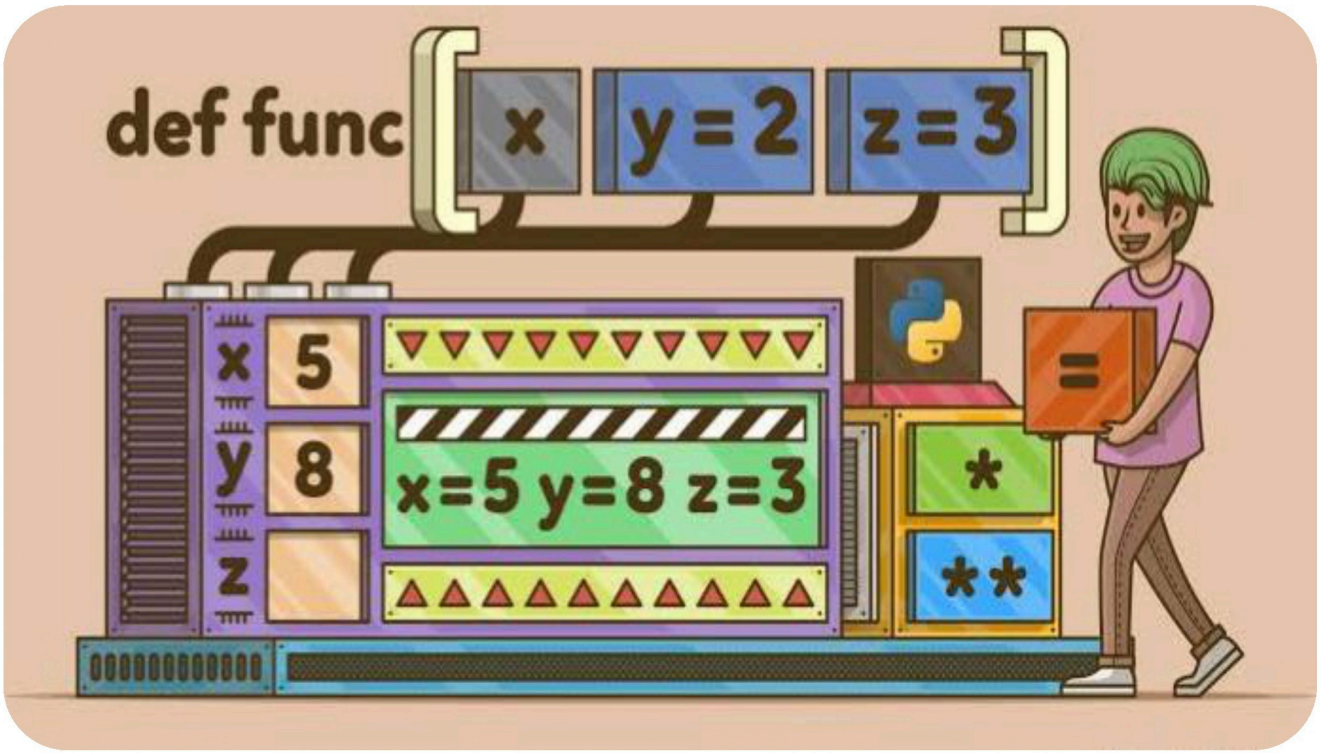


Python Functions

- A function is a collection of related assertions that performs a mathematical, analytical, or evaluative operations.
- Sometimes we have to design our function to perform some operation.
- Python functions are simple to define and essential to intermediate-level programming.
- The exact criteria hold to function names as they do to variable names.
- The goal is to group up certain often performed actions and define a function. Rather than rewriting the same code block over and over for varied input variables, we may call the function and repurpose the code included within it with different variables.
- If a group of statements is repeatedly required, then it is not recommended to write them separately every time.
- If a group of statements is repeatedly required, then it is not recommended to write them separately every time.
- We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting. This unit is nothing but function.

  Note: In other languages, functions are known as methods, procedures, subroutines, etc.

Types of Function

- **Built-in Functions** — The functions which are coming along with Python software automatically, are called built-in functions or pre-defined functions. Eg: `id()`, `type()`, `input()`, `eval()`
- **User-Defined Functions** — The functions which are developed by programmers explicitly according to business requirements are called user-defined functions.

Difference between Built-in Function and Pre-defined Function:

In Python, the terms "built-in function" and "predefined function" are often used interchangeably, but they can have slightly different meanings depending on the context.

