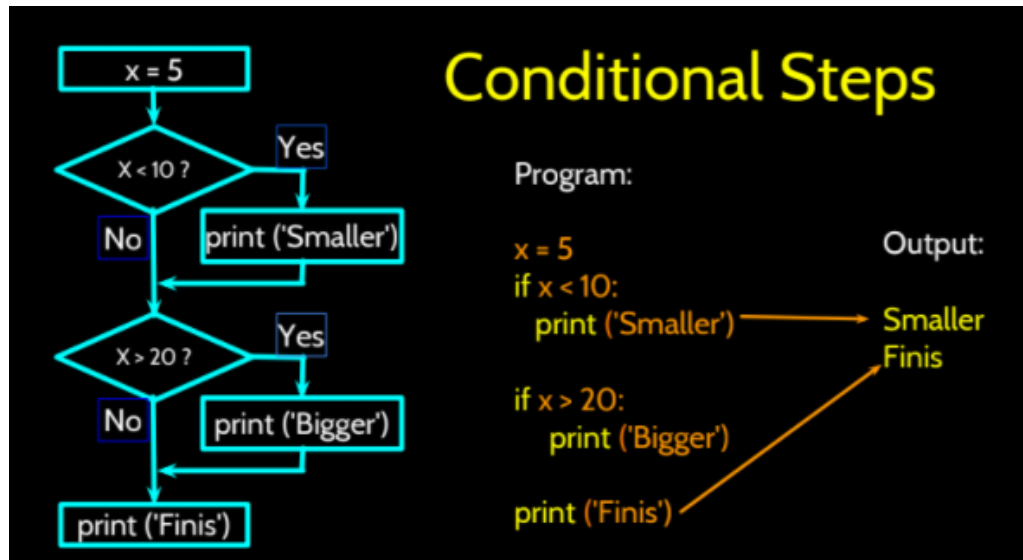


Unit III

Iteration and Recursion: Conditional execution, Alternative execution, Nested conditionals, The return statement, Recursion, Stack diagrams for recursive functions, Multiple assignment, The while statement, Implementing 2-D matrices.



In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement.

The boolean expression after if is called the **condition**. If it is true, then the indented statement gets executed. If not, nothing happens.

4.4. Alternative execution

A second form of the `if` statement is *alternative execution*, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

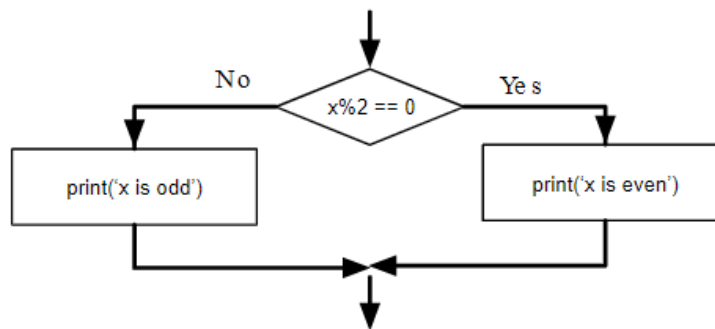
```

→ 1 x = 8
   2 if x % 2 == 0 :
   3     print('x is even')
   4 else :
   5     print('x is odd')
   6 print('All done.')

```

Print output (drag lower right corner to resize)

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.



Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called *branches*, because they are branches in the flow of execution.

A second form of the `if` statement is **alternative execution**, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0:
```

