MODULE I

1.1 PYTHON BASICS

Entering Expressions into the Interactive Shell, The Integer, Floating-Point, and String Data Types, String Concatenation and Replication, Storing Values in Variables, Your First Program, Dissecting Your Program

1.2 FLOW CONTROL

Boolean Values, Comparison Operators, Boolean Operators, Mixing Boolean and Comparison Operators, Elements of Flow Control, Program Execution, Flow Control Statements, Importing Modules, Ending a Program Early with sys.exit()

1.3 FUNCTIONS

def Statements with Parameters, Return Values and return Statements, The None Value, Keyword Arguments and print(), Local and Global Scope, The global Statement, Exception Handling, A Short Program: Guess the Number

MODULE I

1.2 FLOW CONTROLS

➢ Boolean values

✓ In Python, integer, floating-point, and string data types have an unlimited number of possible values, the Boolean data type has only two values: **True and False**.

✓ When typed as Python code, the Boolean values True and False lack the quotes you place around strings, and they always start with a capital T or F, with the rest of the word in lowercase.

```
❶ >>> spam = True
  >>> spam
  True
❷ >>> true
  Traceback (most recent call last):
    File "<pyshell#2>", line 1, in <module>
      true
  NameError: name 'true' is not defined
❸ >>> True = 2 + 2
  SyntaxError: assignment to keyword
```

Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If you don't use the proper case ❷ or you try to use True and False for variable names ❸, Python will give you an error message.

➢ Comparison Operators

✓ Comparison operators compare two values and evaluate down to a single Boolean value.

✓ List of Comparison Operators are:

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

- ✓ These operators evaluate to True or False depending on the values we give them.
- ✓ Example:

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

- ✓ As expected, == (equal to) evaluates to True when the values on both sides are the same, and != (not equal to) evaluates to True when the two values are different.
- ✓ The == and != operators can actually work with values of any data type.

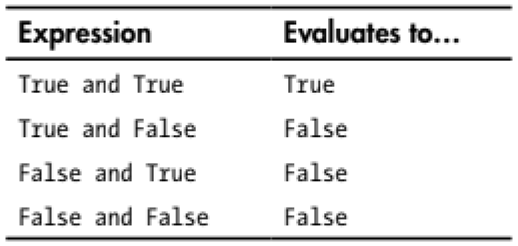
```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
>>> 42 == '42'
False
```

- ✓ The <, >, <=, and >= operators, on the other hand, work properly only with integer and floating-point values.

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
>>> eggCount <= 42
True
>>> myAge = 29
>>> myAge >= 10
True
```

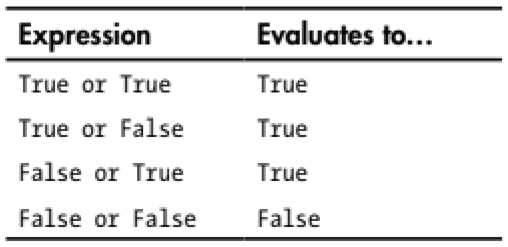
➢ BOOLEAN OPERATORS

- ✓ The three Boolean operators (and, or, and not) are used to compare Boolean values.
- ✓ Like comparison operators, they evaluate these expressions down to a Boolean value.
- ✓ Binary Boolean Operators : The and and or operators always take two Boolean values (or expressions), so they're considered binary operators.
- ✓ The and operator evaluates an expression to True if both Boolean values are True; otherwise, it evaluates to False.
- ✓ Truth table of **and** operator:



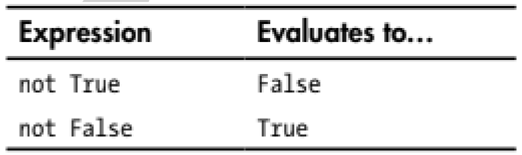
Expression	Evaluates to...
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

- ✓ The or operator evaluates an expression to True if either of the two Boolean values is True. If both are False, it evaluates to False.
- ✓ Truth table of **or** operator:



Expression	Evaluates to...
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

- ✓ The not operator operates on only one Boolean value (or expression). The not operator simply evaluates to the opposite Boolean value.
- ✓ Truth table of **not** operator:



Expression	Evaluates to...
`not True`	`False`
`not False`	`True`

➢ FLOW CONTROL STATEMENTS

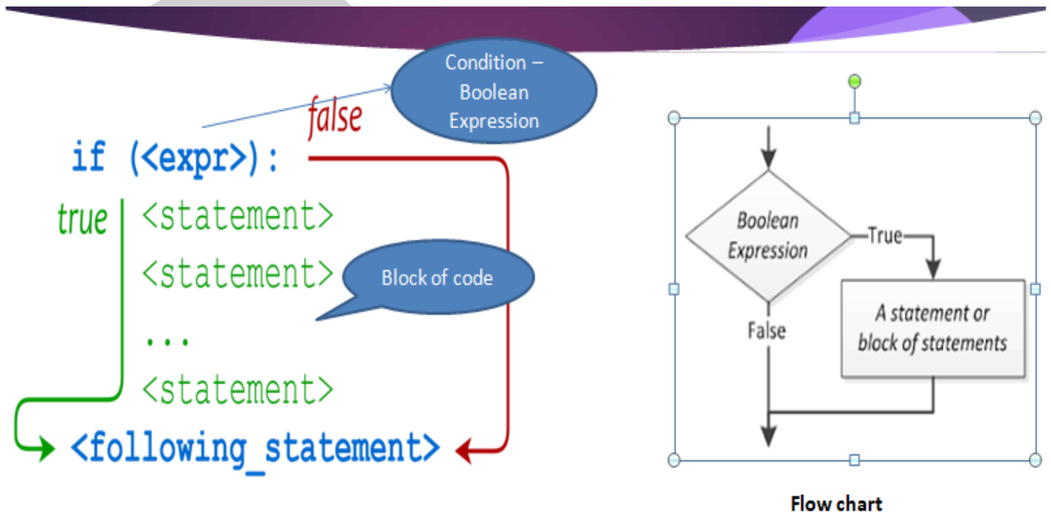
- ✓ A program is just a series of instructions which will be executed sequentially.
- ✓ But the real strength of programming isn't just running (or executing) one instruction after

another

- ✓ Based on how the expressions evaluate, the program can decide to skip instructions, repeat them, or choose one of several instructions to run.
- ✓ A Flow Control Statement defines the flow of the execution of the Program
- ✓ Flow control statements can decide which Python instructions to execute under which conditions.

➢ ELEMENTS OF FLOW CONTROL STATEMENTS

- ✓ Flow control statements
  - start with a part called the **condition**, and
  - followed by a block of code called the **clause**.
- ✓ **Conditions** are boolean expressions that evaluate down to a Boolean value, True or False.
- ✓ A flow control statement decides what to do based on whether its condition is True or False, and almost every flow control statement uses a condition.
- ✓ **Block of Code(Claus) :** Lines of Python code can be grouped together in blocks.
- ✓ We can tell when a block begins and ends from the indentation of the lines of code.
- ✓ There are three rules for blocks.
- ▪ 1. Blocks begin when the indentation increases.
- ▪ 2. Blocks can contain other blocks.
- ▪ 3. Blocks end when the indentation decreases to zero or to a containing block's indentation.
- ✓ **Example:**



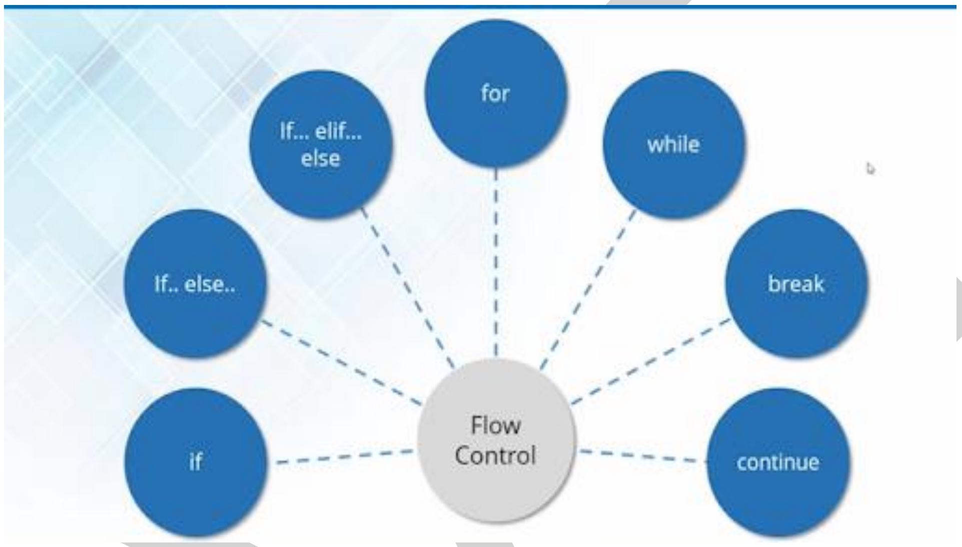
➢ TYPES OF FLOW CONTROL STATEMENTS

In Python, there are 3 types of flow control statements

→ Conditional Statements(Decision Statements)

→ Looping Statements

→ Break, Continue and Pass Statements



➢ CONDITIONAL STATEMENTS (DECISION MAKING)

→ Python language provide the following conditional (Decision making) statements.

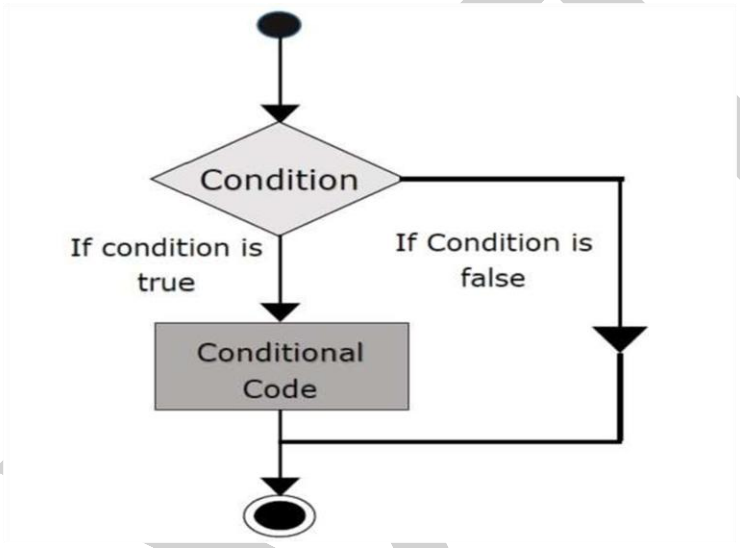
- if statement
- if...else statement
- if...elif...else staement
- Nested if..else statement

❖ The if statement

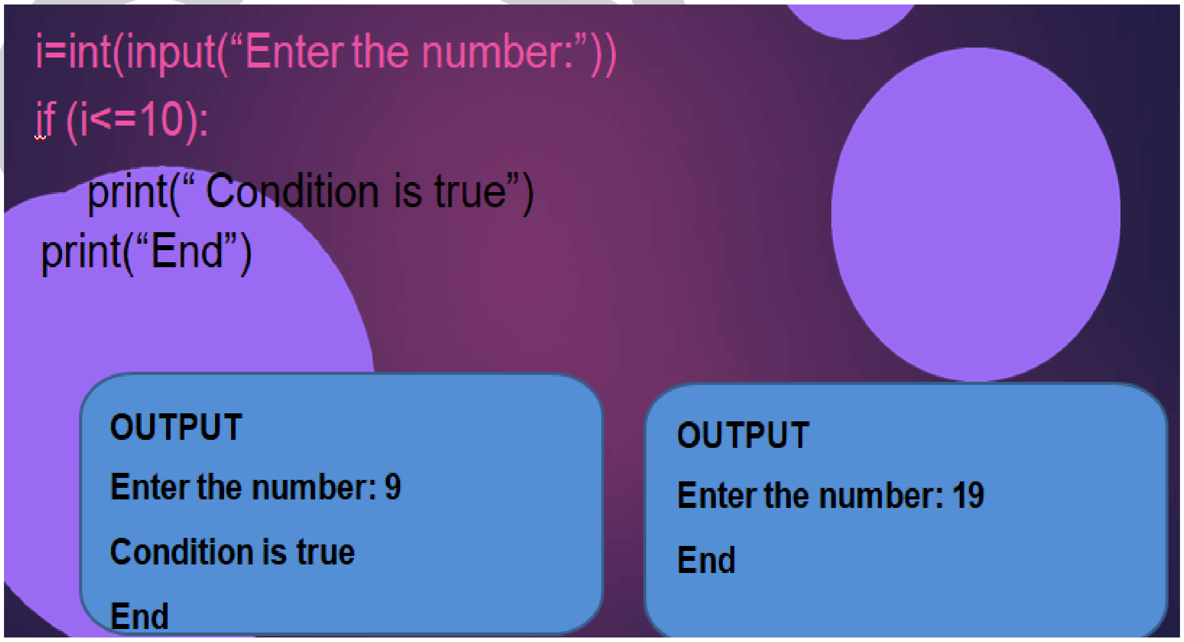
- The if statement is a decision making statement.
- An if statement's clause (that is, the block following the if statement) will execute if the statement's condition is True. The clause is skipped if the condition is False.
- **Syntax:**

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

- In Python, an if statement consists of the following:
    - The if keyword
    - A condition (that is, an expression that evaluates to True or False)
    - A colon
    - Starting on the next line, an indented block of code (called the if clause)
- **Flowchart** is as follows:



- **Example:**



