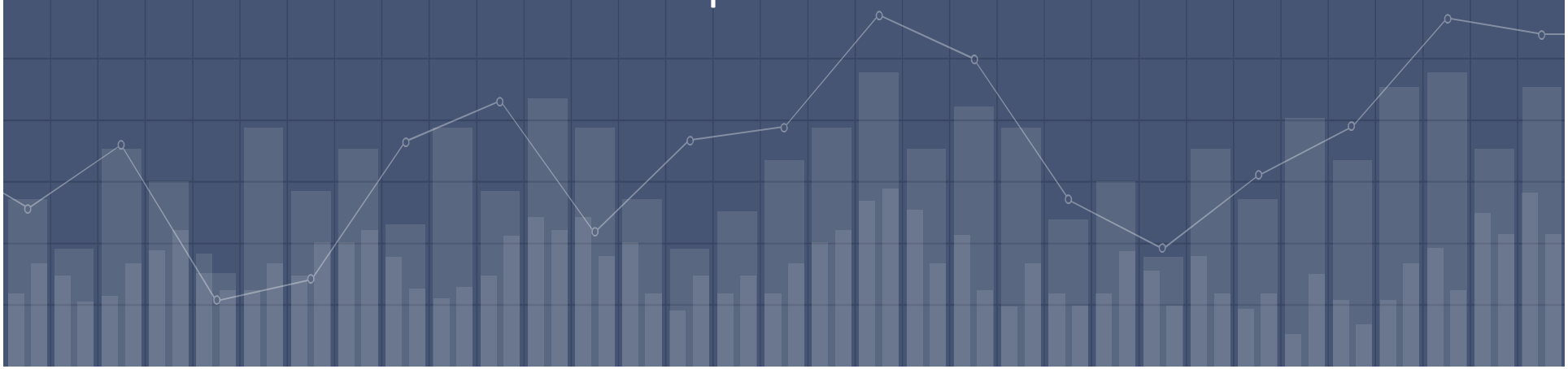
Decision Making Statement, Loop & functions in Python

Created by :- The Easylearn Academy
9662512857

Lines and Indentation

- Python provides no braces to indicate blocks of code for class and function definitions or decision making statement & loop.
- Blocks of code is denoted by line indentation, this rule must be strictly followed else one will get run time error.
- The number of spaces in the indentation is your choice, but all statements within the block must be indented the with same number of space.



Introduction to decision making

- 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:

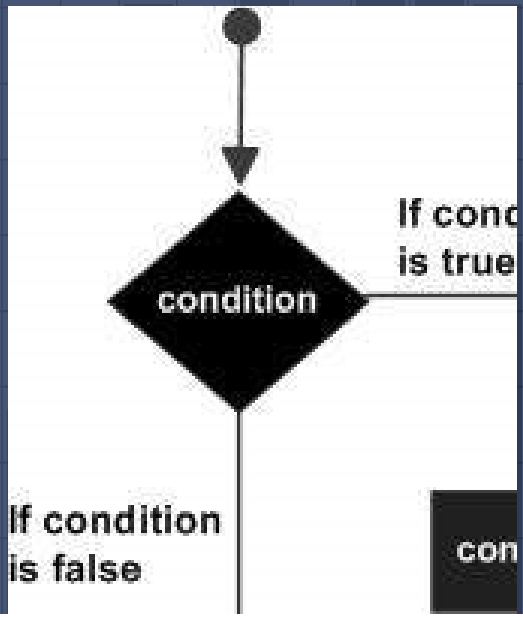
1. if statement
2. if..else statements
3. nested if statements
4. if-elif else ladder

- **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/relational 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 relational_expression(s):
    # Statements that will execute only  if relational expression is true

