

# Tutorial\_Session1

September 28, 2018

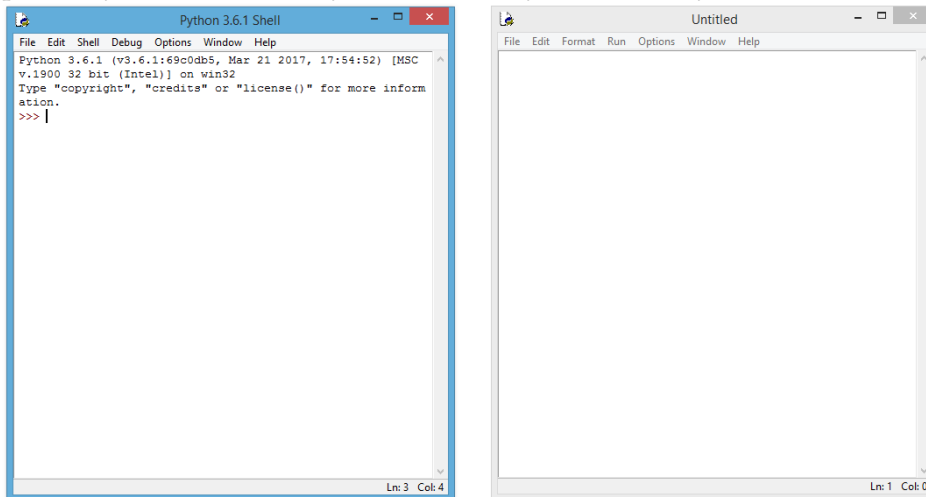
---

## 1 PYTHON

- Interactive, interpreted, and object-oriented programming language.
- Simple syntax
- Developed by Guido Van Rossum in 1991 at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- Python is a Beginner's Language Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.
- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

### 1.1 PYTHON PROGRAMMING ENVIRONMENT

- Available on a wide variety of platforms including Windows, Linux and Mac OS X.
- Official Website: [python.org](http://python.org)
- IDLE stands for Integrated Development and Learning Environment. Python IDLE comprises Python Shell and Python Editor. Python Shell Python Editor



---

---

## 2 First Python Program

- Interactive Mode Programming

python  
get a prompt

- Script Mode Programming

Write a program with any editor and save it with .py extension  
python test.py

### 2.1 Display on screen

```
In [1]: print('hello world')
```

```
hello world
```

---

### 2.2 Lines and Indentation

- Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.
  - The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.
- 

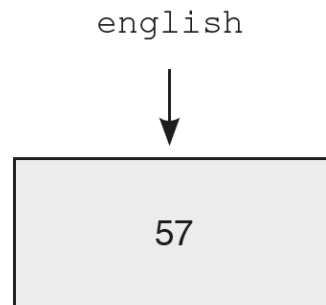
### 2.3 Names (Variables) and Assignment Statements

- Variables provide a means to name values so that they can be used and manipulated later.
- 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.
- Assignment Statement: Statement that assigns value to a variable.

```
In [2]: english = 57  
        print(english)
```

```
57
```

Python associates the **name** (variable) **english** with value **57** i.e. the name (variable) **english** is assigned the value **57**, or that the name (variable) **english** refers to value **57**. Values are also called **objects**.



### 2.3.1 Rules for creating a name (variable)

- Must begin with a letter or \_ (underscore character)
- May contain any number of letters, digits, or underscore characters. No other character apart from these is allowed.

### 2.3.2 Multiple Assignments

- Used to enhance the readability of the program.

```
In [3]: msg, day, time = 'Meeting', 'Mon', '9'
        totalMarks = count = 0
```

---

## 2.4 Arithmetic Operators

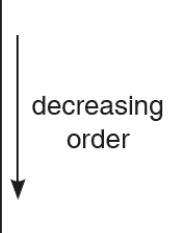
```
In [4]: print("18 + 5 =", 18 + 5)    #Addition
        print("18 - 5 =", 18 - 5)    #Subtraction
        print("18 * 5 =", 18 * 5)    #Multiplication
        print("27 / 5 =", 27 / 5)    #Division
        print("27 // 5 =", 27 // 5)  #Integer Division
        print("27 % 5 =", 27 % 5)    #Modulus
        print("2 ** 3 =", 2 ** 3)    #Exponentiation
        print("-2 ** 3 =", -2 ** 3)  #Exponentiation
```

```
18 + 5 = 23
18 - 5 = 13
18 * 5 = 90
27 / 5 = 5.4
27 // 5 = 5
27 % 5 = 2
2 ** 3 = 8
-2 ** 3 = -8
```

```
In [5]: print("'how' + ' are' + ' you?':", 'how' + ' are' + ' you?')
        print("'hello' * 5          :", 'hello' * 5)
```

```
'how' + ' are' + ' you?': how are you?
'hello' * 5          : hellohellohellohellohello
```