```
i=int(input("Enter the number:"))
if (i<=10):
    print(" Condition is true")
print("End")
```

```
OUTPUT
Enter the number: 9
Condition is true
End
```

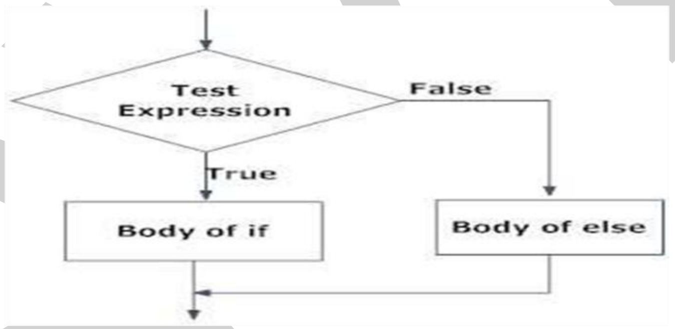
```
OUTPUT
Enter the number: 19
End
```

➢ The if..else statement

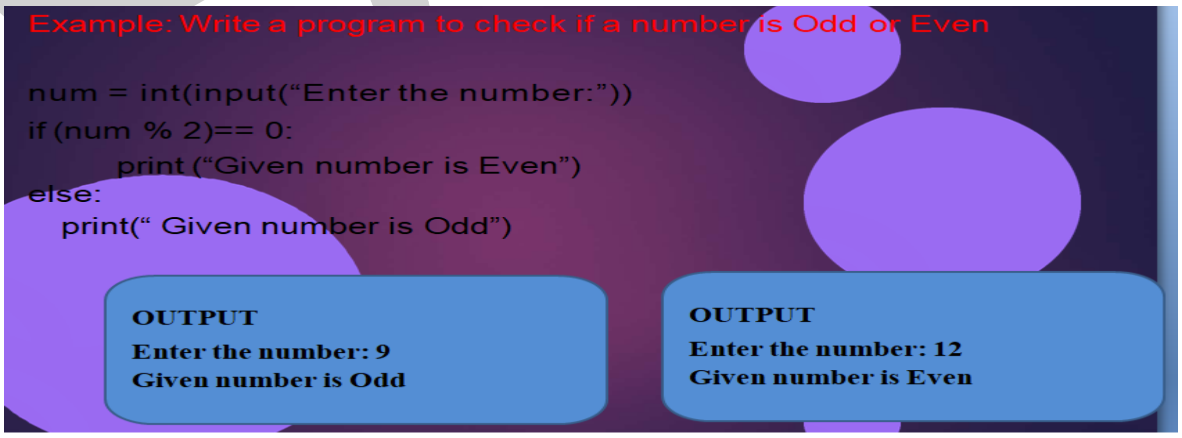
- An if clause can optionally be followed by an else statement.
- The else clause is executed only when the if statement's condition is False.
- The if…else statement is called alternative execution, in which there are two possibilities and the condition determines which one gets executed.
- **Syntax:**

```
if test expression:
        statements
else:
        statements
```

- An else statement always consists of the following:
  - The else keyword
  - A colon
  - Starting on the next line, an indented block of code (called the else clause)
- **Flowchart** is as follows:



- **Example:**



Example: Write a program to check if a number is Odd or Even

```
num = int(input("Enter the number:"))
if (num % 2)== 0:
        print ("Given number is Even")
else:
   print(" Given number is Odd")
```

```
OUTPUT
Enter the number: 9
Given number is Odd
```

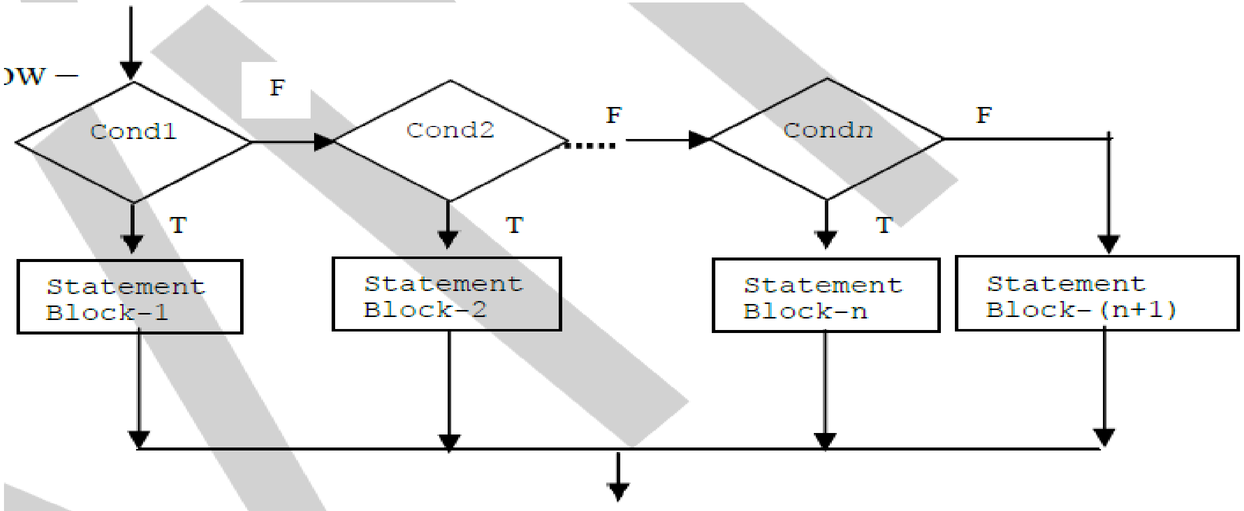
```
OUTPUT
Enter the number: 12
Given number is Even
```

➢ elif Statements

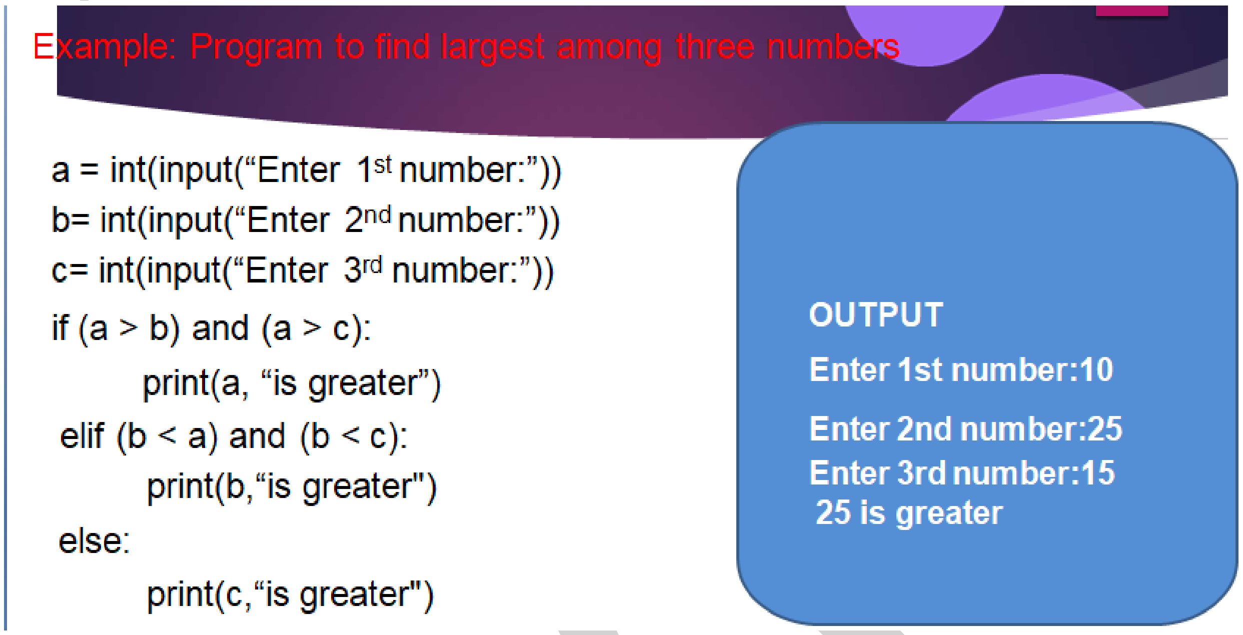
- The elif statement always follows an if or another elif statement.
- It provides another condition that is checked only if any of the previous conditions were False.
- In code, an elif is a keyword in Python in replacement of else if to place another condition in the program. This is called chained conditional.
- Elif statement always consists of the following:
    - A condition (that is, an expression that evaluates to True or False)
    - A colon
    - Starting on the next line, an indented block of code (called the elif clause)
- **Syntax:**