1. Built-in Function:
   - In the context of Python, a built-in function refers to a function that is available as part of the core Python language, and does not require any additional imports or external libraries to use. These functions are always available and can be used directly without any special setup.
   - Example: `print()`, `type()`, `input()`, `range()`, `len()`, `max()`, `min()`, and many others.
2. Pre-defined Function:
   - Pre-defined functions are functions that are defined in modules or libraries, which need to be imported before use.
   - This can include functions that are part of the Python standard library (e.g., `math.sqrt()`, `random.randint()`), as well as functions defined in third-party libraries or custom modules.

In summary, built-in functions are always available, while pre-defined functions require an import statement to be used.

Benefits of Functions:

1. Larger code can be broken up into pieces (Code modularity)
2. Works on the philosophy of write once use forever! (Code Reusability)
3. Code is organized and coherent (Code Readability)

Advantages of Functions:

- By including functions, we can prevent repeating the same code of block repeatedly.
- Python functions, once defined, can be called many times and from anywhere in a program.
- If our Python program is large, it can be separated into numerous functions which is simple to track.
- The key accomplishment of Python functions is we can return as many outputs as we want with different arguments.

Built-in Function

```
abs():
num = -7
print(f"The absolute value of {num} is {abs(num)}")

The abs() function returns the absolute value of a number.
It removes the sign (positive or negative) from a number.
```

```
pow():
base = 2
exp = 3
print(f"{base} raised to the power {exp} is {pow(base, exp)}")

The pow() function raises the first argument to the power of the second argument.
If a third argument is provided, it calculates the result modulo that value.
```

```
min():

print(f"The minimum of 4, 7, and 2 is {min(4, 7, 2)}")

The min() function returns the smallest item from a collection of items.
```

```
max():
print(f"The maximum of 9, 3, and 6 is {max(9, 3, 6)}")

The max() function returns the largest item from a collection of items.
```

```
divmod():

quotient, remainder = divmod(10, 3)
print(f"10 divided by 3 gives a quotient of {quotient} and a remainder of {remainder}")

The divmod() function returns a tuple containing the quotient and remainder of dividing
two numbers.
If you are storing value of the Function into 1 variable it will be stored in Tuple
```

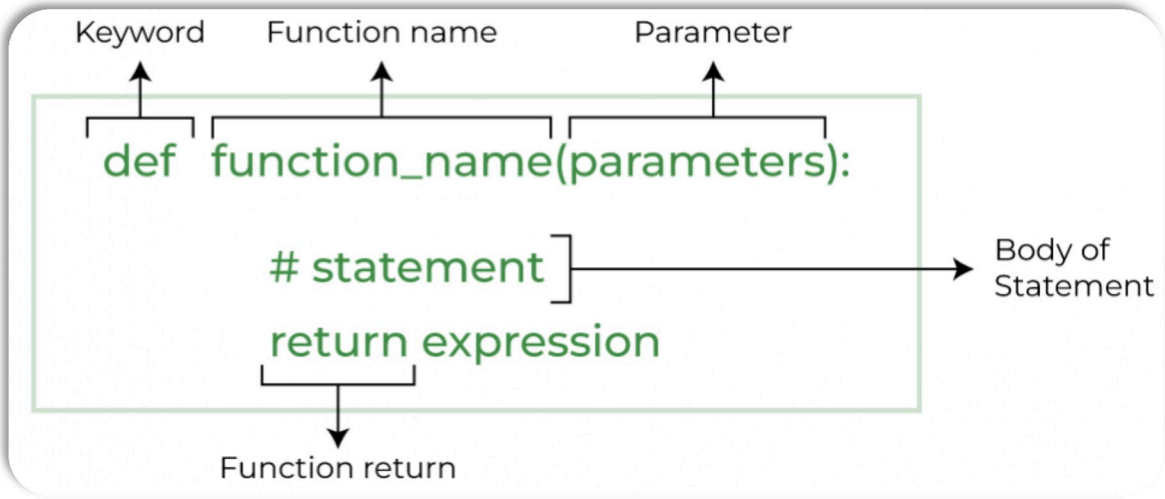
```
round():
num = 3.14159
print(f"Rounded value of {num} with 2 decimal places is {round(num, 2)}")   # prints
3.14

The round() function rounds a number to a specified number of decimal places.
```

```
bin():
num = 10
print(f"The binary representation of {num} is {bin(num)}")

The bin() function converts an integer to a binary string prefixed with '0b'.
```

```
oct():
num = 10
print(f"The octal representation of {num} is {oct(num)}")

The oct() function converts an integer to an octal string prefixed with '0o'.
```

```
hex():
num = 16
print(f"The hexadecimal representation of {num} is {hex(num)}")

The hex() function converts an integer to a hexadecimal string prefixed with '0x'.
```

Syntax of Functions:



Which keywords are used while creating a function?

