

PYTHON

UNIT 2

Contents

Boolean Expressions

- ▶ Compound Boolean expressions
- ▶ Decision Statements
 - ▶ If statements
 - ▶ If else statements
 - ▶ Nested if Statements
 - ▶ Multiway if elif-else statements
- ▶ Loops
 - ▶ The while statement
 - ▶ Range functions
 - ▶ For statement
 - ▶ Nested loops
 - ▶ Break and continue statements
 - ▶ Infinite Loops

Boolean Expressions in Python

A Boolean value is either true or false. In Python, the two Boolean values are **True** and **False**, and the Python type is bool.

A **Boolean expression** is an expression that evaluates to produce a result which is a Boolean value. The `==` operator is one of six common **comparison operators** which all produce a bool result.

For example, the operator `==` tests if two values are equal. It produces (or *yields*) a Boolean value:

```
print("Is five equal 5 to the result of 3 + 2?")
print(5 == (3 + 2))
print("Does five equal six?")
print(5 == 6)
```

OUTPUT

Is five equal 5 to the result of 3 + 2?

True

Does five equal six?

False

```
x == y # Produce True if ... x is equal to y
x != y # ... x is not equal to y
x > y # ... x is greater than y
x < y # ... x is less than y
x >= y # ... x is greater than or equal to y
x <= y # ... x is less than or equal to y
```

A Boolean value is either true or false. In Python, the two Boolean values are **True** and **False**, and the Python type is bool.

- ▶ 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.
- ▶ EG:

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

Will produce result as:

True

False

False

```
>>> 5==5
True
>>> bool(6)
True
>>> bool(-25)
True
>>> bool("hai")
True
>>> bool([1,2,3])
True
>>> bool('')
False
>>> bool([])
False
>>> bool(False)
False
>>> bool(None)
False
....
```

```
Eg: x = "Hello"  
    y = 15  
    print(bool(x))  
    print(bool(y))
```

Will both produce True.

In fact, there are not many values that evaluate to `False`, except empty values, such as `()`, `[]`, `{}`, `""`, the number `0`, and the value `None`. And of course the value `False` evaluates to `False`.

```
bool(False)  
bool(None)  
bool(0)  
bool("")  
bool()  
bool([])  
bool({})
```

Compound Boolean Expression

A combination of two or more Boolean expression using logical operators are called compound boolean expression.

- The and, or , not are the basic Boolean operators.
- The not operator: it's a unary operator, it takes a single operand and inverts its boolean value.
- And operator: it is a binary operator. It s value is true if both operands are true.
- Or operator: it is binary operator.it is true if at least one of the operand is true.

The OR operator

Using the OR operator, we can create a compound expression that is true when *either* of two conditions are true.

One way to implement that logic is with two separate `if` statements, A much better approach is to use an **OR** operator to combine those two conditions.

```
if (temperature < 40 or weather == "rain"):
    print("Wear a jacket!")
```

Another example:

```
AnimalString = 'We were looking for a mouse in our house.'
```

```
s = AnimalString
```

```
if 'dolphin' in s or 'mouse' in s or 'cow' in s:
```

```
    print('1: At least one listed animal is in the AnimalString.')
```

Operator	Description	Example
and	Returns True if both statements are true	$x < 5$ and $x < 10$
or	Returns True if one of the statements is true	$x < 5$ or $x < 4$
not	Reverse the result, returns False if the result is true	$\text{not}(x < 5 \text{ and } x < 10)$

The AND operator

Using the **AND** operator, we can create a compound expression that is true only when *both* of the conditions are true.

For example, here's a program where one of the expressions (`time_available_minutes > 120`) is true.

Because we're using "and" and not "or", the compound expression evaluates to false, so the else clause is executed.

```
money_available_dollars = 3
time_available_minutes = 180
if money_available_dollars > 10 and time_available_minutes > 120:
    print("Go out with friends")
else:
    print("Stay home")
```


The NOT operator

Using the **NOT** operator, we can reverse the truth value of an entire expression, from true to false or false to true.

With `not`, you can negate the truth value of any Boolean expression or object. This functionality makes it worthwhile in several situations:

- Checking **unmet conditions** in the context of **if** statements and **while** loops
- Inverting the truth value** of an object or expression
- Checking if a **value is not in a given container**
- Checking for an **object's identity**

The `not` keyword is a logical operator, and is used to reverse the result of the conditional statement:

Example

Test if `a` is NOT greater than `b`:

```
a = 33
```

```
b = 200
```

```
if not a > b:
```

```
    print("a is NOT greater than b")
```

Decision Statements

In Python, decision making statements are those that decide whether a block of statements has to execute or not based on a condition. Decision making statements are also called **Conditional Statements**.

- Python supports variety of decision making statements.
 - If statement
 - if else statement
 - Nested if statement
 - if elif else statement

- Decision-making statements in programming languages decide the direction(Control Flow) of the flow of program execution.
- 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.
- Decisions in a program are used when the program has conditional choices to execute a code block. Let's take an example of traffic lights, where different colors of lights lit up in different situations based on the conditions of the road or any specific rule.

If statements

The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not.