```
if condition1:
       Statement Block-1
elif condition2:
       Statement Block-2
       |
       |
       |
       |
elif condition_n:
       Statement Block-n
else:
       Statement Block-(n+1)
```

- **Flowchart** is as follows:



- Here, the conditions are checked one by one sequentially.
- If any condition is satisfied, the respective statement block will be executed and further conditions are not checked.
- Note that, the last else block is not necessary always.
- **Example:**



Example: Program to find largest among three numbers

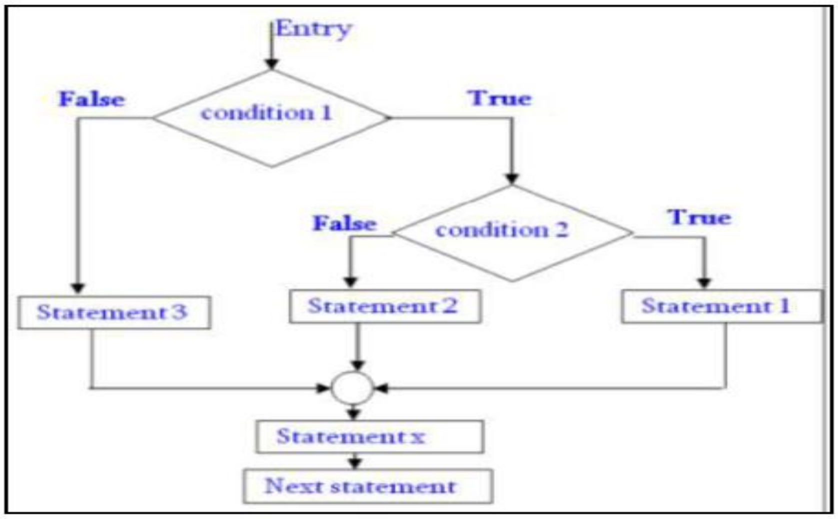
```
a = int(input("Enter  1st number:"))
b= int(input("Enter  2nd number:"))
c= int(input("Enter  3rd number:"))
if (a > b) and (a > c):
        print(a, "is greater")
 elif (b < a) and (b < c):
        print(b,"is greater")
 else:
        print(c,"is greater")
```

OUTPUT

```
Enter 1st number:10
Enter 2nd number:25
Enter 3rd number:15
 25 is greater
```

➢ Nested if..else statements

- We can write an entire if… else statement in another if… else statement called nesting, and the conditional statement is called nested conditional statements.
- It can be done in multiple ways depending on programmer's requirements.
- Flowchart Illustrating an Example:



- Examples are given below –

Ex1.

```
marks=float(input("Enter marks:"))
if marks>=60:
    if marks<70:
        print("First Class")
    else:
        print("Distinction")
```

Sample Output:

```
Enter
marks:6
8 First
Class
```

Here, the outer condition marks>=60 is checked first. If it is true, then there are two branches for the inner conditional. If the outer condition is false, the above code does nothing.

Ex2.

```
gender=input("Enter gender:")
age=int(input("Enter age:"))

if gender == "M" :
    if age >= 21:
        print("Boy, Eligible for Marriage")
    else:
        print("Boy, Not Eligible for Marriage")
elif gender == "F":
    if age >= 18:

        print("Girl, Eligible for Marriage")

    else:
        print("Girl, Not Eligible for Marriage")
```

Sample Output:

```
Enter
gender: F
Enter
age: 17
Girl, Not Eligible for Marriage
```

➢ LOOPING/ITERATION STATEMENTS

- ✓ Iteration is a processing of repeating some task.
- ✓ In a real time programming, we require a set of statements to be repeated certain number of times and/or till a condition is met.
- ✓ Constructs that are available in python are : **while** and **for loop**

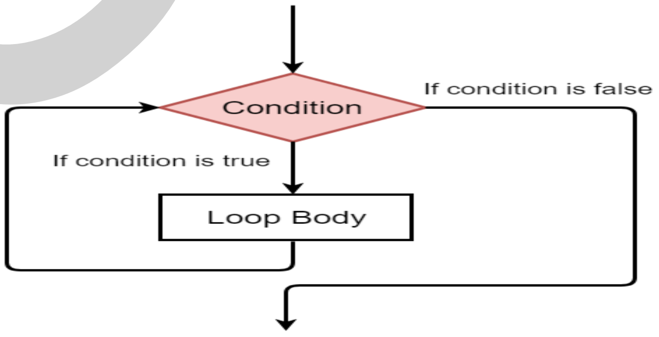
➢ While Loop

- We can make a block of code to execute repeatedly with a while statement.
- The code in a while clause will be executed as long as the while statement's condition is True.
- In code, a while statement always consists of the following:
  - The while keyword
  - A condition (that is, an expression that evaluates to True or False)
  - A colon
  - Starting on the next line, an indented block of code (called the while clause)
- **Syntax:**

```
while condition:
     statement_1
     statement_2
     …………….
     statement_n

statements_after_while
```

- **Flowchart** is as follows:



- Here, ***while*** is a keyword, the flow of execution for a while statement is as below.
- The condition is evaluated first, yielding True or False
- If the condition is false, the loop is terminated and statements after the loop will be executed.
- If the condition is true, the body will be executed which comprises of the statement_1 to statement_n and then goes back to condition evaluation.
- Consider an example –

```
n=1
while n<=5:
        print(n)            #observe indentation
        n=n+1
print("over")
```

The output of above code segment would be –

```
1
2
3
4
5
over
```

- In the above example, a variable n is initialized to 1. Then the condition n<=5 is being checked. As the condition is true, the block of code containing print statement print(n) and increment statement (n=n+1)are executed. After these two lines, condition is checked again. The procedure continues till condition becomes false, that is when n becomes 6. Now, the while-loop is terminated and next statement after the loop will be executed. Thus, in this example, the loop is ***iterated*** for 5 times.
- Consider another example –

```
n=5
while n>0:
        print(n)            #observe indentation
        n=n-1
print("Blast off!")
```

The output of above code segment would be –

```
5
4
3
2
1
Blast off!
```

- Iteration is referred to each time of execution of the body of loop.
- Note that, a variable n is initialized before starting the loop and it is incremented/decremented inside the loop. Such a variable that changes its value for every iteration and controls the total execution of the loop is called as ***iteration variable*** or ***counter variable***. If the count variable is not updated properly within the loop, then the loop may not terminate and keeps executing infinitely.

➢ Definite Loops using *for*

- The *while* loop iterates till the condition is met and hence, the number of iterations are usually unknown prior to the loop. Hence, it is sometimes called as *indefinite loop*.
- When we know total number of times the set of statements to be executed, ***for*** loop will be used. This is called as a *definite loop***.** The for-loop iterates over a set of numbers, a set of words, lines

```
for var in list/sequence:
     statement_1
     statement_2
     ..................
     statement_n

statements_after_for
```

in a file etc. The syntax of for-loop would be –

Here, *for* and *in*	are keywords
list/sequence	is a set of elements on which the loop is iterated. That is, the loop will be executed till there is an element in list/sequence
statements	constitutes body of the for loop

- **Example:** In the below given example, a ***list*** names containing three strings has been created. Then the counter variable x in the *for*-loop iterates over this *list*. The variable x takes the elements in names one by one and the body of the loop is executed.

```
names=["Ram", "Shyam", "Bheem"]
for x in names:
        print("Happy New Year",x)
print('Done!')
```

The output would be –

```
Happy New Year Ram
Happy New Year Shyam
Happy New Year Bheem
Done!
```

NOTE: In Python, list is an important data type. It can take a sequence of elements of different types. It can take values as a comma separated sequence enclosed within square brackets. Elements in the list can be extracted using index (just similar to extracting array elements in C/C++ language). Various operations like indexing, slicing, merging, addition and deletion of elements etc. can be applied on lists. The details discussion on Lists will be done in Module 2.

- The *for* loop can be used to print (or extract) all the characters in a string as shown below –

```
for i in "Hello":
      print(i, end='\t')
```

Output:

```
H       e       l       l       o
```

- When we have a fixed set of numbers to iterate in a *for* loop, we can use a function ***range()***. The function *range()* takes the following format –

```
range(start, end, steps)
```

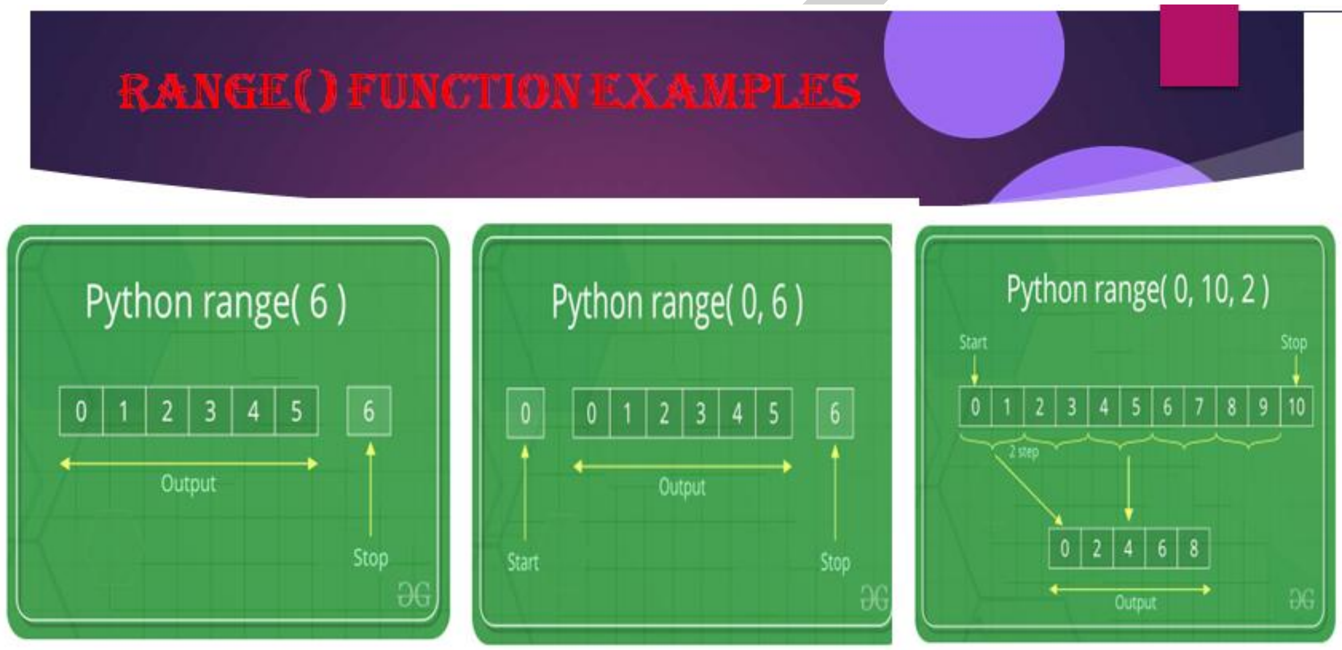
- The start and end indicates starting and ending values in the sequence, where end is excluded in the sequence (That is, sequence is up to end-1). The default value of start is 0. The argument steps indicates the increment/decrement in the values of sequence with the default value as 1. Hence, the argument steps is optional.

range(stop) takes one argument.

range(start, stop) takes two arguments.

range(start, stop, step) takes three arguments.

- Let us consider few examples on usage of *range()* function.



Ex1. Printing the values from 0 to 4 –