While creating functions, we can use 2 keywords

1. def (Mandatory)
2. return (Optional)

Example of Functions

```
def Squares(num):
    # This function computes the square of the number
    return num**2


val = int(input("Enter the number to calculate it's square: "))

print(f"The square of {val} is: {Squares(val)}")


OUTPUT:
Enter the number to calculate it's square: 5
The square of 5 is: 25
```

Passing arguments to function by calling it

```
Here the function call is done before the function defination
Here we first called the function and then defined the function

print(f"The square is: {Squares(6)}")

def Squares(num):
    # This function computes the square of the number
    return num**2

OUTPUT:
The square is: 36
```

```
def Greet(name):
    print(f"\nHello {name}!\nHave a Good Day!")

name = input("Enter your name: ")
Greet(name)

OUTPUT:
Enter your name: Aman

Hello Aman!
Have a Good Day!


You have to pass parameters to this function
Greet()
TypeError: Greet() missing 1 required positional argument: 'name'
```

Write a function to check whether the given number is even or odd?

```
def Even_Odd(num):
    """
    BELOW IS THE DOC STRING
    This function tells is a given number is odd or even
    INPUT - Any Valid Integer
    OUTPUT - Even or Odd
    CREATED By - XYZ_Name
    """
    if num % 2 == 0:
        print("Even")
    else:
        print("Odd")

number = int(input("Enter any Number: "))
Even_Odd(number)


OUTPUT:
Enter any Number: 5
Odd
```

If we want to retrieve the document string of any function, then we can access it by using the following approach

```
print(Even_Odd.__doc__)
```

```
    BELOW IS THE DOC STRING
    This function tells is a given number is odd or even
    INPUT - Any Valid Integer
    OUTPUT - Even or Odd
    CREATED By - XYZ_Name
```

What is the difference between `return` and `print`?

- The return keyword allows you to save the result of the output of a function as a variable.
- The `print( )` function simply displays the output to you but doesn't save it for future use.

```
def Print_Sum(a, b):
    print(f"Addition of {a} and {b} is {a+b}")

def Return_Sum(a, b):
    return (a+b)

Print_Sum(10, 5)   # Addition of 10 and 5 is 15
Return_Sum(9, 8)   # Nothing is printed as an Output because it is a return
                   function it will return something not print anything
```

How we can save the result returned by the function for later use?

- We have to create a variable for storing the returned value of a function.
- And have to call the function and directly assign the calling of the function to the variable.

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


My_Result = Return_Sum(9, 8)
print(My_Result)

OUTPUT:
17
```

When a function can't return anything, what return type will be considered for that function?

- When a function does not return anything, actually it returns None as the output of the function.

```
def Return_Sum(a, b):
    a+b

My_Result = Return_Sum(9, 8)
print(My_Result)

OUTPUT: None
```

```
def Square(My_List):
    """Finding Square of items in the List"""
    squares = []
    for i in My_List:
        squares.append(i**2)
    return squares

lst = [1, 2, 3, 4, 5]
print(f"The List is: {lst}")
# Calling the Function
print(f"The Square of items in list is: {Square(lst)}")
```

```
OUTPUT:
The List is: [1, 2, 3, 4, 5]
The Square of items in list is: [1, 4, 9, 16, 25]
```

Function Arguments: Externally we are passing values to the parameters, those values are called arguments.

- Default Arguments
- Keyword Arguments
- Required Arguments
- Variable-length Arguments

1. Default Arguments:

- A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called.
- There is no meaning in using a default argument if you pass an exact number of arguments that match the parameters.
- If value is passed for the default argument then that default value will be overwritten.
- We have to specify default arguments from right to left, if we specify from left to right then it will give an error.
- In the case of parameters and default arguments, the priority is given to parameters, and if no parameters then only to default arguments.
- If parameters are passed and that variable has a default value then that default value is overwritten. Sometimes we can provide default values for our positional arguments.
- If we are not passing any name then only the default value will be considered.