- The statement must be indented at least one space right of the if statement
- In case there is more than one statement after the if condition, then each statement must be indented using the same number of spaces to avoid indentation errors.
- The statement within the if block are executed if the Boolean expression evaluates to true

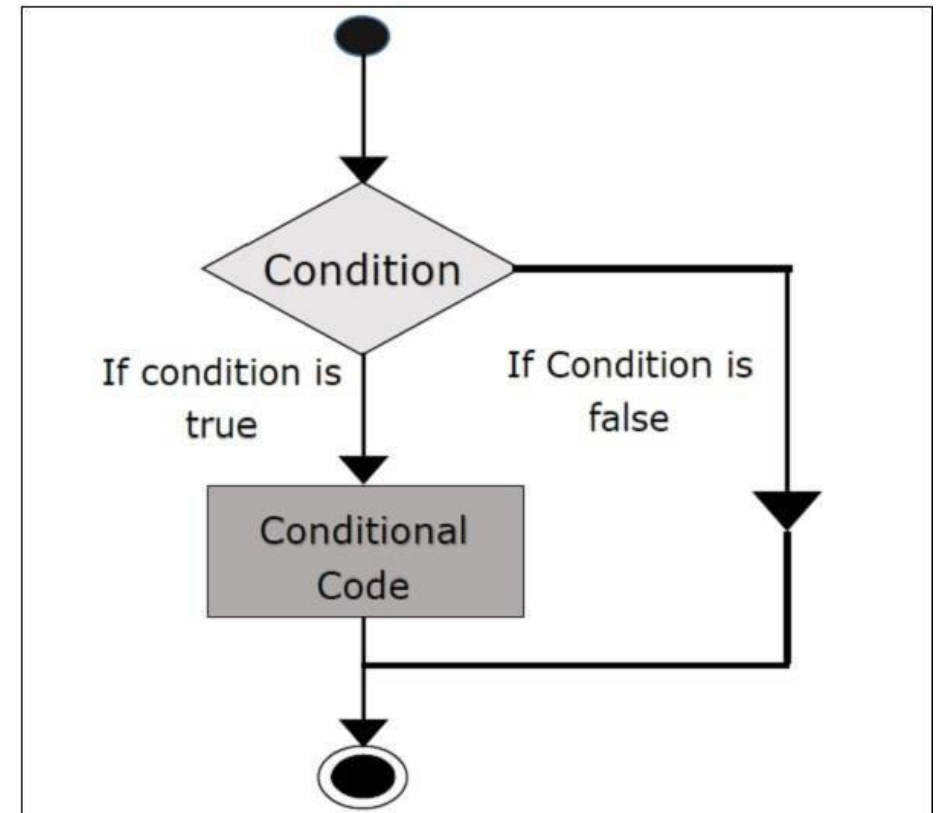
Syntax:

`if condition:`

```
    # Statements to execute if condition is true
```

```
test.py - C:\Users\PC\AppData\Local\Programs\Python\Python38-32\test.py (3.8.3)
File Edit Format Run Options Window Help
num1=eval(input("Enter the first number"))
num2=eval(input("Enter the second number"))
if num1==num2:
    print("Both numbers are equal")

Python 3.8.3 Shell
File Edit Shell Debug Options Window Help
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:\Users\PC\AppData\Local\Programs\Python\Python38-32\test.py ====
Enter the first number15
Enter the second number15
Both numbers are equal
>>> |
```



If-else Statements

In conditional if Statement the additional block of code is merged as else statement which is performed when if condition is false.

Syntax:

```
if (condition):  
    # Executes this block if condition is true  
else:  
    # Executes this block if condition is false
```

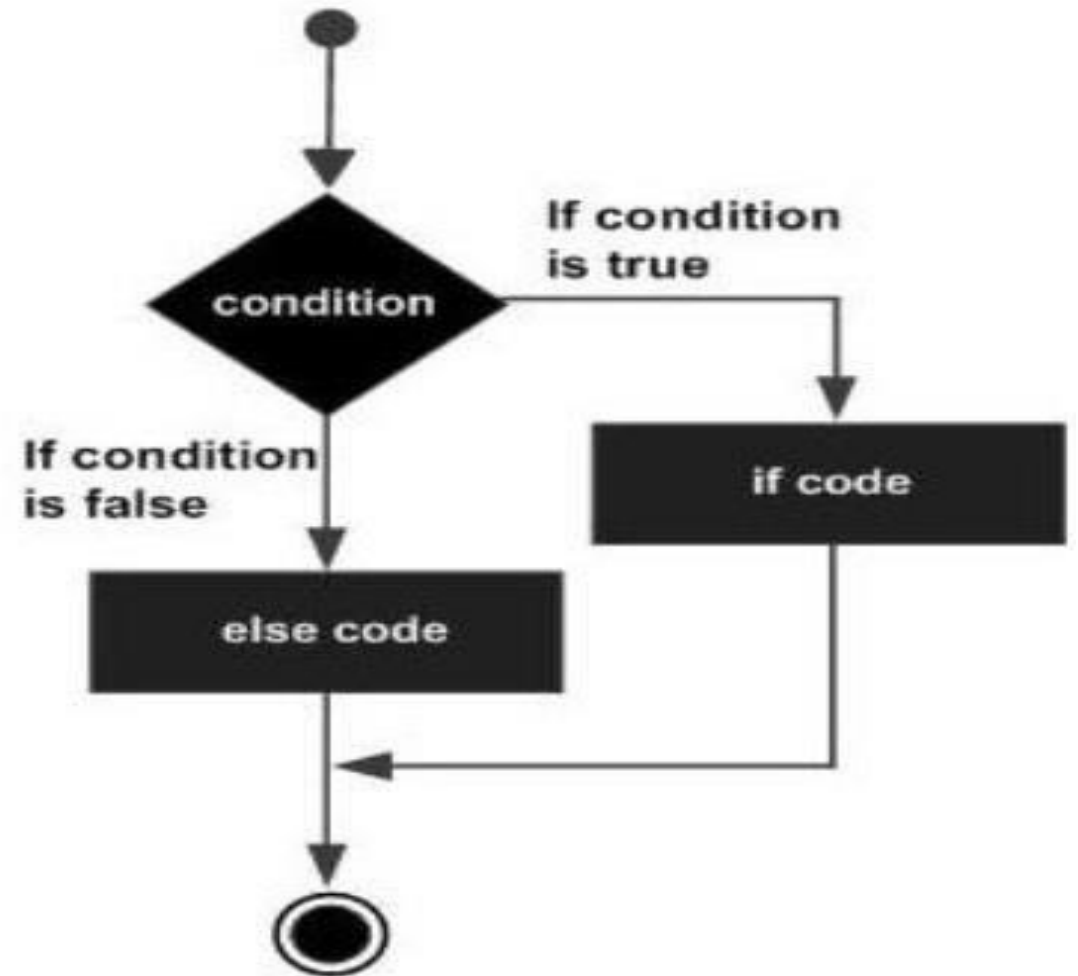
Eg:

```
x= 3  
if == 4:  
    print("Yes")  
else:  
    print("No")
```

```
# if..else chain statement
letter = "A"
if letter=="B":
    print("letter is B")
else:
    if letter == "C":
        print("letter is C")
    else:
        if letter == "A":
            print("letter is A")
        else:
            print("letter isn't A, B and C")
```