```
for i in range(5):
        print(i, end= '\t')
```

Output:

```
0        1        2        3        4
```

Here, 0 is the default starting value. The statement range(5)is same as range(0,5) and range(0,5,1).

Ex2. Printing the values from 5 to 1 –

```
for i in range(5,0,-1):
        print(i, end= '\t')
```

Output:

```
5	4	3	2	1
```

The function range(5,0,-1)indicates that the sequence of values are 5 to 0(excluded) in steps of -1 (downwards).

Ex3. Printing only even numbers less than 10 –

```
for i in range(0,10,2):
	print(i, end= '\t')
```

Output:

```
0	2	4	6	8
```

Points to remember about Python range() function :

→ range() function only works with the integers i.e. whole numbers.

→ All argument must be integers. User can not pass a string or float number or any other type in

→ a start, stop and step argument of a range().

→ All three arguments can be positive or negative.

→ The step value must not be zero. If a step is zero python raises a ValueError exception.

→ range() is a type in Python

➢ Loop Patterns

The *while*-loop and *for*-loop are usually used to go through a list of items or the contents of a file and to check maximum or minimum data value. These loops are generally constructed by the following procedure –

- Initializing one or more variables before the loop starts
- Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop
- Looking at the resulting variables when the loop completes

The construction of these loop patterns are demonstrated in the following examples.

Counting and Summing Loops: One can use the *for* loop for counting number of items in the list as shown –

```
count = 0
for i in [4, -2, 41, 34, 25]:
        count = count + 1
print("Count:", count)
```

- Here, the variable count is initialized before the loop. Though the counter variable is not being used inside the body of the loop, it controls the number of iterations.
- The variable count is incremented in every iteration, and at the end of the loop the total number of elements in the list is stored in it.
- One more loop similar to the above is finding the sum of elements in the list –

```
total = 0
for x in [4, -2, 41, 34, 25]:
        total = total + x
print("Total:", total)
```

- Here, the variable total is called as ***accumulator*** because in every iteration, it accumulates the sum of elements. In each iteration, this variable contains *running total of values so far*.

NOTE: In practice, both of the counting and summing loops are not necessary, because there are built-in functions len()and sum()for the same tasks respectively.

Maximum and Minimum Loops: To find maximum element in the list, the following code can be used –

```
big = None
print('Before Loop:', big)
for x in [12, 0, 21,-3]:
        if big is None or x > big :
                big = x
        print('Iteration Variable:', x, 'Big:', big)
print('Biggest:', big)
```

Output:

```
Before Loop: None
Iteration Variable: 12          Big: 12
Iteration Variable: 0           Big: 12
Iteration Variable: 21          Big: 21
Iteration Variable: -3          Big: 21
Biggest: 21
```

- Here, we initialize the variable big to None. It is a special constant indicating empty.
- Hence, we cannot use relational operator == while comparing it with big. Instead, the ***is*** operator must be used.
- In every iteration, the counter variable x is compared with previous value of big. If x > big, then xis assigned to big.
- Similarly, one can have a loop for finding smallest of elements in the list as given below –

```
small = None
print('Before Loop:', small)
for x in [12, 0, 21,-3]:
    if small is None or x < small :
        small = x
    print('Iteration Variable:', x, 'Small:', small)
print('Smallest:', small)
```

Output:

```
Before Loop: None
Iteration Variable: 12      Small: 12
Iteration Variable: 0       Small: 0
Iteration Variable: 21      Small: 0
Iteration Variable: -3      Small: -3
Smallest: -3
```

NOTE: In Python, there are built-in functions max() and min() to compute maximum and minimum values among. Hence, the above two loops need not be written by the programmer explicitly. The inbuilt function min()has the following code in Python –

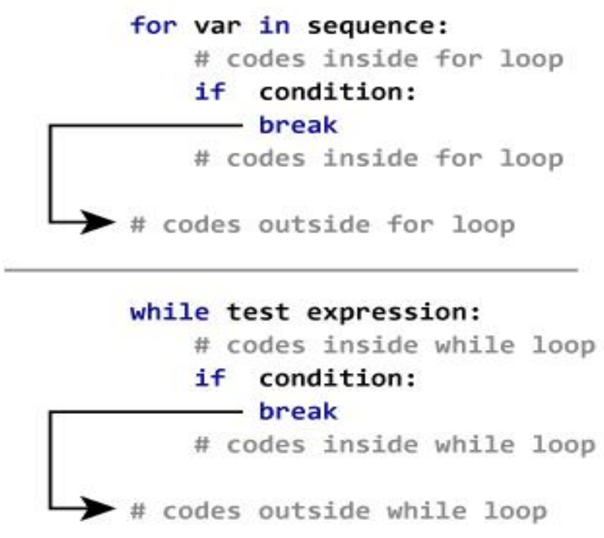
```
def min(values):
        smallest = None
        for value in values:
                if smallest is None or value < smallest:
                 smallest = value
        return smallest
```

➢ Infinite Loops, *break* and *continue*

- A loop may execute infinite number of times when the condition is never going to become false.
- For example,

```
n=1
while True:
        print(n)
        n=n+1
```

- Here, the condition specified for the loop is the constant True, which will never get terminated. Sometimes, the condition is given such a way that it will never become false and hence by restricting the program control to go out of the loop. This situation may happen either due to wrong condition or due to not updating the counter variable.
- In some situations, we deliberately want to come out of the loop even before the normal termination of the loop. For this purpose ***break*** statement is used.
- The same is illustrated below:

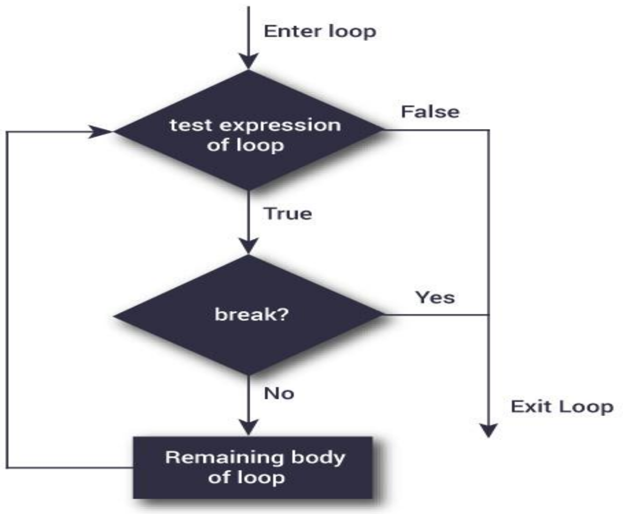


```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```



- The following example depicts the usage of *break***.** Here, the values are taken from keyboard until a negative number is entered. Once the input is found to be negative, the loop terminates.

```
while True:
          x=int(input("Enter a number:"))
           if x>= 0:
                    print("You have entered ",x)
          else:
                    print("You have entered a negative number!!")
                    break                  #terminates the loop
```

Output:

```
Enter a number:23
You have entered 23
Enter a number:12
You have entered 12
Enter a number:45
You have entered 45
Enter a number:0
You have entered 0
 Enter a number:-2
You have entered a negative number!!
```

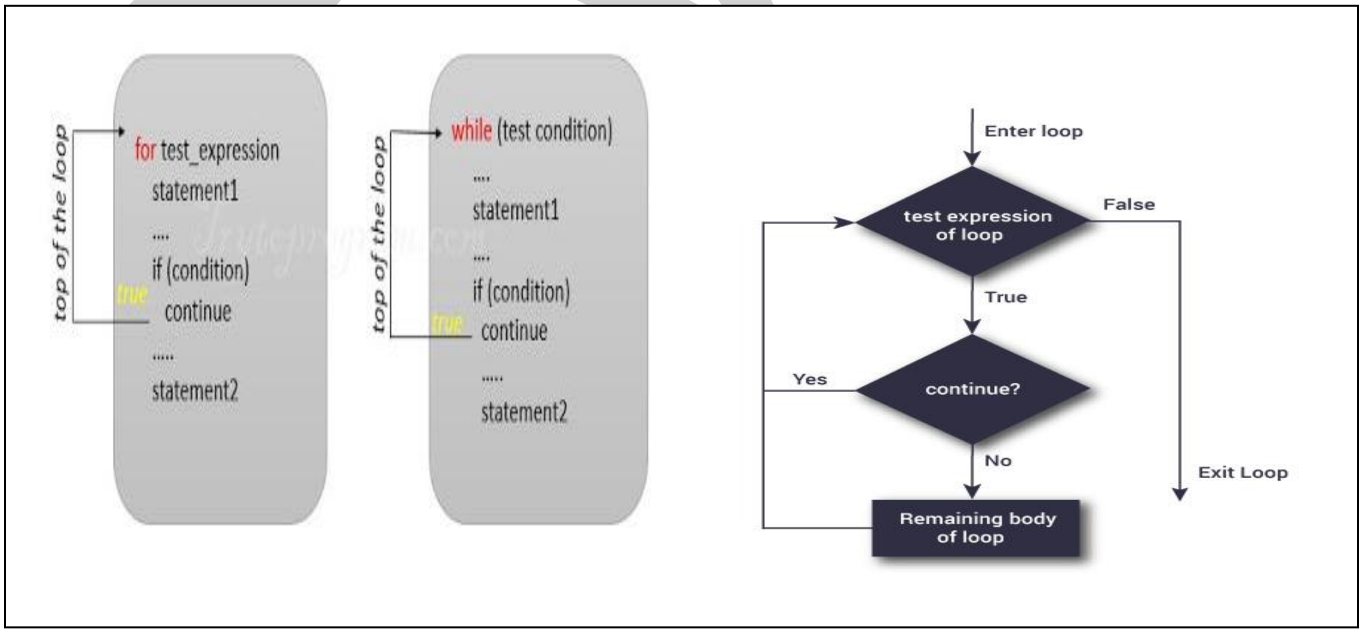
- In the above example, we have used the constant True as condition for while-loop, which will never become false. So, there was a possibility of infinite loop. This has been avoided by using *break* statement with a condition.
- The condition is kept inside the loop such a way that, if the user input is a negative number, the loop terminates. This indicates that, the loop may terminate with just one iteration (if user gives negative number for the very first time) or it may take thousands of iteration (if user keeps on giving only positive numbers as input). Hence, the number of iterations here is unpredictable.
- But, we are making sure that it will not be an infinite-loop, instead, the user has control on the loop.
- Another example for usage of while with break statement: the below code takes input from the user until they type done:

```
while True:
    line = input(">")
    if line == 'done':
        break
    print(line)
print('Done!')
```

- In the above example, since the loop condition is True, so the loop runs repeatedly until it hits the break statement.
- Each time it prompts the user to enter the data. If the user types done, the brak statement exits the loop. Otherwise the program echoes whatever the user types and goes back ti the top of the loop.
- **Output will be:**

```
>hello
hello
>finished
finished
>done
Done!
```

- Sometimes, programmer would like to move to next iteration by skipping few statements in the loop, based on some condition with current iteration. For this purpose ***continue*** statement is used.



