

PYTHON PROGRAMMING

INTRODUCTION DATA, EXPRESSIONS, STATEMENTS

Introduction to Python and installation, data types: Int, float, Boolean, string, and list; variables, expressions, statements, precedence of operators, comments; modules, functions--
- function and its use, flow of execution, parameters and arguments.

Introduction to Python and installation:

Python is a widely used general-purpose, high level programming language. It was initially designed by **Guido van Rossum in 1991** and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions- **Python 2 and Python 3**.

- On 16 October 2000, Python 2.0 was released with many new features.
- On 3rd December 2008, Python 3.0 was released with more testing and includes new features.

Beginning with Python programming:

1) Finding an Interpreter:

Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like <https://ide.geeksforgeeks.org/>, <http://ideone.com/> or <http://codepad.org/> that can be used to start Python without installing an interpreter.

Windows: There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

2) Writing first program:

Script Begins

Statement1
Statement3

Script

Why to use Python:

The following are the primary factors to use python in day-to-day life:

1. Python is object-oriented

Structure supports such concepts as polymorphism, operation overloading and multiple inheritance.

2. Indentation

Indentation is one of the greatest feature in python

3. It's free (open source)

Downloading python and installing python is free and easy

4. It's Powerful

- Dynamic typing
- Built-in types and tools
- Library utilities
- Third party utilities (e.g. Numeric, NumPy, sciPy)
- Automatic memory management

5. It's Portable

- Python runs virtually every major platform used today
- As long as you have a compatible python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.

6. It's easy to use and learn

- No intermediate compile
- Python Programs are compiled automatically to an intermediate form called byte code, which the interpreter then reads.
- This gives python the development speed of an interpreter without the performance loss inherent in purely interpreted languages.
- Structure and syntax are pretty intuitive and easy to grasp.

7. Interpreted Language

Python is processed at runtime by python Interpreter

8. Interactive Programming Language

Users can interact with the python interpreter directly for writing the programs

9. Straight forward syntax

The formation of python syntax is simple and straight forward which also makes it popular.

Installation:

There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

Steps to be followed and remembered:

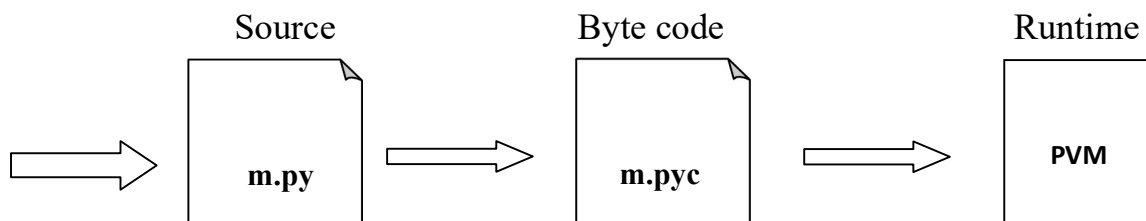
- Step 1: Select Version of Python to Install.
- Step 2: Download Python Executable Installer.
- Step 3: Run Executable Installer.
- Step 4: Verify Python Was Installed On Windows.
- Step 5: Verify Pip Was Installed.
- Step 6: Add Python Path to Environment Variables (Optional)



Working with Python

Python Code Execution:

Python's traditional runtime execution model: Source code you type is translated to byte code, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.



Source code extension is .py
Byte code extension is .pyc (Compiled python code)

There are two modes for using the Python interpreter:

- Interactive Mode
- Script Mode

Running Python in interactive mode:

Without passing python script file to the interpreter, directly execute code to Python prompt. Once you're inside the python interpreter, then you can start.

```
>>> print("hello world")
```

```
hello world
```

Relevant output is displayed on subsequent lines without the >>> symbol

```
>>> x=[0,1,2]
```

Quantities stored in memory are not displayed by default.