Output:

letter is A



Nested if Statements

if statement can also be checked inside other if statement. This conditional statement is called a nested if statement. This means that inner if condition will be checked only if outer if condition is true and by this, we can see multiple conditions to be satisfied.

Syntax:

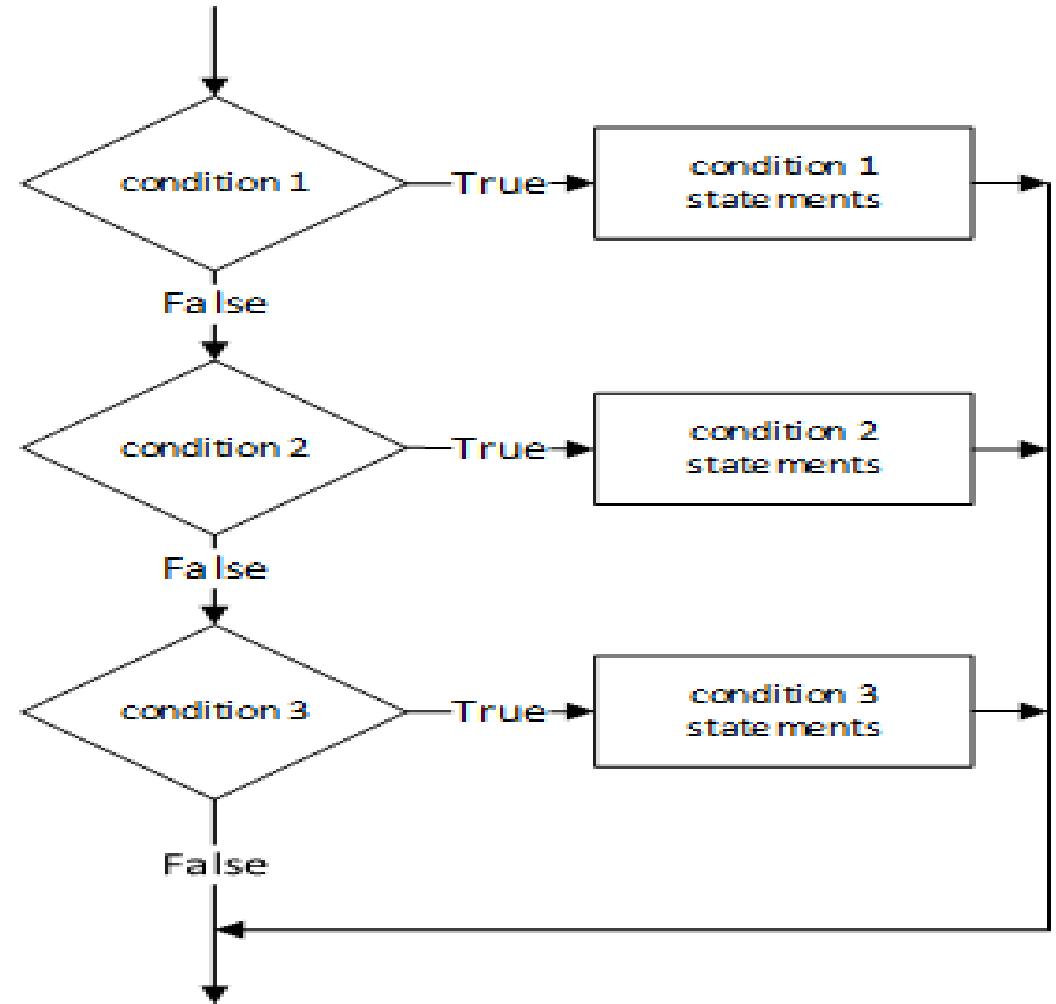
```
if (condition1):  
    # Executes when condition1 is true  
    if (condition2):  
        # Executes when condition2 is true  
    # if Block is end here  
# if Block is end here
```



```
# Nested if statement example
num= 10
if num>5:
    print("Bigger than 5")
    if num <= 15:
        print("Between 5 and 15")
```

OUTPUT

Bigger than 5 Between 5 and 15



if elif-else statements

The if-elif statement is shortcut of if..else chain. While using if-elif statement at the end else block is added which is performed if none of the above if-elif statement is true.