```
def Greet(msg, name = "Guest"):
    print(msg, name)

Greet("Welcome")
Greet("Welcome", "Aman")

OUTPUT:
Welcome Guest
Welcome Aman
```

```
def Funct(num1, num2 = 40):
    print("Number 1 is:",num1)
    print("Number 2 is:",num2)


# Calling a function and passing only one argument.
print("PASSING ONLY ONE ARGUMENT")
Funct(100)

# Calling a function and passing two arguments.
print("\nPASSING TWO ARGUMENT")
Funct(90,20)  # 40 will be overwritten by 20
```

```
OUTPUT:
Number 1 is: 100
Number 2 is: 40

PASSING TWO ARGUMENT
Number 1 is: 90
```

2. Keyword Arguments:

- The arguments in a function called are connected to keyword arguments. If we provide keyword arguments while calling a function, the user uses the parameter label to identify which parameters value it is.
- Since the Python interpreter will connect the keywords given to link the values with its parameters, we can omit some arguments or arrange them out of order.
- The parameter name and argument name must be the same.
- If you have so many parameters in your function then it is hard to remember the order of all parameters, then you can simply pass arguments by their names rather than remembering their order.
- Here the order of arguments is not important but the number of arguments must be matched.

```
Python code to demonstrate the use of keyword arguments

Defining a function
def Funct(num1, num2 = 40):
    print("Number 1 is:",num1)
    print("Number 2 is:",num2)


Calling a function without using keyword argument
print("PASSING VALUES WITHOUT ARGUMENT")
Funct(100, 200)

OUTPUT:
PASSING VALUES WITHOUT ARGUMENT
Number 1 is: 100
Number 2 is: 200



Calling a function by using keyword argument
print("\nPASSING VALUES WITH KEYWORD ARGUMENTS")
Funct(num2 = 90, num1 = 20)

OUTPUT:
PASSING VALUES WITH KEYWORD ARGUMENTS
Number 1 is: 20
Number 2 is: 90
```

```
print(pow(2,3))   # prints 8


print(pow(y=3, x=2))

OUTPUT:
TypeError: pow() missing required argument 'base' (pos 1)
```

3. Required Arguments:

- These are also called positional arguments.
- These are the arguments passed to function in the correct positional order.
- The number of arguments and position of arguments must be changed. If we change the order then the result may be changed.
- The arguments are given to a function while calling in a pre-defined positional sequence are required arguments. The count of required arguments in the method call must be equal to the count of arguments provided while defining the function.
- We must send two arguments to the function “function( )” in the correct order, or it will return a syntax error.
- This tells that number of parameters must match the total number of arguments.
- Required arguments are Number of parameters = Number of arguments

```
Python code to demonstrate the use of required arguments

def Funct(num1, num2):
    print("Number 1 is:",num1)
    print("Number 2 is:",num2)


Funct(100, 40)  # This Satisfies the required Arguments

OUTPUT:
Number 1 is: 100
Number 2 is: 40


Funct(30)          # This will give an error
OUTPUT: TypeError: Funct() missing 1 required positional argument: 'num2'



Funct(30, 50, 90)  # This will give an error
OUTPUT: TypeError: Funct() takes 2 positional arguments but 3 were given
```

4. Variable-length Arguments

- We can use special characters in Python functions to pass as many arguments as we want in a function. There are two types of characters that we can use for this purpose:
  - ***args:** These are Non-Keyword Arguments
  - ****kwargs:** These are Keyword Arguments
  - Here only * and ** are mandatory, **args** and **kwargs** are not at all mandatory.
- What we have seen in keyword arguments is, the names are keywords, In the same way, if we want to pass keywords to the argument in case of variable-length arguments, we use **
- ** denotes the keyword argument in the variable-length argument.
- If I have to pass like num1=20, num2=40 means I want to pass in a keyword-like fashion then we can use these **
- ** is used for storing value in a key-value pair.
- ‘*’ is used to store a single value, so * is sufficient for that.
- **By default the output of single starred variable (*) variable is in the form of a tuple**
- **By default, the output of double starred variable (* *) variable is in the form of a dictionary**

```
Python code to demonstrate the use of variable-length arguments

Defining a function
def Funct(*args):
    print("Numbers are:",args)


Funct(10, 25, 78)
Funct(17, 78, 96, 48, 56, 47)
Funct(12)