- For example, we would like to find the sum of 5 even numbers taken as input from the keyboard. The logic is –
- Read a number from the keyboard
- If that number is odd, without doing anything else, just move to next iteration for reading another number
- If the number is even, add it to *sum* and increment the accumulator variable.
  - When accumulator crosses 5, stop the program
  - The program for the above task can be written as –

```
sum=0
count=0
while True:
        x=input("Enter a number:")
        if x%2!=0:
            continue
        else:
            sum+=x
            count+=1
        if count==5:
            break
print("Sum= ", sum)
```

Output:

```
Enter a number: 23
Enter a number: 67
Enter a number: 789
Enter a number: 78
Enter a number: 5
Enter a number: 7
Sum=  891
```

- Example of a loop that copies its input until the user types “done”, but treats lines that start with the hash character as lines not to be printed

```
while True:
  line=input('>')
  if line[0] == '#':
    continue
  if line =='done':
    break
  print(line)
print('Done!')
```

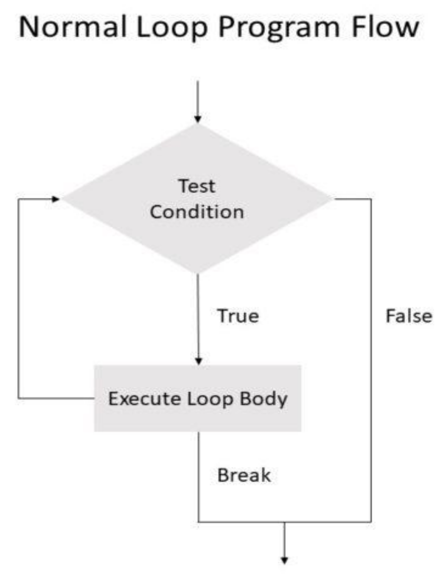
Output:

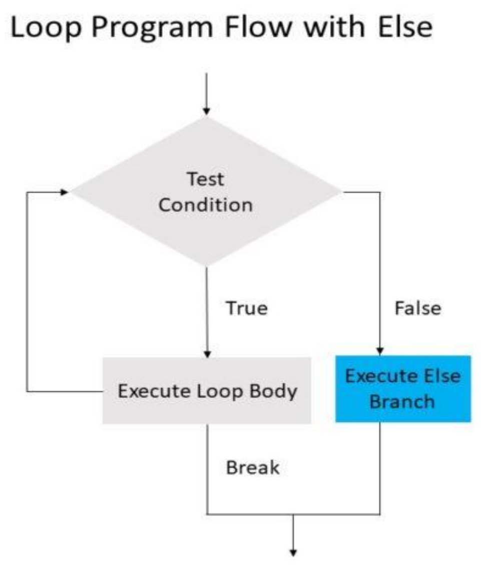
```
> hello there
hello there
> #dont print this
> print this!
print this!
> done
Done!
```

- Above, all lines are printed except the one that starts with ‘#’ because whenth econtinue is executed, it ends the current iteration and jumps back to the while statement to start the next iteration, thus skipping the print statement.

➢ Else block with iteration statements

- ✓ A while loop may have an else statement after it[optional].
- ✓ When the condition becomes false, the block under the else statement (clause) is executed.
- ✓ However, it doesn’t execute if you break out of the loop or if an exception is raised.





- **Example1:**

```
a=3
while(a>0):
        print(a)
        a=a-1
else:
        print("Reached 0")
print("OVER")
```

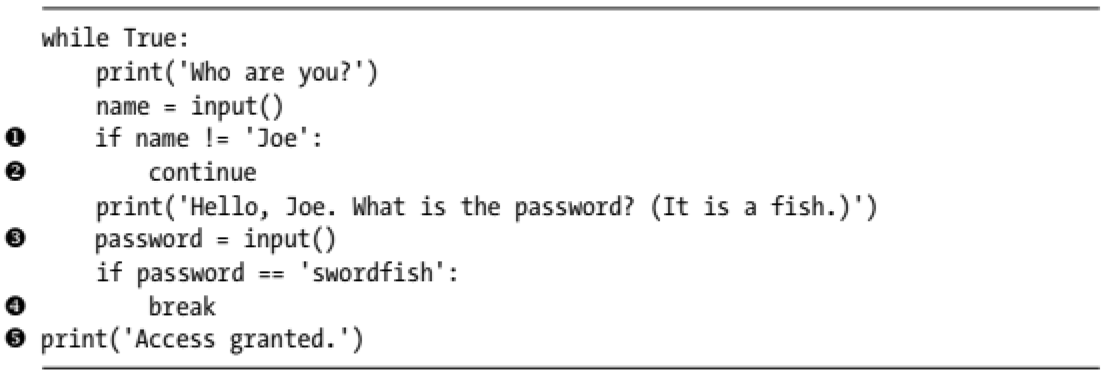
```
OUTPUT:
3
2
1
Reached 0
OVER
```

- **Example2:**

```
for i in range(10):
        print(i)
        if(i==7):
            break
else:
        print("Reached else")
print("OVER")
```

```
OUTPUT:
0
1
2
3
4
5
6
7
OVER
```

➢ Example 3: Write a Program that asks for a name and password.



```
  while True:
      print('Who are you?')
      name = input()
❶     if name != 'Joe':
❷         continue
      print('Hello, Joe. What is the password? (It is a fish.)')
❸     password = input()
      if password == 'swordfish':
❹         break
❺ print('Access granted.')
```

- ✓ If the user enters any name besides Joe (1), the continue statement (2) causes the program execution to jump back to the start of the loop.
- ✓ When it reevaluates the condition, the execution will always enter the loop, since the condition is simply the value True.
- ✓ Once the name = 'Joe' is entered, then, the user is asked for a password (3). If the password entered is swordfish, then the break statement (4) is run, and the execution jumps out of the while loop to print Access granted (5).
- ✓ Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start of the loop.

➢ Pass statement

- When we need a particular loop, class, or function in our program, but don't know what goes in it, we place the pass statement in it.
- It is a null statement.
- The pass statement is useful when you don't write the implementation of a function but you want to implement it in the future.
- The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored.
- The interpreter does not ignore it, but it performs a no-operation (NOP).
- Empty code is not allowed in loops, function definitions, class definitions, or in if statements.
- **Syntax of pass:**

  pass

- Example:

```
for x in [0, 1, 2]:
        pass
```

Nested Loops

- We can also nest a loop inside another.
- We can put a for loop inside a while, or a while inside a for, or a for inside a for, or a while inside a while.
- Example:

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

OUTPUT:

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

Importing Modules

- All Python programs can call a basic set of functions called built-in functions, including the print(), input(), and len() functions
- Python also comes with a set of modules called the standard library.
- Each module is a Python program that contains a related group of functions that can be embedded in the programs.
- For example, the math module has mathematics-related functions, the random module has random number–related functions, and so on.
- Before we can use the functions in a module, we must import the module with an import statement.

- In code, an import statement consists of the following:
    - The import keyword
    - The name of the module
- Optionally, more module names, as long as they are separated by commas
- Example with the random module, which will give us access to the random.randint() function.

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

- When we run this program, the output will look something like this:

```
4
1
8
4
1
```

- The random.randint() function call evaluates to a random integer value between the two integers that we pass to it.
- Since randint() is in the random module, we must first type **random.** in front of the function name to tell Python to look for this function inside the random module.
- Here's an example of an import statement that imports four different modules:

```
import random, sys, os, math
```

- Now we can use any of the functions in these four modules.
- An alternative form of the import statement is composed of the from keyword, followed by the module name, the import keyword, and a star;
- For example, **from random import ***
- With this form of import statement, calls to functions in random will not need the **random.** Prefix.
- However, using the full name makes for more readable code, so it is better to use the normal form of the import statement.

➢ Ending A Program Early With Sys.Exit()

- Program Termination happens when the program execution reaches the bottom of the instructions. However, we can cause the program to terminate, or exit, by calling the **sys.exit()** function.
- Since this function is in the sys module, we have to import sys before program uses it.
- **Example:**

```
import sys
while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
print('You typed ' + response + '.')
```

- This program has an infinite loop with no break statement inside. The only way this program will end is if the user enters exit, causing sys.exit() to be called. When response is equal to exit, the program ends.
- Since the response variable is set by the input() function, the user must enter exit in order to stop the program.

1.3 FUNCTIONS

- Functions are the building blocks of any programming language.
- A sequence of instructions intended to perform a specific independent task is known as a *function*.
- In this section, we will discuss various types of built-in functions, user-defined functions, applications/uses of functions etc.

→ Function Calls

- A function is a named sequence of instructions for performing a task.
- When we define a function we will give a valid name to it, and then specify the instructions for performing required task.
- Later, whenever we want to do that task, a function is *called* by its name.

- Consider an example:

```
>>> type(15)
<class 'int'>
```

  - Here *type* is a function name, 15 is the argument to a function and <class 'int'> is the result of the function.
  - Usually, a function *takes* zero or more arguments and *returns* the result.