Syntax:-

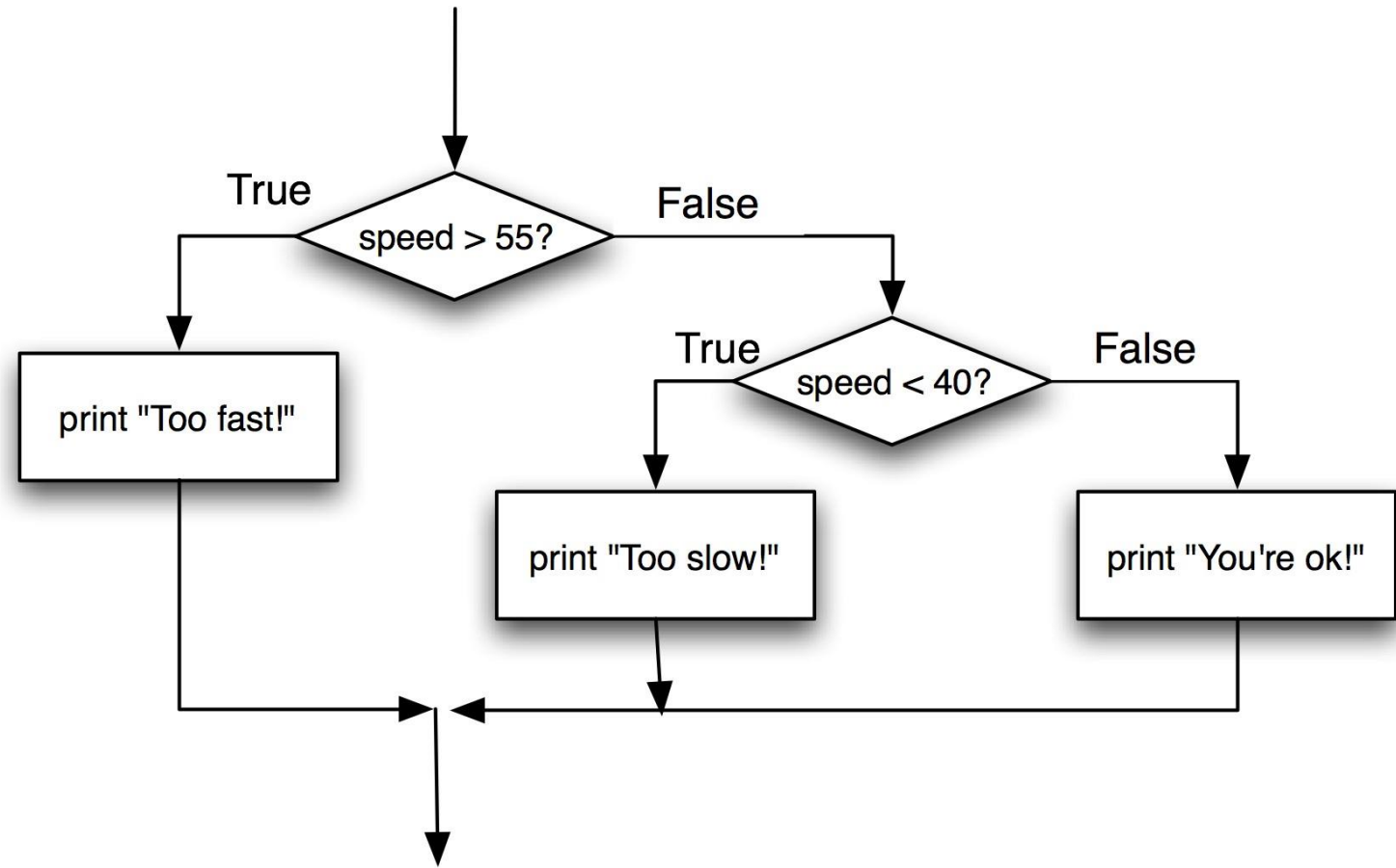
```
if (condition):  
    statement  
elif (condition):  
    statement  
.  
.  
else:  
    statement
```

```
# if-elif statement example
```

```
letter = "A"  
if letter == "B":  
    print("letter is B")  
elif letter == "C":  
    print("letter is C")  
elif letter == "A":  
    print("letter is A")  
else:  
    print("letter isn't A, B or C")
```

OUTPUT

```
letter is A
```



Loops

- **The while Loop**
- The range() function
- **The for loop**
- **Nested Loop**
- The break statement
- The continue statement
- Infinite loops

A **loop** is a control structure that can execute a statement or group of statements repeatedly. Python has three types of loops: while loops, for loops, and nested loops.

Sr.No.	Name of the loop	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	This type of loop executes a code block multiple times and abbreviates the code that manages the loop variable.
3	Nested loops	We can iterate a loop inside another loop.