OUTPUT:
Numbers are: (10, 25, 78)
Numbers are: (17, 78, 96, 48, 56, 47)
Numbers are: (12,)
```

```
Calculating sum of numbers
def Sum(*args):
    Total = 0
    for val in args:
        Total += val

    print("Sum:",Total)


Sum()              # Sum: 0
Sum(10)            # Sum: 10
Sum(10, 20, 30)    # Sum: 60
```

What happens if we are passing multiple arguments with keywords to a single starred variable?

If we are passing multiple arguments with keywords to a single starred variable then it will give an error

```
def Funct(*args):
    print("Numbers are:",args)


Funct(num1 = 10, num2 = 25, num3 = 78)

OUTPUT: TypeError: Funct() got an unexpected keyword argument 'num1'
```

So we have to give the ** for accessing the values in the form of keyword arguments

```
def Funct(**kwargs):
    print("Numbers are:",kwargs)


Funct(num1 = 10, num2 = 25, num3 = 78)

OUTPUT:
Numbers are: {'num1': 10, 'num2': 25, 'num3': 78}
```

```
def Vals(n1, *s):
    print(n1)
    print(s)

Vals(10)
OUTPUT:
10
()

Vals(10, 1, 2, 3, 4, 5)
OUTPUT:
10
(1, 2, 3, 4, 5)

Vals()
OUTPUT: TypeError: Vals() missing 1 required positional argument: 'n1'


def Vals(*s, n1):
    print(n1)
    print(s)

Vals(10)
OUTPUT: TypeError: Vals() missing 1 required keyword-only argument: 'n1'

Vals(10, 1, 2, 3, 4, 5)
OUTPUT: TypeError: Vals() missing 1 required keyword-only argument: 'n1'
```

Applying *args on the function

```
def Avg_Num(*num):
    sum = 0; length = len(num)

    for i in num:
        sum += i
    print(f"The Average of Numbers: {sum/length}")


Avg_Num(3, 4)           # prints The Average of Numbers: 3.5
Avg_Num(1, 2, 3, 4)     # prints The Average of Numbers: 2.5
```

Applying *kwargs on the function

```
def Avg_Num(**nums):
    sum = 0; length = len(nums)

    for key, value in nums.items():
        sum += value
    print(f"The Average of Numbers: {sum/length}")


Avg_Num(a1 = 3, a2 = 4)                 # prints The Average of Numbers: 3.5
Avg_Num(x1 = 1,x2 = 2,x3 = 3,x4 = 4)    # prints The Average of Numbers: 2.5
```

Return Statement

- We write a return statement in a function to leave a function and give the calculated value when a defined function is called.
- return < expression to be returned as output >
- An argument, a statement, or a value can be used in the return statement, which is given as output when a specific task or function is completed. If we do not write a return statement, then the None object is returned by a defined function.
- The return statement is not mandatory, we can have a function without a return statement.
- In C, C++, and Java we are using NULL keyword to signify that no value, so in Python, we do not have NULL, here we are using the None
- If we don't want to use a return statement in a function, then we have to do all the operations and display in that function so no need to use return.

Pass Statement

- Function definitions can not be empty, but if you for some reason have a function definition with no content, put it in the pass statement to avoid getting an error.
- This keyword is used when we go for writing a program but we are not sure about the logic.
- This is used when we want to define a function, but we don't know internally which code of blocks should be written. But we don't want that function to affect the whole code.
- This statement is used when you don't want an error when no code is written in the body of the function.
- Pass is used for ignoring a particular function.
- Pass statement is something where we will be defining the function but we do not know what the function is going to interpret.
- Suppose you have to give code to a client, and if it contains some arithmetic operation but it's not clear but you want that code to run without any error then use a pass statement.
- By using the pass keyword, it will bypass that function.

