Decision Making Statement, Loop & functions in Python

Created by :- The Easylearn Academy
9662512857

introduction

- There comes situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next.
- Similar situations arises in programming also where we need to make some decisions and based on these decision we will execute the next block of code.
- Decision making statements in programming languages decides the direction of flow of program execution.
- Decision structures evaluate multiple expressions which produce True or False as outcome.
- You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.
- **Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value.**
- Decision making statements available in python are:

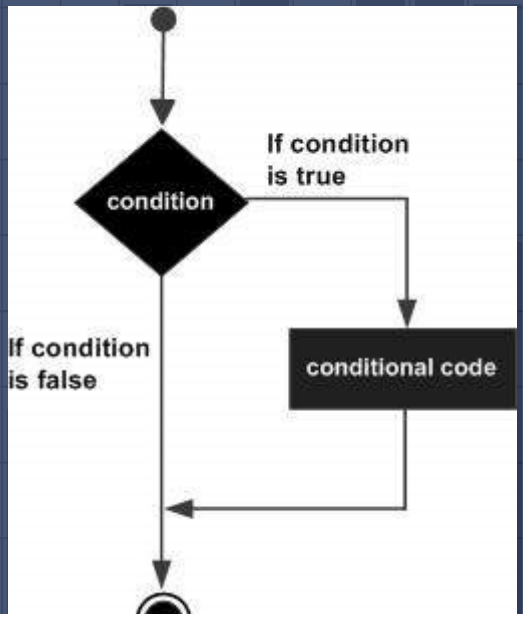
1. if statement
2. if..else statements
3. nested if statements
4. if-elif else ladder

- **Python does not support switch decision making statement at all**

If statement

- if statement is the most simple decision making statement.
- It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.
- If the Boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed.
- If Boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

```
if condition:
    # Statements to execute if condition is true
#statements to be executed after if block
```



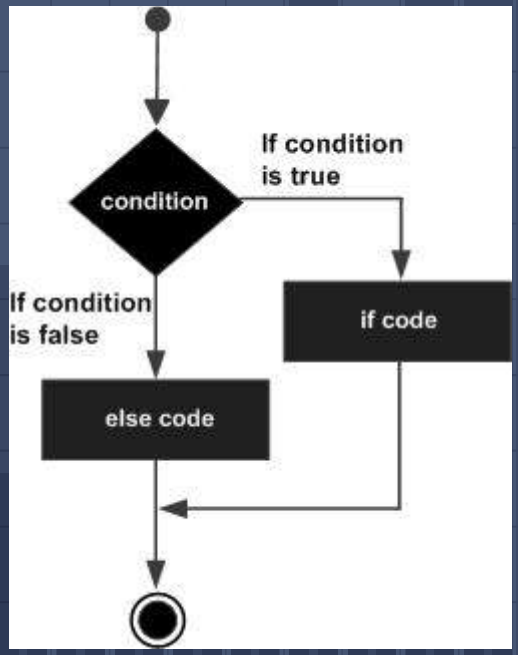
Example of if statement

```
a = int ( input ( "Enter a number: " ) )
b = int ( input ( "Enter another number: " ) )
if a>b: # < > <= >= == !=/<>
   print ( "First number is greater than second number " )
Print(“Good Bye”)
```

if- else statement

- The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false.
- Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.
- In if else statement ,if statement is followed by an optional else statement & if the expression results in FALSE, then else statement gets executed.

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



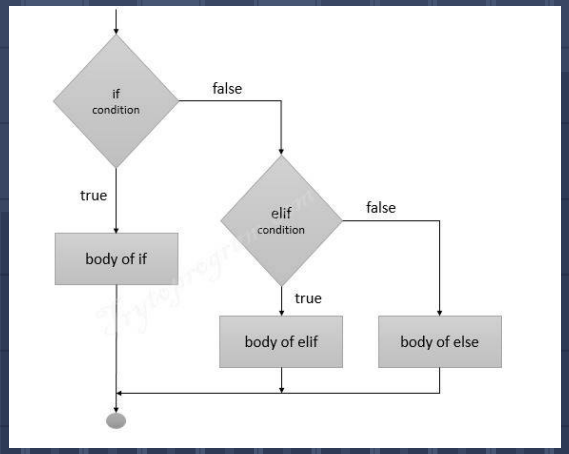
example

```
a = int(input(“enter first number”))
b = int(input(“enter second number”))

if a > b:
   print(“first number is bigger")
   print(“and first number is “,a)
else:
   print(“second number is bigger")
   print(“and second number is “,b)
```