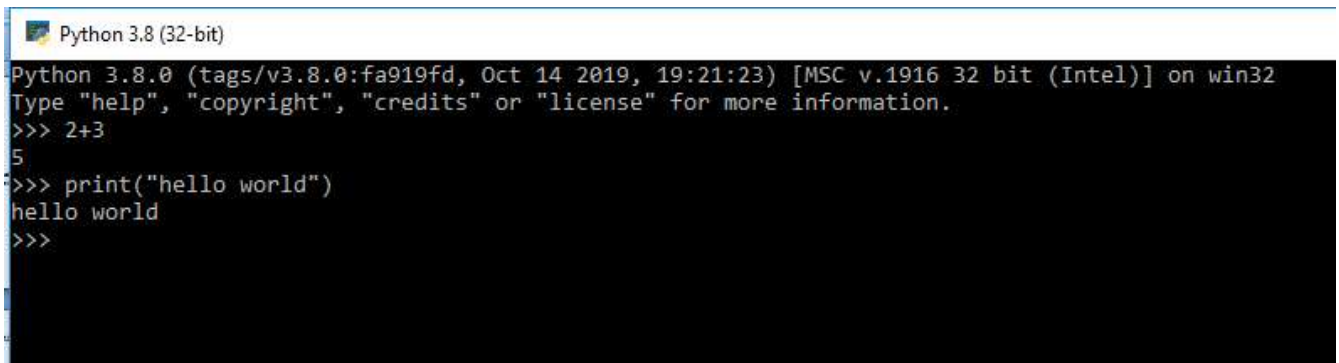
```
>>> x
```

#If a quantity is stored in memory, typing its name will display it.

```
[0, 1, 2]
```

```
>>> 2+3
```

```
5
```

A screenshot of a Windows command prompt window titled "Python 3.8 (32-bit)". The window shows the Python 3.8.0 shell interface. The prompt is "Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32". Below the prompt, the user enters "Type 'help', 'copyright', 'credits' or 'license' for more information." followed by the interactive prompt ">>>". The user enters "2+3", and the interpreter outputs "5". Then, the user enters ">>> print('hello world')", and the interpreter outputs "hello world". Finally, the user enters ">>>" again, and the prompt ">>>" is displayed on the next line.

The chevron at the beginning of the 1st line, i.e., the symbol >>> is a prompt the python interpreter uses to indicate that it is ready. If the programmer types 2+6, the interpreter replies 8.

Running Python in script mode:

Alternatively, programmers can store Python script source code in a file with the .py extension, and use the interpreter to execute the contents of the file. To execute the script by the interpreter, you have to tell the interpreter the name of the file. For example, if you have a script name MyFile.py and you're working on Unix, to run the script you have to type:

python MyFile.py

Working with the interactive mode is better when Python programmers deal with small pieces of code as you can type and execute them immediately, but when the code is more than 2-4 lines, using the script for coding can help to modify and use the code in future.

Example:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy>python e1.py
resource open
the no cant be divisibile zero division by zero
resource close
finished
```

Data types:

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

Int:

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
>>> print(24656354687654+2)
```

```
24656354687656
```

```
>>> print(20)
```

```
20
```

```
>>> print(0b10)
```

```
2
```

```
>>> print(0B10)
```

```
2
```

```
>>> print(0X20)
```

```
32
```

```
>>> 20
```

```
20
```

```
>>> 0b10
```

```
2
```

```
>>> a=10
```

```
>>> print(a)
```

```
10
```

To verify the type of any object in Python, use the type() function:

```
>>> type(10)
```

```
<class 'int'>
```

```
>>> a=11
```

```
>>> print(type(a))
```

```
<class 'int'>
```

Float:

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Float can also be scientific numbers with an "e" to indicate the power of 10.

```
>>> y=2.8
```

```
>>> y
```

```
2.8
```

```
>>> y=2.8
```

```
>>> print(type(y))
```

```
<class 'float'>
```

```
>>> type(.4)
```

```
<class 'float'>
```

```
>>> 2.
```

2.0

Example:

```
x = 35e3
```

```
y = 12E4
```

```
z = -87.7e100
```

```
print(type(x))
```

```
print(type(y))
```

```
print(type(z))
```

Output:

```
<class 'float'>
```

```
<class 'float'>
```

```
<class 'float'>
```

Boolean:

Objects of Boolean type may have one of two values, True or False:

```
>>> type(True)
```

```
<class 'bool'>
```

```
>>> type(False)
```

```
<class 'bool'>
```

String:

1. Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

- 'hello' is the same as "hello".
- Strings can be output to screen using the print function. **For example: print("hello").**

```
>>>print("scodeen
```

```
college")
```

```
>>> type("scodeen college")
```

```
<class 'str'>
```



```
>>> print(scodeen college')
```

```
scodeen college
```

```
>>> " "
```

```
' '
```

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>> print("scodeen is an autonomous (') college")scodeen is an autonomous (') college
```

```
>>> print('scodeen is an autonomous (") college')scodeen is an autonomous (")
```

college **Suppressing Special Character:**

Specifying a backslash (\) in front of the quote character in a string “escapes” it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>> print("scodeen is an autonomous (\') college")
```

```
scodeen is an autonomous (')
```

Escape Sequence	Usual Interpretation of Character(s) After Backslash	“Escaped” Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\newline	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence `\t`:

```
>>> print("a\tb")
a      b
```

List:

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
```

```
>>> print(list1)
```

```
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
```

```
-----
```

```
>>> x=list()
```

```
>>> x
```

```
[]
```

```
-----
```

```
>>> tuple1=(1,2,3,4)
```

```
>>> x=list(tuple1)
```

```
>>> x
```

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

Variables:

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

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

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Assigning Values to Variables:

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

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

For example –

```
a= 100          # An integer assignment
```

```
b = 1000.0      # A floating point
```

```
c = "John"      # A string
```

```
print (a)
```

```
print (b)
```

```
print (c)
```

This produces the following result –

100

1000.0

John

Multiple Assignment:

Python allows you to assign a single value to several variables simultaneously.

For example :

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example –

```
a,b,c = 1,2,"scodeen"
```

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

Output Variables:

The Python print statement is often used to output variables.

Variables do not need to be declared with any particular type and can even change type after they have been set.

```
x = 5          # x is of type int
x = "scodeen " # x is now of type str
print(x)
```

Output: scodeen

To combine both text and a variable, Python uses the “+” character:

Example

```
x = "awesome"
print("Python is " + x)
```

Output

Python is awesome

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "  
y = "awesome"  
z = x + y  
print(z)
```

Output:

Python is awesome

Expressions:

An expression is a combination of values, variables, and operators. An expression is evaluated using assignment operator.

Examples: $Y = x + 17$

```
>>> x=10
```

```
>>> z=x+20
```

```
>>> z
```

```
30
```

```
>> x=10
```

```
>>> y=20
```

```
>>> c=x+y
```

```
>>> c
```

```
30
```

A value all by itself is a simple expression, and so is a variable.

```
>>> y=20
```

```
>>> y
```

Python also defines expressions only contain identifiers, literals, and operators. So,

Identifiers: Any name that is used to define a class, function, variable module, or object is an identifier.

Literals: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

Operators: In Python you can implement the following operations using the corresponding tokens.

Operator	Token
add	+
subtract	-
multiply	*
Integer Division	/
remainder	%
Binary left shift	<<
Binary right shift	>>
and	&
or	\
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Check equality	==
Check not equal	!=

Conditional expression:

Syntax: true_value if Condition else false_value

```
>>> x = "1" if True else "2"
```

```
>>> x
```

```
'1'
```

Statements:

A statement is an instruction that the Python interpreter can execute. We have normally two basic statements, the assignment statement and the print statement. Some other kinds of statements that are if statements, while statements, and for statements generally called as control flows.

Examples:

An assignment statement creates new variables and gives them values:

```
>>> x=10
```

```
>>> college="scodeen"
```

An print statement is something which is an input from the user, to be printed / displayed on to the screen (or) monitor.

```
>>> print("scodeen
```

```
colege")scodeen college
```

Precedence of Operators:

Operator precedence affects how an expression is evaluated.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies $3*2$ and then adds into 7.

Example 1:

```
>>> 3+4*2
```

```
11
```

Multiplication gets evaluated before the addition operation

```
>>> (10+10)*2
```

```
40
```

Parentheses () overriding the precedence of the arithmetic operators

Example 2:

```
a = 20
```

```
b = 10
```

```
c = 15
```

```
d = 5
```

```
e = 0
```

```
e = (a + b) * c / d    #( 30 * 15 ) / 5  
print("Value of (a + b) * c / d is ", e)
```

```
e = ((a + b) * c) / d  # (30 * 15 ) / 5  
print("Value of ((a + b) * c) / d is ", e)
```

```
e = (a + b) * (c / d);  # (30) * (15/5)  
  
print("Value of (a + b) * (c / d) is ", e)
```

```
e = a + (b * c) / d;    # 20 + (150/5)  
print("Value of a + (b * c) / d is ", e)
```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/opprec.py