→ Built-in Functions

- Python provides a rich set of built-in functions for doing various tasks.
- The programmer/user need not know the internal working of these functions; instead, they need to know only the purpose of such functions.
- Some of the built in functions are given below –

- **max():** This function is used to find maximum value among the arguments. It can be used for numeric values or even to strings.
  - max(10, 20, 14, 12) **#maximum of 4 integers**
    20
  - max("hello world")
    'w' **#character having maximum ASCII code**
  - max(3.5, -2.1, 4.8, 15.3, 0.2)
    15.3 **#maximum of 5 floating point values**

- **min():** As the name suggests, it is used to find minimum of arguments.
  - min(10, 20, 14, 12) **#minimum of 4 integers**
    10
  - min("hello world")
    ' ' **#space has least ASCII code here**
  - min(3.5, -2.1, 4.8, 15.3, 0.2)
    -2.1 **#minimum of 5 floating point values**

- **len():** This function takes a single argument and finds its length. The argument can be a string, list, tuple etc.

- len("hello how are you?")

  18

There are many other built-in functions available in Python. They are discussed in further Modules, wherever they are relevant.

→ Type Conversion Functions

- As we have seen earlier (while discussing *input()* function), the type of the variable/value can be converted using functions ***int(), float(), str()***.
- Python provides built-in functions that convert values from one type to another
- Consider following few examples –

```
➢ int('20')                 #integer enclosed within single quotes
     20                     #converted to integer type
➢ int("20")                 #integer enclosed within double quotes
      20

➢ int("hello")              #actual string cannot be converted to int

    Traceback (most recent call last):
    File "<pyshell#23>", line 1, in <module> int("hello")
    ValueError: invalid literal for int() with base 10: 'hello'
➢ int(3.8)                  #float value being converted to integer
     3                      #round-off will not happen, fraction is ignored
➢ int(-5.6)
     -5
➢ float('3.5')              #float enclosed within single quotes
     3.5                    #converted to float type
➢ float(42)                 #integer is converted to float
    42.0
➢ str(4.5)                  #float converted to string
    '4.5'
➢ str(21)                   #integer converted to string
```

```
'21'
```

→ Random Numbers

- Most of the programs that we write are *deterministic*.
- That is, the input (or range of inputs) to the program is pre-defined and the output of the program is one of the expected values.
- But, for some of the real-time applications in science and technology, we need randomly generated output. This will help in simulating certain scenario.
- Random number generation has important applications in games, noise detection in electronic communication, statistical sampling theory, cryptography, political and business prediction etc. These applications require the program to be *nondeterministic.*
- There are several algorithms to generate random numbers. But, as making a program completely *nondeterministic* is difficult and may lead to several other consequences, we generate ***pseudo-random numbers***.
- That is, the type (integer, float etc) and range (between 0 and 1, between 1 and 100 etc) of the random numbers are decided by the programmer, but the actual numbers are unknown.
- Moreover, the algorithm to generate the random number is also known to the programmer. Thus, the random numbers are generated using deterministic computation and hence, they are known as pseudo-random numbers!!
- Python has a module ***random*** for the generation of random numbers. One has to *import* this module in the program. The function used is also ***random()***.
- By default, this function generates a random number between 0.0 and 1.0 (excluding 1.0).
- For example –

```
import random                 #module random is imported
print(random.random())        #random() function is invoked
0.7430852580883088            #a random number generated
print(random.random())
0.5287778188896328            #one more random number
```

- Importing a module creates an object.

- Using this object, one can access various functions and/or variables defined in that module. Functions are invoked using a dot operator.
- There are several other functions in the module *random* apart from the function *random()*. (Do not get confused with module name and function name. Observe the parentheses while referring a function name).
- Few are discussed hereunder:

➢ **randint():** It takes two arguments *low* and *high* and returns a random integer between these two arguments (both *low* and *high* are inclusive). For example,

```
>>>random.randint(2,20)
    14                              #integer between 2 and 20 generated
>>> random.randint(2,20) 10
```

➢ **choice():** This function takes a sequence (a *list* type in Python) of numbers as an argument and returns one of these numbers as a random number.

➢ For example,

```
>>> t=[1,2, -3, 45, 12, 7, 31, 22]    #create a list t
>>> random.choice(t)                  #t is argument to choice()
     12                               #one of the elements in t
>>> random.choice(t)
     1                                #one of the elements in t
```

Various other functions available in *random* module can be used to generate random numbers following several probability distributions like Gaussian, Triangular, Uniform, Exponential, Weibull, Normal etc.

→ Math Functions

- Python provides a rich set of mathematical functions through the module ***math***.
- To use these functions, the ***math*** module has to be imported in the code.
- Some of the important functions available in *math* are given hereunder

➢ **sqrt():** This function takes one numeric argument and finds the square root of that argument.

```
>>> math.sqrt(34)              #integer argument 5.830951894845301
>>> math.sqrt(21.5)            #floating point argument 4.636809247747852
```

- **pi:** The constant value ***pi*** can be used directly whenever we require.

```
>>>print (math.pi) 3.141592653589793
```

- **log10():** This function is used to find logarithm of the given argument, to the base 10.

```
>>> math.log10(2) 0.3010299956639812
```

- **log():** This is used to compute natural logarithm (base e) of a given number.

```
>>> math.log(2)
        0.6931471805599453
```

- **sin():** As the name suggests, it is used to find *sine* value of a given argument. Note that, the argument must be in radians (not degrees). One can convert the number of degrees into radians by multiplying pi/180 as shown below –

```
>>>math.sin(90*math.pi/180)             #sin(90) is 1
   1.0
```

- **cos():** Used to find *cosine* value –

```
>>>math.cos(45*math.pi/180)
    0.7071067811865476
```

- **tan():** Function to find tangent of an angle, given as argument.

```
>>> math.tan(45*math.pi/180) 0.9999999999999999
```

- **pow():** This function takes two arguments x and y, then finds x to the power of y.

```
>>> math.pow(3,4)
     81.0
```

→ **Adding New Functions (User-defined Functions)**

- Python facilitates programmer to define his/her own functions.
- The function written once can be used wherever and whenever required.

- The syntax of user-defined function would be –

```
def fname(arg_list):
        statement_1
        statement_2
        ………..
        …
        Statement
        _n return
        value
```

➢ Here

def	is a keyword indicating it as a function definition.
fname	is any valid name given to the function
arg_list	is list of arguments taken by a function. These are treated as inputs to the function from the position of function call. There may be zero or more arguments to a function.
statements	are the list of instructions to perform required task.
return	is a keyword used to return the output *value*. This statement is optional

- The first line in the function ***def*** fname(arg_list)is known as ***function header/definition***. The remaining lines constitute ***function body***.
- The function header is terminated by a colon and the function body must be indented.
- To come out of the function, indentation must be terminated.
- Unlike few other programming languages like C, C++ etc, there is no *main()* function or specific location where a user-defined function has to be called.
- The programmer has to invoke (call) the function wherever required.
- Consider a simple example of user-defined function –

```
def myfun():
    print("Hello")
    print("Inside the function")


print("Example of function")
myfun()
print("Example over")
```

Observe indentation, Statements outside the function without indentation. myfun() is called here.

- The output of above program would be –

```
Example of function
 Hello
Inside the function
Example over
```

- The function definition creates an object of type *function*.
- In the above example, myfun is internally an object.
- This can be verified by using the statement –

```
>>>print(myfun)                         # myfun without parenthesis
        <function myfun at 0x0219BFA8>
>>> type(myfun)                         # myfun without parenthesis
        <class 'function'>
```

- Here, the first output indicates that myfun is an object which is being stored at the memory address 0x0219BFA8 (0x indicates octal number).
- The second output clearly shows myfun is of type function.

(**NOTE:** In fact, in Python every type is in the form of class. Hence, when we apply *type* on any variable/object, it displays respective class name. The detailed study of classes will be done in Module 4.)

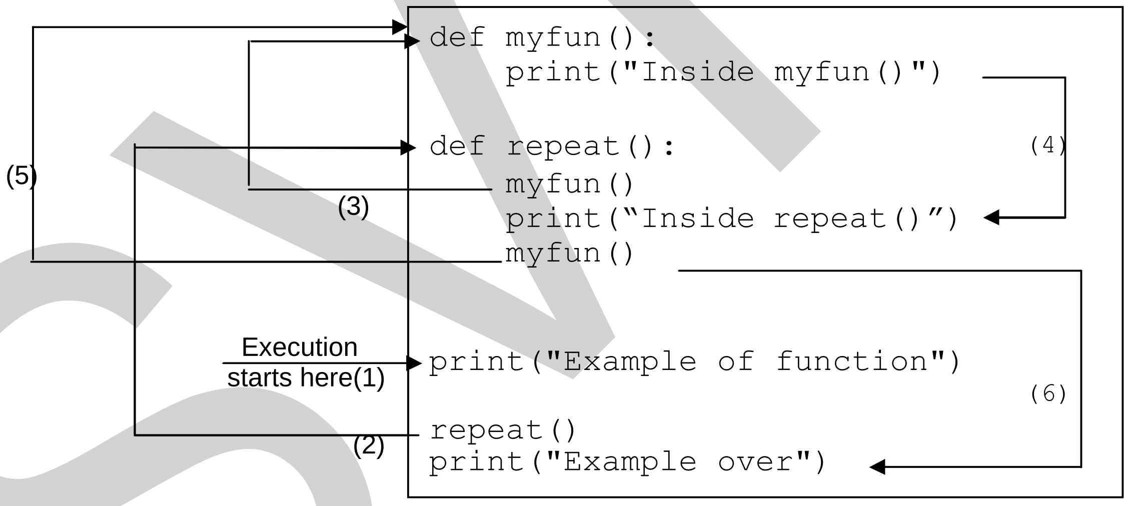
- The *flow of execution* of every program is sequential from top to bottom, a function can be invoked only after defining it.
- Usage of function name before its definition will generate error. Observe the following code:

```
print("Example of function")
myfun()                                    #function call before definition
 print("Example over")

def myfun():                               #function definition is here
print("Hello")
        print("Inside the function")
```

- The above code would generate error saying

  NameError: name 'myfun' is not defined
- Functions are meant for code-reusability. That is, a set of instructions written as a function need not be repeated. Instead, they can be called multiple times whenever required.
- Consider the enhanced version of previous program as below –



```
def myfun():
    print("Inside myfun()")

def repeat():
    myfun()
    print("Inside repeat()")
    myfun()


print("Example of function")

repeat()
print("Example over")
```

The output is –

```
Example of function
 Inside myfun()
 Inside repeat()
Inside myfun()
Example over
```

- Observe the output of the program to understand the flow of execution of the program.
- Initially, we have two function definitions myfun()and repeat()one after the other. But, functions are not executed unless they are called (or invoked). Hence, the first line to execute in the above program is –

  print("Example of function")
- Then, there is a function call repeat(). So, the program control jumps to this function. Inside repeat(), there is a call for myfun().
- Now, program control jumps to myfun()and executes the statements inside and returns back to repeat() function. The statement print("Inside repeat()") is executed.
- Once again there is a call for myfun()function and hence, program control jumps there. The function myfun() is executed and returns to repeat().
- As there are no more statements in repeat(), the control returns to the original position of its call. Now there is a statement print("Example over")to execute, and program is terminated.