```
    print 'x is even'
```

```
else:
```

```
    print 'x is odd'
```

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution.

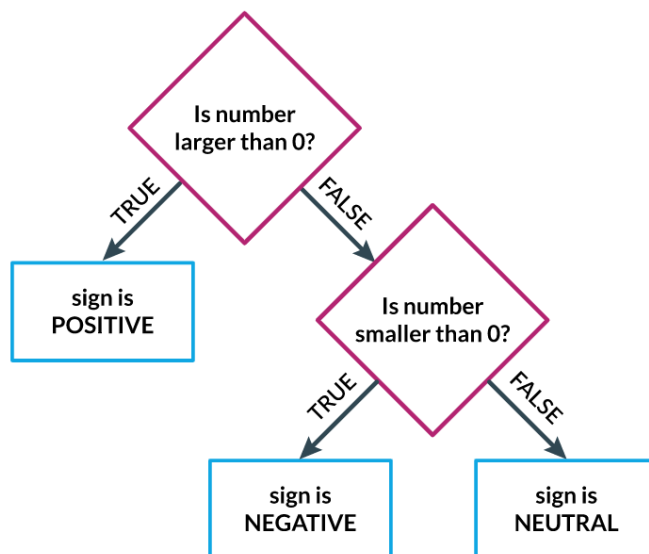
Nested conditionals

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements.

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct. Imagine a program that reports whether a number is positive, negative, or zero. That program needs to select from 3 paths.

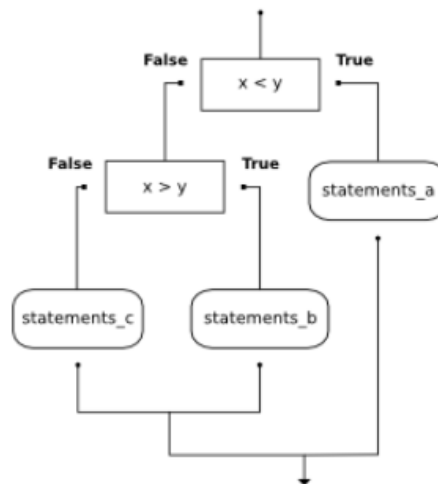


One conditional can also be **nested** within another. For example, assume we have two integer variables, `x` and `y`. The following pattern of selection shows how we might decide how they are related to each other.

```
if x < y:
    print("x is less than y")
else:
    if x > y:
        print("x is greater than y")
    else:
        print("x and y must be equal")
```

The outer conditional contains two branches. The second branch (the else from the outer) contains another `if` statement, which has two branches of its own. Those two branches could contain conditional statements as well.

The flow of control for this example can be seen in this flowchart illustration.



Here is a complete program that defines values for `x` and `y`. Run the program and see the result. Then change the values of the variables to change the flow of control.

Return statement

A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value `None` is returned.

<pre># Python program to demonstrate return statement def add(a, b): # returning sum of a and b return a + b # calling function res = add(2, 3) print(res)</pre>	<p>Output</p> <p>5</p>
--	--------------------------------------

Recursion in Python

The term Recursion can be defined as the process of defining something in terms of itself. In simple words, it is a process in which a function calls itself directly or indirectly.

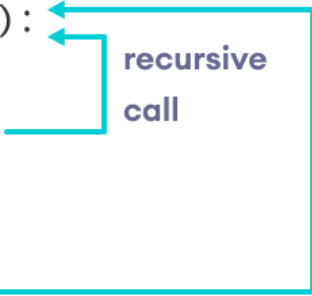
Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

```
def recurse():  
    ...  
    recurse()  
    ...  
  