if-elif-else ladder

- The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code whose conditions evaluates to TRUE as well as skip checking remaining conditions.
- It is specially used to check more then one condition at same level using any relational operations like < > <= >= == <>.
- The if statements are executed from the top down.
- the **elif** statement is optional. but there can be an arbitrary number of **elif** statements following an **if**.
- As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.
- If none of the conditions is true, then the final else statement will be executed.

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```



example

```
a = int(input(“enter first number”))
b = int(input(“enter second number”))

if a > b:
   print("a is greater")
elif a == b:
   print("both are equal")
else:
   print("b is greater")
```

nested if statement

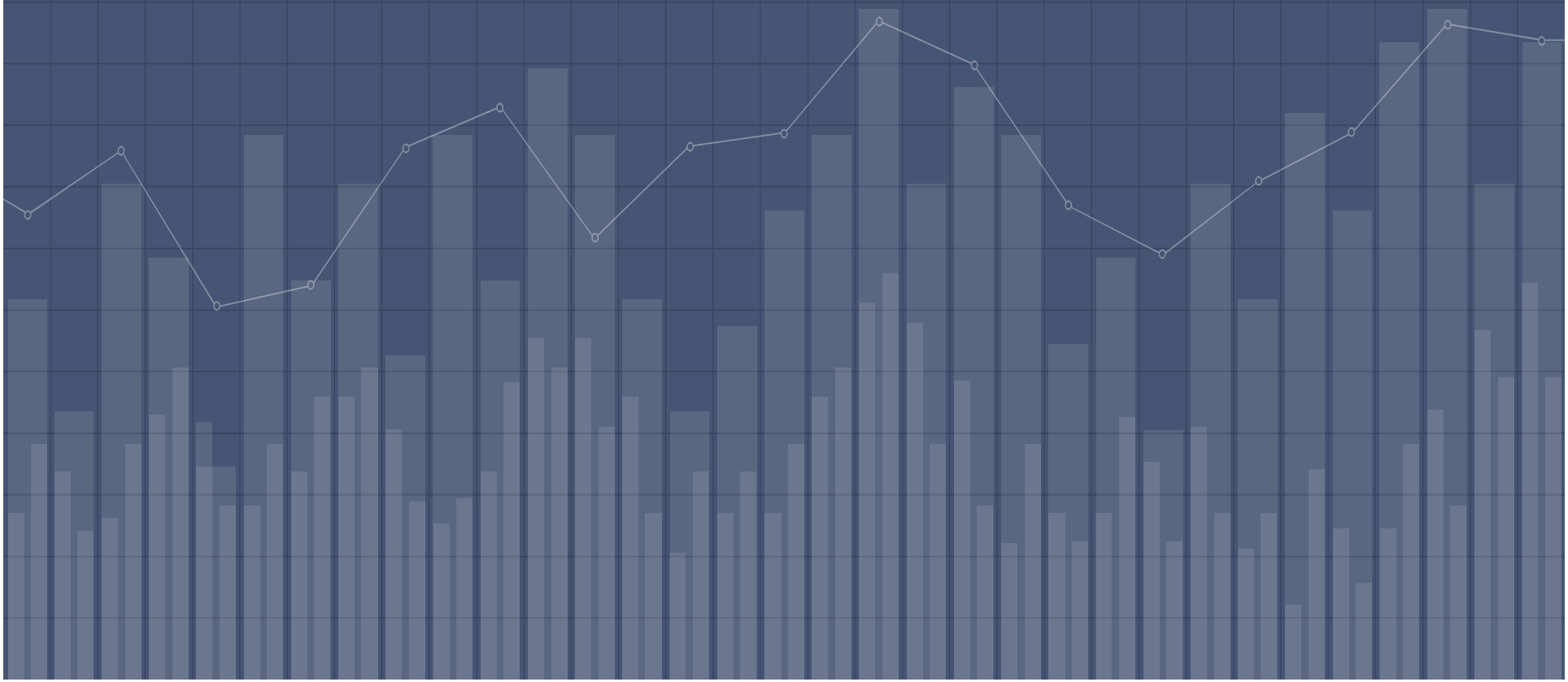
- Sometimes we need to check one or more condition inside another condition. In such a cases we need to use nested if statement in python.
- Nested if statements means an if statement inside another if statement.
- Python allows us to nest if statements within if statements. So we can place an if statement inside another if statement.