#statements that will always executed after if block
```



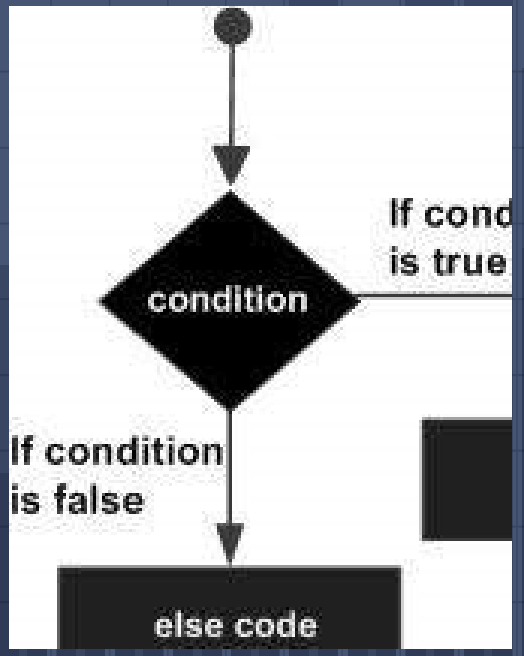
Example of if statement

```
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)
```



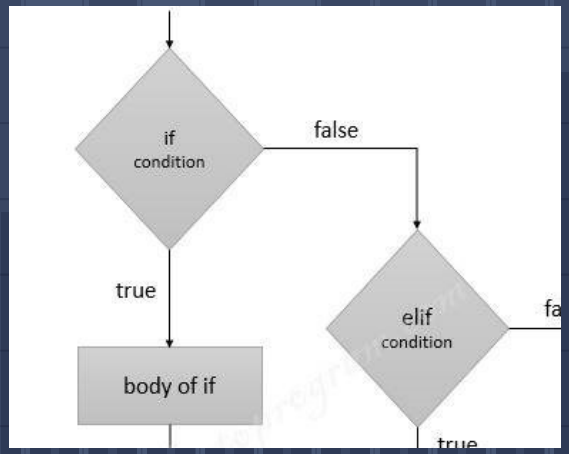
example

```
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)