recurse()
```



```
def facto(n):  
    if n == 1:  
        return n  
    else:  
        return n*facto(n-1)  
# take input from the user  
num = int(input("Enter a number: "))  
# check is the number is negative  
print(facto(num))
```

Stack Diagrams for Recursive Functions

stack diagram to represent the state of a program during a function call. The same kind of diagram can make it easier to interpret a recursive function.

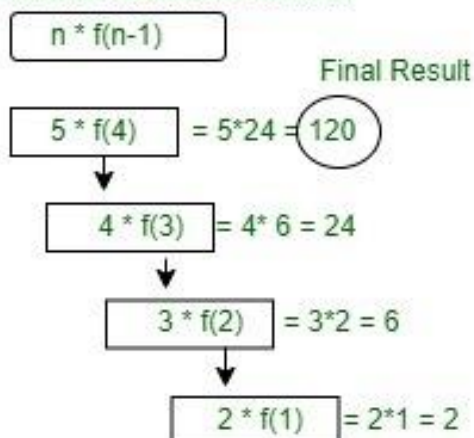
Remember that every time a function gets called it creates a new instance that contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

This figure shows a stack diagram for `countdown`, called with `n = 3`:

	main
n: 3	countdown
n: 2	countdown
n: 1	countdown
n: 0	countdown

For user input : 5

Factorial Recursion Function



Multiple Assignment Statement

The basic assignment statement does more than assign the result of a single expression to a single variable. The assignment statement also copes nicely with assigning multiple variables at one time. The left and right side must have the same number of elements. For example, the following script has several examples of multiple assignment.

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

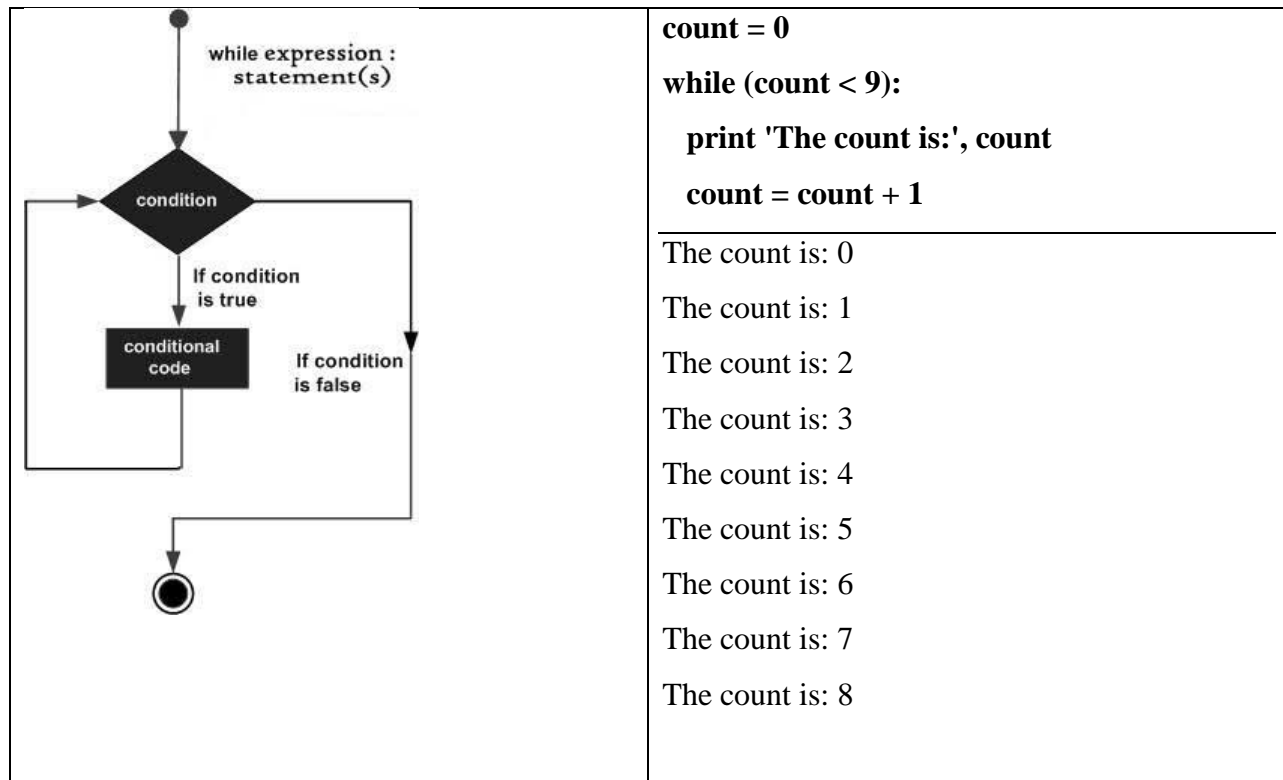
```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location.

<pre>x, y, z = "Orange", "Banana", "Cherry"</pre>	<pre>x = y = z = "Orange"</pre>
<pre>print(x)</pre>	<pre>print(x)</pre>
<pre>print(y)</pre>	<pre>print(y)</pre>
<pre>print(z)</pre>	<pre>print(z)</pre>

The while statement

In Python, **While Loops** is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed. While loop falls under the category of **indefinite iteration**. Indefinite iteration means that the number of times the loop is executed isn't specified explicitly in advance.



2- D Array :

Two dimensional array is an array within an array. It is an array of arrays. In this type of array the position of an data element is referred by two indices instead of one. So it represents a table with rows and columns of data. In the below example of a two dimensional array, observe that each array element itself is also an array. A **2D array** is an array of arrays that can be represented in matrix form like rows and columns. In this array, the position of data elements is defined with two indices instead of a single index.

```

Student_dt = [ [72, 85, 87, 90, 69], [80, 87, 65, 89, 85], [96, 91, 70, 78, 97], [90, 93, 91, 90, 94], [57, 89, 82,
#print(student_dt[])
print(Student_dt[1]) # print all elements of index 1
print(Student_dt[0]) # print all elements of index 0
print(Student_dt[2]) # print all elements of index 2
print(Student_dt[3][4]) # it defines the 3rd index and 4 position of the data element.
  
```

Output:

```

[80, 87, 65, 89, 85]
[72, 85, 87, 90, 69]
[96, 91, 70, 78, 97]
94
  
```