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   elif expression4:
      statement(s)
   else:
      statement(s)
else:
   statement(s)
```

example

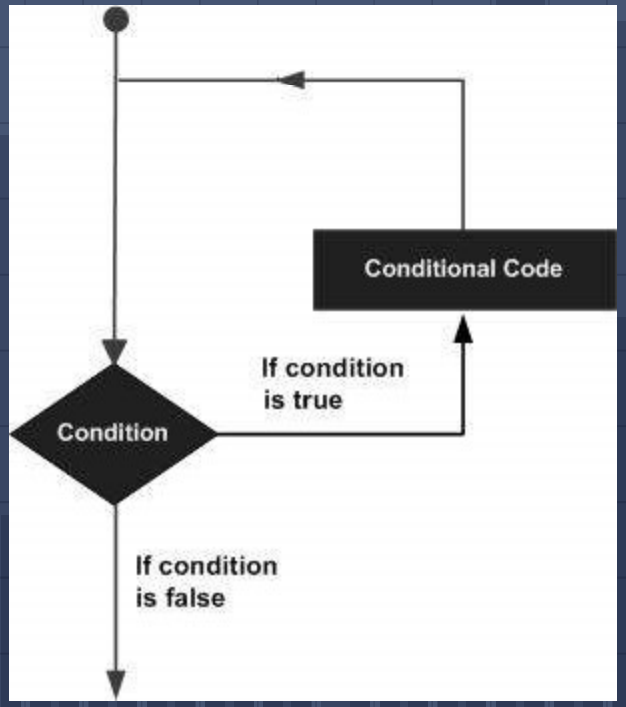
```
i = 10
if (i == 10):
    #  First if statement
    if (i < 15):
        print ("i is smaller than 15")

   if (i < 12):
        print ("i is smaller than 12 too")
    else:
        print ("i is greater than or equal to 12")
```

Loops in python



Introduction to loop

- There may be a situation when you need to execute a block of code several number of times.
- Programming languages provide various control structures that allow for more complicated execution paths.
- **A loop statement allows us to execute a statement or group of statements multiple times.**



Types of loops

- **While loop**
- Repeats a statement or group of statements while a given condition is TRUE. While loop test the condition before executing the loop body if condition is true then only it execute certain code block. It is entry control type of loop

- **For loop**
- Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. It also check condition before executing the loop body. It is entry control type of loop

- **Nested loops**
- When we use loop inside loop, it is called nested loop. It means we can use one or more loop inside any another while, for or do..while loop.

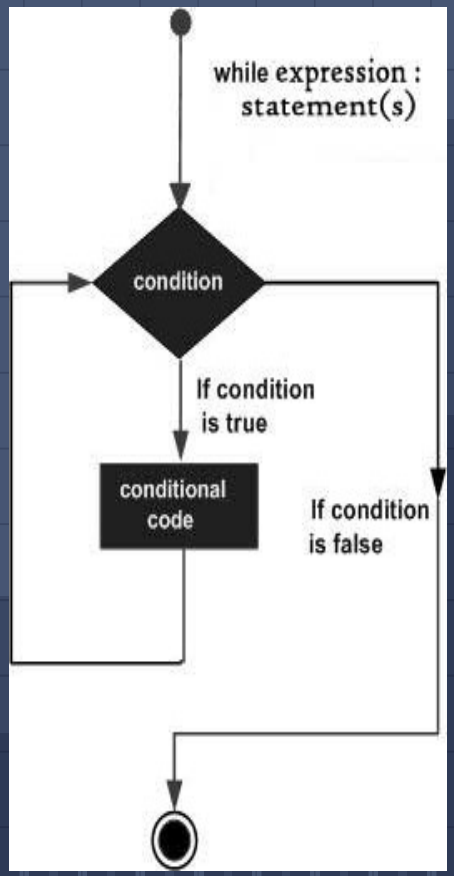
- there is no "**do** ... **while**" **loop** in **Python**. A properly constructed **while loop can do** the same. If you have any problems, give us a simplified idea of what you want to accomplish.

While loop

- A **while** loop statement in Python repeatedly executes a code block as long as a given condition is true.
- **Syntax**