While Loops

A while loop will repeatedly execute a code block as long as a condition evaluates to True. The condition of a while loop is always checked first before the block of code runs. If the condition is not met initially, then the code block will never run.

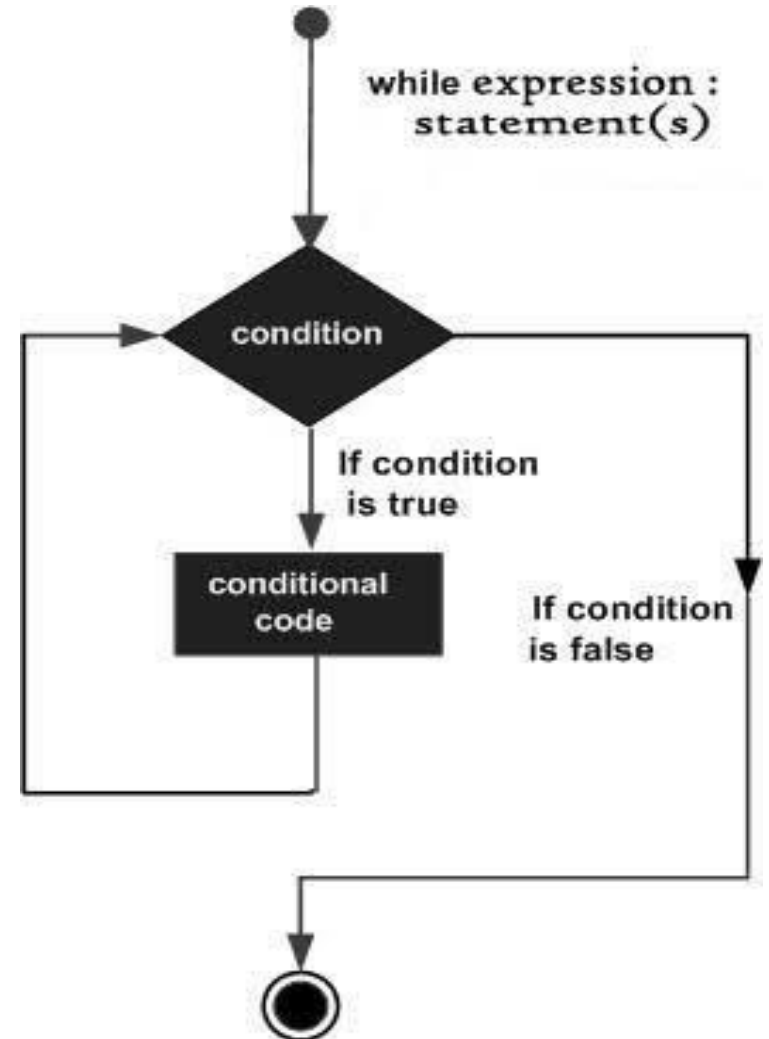
Syntax:

```
while <condition>:  
    { code block }
```

Eg:

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

Will print from 1 to 5



Using else Statement with while Loops

As discussed earlier in the for loop section, we can use the else statement with the while loop also. It has the same syntax.

- Python supports to have an **else** statement associated with a loop statement.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Eg:

#Python program to show how to use else statement with the while loop

```
counter = 0
```

```
while (counter < 10):
```

```
    counter = counter + 3
```

```
    print("Python Loops")
```

```
else:                                # Once the condition of while loop gives False this statement will be executed
```

```
    print("Code block inside the else statement")
```

OUTPUT

Python Loops

Python Loops

Python Loops

Python Loops

Code block inside the else statement

```
>>> i=1
>>> while (i<10) :
        print ('the enetered number is ',i)
        i+=1
else:
        print ('Program ends')
```

```
the enetered number is 1
the enetered number is 2
the enetered number is 3
the enetered number is 4
the enetered number is 5
the enetered number is 6
the enetered number is 7
the enetered number is 8
the enetered number is 9
Program ends
```

```
>>> |
```


The for Loop

Python's for loop is designed to repeatedly execute a code block while iterating through a list, tuple, dictionary, or other iterable objects of Python. The process of traversing a sequence is known as iteration.

Syntax of the for Loop

```
for value in sequence:  
    { code block }
```

In this case, the variable value is used to hold the value of every item present in the sequence before the iteration begins until this particular iteration is completed.

Loop iterates until the final item of the sequence are reached.