➢ Parameters and Arguments

- In the previous section, we have seen simple example of a user-defined function, where the function was without any argument.
- But, a function may take arguments as an input from the calling function.
- Consider an example of a function which takes a single argument as below –

```
def test(var):
    print("Inside test()")
    print("Argument is ",var)


print("Example of function with arguments")
x="hello"
test(x)
y=20
test(y)
print("Over!!")
```

➢ The output would be –

```
Example of function with arguments
 Inside test()
Argument is
hello
Inside test()
Argument is 20
 Over!!
```

- In the above program, var is called as *parameter* and x and y are called as *arguments*.
- The argument is being passed when a function test() is invoked. The parameter receives the argument as an input and statements inside the function are executed.
- As Python variables are not of specific data types in general, one can pass any type of value to the function as an argument.
- Python has a special feature of applying multiplication operation on arguments while passing them to a function. Consider the modified version of above program –

```
def test(var):
    print("Inside test()")
    print("Argument is ",var)


print("Example of function with arguments")
x="hello"
test(x*3)
y=20
test(y*3)
print("Over!!")
```

➢ The output would be –

```
Example of function with arguments
 Inside test()
Argument is hellohellohello                    #observe repetition
Inside test()
Argument is 60                                 #observe multiplication
 Over!!
```

- One can observe that, when the argument is of type *string*, then multiplication indicates that string is repeated 3 times.

- Whereas, when the argument is of numeric type (here, integer), then the value of that argument is literally multiplied by 3.

➢ Fruitful Functions and void Functions

- A function that performs some task, but do not return any value to the calling function is known as ***void function***. The examples of user-defined functions considered till now are void functions.
- The function which returns some result to the calling function after performing a task is known as ***fruitful function.*** The built-in functions like mathematical functions, random number generating functions etc. that have been considered earlier are examples for fruitful functions.
- One can write a user-defined function so as to return a value to the calling function as shown in the following example –

```
def sum(a,b):
    return a+b

x=int(input("Enter a number:"))
y=int(input("Enter another number:"))

s=sum(x,y)
print("Sum of two numbers:",s)
```

➢ The sample output would be –

```
Enter a number:3
Enter another number:4
Sum of two numbers: 7
```

- In the above example, The function sum() take two arguments and returns their sum to the receiving variable s.
- When a function returns something and if it is not received using a LHS variable, then the return value will not be available.
- For instance, in the above example if we just use the statement sum(x,y) instead of s=sum(x,y), then the value returned from the function is of no use.
- On the other hand, if we use a variable at LHS while calling void functions, it will receive None. For example,

```
p= test(var)        #function used in previous example
print(p)
```

- Now, the value of p would be printed as None. Note that, None is not a string, instead it is of type class 'NoneType'. This type of object indicates *no value*.

➢ Why Functions?

Functions are essential part of programming because of following reasons –

- ➢ Creating a new function gives the programmer an opportunity to name a group of statements, which makes the program easier to read, understand, and debug.
- ➢ Functions can make a program smaller by eliminating repetitive code. If any modification is required, it can be done only at one place.
- ➢ Dividing a long program into functions allows the programmer to debug the independent functions separately and then combine all functions to get the solution of original problem.
- ➢ Well-designed functions are often useful for many programs. The functions written once for a specific purpose can be re-used in any other program.

➢ Local and Global Scope

- Parameters and variables that are assigned in a called function are said to exist in that function's local scope.
- Variables that are assigned outside all functions are said to exist in the global scope.
- A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable.
- A variable must be one or the other; it cannot be both local and global.
- Scope - container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten.
- The global scope is created when the program begins. When the program terminates, the global scope is destroyed, and all its variables are forgotten.
- A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten.

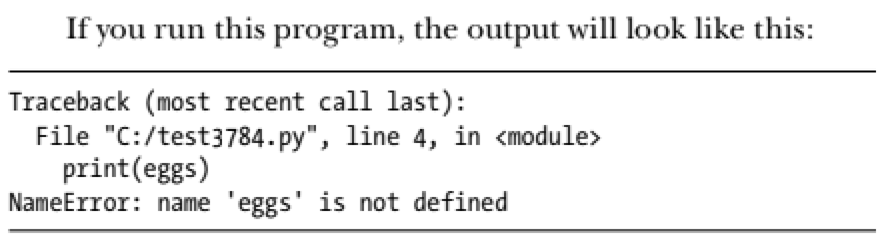
- Points to be remembered are
    - Local Variables Cannot Be Used in the Global Scope
    - Local Scopes Cannot Use Variables in Other Local Scopes
    - Global Variables Can Be Read from a Local Scope
    - Local and Global Variables with the Same Name

→ Local Variables Cannot Be Used in the Global Scope

- Consider the below program

```
def spam():
        eggs = 31337  #local variable


spam()
print(eggs)  # local variable cannot be used in global scope
```



If you run this program, the output will look like this:

```
Traceback (most recent call last):
  File "C:/test3784.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

- The error happens because the eggs variable exists only in the local scope created when spam() is called.
- Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs.
- So when program tries to run print(eggs), Python gives an error saying that eggs is not defined.
- Because, when the program execution is in the global scope, no local scopes exist, so there can't be any local variables.
- Therefore only global variables can be used in the global scope.

➢ Local Scopes Cannot Use Variables in Other Local Scopes

- A new local scope is created whenever a function is called, including when a function is called from another function.
- Consider this program:

```
def spam():
        eggs = 99
        bacon()
        print(eggs)
def bacon():
        ham = 101
        eggs = 0
spam()
```

- When the program starts, the spam() function is called, and a local scope is created.
- The local variable eggs is set to 99. Then the bacon() function is called, and a second local scope is created.
- Multiple local scopes can exist at the same time. In this new local scope, the local variable ham is set to 101, and a local variable eggs—which is different from the one in spam()'s local scope—is also created and set to 0.
- When bacon() returns, the local scope for that call is destroyed.
- The program execution continues in the spam() function to print the value of eggs ,and since the local scope for the call to spam() still exists here, the eggs variable is set to 99. The program prints 99.
- The upshot is that local variables in one function are completely separate from the local variables in another function.

➢ Global Variables Can Be Read from a Local Scope

- Consider the following program:

```
def spam():
        print(eggs)
eggs = 42   # global variable
spam()
print(eggs)
```

- Since there is no parameter named eggs or any code that assigns eggs a value in the spam() function, when eggs is used in spam(), Python considers it a reference to the global variable eggs.
- This is why 42 is printed.

➢ Local and Global Variables with the Same Name

- Consider the following program:

```
def spam():
        eggs = 'spam local'
        print(eggs) # local variable in spam
def bacon():
        eggs = 'bacon local'
        print(eggs) # local varible in bacon
        spam()
        print(eggs) # local variablein bacon
eggs = 'global'
bacon()
print(eggs) # global variable
```

When you run this program, it outputs the following:

```
bacon local
spam local
bacon local
global
```

- There are actually three different variables in this program. The variables are as follows:
  - ✓ A variable named eggs that exists in a local scope when spam() is called.
  - ✓ A variable named eggs that exists in a local scope when bacon() is called.
  - ✓ A variable named eggs that exists in the global scope.

➢ Global statement

- If we need to modify a global variable from within a function, we can use the global statement.
- Example;

```
def spam():
        global eggs
        eggs = 'spam'
eggs = 'global'
spam()
print(eggs)
```

- Here, we have a statement as global eggs at the top of a function, it tells Python that, “In this function, eggs refers to the global variable, so don’t create a local variable with this name.”
- Output will be **spam**

➢ Exception Handling

- There is a chance of runtime error while doing some program.
- One of the possible reasons is wrong input.
- For example, consider the following code segment –

```
a=int(input("Enter a:"))
b=int(input("Enter b:"))
c=a/b
print(c)
```

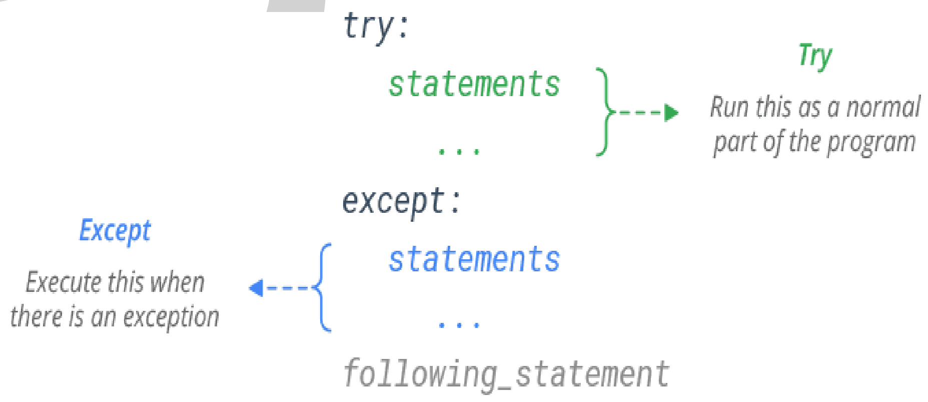
- When you run the above code, one of the possible situations would be –

```
Enter a:12
Enter b:0
Traceback (most recent call last):
    c=a/b
ZeroDivisionError: division by zero
```

- For the end-user, such type of system-generated error messages is difficult to handle.
- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception.
- An exception is a Python object that represents an runtime error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

→ Exception handling

- Exceptions can be handled with try and except statements.
- The code that could potentially raise an exception is put in a try clause. The program execution moves to the start of the following except clause if an exception happens.
- The ***try*** block contains the statements involving suspicious code and the ***except*** block contains the possible remedy (or instructions to user informing what went wrong and what could be the way to get out of it).
- If something goes wrong with the statements inside ***try*** block, the ***except*** block will be executed.
- Otherwise, the except-block will be skipped.