### 2.4.1 Precedence of Arithmetic Operators

() (parentheses)	
** (exponentiation)	
- (negation)	
/ (division) // (integer division) * (multiplication) % (modulus)	
+ (addition) - (subtraction)	

## 2.5 Shorthand Operators

```
In [6]: a = 6
        a = a + 5
        print(a)
        a = 6
        a += 5
        print(a)
```

```
11
11
```

## 2.6 Relational Operators

- Used for comparing two expressions and yield True or False.
- The arithmetic operators have higher precedence than the relational operators.

```
In [7]: print("23 < 25 :", 23 < 25)           #less than
        print("23 > 25 :", 23 > 25)           #greater than
        print("23 <= 23 :", 23 <= 23)         #less than or equal to
        print("23 - 2.5 >= 5 * 4 :", 23 - 2.5 >= 5 * 4) #greater than or equal to
        print("23 == 25 :", 23 == 25)         #equal to
        print("23 != 25 :", 23 != 25)         #not equal to
```

```
23 < 25 : True
23 > 25 : False
23 <= 23 : True
```

```
23 - 2.5 >= 5 * 4 : True
23 == 25 : False
23 != 25 : True
```

- When the relational operators are applied to strings, strings are compared left to right, character by character, based on their ASCII codes, also called ASCII values.

```
In [8]: print("'hello' < 'Hello' :", 'hello' < 'Hello')
        print("'hi' > 'hello'      :", 'hi' > 'hello')
```

```
'hello' < 'Hello' : False
'hi' > 'hello'      : True
```

---

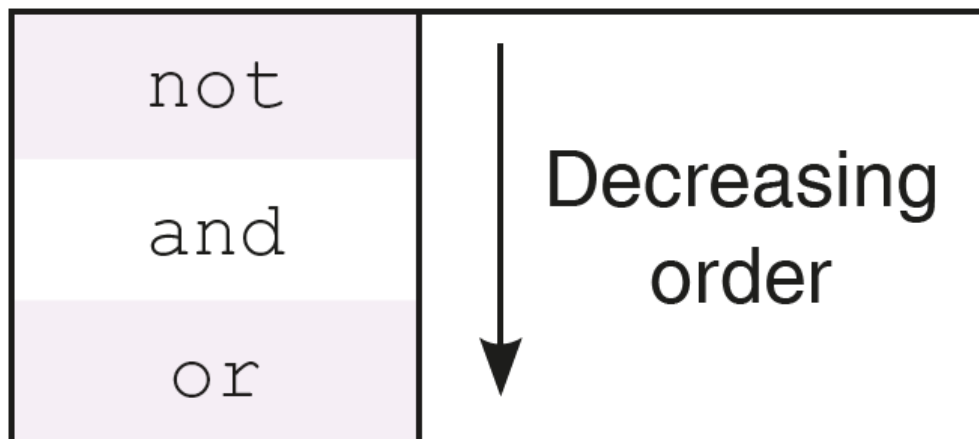
## 2.7 Logical Operators

- The logical operators not, and, and or are applied to logical operands True and False, also called Boolean values, and yield either True or False.
- As compared to relational and arithmetic operators, logical operators have the least precedence level.

```
In [9]: print("not True:", not True)           #not operator
        print("10 < 25 and 5 > 6 :", 10 < 25 and 5 > 6) #and operator
        print("10 < 25 or 5 > 6  :", 10 < 25 or 5 > 6)  #or operator
```

```
not True: False
10 < 25 and 5 > 6 : False
10 < 25 or 5 > 6  : True
```

### 2.7.1 Precedence of Logical Operators



## 2.8 Python Keywords

- Reserved words that are already defined by the Python for specific uses.

```
In [10]: import keyword
         print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'd
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else',
'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass',
'raise', 'return', 'try', 'while', 'with', 'yield']
```

---

## 2.9 Functions

- Functions provide a systematic way of problem solving by dividing the given problem into several sub-problems, finding their individual solutions, and integrating the solutions of individual problems to solve the original problem.
- This approach to problem solving is called stepwise refinement method or modular approach.