Else in For Loop

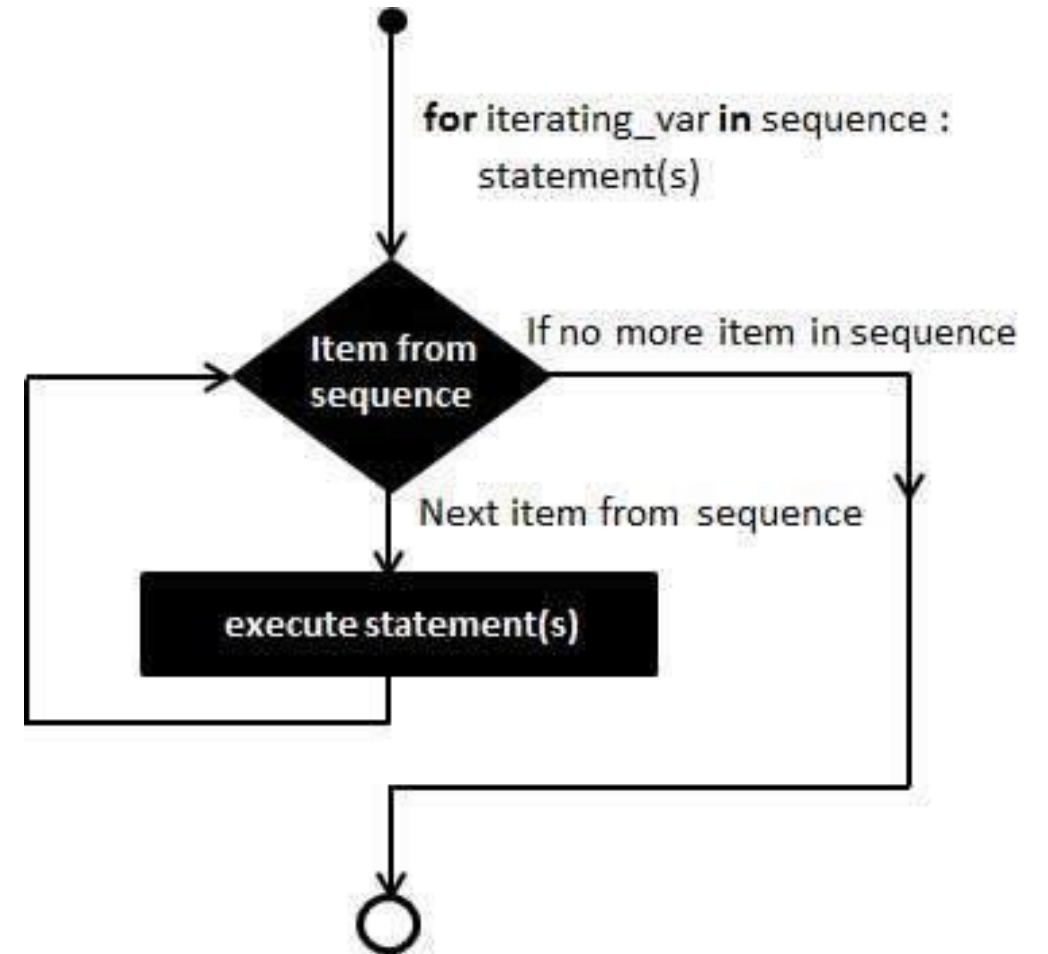
The **else** keyword in a **for** loop specifies a block of code to be executed when the loop is finished:

- ▶ Python supports to have an **else** statement associated with a loop statement
 - ▶ If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
Eg:

```
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

OUTPUT

```
0  
1  
2  
3  
4  
5  
Finally finished!
```



Python program to show how if-else statements work

```
string = "Python Loop"
```

Initiating a loop

```
for s in a string:
```

```
    # giving a condition in if block
```

```
    if s == "o":
```

```
        print("If block")
```

```
    # if condition is not satisfied then else block will be executed
```

```
    else:
```

```
        print(s)
```

OUTPUT

P

Y

t

h

If block

n

L

If block

If block

p

```
# Python program to show how the for loop works
```

```
# Creating a sequence which is a tuple of numbers
```

```
numbers = [4, 2, 6, 7, 3, 5, 8, 10, 6, 1, 9, 2]
```

```
# variable to store the square of the number
```

```
square = 0
```

```
# Creating an empty list
```

```
squares = []
```

```
# Creating a for loop
```

```
for value in numbers:
```

```
    square = value ** 2
```

```
    squares.append(square)
```

```
print("The list of squares is", squares)
```

OUTPUT

The list of squares is [16, 4, 36, 49, 9, 25, 64, 100, 36, 1, 81, 4]

Range Function

The built-in function `range()` is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions.

`range(start, stop, step size)`

If the step size is not specified, it defaults to 1. Start has default value zero.

➤ Range function has one , two , or three parameters.

➤ The last two parameters are optional.

➤ The general form of range function

➤ `List(range(1,6))`

`[1,2,3,4,5]`

Python For Loop with a step size

This code uses a for loop in conjunction with the `range()` function to generate a sequence of numbers starting from 0, up to (but not including) 10, and with a step size of 2. For each number in the sequence, the loop prints its value using the `print()` function. The output will show the numbers 0, 2, 4, 6, and 8.