```
def Cube(num):
    # We don't know the Logic here

def Square(num):
    print("The Square of number is:",num**2)

val = int(input("Enter the Number: "))
Square(val)
```

```
def Square(num):
    ^
IndentationError: expected an indented block after function definition
```

```
def Cube(num):
    # We don't know the Logic here
    pass

def Square(num):
    print("The Square of number is:",num**2)

val = int(input("Enter the Number: "))
```

```
OUTPUT:
Enter the Number: 5
The Square of number is: 25
```

Recursive Functions

- In Python, it is possible for a function to call itself. These types of construct are termed recursive functions.
- Recurring or recursion is getting repeated again and again.
- If we are not putting any condition in a return then it will indicate that, stop the recursion.
- Why do we call it a recursive function, because it is calling itself again and again?
- Questions asked in recursion are, what will be the output after 4th recursion or 5th recursion?

The main advantages of recursive functions are:

1. We can reduce the length of the code and improves readability.
2. We can solve complex problems very easily.

1. Python Function to find the factorial of a given number with recursion

```
def Factorial(x):
    if x == 1:
        return 1
    else:
        return (x*Factorial(x-1))

num = int(input("Enter a number to find out Factorial: "))
Ans = Factorial(num)

print(f"The Factorial of {num} is {Ans}")
```

```
OUTPUT:
Enter a number to find out Factorial: 5
The Factorial of 5 is 120
```

2. Write a program to create a recursive function to calculate the sum of numbers

```
def Addition(num):
    if num>0:
        return (num + Addition(num-1))
    else:
        return 0

val = int(input("Enter the number: "))
print(f"The Addition of number from 0 to {val} is: {Addition(val)}")
```

```
OUTPUT:
Enter the number: 10
The Addition of number from 0 to 10 is: 55
```

3. Python program to print the Fibonacci series

```
def Fibonacci(num):
    if num == 0:
        return 0
    elif num == 1:
        return 1
    else:
        return Fibonacci(num-1) + Fibonacci(num-2)

val = int(input("Enter the Number: "))

for i in range(0,val):
    print(Fibonacci(i), end = "  ")
```

```
OUTPUT:
Enter the Number: 10
0  1  1  2  3  5  8  13  21  34
```

Lambda Function

- Above are all the user-defined functions and we have seen how to use user-defined functions.
- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.
- Use lambda functions when an anonymous function is required for a short period.
- A lambda function is a one-linear function
- In one line, we are giving all the things, that are definitions, expressions, and arguments.
- You can have n number of parameters but have only one expression.
- After the definition of lambda, there should be an expression only, you can not put a variable in it.
- lambda is also called a one-line expression function.
- The lambda function must be in one line only, if you are using it in multiple lines then it will throw an error.
- Sometimes we can declare a function without any name, such type of nameless functions are called anonymous functions or lambda functions.
- The main purpose of the anonymous function is just for instant use (i.e. for one-time usage)
- Syntax: `lambda argument_list: expression`

Advantages

- Good for simple logical operations that are easy to understand. This makes the code more readable too.
- Good when you want a function that you will use just one time.

Disadvantages

- They can only perform one expression. It's not possible to have multiple independent operations in one lambda function.
- Bad for operations that would span more than one line in a normal `def` function (For example nested conditional operations). If you need a minute or two to understand the code, use a named function instead.
- Bad because you can't write a doc-string to explain all the inputs, operations, and outputs as you would in a normal `def` function.

Note:

- Lambda Function internally returns expression value and we are not required to write return statement explicitly.
- Sometimes we can pass a function as an argument to another function. In such cases, lambda functions are the best choice.
- We can use lambda functions very commonly with `filter()`, `map()`, and `reduce()` functions because these functions expect to function as arguments.

Normal Function:

- We can define by using `def` keyword

```
def SquareIt(n):
    return n*n
```

Lambda Function:

- We can define by using `lambda` keyword

```
lambda n: n*n
```

```
1. Lambda function to find square of given number

SquareIt = lambda n: n*n

val = int(input("Enter the Number to find Square: "))
print(f"The Square of {val} is {SquareIt(val)}")

OUTPUT:
Enter the Number to find Square: 5
The Square of 5 is 25
```

```
2. Lambda function to find sum of 2 given number

SumIt = lambda a,b: a+b

val1 = int(input("Enter a Number: "))
val2 = int(input("Enter another Number: "))

print(f"The Sum of {val1} and {val2} is {SumIt(val1, val2)}")

OUTPUT:
Enter a Number: 5
Enter another Number: 6
The Sum of 5 and 6 is 11
```

IIFE Execution

Immediately invoked function expression or IIFE. The function is created and then immediately executed

- In IIFE we do not have a provision to take input from users like normal execution.
- Below are examples of IIFE execution.