print(“good bye”)
```

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 relational_expression1:
    statement(s)
elif relational_expression2:
    statement(s)
elif relational_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 decision statement

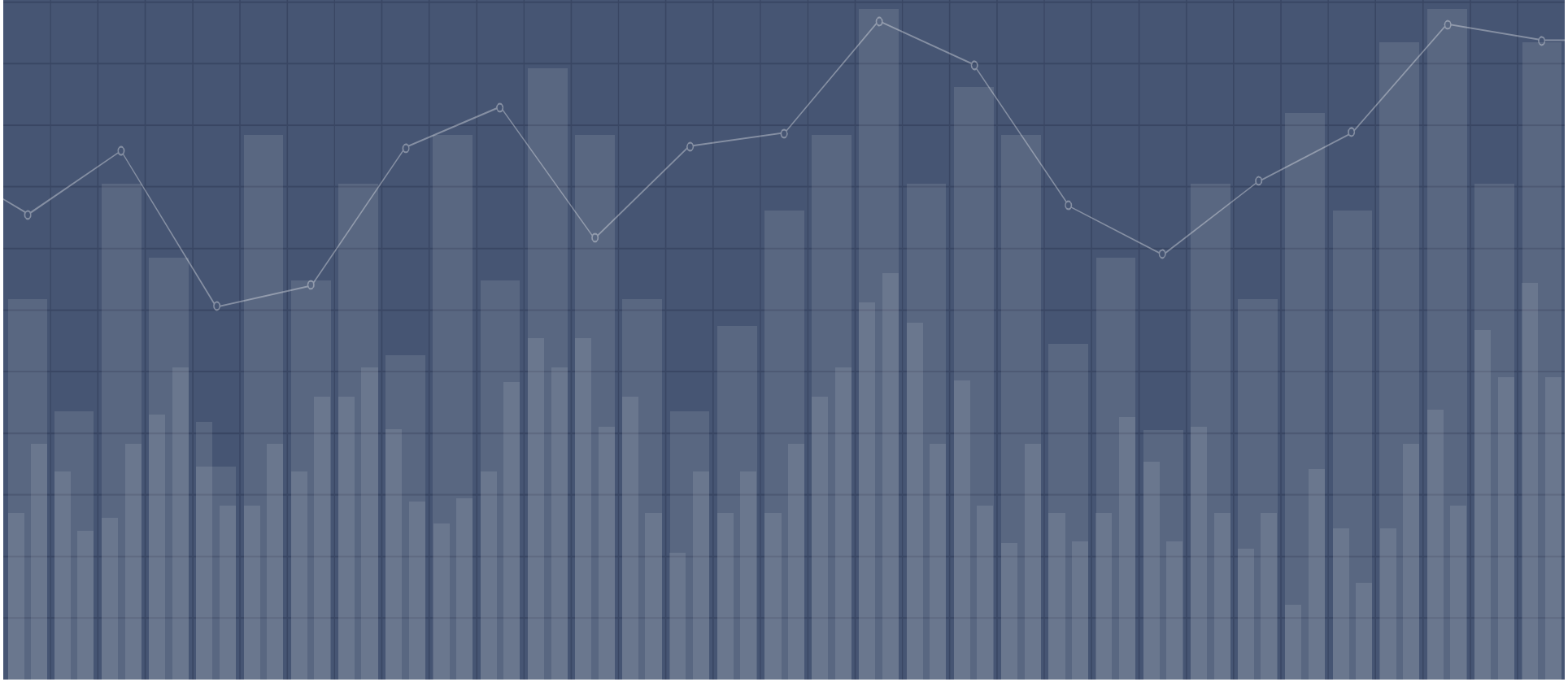
- Sometimes we need to check one or more condition inside another condition. In such a cases we need to use nested if statement in python.
- We can use any type of decision making statement inside any decision making statement.
- For example we can use if decision making statement inside if elif ladder decision making statement and vice versa.

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

example

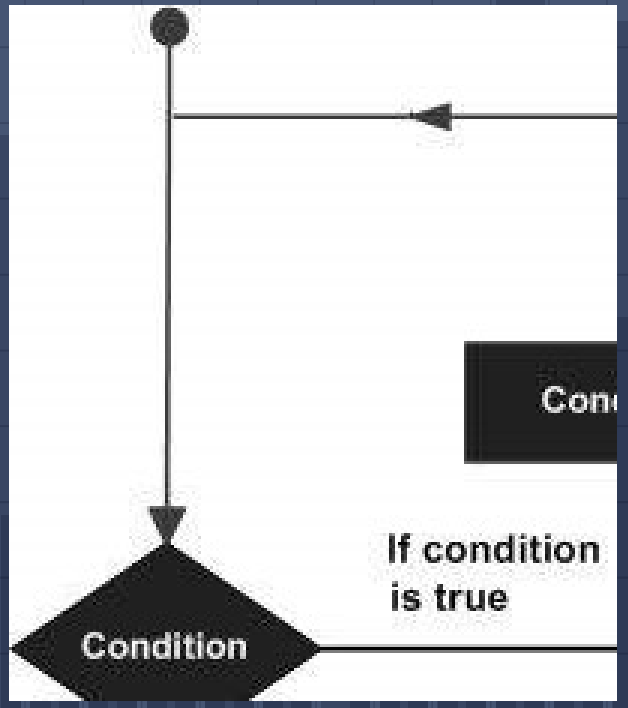
```
i = int(input(“enter any number”))
if i == 10: #outer if decision making statement
    if i < 15: #inner decision making
        print ("i is smaller than 15")

    if i < 12: #inner decision making
        print ("i is smaller than 12 too")
    else:
        print ("i is greater than or equal to 12")
else:
   print(“I is not equal to 10”)
```

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 to do such task with the help of loop.
- **A loop statement allows us to execute a code block multiple times till some condition is true.**



Types of loops

- **While loop**
- Repeats a block of code while a given condition is TRUE. While loop test the condition before executing the code block 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.

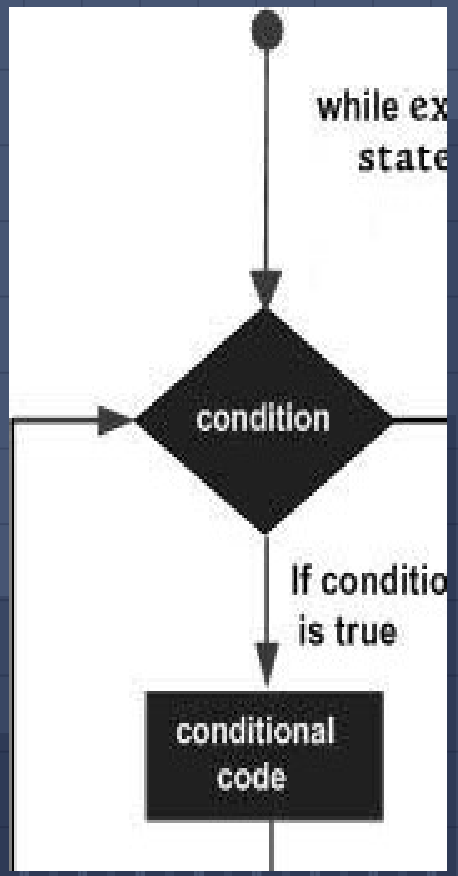
- there is no "**do** ... **while**" **loop** in **Python**. A properly constructed **while loop can do** the same. If you have any problems, give us a simplified idea of what you want to accomplish.

While loop

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