```
print(range(15))
```

```
print(list(range(15)))
```

```
print(list(range(4, 9)))
```

```
print(list(range(5, 25, 4)))
```

OUTPUT

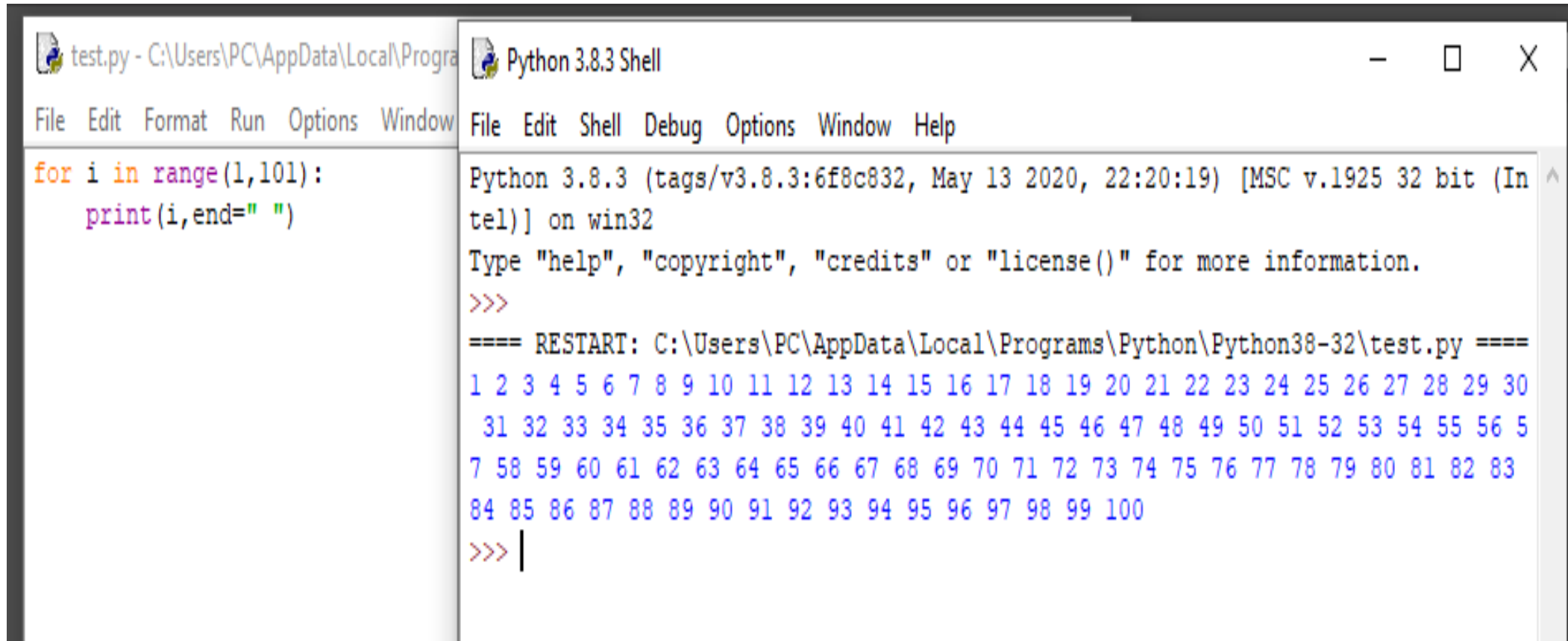
```
range(0, 15)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```
[4, 5, 6, 7, 8]
```

```
[5, 9, 13, 17, 21]
```

Print numbers from 1-100 using for loop?



The image shows a screenshot of a Python IDE with two windows. The left window, titled 'test.py', contains the following code:

```
for i in range(1,101):  
    print(i,end=" ")
```

The right window, titled 'Python 3.8.3 Shell', shows the execution output:

```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
==== RESTART: C:\Users\PC\AppData\Local\Programs\Python\Python38-32\test.py ====  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30  
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 5  
7 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83  
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100  
>>> |
```

```
# Python program to iterate over a sequence with the help of indexing
```

```
tuple_ = ("Python", "Loops", "Sequence", "Condition", "Range")
```

```
# iterating over tuple_ using range() function
```

```
for iterator in range(len(tuple_)):
```

```
    print(tuple_[iterator].upper())
```

OUTPUT

PYTHON

LOOPS

SEQUENCE

CONDITION

RANGE

Python Nested Loops

Loops Inside Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Eg:

```
adj = ["red", "big", "tasty"]  
fruits = ["apple", "banana", "cherry"]
```

```
for x in adj:  
    for y in fruits:  
        print(x, y)
```

OUTPUT

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

Loop Control Statements

Continue Statement

- The continue statement is used to skip the rest of the code inside the loop for the current iteration.
- Loop does not terminate but continues with the next iteration
- Continue statement is used to end the current iteration in a for loop or while loop and continues to the next iteration

Python program to show how the continue statement works

Initiating the loop