```
print((lambda x: x*2)(12))      # print's 24
```

```
One Parameter, one argument only
print((lambda x: x*2)(12, 2))

TypeError: <lambda>() takes 1 positional argument but 2 were given
```

```
Using two Parameters, and one expression
print((lambda x,y: x*y)(12, 2)) # print's 24
```

```
Lambda function can have only one expression
print((lambda x,y: x*2, y*2)(12, 2))

NameError: name 'y' is not defined
```

```
Because there are 2 parameters and we have passed only 1 argument
print((lambda x,y: x*y)(12))

TypeError: <lambda>() missing 1 required positional argument: 'y'
```

Difference between lambda function and UDF (User defined function)

- There is only one expression in lambda and there are n number of exceptions in UDF.
- For the lambda function, we are using the lambda keyword and for UDF we are using the def keyword
- We can not call the lambda function multiple times, but we can call UDF an infinite number of times.
- A lambda function is defined for a very short period and used only for that period, but UDF is used for an infinite number of times.
- We are using the return keyword in UDF, in Lambda there is no need for a return keyword.

```
User Defined Function
def Reciprocal(num):
    return 1/num

Lambda Function Definition
Lambda_Reciprocal = lambda num : 1/num

Taking input from user
val = int(input("Enter the value to find out reciprocal: "))

Using the function defined by def keyword
print("\nUsing UDF:",Reciprocal(val))

Using the function defined by lambda keyword
print("\nUsing Lambda Function: ",Lambda_Reciprocal(val))

OUTPUT:
Enter the value to find out reciprocal: 6
Using UDF: 0.16666666666666666
Using Lambda Function:  0.16666666666666666
```

```
list_1 = [1, 2, 3, 4, 5, 6, 8, 9]

even = list(filter(lambda x : x%2==0, list_1))

print(even)         # print's [2, 4, 6, 8]
```

```
list_2 = [1, 2, 3, 4, 5, 6, 8, 9]

list_odd = list(filter(lambda x : x%2==1, list_2))

print(list_odd)     # print's [1, 3, 5, 9]
```

```
squares = [lambda num = a : num**2 for a in range(0, 11)]

for square in squares:
    print(square(), end = "   ")

OUTPUT: 0   1   4   9   16   25   36   49   64   81   100
```

Lambda with if-else

lambda <arguments> : <statement1> if <condition> else <statement2>

```
test = lambda x,y: print(x,"is smaller than",y) if x<y else print(x,"is greater than",y)
test(10, 20)    # print's 10 is smaller than 20
```

```
(lambda x,y: print(x,"is smaller than",y) if x<y else print(x,"is greater than",y))(20, 17)
# prints 20 is greater than 17
```

```
(lambda x,y: print(x,"is smaller than",y) if x<y elif x==y print(x,"is equal to",y) else
print(x,"is greater than",y))(20, 17)

SyntaxError: expected 'else' after 'if' expression
```

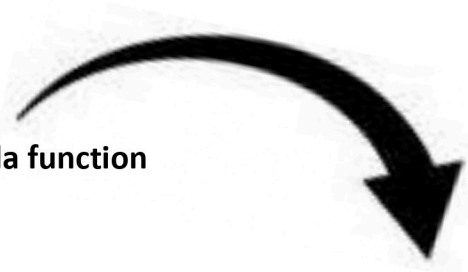
```
(lambda x,y: print(x,"is smaller than",y) if x<y else print(x,"is equal to",y) if x==y else
print(x,"is greater than",y) )(33, 33)

print's 33 is equal to 33
```

filter() function:

- We can use the filter( ) function to filter values from the given sequence based on some condition.
- Where function argument is responsible to perform conditional check sequence can be a list, tuple, or string
- **Syntax:**

```
filter(function, sequence)
```

Program to filter only even numbers from the list by using the filter( ) function.



1. Without using the lambda function

```
def isEven(x):
    if x%2 == 0:
        return True
    else:
        return False

l = [0, 5, 10 ,15, 20, 25, 30]
l1 = list(filter(isEven,l))
print(l1)

OUTPUT:
[0, 10, 20, 30]
```

2. With using the lambda function

```
isEven = lambda x : x%2 == 0

l = [0, 5, 10, 15, 20, 25, 30]
l1 = list(filter(isEven,l))

print(l1)

OUTPUT:
[0, 10, 20, 30]
```

map() function:

- For every element present in the given sequence, apply some functionality and generate a new element with the required modification. For this requirement, we should go for the map() function.
- Ex: For every element present in the list perform a double and generate a new list of doubles.
- The function can be applied to each element of the sequence and generates a new sequence.
- **Syntax:**