```
while condition:
    statement(s)
```

- Here, **statement(s)** may be a single statement or a block of statements.
- **The condition may be any expression, and true is any non-zero value.**
- The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.
- In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code.
- Python uses indentation as its method of grouping statements.

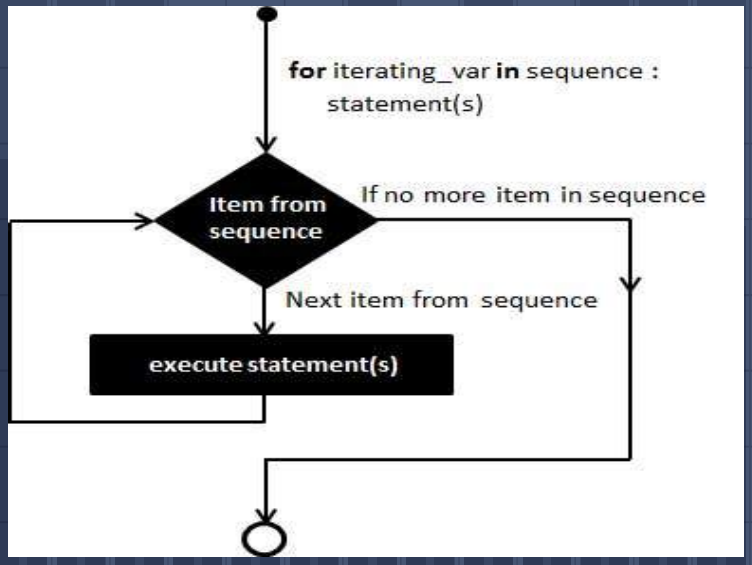


example

```
# Prints out 0,1,2,3,4

count = 0
while count < 5: # 5<5
    print(count) #2
    count=count + 1  # 3
print(“good bye”)
```

For loop

- For loop is used to iterate over the items of any sequence, such as a list or a string.
- for iterating_var in sequence:
- statements(s)
- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable. Next, the statements block is executed.
- Each item in the list is assigned to iterating variable, and the statement(s) block is executed until the entire sequence is exhausted.



example

```
# Prints out the numbers 0,1,2,3,4
for count in range(5):
    print(count)

# Prints out 3,4,5
for x in range(3, 6):
    print(x)

for letter in 'Python':
   print (letter)

fruits = ['banana', 'apple',  'mango'] #list
for item in fruits:
   print (item)
```

Example of reverse for loop

```
for x in range(6,3,-1):
    print(x,end='') ## print output in same line
print()
cars = ['three','two','one']
for c in reversed(cars):
   print(c,end='')
```

Nested loop

- When one use loop inside another loop, then it is called nested loop.
- One can do nesting of loop in this manner upto any level.
- Python allows nesting of loop like other programming languages.
- One can use for loop inside while loop or while loop inside for loop or for loop inside for loop or while loop inside while loop.

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)

while expression:
   while expression:
      statement(s)
   statement(s)
```

example

```
for i in range(1,5): #outer loop
      for j in range(1,i+1): #inner loop
        print j,
      print
```

Output

- 1
- 1 2
- 1 2 3
- 1 2 3 4

Function in python

- In Python, a function is a group of statements that performs a specific task.
- A function is named block of code which only runs when it is called.
- You can pass data, known as parameters, into a function. Function mainly process the passed data & may return some data as a result.
- One should create a function for repetitive task and should run it when required.
- Therefore functions help break our program into smaller and modular parts.
- As our program grows larger and larger, functions make it more organized and manageable.
- There are mainly two types of functions
  built-in or
  user-defined.

  It helps the program to be concise, non-repetitive, and organized.

Characteristic of functions

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- every function name ends with a colon (:).
- The statement return [expression] exits a function,
- Its optional and can be used to return value to the caller.
- A return statement with no arguments is the same as return None.
- All the code in function should be given with tab like we do in decision making and loops.