```
while condition (relational expression):
    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.



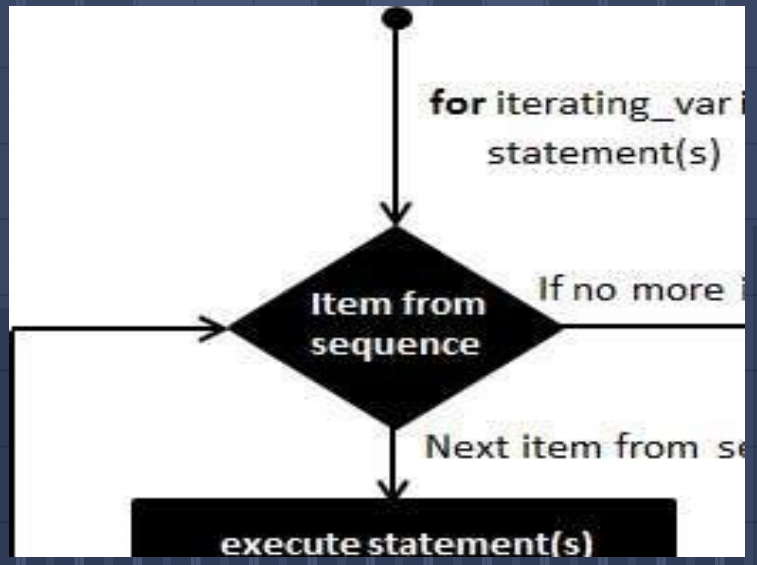
example

```
# write a program to Print 1 2 3 4 5 6 7 8 9 10 … 10000
count = 1 #create variable count and store 1
while count < 10000: # 3<10000
    #loop body
    print(count) #3
    count=count + 1  #4
print(“good bye”)
```

For loop

- For loop is used block of code on list/tuples/dictionary/set & string.
- It will execute code block as many times as no of items in list/tuples/dictionary or no of characters in string.

  for variable-name in list/tuple/dictionary/set/string:
- 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

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

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

# Prints out 11,12,13,13,15,16,17,18,19,20
for number in range(11,21):
    print(number)

for letter in 'Python':
    print (letter)
```

Example of reverse for loop

```
cars = [‘BMW’,‘AUDI’,‘Tesla’]
for item in reversed(cars):
   print(item,end=‘ ‘)


for number in range(6,3,-1):
      #6 is including but 3 is not including
     print(number,end=‘ ‘) ## print output in same
   line
print()
```

Nested loop

- When one use loop inside another loop, then it is called nested loop.
- One can do nesting of loop in this manner up to 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 variable in list/tuple/dictionary/string: #outer loop
   for variable in list/tuple/dictionary/string: #inner loop
      statements(s)
   statements(s)

while condition: #outer loop
   while condition: #inner loop
      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.

  every function name ends with a colon :
- Last line of function should be return keyword
- If function return value then we should give variable after return.
- A return statement without variable is the same as return None.
- All the code in function should be given with tab like we give tab in decision making and loops.

Example

```
def getsquare(number): #10 actual argument
    square = number * number
    #square variable is local variable ( local variable can
    be accessed inside function)
    return square

num = int(input("Enter number")) #10 formal argument

# Now you can call getSquare function
answer = getsquare(num) #calling function/use function
print(answer)
```

Types of functions

1. **Without return value Without argument**
   Function without argument and without return value is called without return value Without argument function

2. **without return value With argument**
   Function that has one or more arguments and do not return any value is called without return value with argument function.

3. With return value Without argument
   a function that has no argument but return value is called with return value without argument function.

4. With return value With argument
   Function that has one or more arguments and also return value is called with return value with argument function.

Default Argument function

- 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)
Result = nsquare(2) #only one argument is provided
print("The square of the sum of 2 and 2 is : ",Result)
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 an example.

Example of keyword arguments