```
map(function, sequence)
```

1. Without using the lambda function

```
def DoubleIt(x):
    return x*2

l = [1, 2, 3, 4, 5]
l1 = list(map(DoubleIt, l))

print(l1)

OUTPUT:
[2, 4, 6, 8, 10]
```

2. With using the lambda function

```
l = [1, 2, 3, 4, 5]
l1 = list(map(lambda x : x*2, l))

print(l1)

OUTPUT:
[2, 4, 6, 8, 10]
```

- We can apply the map( ) function on multiple lists also. But make sure all list should have the same length.
- **Syntax:**

```
map(lambda x,y : x*y, l1, l2)
```

- x is from l1 and y is from l2.

```
l = [1, 2, 3, 4, 5]

l1 = list(map(lambda x : x*x, l))

print(l1)

OUTPUT:
[1, 4, 9, 16, 25]
```

```
l1 = [1, 2, 3, 4, 5]
l2 = [2, 3, 4, 5, 6]

l3 = list(map(lambda x,y : x*y, l1,l2))

print(l3)

OUTPUT:
[2, 6, 12, 20, 30]
```

reduce()function:

- reduce( ) function reduces the sequence of elements into a single element by applying the specified function.
- reduce( ) function present in the functools module and hence we should write an import statement.
- **Syntax:** `reduce(function,sequence)`

```
from functools import *

l = [10, 20, 30, 40, 50]
result = reduce(lambda x,y : x+y, l)

print(result)       # print's 150
```

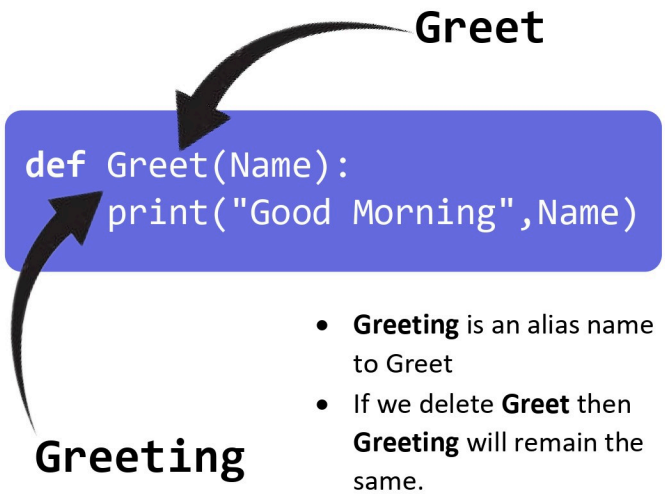
In Python, everything is treated like what?

- In Python, everything is treated as an object.
- Even functions are also internally treated as objects only.

Function Aliasing:

- For the existing function, we can give another name, which is nothing but function aliasing.
- We have seen that if x = 10 and y = 10, both are referring to the same object and hence they both are pointing to the same object, likewise function aliasing.

greet and **greeting** both are referring to the same function



- **Greeting** is an alias name to Greet
- If we delete **Greet** then **Greeting** will remain the same.

```
def Greet(Name):
    print("Good Morning",Name)

Greeting = Greet

Greet("Raman")          # print's Good Morning Raman
Greeting("Govind")      # print's Good Morning Govind

print(id(Greet))        # print's 2291229237920
print(id(Greeting))     # print's 2291229237920
```

Can we access a function by its alias name, if we deleted its original name?

- If we delete one name still we can access the function by using an alias name

```
del Greet
Greeting("Robin")       # print's Good Morning Robin
```

Nested Functions

- We can declare a function inside another function, such types of functions are called Nested functions.
- We can not call the inner function outside the outer function body, otherwise, it will throw an error.

```
def Outer():
    print("Outer Function Started!")

    def Inner():
        print("\nInner Function Started!")
    Inner()

Outer()

OUTPUT:
Outer Function Started!
Inner Function Started!
```

```
def Outer():
    print("Outer Function Started!")

    def Inner():
        print("\nInner Function Started!")
    Inner()

Outer()
Inner()

OUTPUT:
Outer Function Started!
Inner Function Started!
NameError: name 'Inner' is not defined!
```

```
def Func():
    def x(a, b):
        return a+b
    def y(a, b):
        return a-b
    return x

val = Func()(3,4)   # Passing values to nested function inside f()
print(val)          # print's 7
```