Python gets stuck in this loop. As explained earlier **pass** is roughly to statements as None is to objects — an explicit nothing.

Using Ellipses ... as pass Alternative

Python 3.X allows ellipses coded as three consecutive dots ...to be used in place of **pass** statement. This **...** can serve as an alternative to the pass statement.

For example if we create a function which does not do anything especially for code to be filled in later, then we can make use of **...**

```
def func1():
    ...                 # Alternative to pass

def func2(): ...        # Works on same line too

func1()                 # Does nothing if called
func2()                 # Does nothing if called
```