Value of (a + b) * c / d is 90.0

Value of ((a + b) * c) / d is 90.0

Value of (a + b) * (c / d) is 90.0

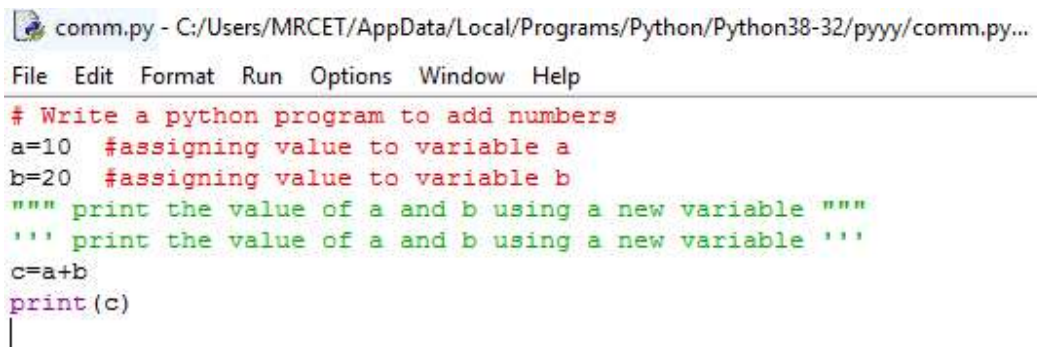
Value of a + (b * c) / d is 50.0

Comments:

Single-line comments begins with a hash(#) symbol and is useful in mentioning that the whole line should be considered as a comment until the end of line.

A Multi line comment is useful when we need to comment on many lines. In python, triple double quote(“ “ “”) and single quote(‘ ‘ ‘’)are used for multi-line commenting.

Example:



comm.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py...

File Edit Format Run Options Window Help

```
# Write a python program to add numbers
a=10 #assigning value to variable a
b=20 #assigning value to variable b
""" print the value of a and b using a new variable """
''' print the value of a and b using a new variable '''
c=a+b
print(c)
|
```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py 30

CONTROL FLOW, LOOPS

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: while, for, break, continue.

Control Flow, Loops:

Boolean Values and Operators:

A boolean expression is an expression that is either true or false. The following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

```
>>> 5 == 5
```

```
True
```

```
>>> 5 == 6
```

```
False
```

`True` and `False` are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
```

```
<class 'bool'>
```

```
>>> type(False)
```

```
<class 'bool'>
```

The `==` operator is one of the relational operators; the others are: `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`

Note:

All expressions involving relational and logical operators will evaluate to either `true` or `false`

Conditional (if):

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.

if Statement Flowchart:

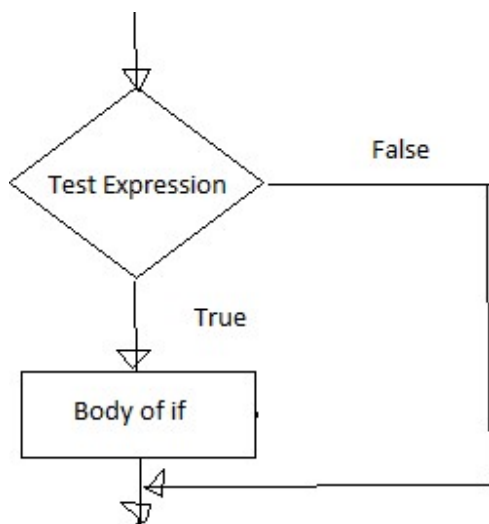


Fig: Operation of if statement

Example: Python if Statement

```
a = 3
```

```
if a > 2:
```

```
    print(a, "is greater")
```

```
print("done")
```

```
a = -1
```

```
if a < 0:
```

```
    print(a, "a is smaller")
```

```
print("Finish")
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-  
32/pyyy/if1.py 3 is greater  
done  
-1 a is smaller  
Finish
```