- Example
- ```
  def getsquare(number): #10 actual argument
  ```
- ```
    square = number * number
  ```
  ```
    return square
  ```
- ```
  # Now you can call myfunction function
  ```

- ```
  num = int(input("Enter number")) #10 formal argument
  ```
- ```
  answer = getsquare(num) #calling function/use function
  ```
- ```
  print(answer)
  ```

Default Argument Values

- In function's parameters list one can specify a default value(s) for one or more arguments.
- A default value can be written in the format "argument1 = value", therefore one will have the option to declare or not declare a value for those arguments. See the following example.

```
def nsquare(x, y = 2): #here x is required and y is optional

    return (x*x + 2*x*y + y*y)

print("The square of the sum of 2 and 2 is : ", nsquare(2))

print("The square of the sum of 2 and 4 is : ", nsquare(2,4))
```

Keyword Arguments:

- We have already learned how to use default arguments values, functions can also be called using keyword arguments.
- **Arguments which are preceded with a variable name followed by a '=' sign (e.g. var_name=") are called keyword arguments.**
- All the keyword arguments passed must match one of the arguments accepted by the function.
- It is used to change order of argument or to skip some argument at the time of calling function.
- See the following example.

```
def marks(english, math = 85, science = 80):
    print('Marks in : English is - ', english,', Math - ',math, ',
    Science - ',science)
Return
marks(71, 77)
marks(65, science = 74)
marks(science = 70, math = 90, english = 75)
output
Marks in : English is -  71 , Math -  77 , Science -  80
Marks in : English is -  65 , Math -  85 , Science -  74
Marks in : English is -  75 , Math -  90 , Science -  70
```

Arbitrary Argument Lists

- Sometimes, we do not know in advance the number of arguments that will be passed into a function.
- Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.
- In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument.
- Here is an example.

```
def SayHello(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello Mr/Miss/M.R.S.",name)
return

SayHello("Ram","ramesh","meena","haresh")
```

Recursion

- When function call itself repeatedly until some condition is true or false ,then it is called recursion.
- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- **Advantages of Recursion**
  - Recursive functions make the code look clean and elegant.
  - A complex task can be broken down into simpler sub-problems using recursion.
  - Sequence generation is easier with recursion than using some nested iteration.
- **Disadvantages of Recursion**
  - Sometimes the logic behind recursion is hard to follow through.
  - Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
  - Recursive functions are hard to debug.

Some important points about recursion.

1. Each function called in recursion will always complete its execution.
2. Function called in recursion will always complete in reverse order of calling.
3. If function has parameters means input or local variables then it will created for each every function call separately.

example

```
 An example of a recursive function to
# find the factorial of a number

def calc_factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))

num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

lambda functions

- In Python, anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.
- Hence, anonymous functions are also called lambda functions.
- Syntax
- Function-name = lambda input: expression
- # Program to show the use of lambda functions
- double = lambda x: x * 2
- # Output: 10
- print(double(5))
- Example
- In the above program, lambda x: x * 2 is the lambda function.
- Here *x* is the argument and x * 2 is the expression that gets evaluated and returned.
- This function has no name. It returns a function object which is assigned to the identifier double. We can now call it as a normal function. The statement
- double = lambda x: x * 2

Where and where not to use lamda?

- For that one need to understand difference between expression and statement.
- Expression returns or evaluate to value while statement return nothing.
- If it doesn't return a value, it isn't an expression and can't be put into a lambda.
- If you can imagine it in an assignment statement, on the left-hand side of the equals sign, it is an expression and can be put into a lambda.
- There are some informal rules
- Assignment statements cannot be used in lambda. In Python, assignment statements don't return anything, not even None (null).
- Simple things such as mathematical operations, string operations, list comprehensions, etc. are OK in a lambda.
- Function calls are expressions. It is OK to put a function call in a lambda, and to pass arguments to that function. Doing this wraps the function call (arguments and all) inside a new, anonymous function.
- In Python 3, *print* became a function, so in Python 3+, *print(…)* can be used in a lambda.
- Even functions that return None, like the *print* function in Python 3, can be used in a lambda.
- Conditional expressions, which were introduced in Python 2.5, are expressions (and not merely a different syntax for an *if/else* statement). They return a value, and can be used in a lambda.