```
try:
    statements
       ...
except:
    statements
       ...
following_statement
```

Try
Run this as a normal part of the program
Except
Execute this when there is an exception

- Consider the example –

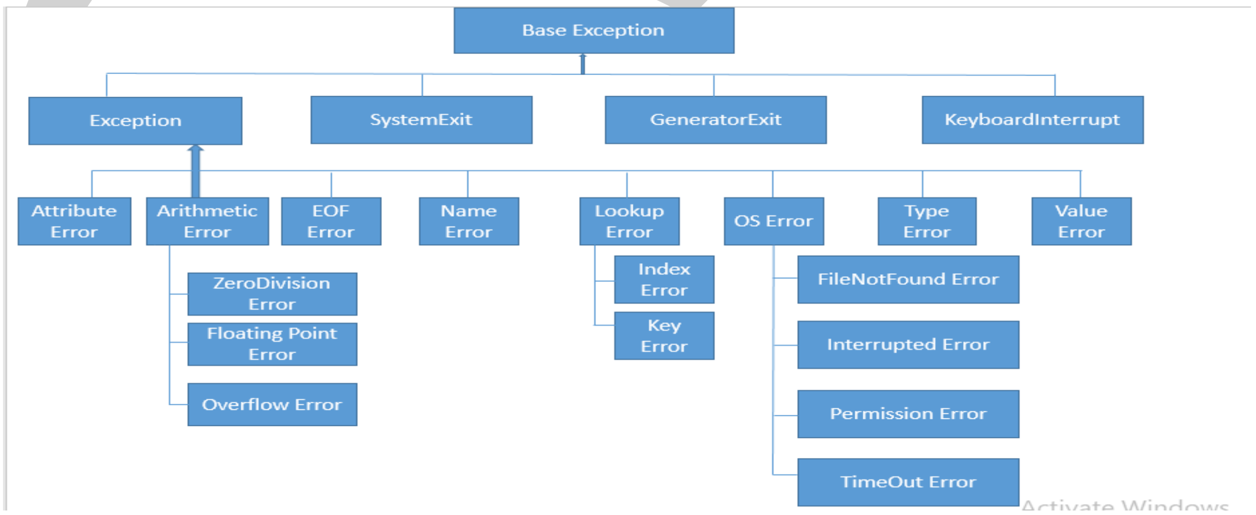
```
a=int(input("Enter a:"))
b=int(input("Enter b:"))
 try:
        c=/b
        print(c)
except:
        print("Division by zero is not possible")
print("Program Ends")
```

Output:

```
Enter a:12
Enter b:0
Division by zero is not possible
Program Ends
```

- Handling an exception using *try* is called as ***catching*** an exception.
- In general, catching an exception gives the programmer to fix the probable problem, or to try again or at least to end the program gracefully or normally.

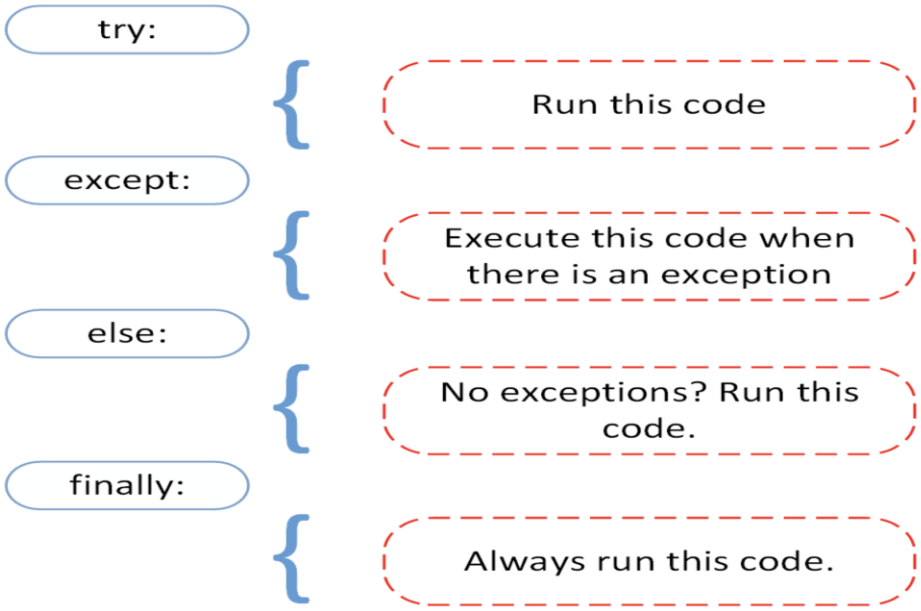
→ Types of Exceptions



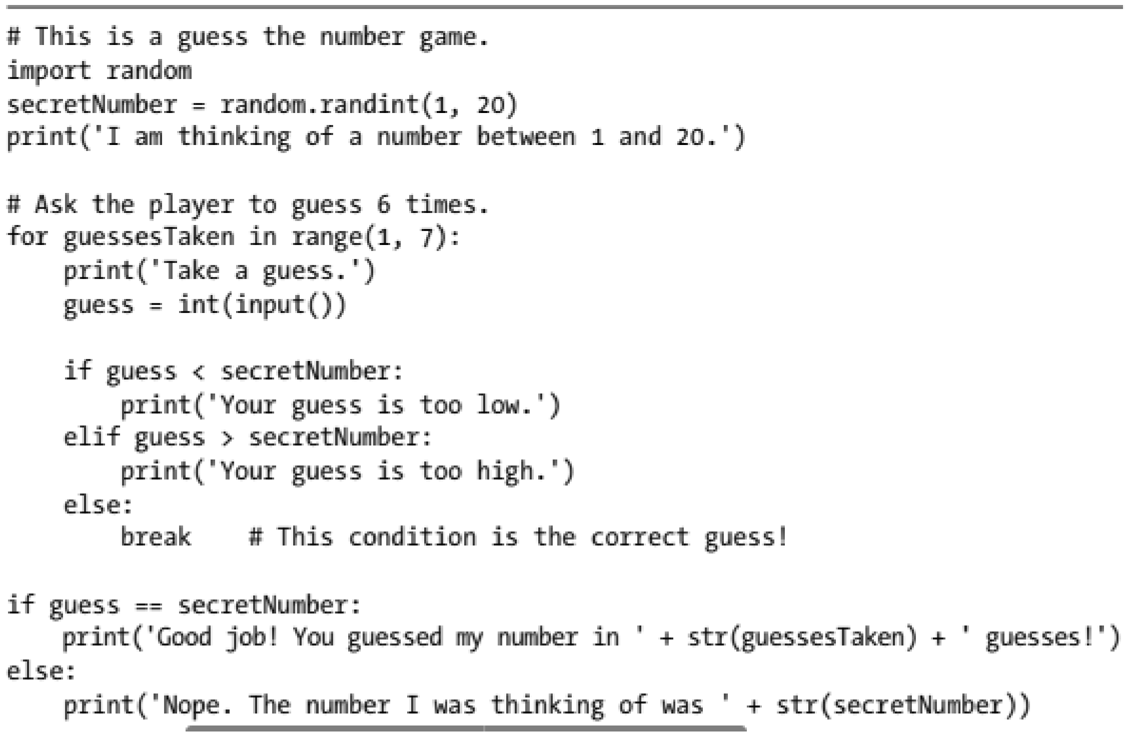
Standard Exceptions in Python

Sl No	Name	Purpose
1.	Exception	Base class for all exceptions
2.	ArithmeticError	Base class for all errors that occur for numeric calculations
3.	OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
4.	FloatingPointError	Raised when a floating point calculation fails
5.	ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
6.	EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
7.	ImportError	Raised when an import statement fails.
8.	KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+C
9.	NameError	Raised when an identifier is not found in the local or global namespace.
10.	IOError	Raised when an input/output operation fails.
11.	SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the python interpreter does not exit.
12.	SystemExit	Raised when python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
13.	TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
14.	ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
15.	RuntimeError	Raised when a generated error does not fall into any category.

Note: There are other two Exception Handling Keywords:


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

➢ A Short Program: Guess the Number



```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())

    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break    # This condition is the correct guess!

if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

Special parameters of print() – *sep* and *end* :

Consider an example of printing two values using *print()* as below –

```
>>> x=10
>>> y=20
>>> print(x,y)
10 20                    #space is added between two values
```

Observe that the two values are separated by a space without mentioning anything specific. This is possible because of the existence of an argument ***sep*** in the *print()* function whose default value is white space. This argument makes sure that various values to be printed are separated by a space for a better representation of output.

The programmer has a liberty in Python to give any other character(or string) as a separator by explicitly mentioning it in *print()* as shown below –

```
>>> print("18","2","2018",sep='-') 18-2-
        2018
```

We can observe that the values have been separated by hyphen, which is given as a value for the argument *sep*. Consider one more example –

```
>>> college="SVIT"
>>> address=”BANGALORE"
>>> print(college, address, sep='@')
              SVIT@BANGALORE
```

If you want to deliberately suppress any separator, then the value of *sep* can be set with empty string as shown below –

```
>>> print("Hello","World", sep='')
               HelloWorld
```

You might have observed that in Python program, the *print()* adds a new line after printing the data. In a Python script file, if you have two statements like –

```
        print(“Hello”)
        print(“World”)
```

then, the output would be

```
        H
        e
        l
        l
```

```
o

W
o
r
l
d
```

This may be quite unusual for those who have experienced programming languages like C, C++ etc. In these languages, one has to specifically insert a new-line character (\n) to get the output in different lines. But, in Python without programmer's intervention, a new line will be inserted. This is possible because, the *print()* function in Python has one more special argument ***end*** whose default value itself is new-line. Again, the default value of this argument can be changed by the programmer as shown below (Run these lines using a script file, but not in the terminal/command prompt) –

```
print("Hello", end= '@')
print("World")
```

The output would be –

```
Hello@World
```

Formatting the output:

There are various ways of formatting the output and displaying the variables with a required number of space-width in Python. We will discuss few of them with the help of examples.

- **Ex1:** When multiple variables have to be displayed embedded within a string, the *format()* function is useful as shown below –

```
>>> x=10
>>> y=20
>>> print("x={0}, y={1}".format(x,y))
    x=10, y=20
```

While using *format()* the arguments of *print()* must be numbered as 0, 1, 2, 3, etc. and they must be provided inside the *format()* in the same order.

- **Ex2:** The *format()* function can be used to specify the width of the variable (the number of spaces that the variable should occupy in the output) as well. Consider below given example which displays a number, its square and its cube.

```
for x in range(1,5):
        print("{0:1d} {1:3d} {2:4d}".format(x,x**2, x**3))
```

Output:

```
    1    1    1
    2    4    8
    3    9   27
    4   16   64
```

Here, 1d, 3d and 4d indicates 1-digit space, 2-digit space etc. on the output screen.

- **Ex3:** One can use % symbol to have required number of spaces for a variable. This will be useful in printing floating point numbers.

```
>>> x=19/3
>>> print(x)
6.333333333333333                    #observe number of digits after dot
>>> print("%.3f"%(x))                #only 3 places after decimal point 6.333

>>> x=20/3
>>> y=13/7
>>> print("x= ",x, "y=",y)                     #observe actual digits
        x=6.666666666666667            y= 1.8571428571428572
>>> print("x=%0.4f, y=%0.2f"%(x,y))
       x=6.6667, y=1.86                        #observe rounding off digits
```