```
a=10
```

```
if a>9:
```

```
    print("A is Greater than 9")
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-  
32/pyyy/if2.py A is Greater than 9
```

Alternative if (If-Else):

An else statement can be combined with an if statement. An else statement contains the block of code (false block) that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The else statement is an optional statement and there could be at most only one else Statement following if.

Syntax of if - else :

```
if test expression:
```

```
    Body of if stmts
```

```
else:
```

```
    Body of else stmts
```

If - else Flowchart :

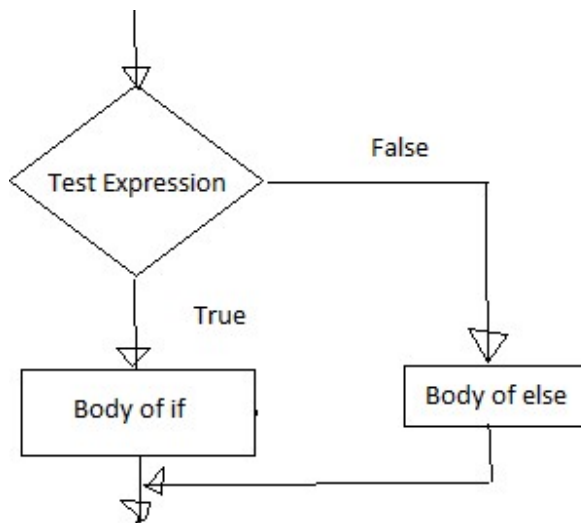


Fig: Operation of if – else statement

Example of if - else:

```
a=int(input('enter the number'))
if a>5:
    print("a is greater")
else:
    print("a is smaller than the input given")
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
enter the number 2
a is smaller than the input given
```

```
a=10
b=20
if a>b:
    print("A is Greater than B")
else:
    print("B is Greater than A")
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/if2.py B is Greater than A
```

Chained Conditional: (If-elif-else):

The elif statement allows us 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, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

Syntax of if – elif - else:

If test expression:

 Body of if stmts

elif test expression:

 Body of elif stmts

else:

 Body of else stmts

Flowchart of if – elif - else:

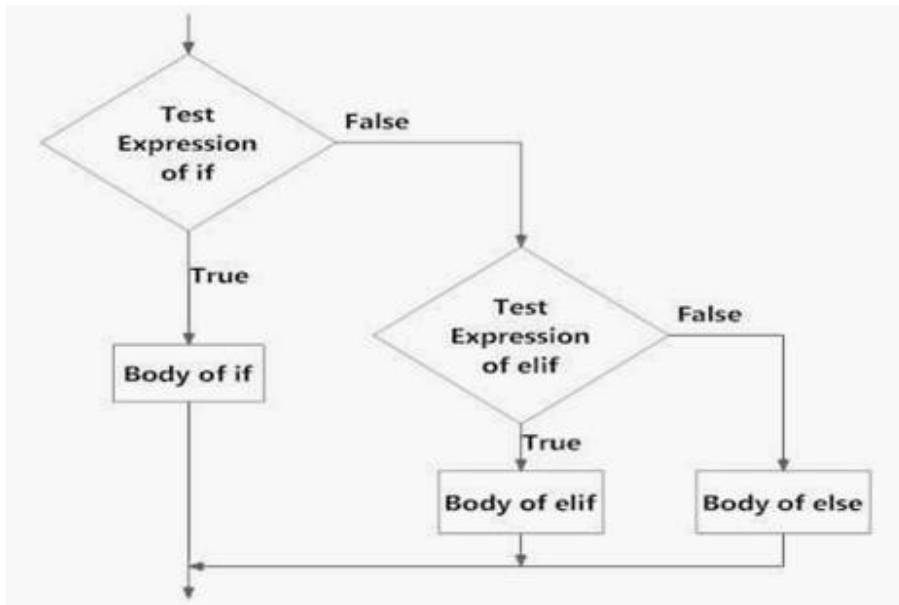


Fig: Operation of if – elif - else statement

Example of if - elif – else:

```
a=int(input('enter the number'))
b=int(input('enter the number'))
c=int(input('enter the number'))
if a>b:
```