```
def getMerit(maths,science,english,computer,history,drawing):
    print(maths,science,english,computer,history,drawing)
    total = maths + science + english
    return    total
maths = 80
science = 90
english = 100
computer = 50
history = 45
drawing = 40
#bad way to call function because it creates wrong merit because
   arguments are passed in wrong sequence
print (getMerit(computer,history,drawing,maths,science,english))

#perfect way to call function (keyword argument style)
print
   (getMerit(computer=computer,history=history,drawing=drawing,
   maths=maths,science=science,english=english))
```

Return multiple value from function

- It is possible that python function can return multiple value.
- Function return multiple value as **tuple**
- Let us see an example

```
def calculation(a, b):
    addition = a + b
    subtraction = a - b
    # return multiple values separated by comma
    return addition, subtraction

# when function return multiple value it return all value as
   tuple (read only list)
result = calculation(40, 10)
print(result) # (50,30)
```

Arbitrary Argument functions

- 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 till some condition is true, then it is called recursion.
- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- Recursion should be avoided as far as possible. Means it should be last option to solve problem.

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.

- **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.

- **Disadvantages of Recursion**
  - Sometimes the logic behind recursion is hard to follow through / difficult to understand.
  - Recursive calls are expensive (inefficient) in terms of resources as they take up a lot of memory and time to solve problem.
  - Recursive functions are hard to debug (errors are difficult to find and solve).

example

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

def getFactorial(num):
    if num == 1:
        return 1
    else:
        return (num * getFactorial(num-1))

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

lambda functions

- A lambda function in Python is a small, anonymous function defined using the lambda keyword.
- Unlike regular functions defined with def, lambda functions are typically used for short, simple operations that can be written in a single line.
- They are often used in situations where a function is needed temporarily, such as in arguments to higher-order functions like map(), filter(), or sorted().
- Syntax
- lambda arguments: expression

1. **lambda**: Keyword to start the function.
2. **arguments**: Inputs (e.g., x, y).
3. **expression**: One line of code that returns a result.

Example

```
multiply = lambda x, y: x * y
print(multiply(2, 3))  # Output: 6
```

Map function

- **Purpose**: Applies a given function to each item in an iterable (e.g., list, tuple) and returns an iterator of the results.
- **Syntax**:
  map(function, iterable, ...)
  function: A function to apply to each item.
  iterable: One or more iterables (e.g., lists).
  Returns: A map object (iterator). Convert to a list with list() if needed.

```
numbers = [1, 2, 3, 4]
squared = map(lambda x: x * x, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

With Multiple Iterables:

```
list1 = [1, 2, 3]
list2 = [10, 20, 30]
result = map(lambda x, y: x + y, list1, list2)
print(list(result))  # Output: [11, 22, 33]
```

filter

- **filter()**
- **Purpose**: Filters items in an iterable based on a function that returns True or False. Only items where the function returns True are kept.
- **Syntax**:

filter(function, iterable)

- function: A function that returns True or False for each item.
- iterable: The iterable to filter.
- Returns: A filter object (iterator). Convert to a list with list().

```
numbers = [1, 2, 3, 4, 5, 6]
evens = filter(lambda x: x % 2 == 0, numbers)
print(list(evens))  # Output: [2, 4, 6]
```

- **Key Points**:
- Use for selecting items based on a condition.
- The function must return a boolean (True or False).
- Returns an iterator, so convert to a list if needed.

sorted()

- **Purpose**: Sorts items in an iterable and returns a new sorted list.
- Syntax
- sorted(iterable, key=None, reverse=False)
- iterable: The iterable to sort (e.g., list, tuple).
- key: A function to determine sorting order (optional).
- reverse: Set to True to sort in descending order (default is False).
- Returns: A new sorted list.
- Example

```
numbers = [3, 1, 4, 2]
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # Output: [1, 2, 3, 4].

numbers = [3, 1, 4, 2]
sorted_numbers = sorted(numbers, reverse=True)
print(sorted_numbers)  # Output: [4, 3, 2, 1]
```