```
for string in "Python Loops":
```

```
    if string == "o" or string == "p" or string == "t":
```

```
        continue
```

```
    print('Current Letter:', string)
```

OUTPUT

Current Letter: P

Current Letter: y

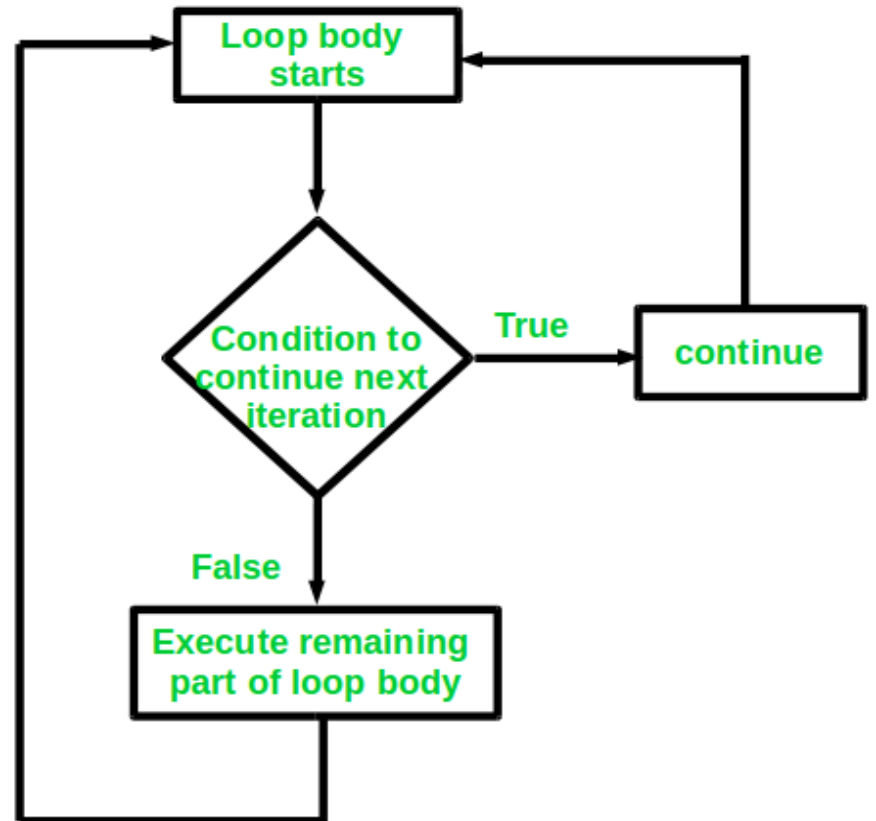
Current Letter: h

Current Letter: n

Current Letter:

Current Letter: L

Current Letter: s



Break Statement

It stops the execution of the loop when the break statement is reached.

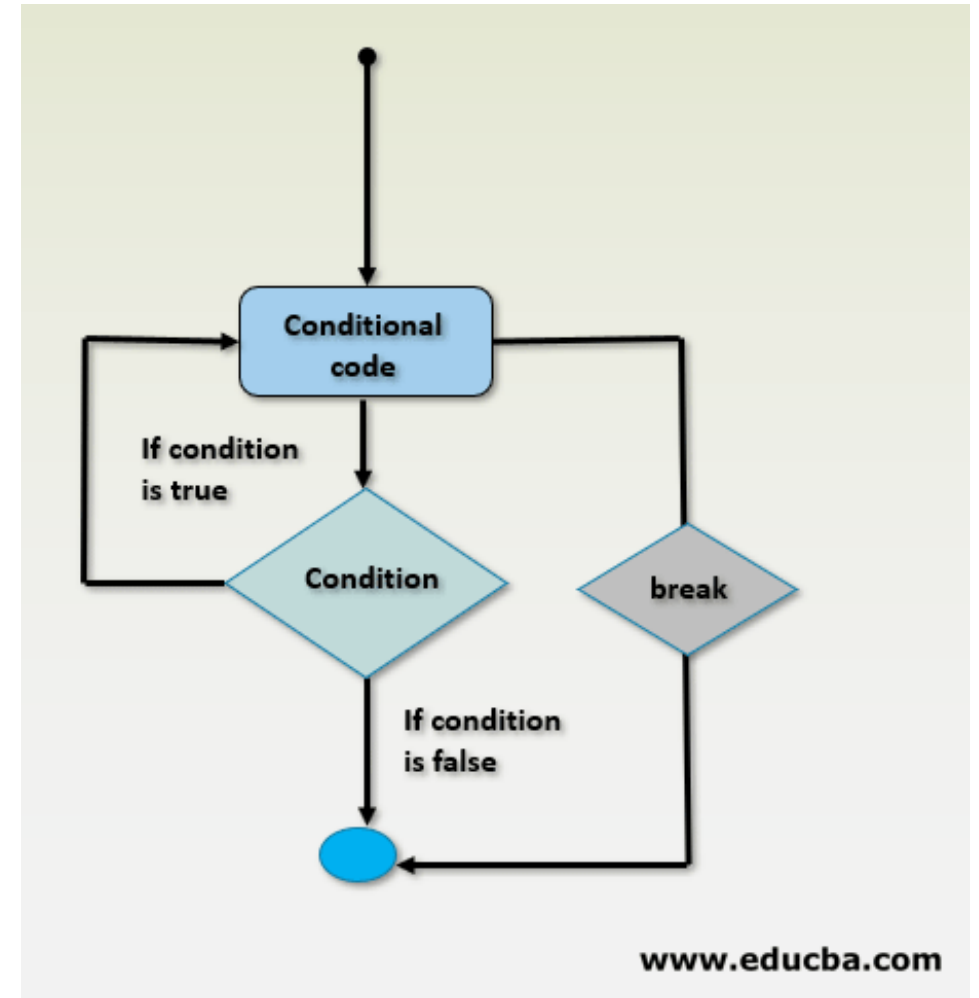
Python program to show how the break statement works

Initiating the loop

```
for string in "Python Loops":  
    if string == 'L':  
        break  
    print('Current Letter: ', string)
```

OUTPUT

Current Letter: P
Current Letter: y
Current Letter: t
Current Letter: h
Current Letter: o
Current Letter: n
Current Letter:



Infinite loop

A loop becomes infinite loop if a condition never becomes FALSE. You must use caution 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.

Loops are generally aimed to repeat a particular set of statements until a given condition is fulfilled. However, there may be a case when a condition is never fulfilled, as a result of which statements are executed again and again. We call this “Infinite looping”. Hence, in other words, **Infinite loops are the loops that run indefinitely until the program is terminated.**

Simple example of Infinite loop:

```
while True:
```

```
    print("True")
```