```
    print("a is greater")
elif b>c:
    print("b is greater")
else:
    print("c is greater")
```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py

enter the number5

enter the number2

enter the number9

a is greater

>>>

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py

enter the number2

enter the number5

enter the number9

c is greater

var = 100

if var == 200:

print("1 - Got a true expression value")

print(var)

elif var == 150:

print("2 - Got a true expression value")

print(var)

elif var == 100:

print("3 - Got a true expression value")

print(var)

else:

print("4 - Got a false expression value")

print(var)

print("Good bye!")

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-

32/pyyy/ifelif.py 3 - Got a true expression value

100

Good bye!

Iteration:

A loop statement allows us to execute a statement or group of statements multiple times as long as the condition is true. Repeated execution of a set of statements with the help of loops is called iteration.

Loops statements are used when we need to run same code again and again, each time with a different value.

Statements:

In Python Iteration (Loops) statements are of three types:

1. While Loop
2. For Loop
3. Nested For Loops

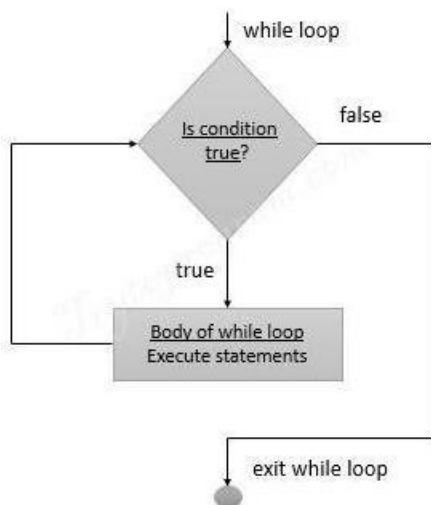
While loop:

- Loops are either infinite or conditional. Python while loop keeps reiterating a block of code defined inside it until the desired condition is met.
- The while loop contains a boolean expression and the code inside the loop is repeatedly executed as long as the boolean expression is true.
- The statements that are executed inside while can be a single line of code or a block of multiple statements.

Syntax:

```
while(expression):  
    Statement(s)
```

Flowchart:



Example Programs:

```
1. _____  
   i=1  
   while i<=6:  
       print("Scodeen  
         college")i=i+1
```

output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/wh1.py

Scodeen college

Scodeen

college

Scodeen

college

Scodeen

college

Scodeen

college

Scodeen

college

```
2. _____  
   i=1
```

```
   while i<=3:  
       print("SCODEEN",end="")  
       j=1  
       while j<=1:  
           print("CSE DEPT",end="")  
           j=j+1  
       i=i+1  
       print()
```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/wh2.py

SCODEEN CSE DEPT

SCODEEN CSE

DEPTSCODEEN

CSE DEPT

3. _____

i=1

```

j=1
while i<=3:
    print("SCODEEN",end="
")

    while j<=1:
        print("CSE DEPT",end="")
        j=j+1
    i=i+1
    print()

```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/wh3.py

```

SCODEEN CSE
DEPTSCODEEN
SCODEEN

```

4. _____

```

i = 1
while (i < 10):
    print (i)
    i = i+1

```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-

32/pyyy/wh4.py 1

```

2
3
4
5
6
7
8
9

```

2. _____

```

a = 1
b = 1
while (a<10):
    print ('Iteration',a)
    a = a + 1

```

$$b = b + 1$$

```
    if (b == 4):
        break
print ('While loop terminated')
```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/wh5.py

Iteration 1

Iteration 2

Iteration 3

While loop terminated

```
count = 0
```

```
while (count < 9):
```

```
    print("The count is:", count)
```

```
    count = count + 1
```

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

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/wh.py

=The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6

The count is: 7

The count is: 8

Good bye!

For loop:

Python **for loop** is used for repeated execution of a group of statements for the desired number of times. It iterates over the items of lists, tuples, strings, the dictionaries and other iterable objects

Syntax: for var in sequence:

Statement(s)

Holds the value of item
in sequence in each iteration

A sequence of values assigned to var in each iteration

Sample Program:

```
numbers = [1, 2, 4, 6, 11, 20]
```

```
seq=0
```

```
for val in numbers:
```

```
    seq=val*val
```

```
    print(seq)
```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/fr.py1

4

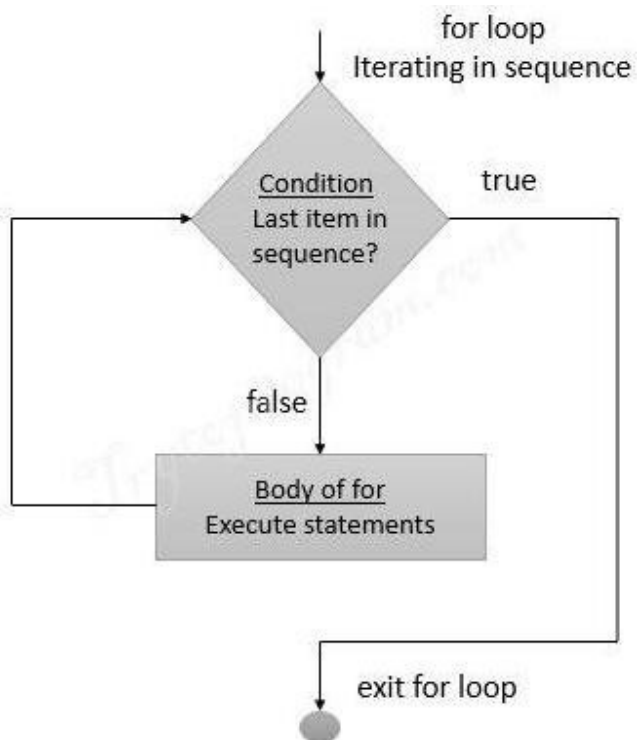
16

36

121

400

Flowchart:



Iterating over a list:

```
#list of items
list = ['M','R','C','E','T']
i = 1

#Iterating over the list
for item in list:
    print ('college ',i,' is ',item)
    i = i+1
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-
32/pyyy/lis.py college 1 is M
college 2 is R
college 3 is C
college 4 is E
college 5 is T
```

Iterating over a Tuple:

```
tuple = (2,3,5,7)
print ('These are the first four prime numbers ')
#Iterating over the tuple
for a in tuple:
    print (a)
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-
32/pyyy/fr3.py These are the first four prime numbers
2
3
5
7
```

Iterating over a dictionary:

```
#creating a dictionary
college = {"ces":"block1","it":"block2","ece":"block3"}

#Iterating over the dictionary to print keys
print ('Keys are:')
```

```
for keys in college:  
    print (keys)
```

```
#Iterating over the dictionary to print values  
print ('Values are:')  
for blocks in college.values():  
    print(blocks)
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-  
32/pyyy/dic.py Keys are:  
ces  
it  
ece  
Values are:  
block1  
block2  
block3
```

Iterating over a String:

```
#declare a string to iterate over  
college = 'SCODEEN'
```

```
#Iterating over the string  
for name in college:  
    print (name)
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-  
32/pyyy/strr.py M  
R  
C  
E  
T
```

Nested For loop:

When one Loop defined within another Loop is called Nested Loops.

Syntax:

```
for val in sequence:  
    for val in sequence:
```


statements

statements

Example 1 of Nested For Loops (Pattern Programs)

```
for i in range(1,6):  
    for j in range(0,i):  
        print(i, end=" ")  
    print("")
```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/nesforr.py 1

2 2

3 3 3

4 4 4 4

5 5 5 5 5

Example 2 of Nested For Loops (Pattern Programs)

```
for i in range(1,6):  
    for j in range(5,i-1,-1):  
        print(i, end=" ")  
    print("")
```

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/nesforr.py

Output:

1 1 1 1 1

2 2 2 2

3 3 3

4 4

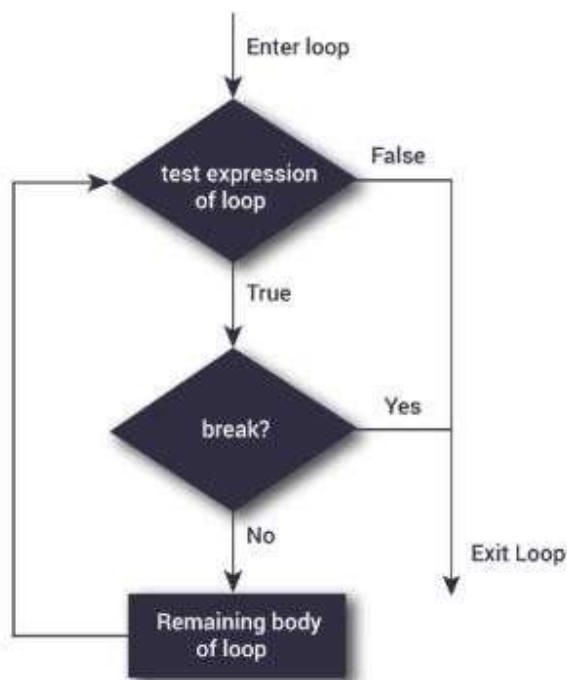
Break and continue:

In Python, **break** and **continue** statements can alter the flow of a normal loop. 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:

The break statement terminates the loop containing it and 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.

Flowchart:



The following shows the working of break statement in for and while loop:

for var in sequence:

 # code inside for loop

 If condition:

 break (if break condition satisfies it jumps to outside loop)

 # code inside for loop

code outside for loop

while test expression

 # code inside while loop

 If condition:

 break (if break condition satisfies it jumps to outside loop)

 # code inside while loop

code outside while loop

Example:

```
for val in "SCODEEN COLLEGE":
```

```
    if val == " ":
```

```
        break
```

```
    print(val)
```

```
print("The end")
```

Output:

M

R

C

E

T

The end

Program to display all the elements before number 88

```
for num in [11, 9, 88, 10, 90, 3, 19]:
```

```
    print(num)
```

```
    if(num==88):
```

```
        print("The number 88 is found")
```

```
        print("Terminating the loop")
```

```
        break
```

Output:

11

9

88

The number 88 is found

Terminating the loop

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

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/br.py =

Current Letter : P

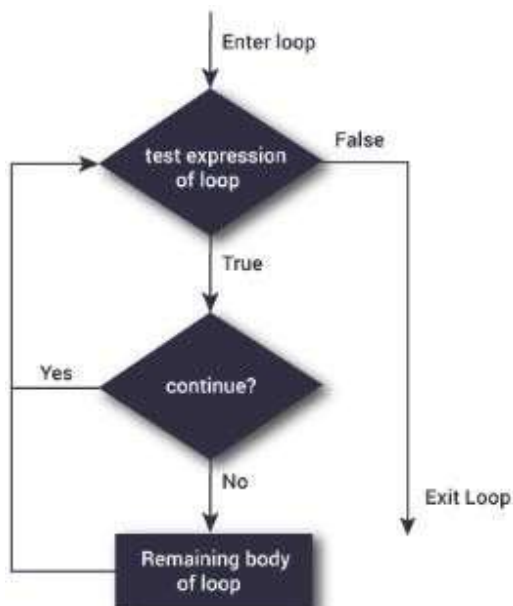
Current Letter : y

Current Letter : t

Continue:

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.

Flowchart:



The following shows the working of break statement in for and while loop:

for var in sequence:

 # code inside for loop

 If condition:

 continue (if break condition satisfies it jumps to outside loop)

 # code inside for loop

code outside for loop

while test expression

 # code inside while loop

 If condition:

 continue (if break condition satisfies it jumps to outside loop)

 # code inside while loop

code outside while loop

Example:

Program to show the use of continue statement inside loops

```
for val in "string":
```

```
    if val == "i":
```

```
        continue
```

```
    print(val)
```

```
print("The end")
```

Output:

C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-

32/pyyy/cont.py s

t

r

n

g

The end

program to display only odd numbers

```
for num in [20, 11, 9, 66, 4, 89, 44]:
```

```
# Skipping the iteration when number is even
if num%2 == 0:
    continue
# This statement will be skipped for all even numbers
print(num)
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-
32/pyyy/cont2.py 11
9
89
```

```
# _____
for letter in "Python": # First Example
    if letter == "h":
        continue
    print ("Current Letter :", letter)
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-
32/pyyy/con1.py Current Letter: P
Current Letter: y
Current Letter: t
Current Letter: o
Current Letter: n
```

Pass:

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.

pass is just a placeholder for functionality to be added later.

Example:

```
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

Output:

```
C:/Users/SCODEEN/AppData/Local/Programs/Python/Python38-32/pyyy/fl.y.py
```

```
>>>
```

Similarly we can also write,

```
def f(arg): pass    # a function that
```

```
does nothing (yet)class C: pass
```

```
    # a class with no
```

```
methods (yet)
```