## 2.10 Built-in Functions

- Predefined functions that are already available in Python.

### 2.10.1 Type Conversion: int, float, str functions

```
In [11]: str(123)
```

```
Out[11]: '123'
```

```
In [12]: int('234')
```

```
Out[12]: 234
```

```
In [13]: int(234.8)
```

```
Out[13]: 234
```

### 2.10.2 input function

- Enables us to accept an input string from the user without evaluating its value.
- The function input continues to read input text from the user until it encounters a newline.

```
In [14]: name = input('Enter a name: ')
         print('Welcome', name)
```

```
Enter a name: Ankur
Welcome Ankur
```

```
In [ ]: costPrice = int(input('Enter cost price: '))
        profit = int(input('Enter profit: '))
        sellingPrice = costPrice + profit
        print('Selling Price: ', sellingPrice)
```

### 2.10.3 eval function

- Used to evaluate the value of a string.

```
In [16]: a = eval('15+10')
        print(a)
```

```
25
```

### 2.10.4 min and max functions

- Used to find maximum and minimum value respectively out of several values.

```
In [17]: a = max(59, 80, 95.6, 95.2)
        b = min('hello', 'how', 'are', 'you', 'Sir')
        print("Minimum Value", b)
        print("Maximum Value", a)
```

```
Minimum Value Sir
Maximum Value 95.6
```

### 2.10.5 Functions from math module

- Used to find maximum and minimum value respectively out of several values.

```
In [18]: import math
        print("math.ceil(3.4) :", math.ceil(3.4))
        print("math.pow(3, 3) :", math.pow(3, 3))
        print("math.sqrt(65) :", math.sqrt(65))
        print("math.sqrt(65) :", round(math.sqrt(65),2))
        print("math.log10(100) :", math.log10(100))
```

```
math.ceil(3.4) : 4
math.pow(3, 3) : 27.0
math.sqrt(65) : 8.06225774829855
math.sqrt(65) : 8.06
math.log10(100) : 2.0
```

### 2.10.6 help function

- Used to know the purpose of a function and how it is used.

```
In [19]: import math
         print(help(math.cos))
```

Help on built-in function cos in module math:

```
cos(...)
cos(x)
```

Return the cosine of x (measured in radians).

None

### 2.11 Function Definition and Call

The **syntax** for a function definition is as follows:

```
def function_name ( comma_separated_list_of_parameters):
    statements
```

Note: Statements below **def** begin with four spaces. This is called **indentation**. It is a requirement of Python that the code following a colon must be indented.

```
In [20]: def triangle():
         '''
         Objective: To print a right angled triangle.
         Input Parameter: None
         Return Value: None
         '''
         '''
         Approach: To use a print statement for each line of output
         '''
         print('*')
         print('* *')
         print('* * *')
         print('* * * *')

         triangle()

*
* *
* * *
* * * *
```



## Invoking the function

```
In [21]: triangle()
```

```
*
* *
* * *
* * * *
```

### 2.11.1 Computing Area of the Rectangle

```
In [22]: def areaRectangle(length, breadth):
        '''
        Objective: To compute the area of rectangle
        Input Parameters: length, breadth numeric value
        Return Value: area - numeric value
        '''
        area = length * breadth
        return area
```

```
In [23]: areaRectangle(7,5)
```

```
Out[23]: 35
```

```
In [24]: help(areaRectangle)
```

```
Help on function areaRectangle in module __main__:
```

```
areaRectangle(length, breadth)
    Objective: To compute the area of rectangle
    Input Parameters: length, breadth numeric value
    Return Value: area - numeric value
```

```
In [25]: def areaRectangle(length, breadth=1):
        '''
        Objective: To compute the area of rectangle
        Input Parameters: length, breadth - numeric value
        Return Value: area - numeric value
        '''
        area = length * breadth
        return area

    def main():
        '''
        Objective: To compute the area of rectangle based on user input
        Input Parameter: None
```

```

    Return Value: None
    '''
    print('Enter the following values for rectangle:')
    lengthRect = int(input('Length : integer value: '))
    breadthRect = int(input('Breadth : integer value: '))
    areaRect = areaRectangle(lengthRect, breadthRect)
    print('Area of rectangle is', areaRect)

main()

```

```

Enter the following values for rectangle:
Length : integer value: 6
Breadth : integer value: 2
Area of rectangle is 12

```

---

## 2.12 Pass by reference vs value

- All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function

```

In [26]: def sum(a,b=1):
        a=5
        print(a)
        print(b)
        return(a+b)

def main():
    x=int(input('Enter integer'))
    y=int(input('Enter integer'))
    print(x)
    print(y)
    z=sum(x,y)
    #sum(b=y, a=x)
    #sum(x)
    print(x)
    print(y)
    print(z)

main()

```

```

Enter integer5
Enter integer2
5
2
5

```

2  
5  
2  
7

```
In [27]: def changeme( mylist ):  
        #This changes a passed list into this function  
        mylist.append(40);  
        print('Values inside the function:')  
        print(mylist)  
        return  
  
        # Now you can call changeme function  
        mylist = [10,20,30];  
        changeme( mylist );  
        print('Values outside the function: ')  
        print(mylist)
```

Values inside the function:

[10, 20, 30, 40]

Values outside the function:

[10, 20, 30, 40]

```
In [28]: def changeme( mylist ):  
        #This changes a passed list into this function  
        mylist = [1,2,3,4]; # This would assign new reference in mylist  
        print('Values inside the function: ')  
        print(mylist)  
        return  
  
        # Now you can call changeme function  
        mylist = [10,20,30];  
        changeme( mylist );  
        print('Values outside the function: ')  
        print(mylist)
```

Values inside the function:

[1, 2, 3, 4]

Values outside the function:

[10, 20, 30]

---

## 2.13 Variable-length arguments

- You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

- Syntax for a function with non-keyword variable arguments is this

`def functionname([formal_args,] var_args_tuple ): "function_docstring" function_suite return ()`  
 An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

```
In [29]: # Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print('Output is: ')
    print(arg1)
    for var in vartuple:
        print(var)
    return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Output is:

10

Output is:

70

60

50

## 2.14 Global and Local variables

- follows LEGB (local, enclosing, global ,built-in)

```
In [30]: a=3
def f():
    def g():
        global a
        #a=4
        print('inside function g, global a=',a)
    #g()
    #global a
    a=5
    g()
    print('inside function f, local a=',a)

f()
print('outside of all function definitions a =',a)
```

```
inside function g, global a= 3
inside function f, local a= 5
outside of all function definitions a = 3
```

---

## 2.15 Control Structures

- Needed for non-sequential and repetitive execution of instructions.

## 2.16 if Conditional Statement

- Used to execute a certain sequence of statements depending upon fulfilment of a particular condition > The general form of **if-elif-else** statement is as follows:  
if < condition1 >: < Sequence S1 of statements to be executed > elif < condition2 >: < Sequence S2 of statements to be executed > elif < condition3 >: < Sequence S3 of statements to be executed > ...  
else: < Sequence Sn of statements to be executed >

### 2.16.1 Problem: Grade assignment on the basis of marks obtained

```
In [31]: def assignGrade(marks):
        '''
        Objective: To assign grade on the basis of marks obtained
        Input Parameter: marks numeric value
        Return Value: grade - string
        '''
        assert marks >= 0 and marks <= 100
        if marks >= 90:
            grade = 'A'
        elif marks >= 70:
            grade = 'B'
        elif marks >= 50:
            grade = 'C'
        elif marks >= 40:
            grade = 'D'
        else:
            grade = 'F'
        return grade

    def main():
        '''
        Objective: To assign grade on the basis of input marks
        Input Parameter: None
        Return Value: None
        '''
        marks = float(input('Enter your marks: '))
```

```

    print('Marks:', marks, '\nGrade:', assignGrade(marks))

if __name__ == '__main__':
    main()

```

Enter your marks: 67  
 Marks: 67.0  
 Grade: C

## 2.17 for Statement

- It is used when we want to execute a sequence of statements (indented to the right of keyword for) a fixed number of times. > Syntax of **for** statement is as follows:  
 for variable in sequence:

```

In [32]: for letter in "hello":
        print(letter)

```

h  
 e  
 l  
 l  
 o

### 2.17.1 Generating sequence of numbers using range function

Syntax:

```
range(start, end, increment)
```

The function call **range(1,n + 1)** produces a sequence of numbers from 1 to n

```

In [33]: start = 1
        limit = 11
        for num in range(start, limit):
            print(num)

```

1  
 2  
 3  
 4  
 5  
 6  
 7  
 8  
 9  
 10

```
In [34]: start = 1
        limit = 11
        step = 2
        for num in range(start, limit, step):
            print(num)
```

1  
3  
5  
7  
9

```
In [35]: start = 30
        limit = -4
        step = -3
        for num in range(start, limit, step):
            print(num)
```

30  
27  
24  
21  
18  
15  
12  
9  
6  
3  
0  
-3

```
In [36]: limit = 5
        for num in range(limit):
            print(num)
```

0  
1  
2  
3  
4

### 2.17.2 Problem: Printing a Triangle

```
In [37]: def rightTriangle(rows):
        '''
```

*Objective: To print a triangle comprising of asterisks*

```

    Input Parameter: rows - numeric
    Return Value: None
    '''
    for i in range(1, rows + 1):
        print('*' * i)

def main():
    '''
    Objective: To compute factorial of a number provided as an input
    Input Parameter: None
    Return Value: None
    '''
    rows = int(input('Enter number of rows: '))
    rightTriangle(rows)

if __name__ == '__main__':
    main()

```

Enter number of rows: 5

```

*
**
***
****
*****

```

### 2.17.3 Problem: Factorial of a number

```

In [38]: def factorial(num):
    '''
    Objective: To compute factorial of a number
    Input Parameter: num - numeric
    Return Value: num! - numeric
    '''
    if num <= 0:
        return 'Factorial Not defined'
    fact = 1
    for i in range(1, num+1):
        fact = fact * i
    return fact

def main():
    '''
    Objective: To compute factorial of a number provided as an input
    Input Parameter: None
    Return Value: None
    '''
    num = int(input('Enter the number: '))

```



```

        fact = factorial(num)
        print("Result:", fact)

if __name__ == '__main__':
    main()

```

Enter the number: 5

Result: 120

## 2.18 while Statement

- It is used for executing a sequence of statements again and again on the basis of some test condition.
- If the test condition holds True, the body of the loop is executed, otherwise the control moves to the statement immediately following the while loop. > Syntax of **while** statement is as follows:

while :

```

In [39]: count, n = 1, 5
        while count < n+1:
            print(count)
            count += 1

```

1  
2  
3  
4  
5

### 2.18.1 Sum of digits of a number

```

In [40]: def sumOfDigits(num):
        '''
        Objective: To compute sum of digits of a number
        Input Parameter: num - numeric
        Return Value: numeric
        '''
        Approach:
            Ignore the sign of number. Initialize sum to zero.
            Extract digits one by one beginning unit's place and keeps on
            adding it to sum.
        '''
        num = abs(num)
        total = 0
        while num >= 1:

```

```

        total += (num % 10)
        num = num // 10
    return total

def main():
    '''
    Objective: To compute sum of digits of a number provided as an input
    Input Parameter: None
    Return Value: None
    '''
    num = int(input('Enter the number: '))
    total = sumOfDigits(num)
    print("Result:", total)

if __name__ == '__main__':
    main()

```

Enter the number: 12345

Result: 15

### 2.18.2 Single Statement Suites

Similar to the if statement syntax, if your while clause consists only of a single statement, it may be placed on the same line as the while header.

```

In [41]: flag = 0
        while (flag): print('Given flag is really true!')
        print("Good bye!")

```

Good bye!

## 2.19 Loop Control Statements:

Loop control statements change execution from its normal sequence.

### 2.19.1 Break Statement

Terminates the loop statement and transfers execution to the statement immediately following the loop. If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

```

In [42]: 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!")

```

```

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!

```

### 2.19.2 Continue Statement

The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The continue statement can be used in both while and for loops.

```

In [43]: for letter in 'Python':    # First Example
        if letter == 'h':
            continue
        print('Current Letter :', letter)

```

```

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

```

### 2.19.3 Pass Statement

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

```

In [44]: def passdemo(x,y):
        pass
        passdemo(2,3)
        print('helo here')

```

```

helo here

```

#### 2.19.4 The else Statement Used with Loops

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. \* If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

```
In [45]: count = 0
        while(count < 5):
            print(count, " is less than 5")
            count = count + 1
        else:
            print(count, " is not less than 5")
```

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

```
In [1]: count = 0
        while(count < 5):
            print(count, " is less than 5")
            count = count + 1
            if count==3:
                break
        else:
            print(count, " is not less than 5")
        print("after while loop")
```

```
0 is less than 5
1 is less than 5
2 is less than 5
after while loop
```