In Python, the function is a block of code defined with a name. We use functions whenever we need to perform the same task multiple times without writing the same code again. It can take arguments and returns the value.

Function improves efficiency and reduces errors because of the reusability of a code. Once we create a function, we can call it anywhere and anytime. The benefit of using a function is reusability and modularity.

Types of Functions Python support two types of functions

1. Built-in function (range(), id(), type(), input(), eval() etc.)
2. User-defined function (Functions which are created by programmer explicitly according to the requirement are called a user-defined function.)

Creating a Function

Use the following steps to to define a function in Python.

1. Use the def keyword with the function name to define a function.
2. Next, pass the number of parameters as per your requirement. (Optional).
3. Next, define the function body with a block of code. This block of code is nothing but the action you wanted to perform.

In Python, no need to specify curly braces for the function body. The only indentation is essential to separate code blocks. Otherwise, you will get an error.

While defining a function, we use two keywords, def (mandatory) and return (optional)

In [39]:

```
#Creating a function without any parameters
# function
def message():
    print("Welcome to NetTech")
```

In [40]:

```
def message():
    print("Welcome to NetTech")


# call function using its name
message()
```

```
Welcome to NetTech
```

Parameter Vs Arguement ?

Generally when people say parameter/argument they mean the same thing, but the main difference between them is that the parameter is what is declared in the function, while an argument is what is passed through when calling the function.

In [8]:

```
#Here, the parameters are a and b, and the arguments being passed through are 5 and 4.
def add(a, b):
    return a+b

add(5, 4)
```

Out[8]:

```
9
```

In [9]:

```
#Creating a function with parameters
# function
def course_func(name, course_name):
    print("Hello", name, "Welcome to NetTech")
    print("Your course name is", course_name)

# call function
course_func('Students', 'Python')
```

```
Hello Students Welcome to NetTech
Your course name is Python
```

In [24]:

```
#This function expects 2 arguments, and gets 2 arguments:

def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Sanvee", "Khot")
```

```
Sanvee Khot
```

In [25]:

```
#This function expects 2 arguments, but gets only 1:

def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-25-0588a048a625> in <module>()
      4   print(fname + " " + lname)
      5
----> 6 my_function("Emil")

TypeError: my_function() missing 1 required positional argument: 'lname'
```

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

In [26]:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("a", "b", "c")
```

```
The youngest child is c
```

In [27]:

```
#You can also send arguments with the key = value syntax.
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "a", child2 = "b", child3 = "c")
```

```
The youngest child is c
```

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

In [31]:

```
def my_function(**girl):
  print("His last name is " + girl["lname"])

my_function(fname = "Sanvee", lname = "Khot")
```

```
His last name is Khot
```

Default Parameter Value

If we call the function without argument, it uses the default value:

In [32]:

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

In [33]:

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

```
apple
banana
cherry
```

Type *Markdown* and LaTeX: α^2

Return values

To let a function return a value, use the return statement:

In [83]:

```
def my_function(x):
   return 5 * x
print(my_function(3))
```

```
15
```

In [21]:

```
# function
def calculator(a, b):
    add = a + b
    # return the addition
    return add

# call function
# take return value in variable
res = calculator(20, 5)

print("Addition :", res)
```

```
Addition : 25
```

In [11]:

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

total=sum(10, 20)
print(total)
total=sum(5, sum(10, 20))
print(total)
```

```
30
35
```

The return value is nothing but a outcome of function.

The return statement ends the function execution.

For a function, it is not mandatory to return a value.

If a return statement is used without any expression, then the None is returned.

The return statement should be inside of the function block.

In [35]:

```
#Return multiple values
def arithmetic(num1, num2):
    add = num1 + num2
    sub = num1 - num2
    multiply = num1 * num2
    division = num1 / num2
    # return four values
    return add, sub, multiply, division

# read four return values in four variables
a, b, c, d = arithmetic(10, 2)

print("Addition: ", a)
print("Subtraction: ", b)
print("Multiplication: ", c)
print("Division: ", d)
```

```
Addition:  12
Subtraction:  8
Multiplication:  20
Division:  5.0
```

Local and Global variable

In general, a variable that is defined in a block is available in that block only. It is not accessible outside the block. Such a variable is called a local variable. Formal argument identifiers also behave as local variables.

In [47]:

```
#local variable
def greet():
    name1 = 'Sanvee'
    print('Hello ', name1)
greet()
```

```
Hello  Sanvee
```

In [48]:

```
print(name1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-48-1b8d5f663f2a> in <module>()
----> 1 print(name1)

NameError: name 'name1' is not defined
```

In [50]:

```
#global variable
name='Sanvee'
def greet():
    print ("Hello ", name)
greet()
```

```
Hello  Sanvee
```

In [51]:

```
print(name)
```

```
Sanvee
```

In [54]:

```
def greet():
    global name
    name = 'Bill Gates'
    print('Hello', name)
greet()
```

```
Hello Bill Gates
```

In [57]:

```
name = 'Steve'
def greet():
    name = 'Bill Gates'
    print('Hello', name)
greet()
```

```
Hello Bill Gates
```

In [58]:

```
print(name)
```

```
Steve
```

In [ ]:

```
name = 'Steve'
def greet():
    global name
    name = 'Bill'
    print('Hello ', name)
```

In [59]:

```
print(name)
```

```
Steve
```

In [64]:

```
greet()
```

```
Hello Bill Gates
```

In [65]:

```
def function1():
    # local variable
    loc_var = 888
    print("Value is :", loc_var)

def function2():

    print("Value is :", loc_var)

function1()
function2()
```

```
Value is : 888

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-65-205a15fd3a93> in <module>()
      9
     10 function1()
---> 11 function2()

<ipython-input-65-205a15fd3a93> in function2()
      6 def function2():
      7
----> 8     print("Value is :", loc_var)
      9
     10 function1()

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

In [66]:

```
global_var = 999

def function1():
    print("Value in 1nd function :", global_var)

def function2():
    print("Value in 2nd function :", global_var)

function1()
function2()
```

```
Value in 1nd function : 999
Value in 2nd function : 999
```

In [67]:

```
# Global variable
global_var = 5

def function1():
    print("Value in 1st function :", global_var)

def function2():
    # Modify global variable
    # function will treat it as a local variable
    global_var = 555
    print("Value in 2nd function :", global_var)

def function3():
    print("Value in 3rd function :", global_var)

function1()
function2()
function3()
```

```
Value in 1st function : 5
Value in 2nd function : 555
Value in 3rd function : 5
```

In [68]:

```
# Global variable
x = 5

# defining 1st function
def function1():
    print("Value in 1st function :", x)

# defining 2nd function
def function2():
    # Modify global variable using global keyword
    global x
    x = 555
    print("Value in 2nd function :", x)

# defining 3rd function
def function3():
    print("Value in 3rd function :", x)

function1()
function2()
function3()
```

```
Value in 1st function : 5
Value in 2nd function : 555
Value in 3rd function : 555
```

Lambda Function

In Python, an 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 of Lambda Function in python

lambda arguments: expression

We use lambda functions when we require a nameless function for a short period of time.

In [74]:

```
x = lambda a : a + 10

print(x(5))
```

```
15
```

In [75]:

```
x = lambda a, b : a * b
print(x(5, 6))
```

```
30
```

In [1]:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

```
13
```

In [2]:

```
#We can declare a lambda function and call it as an anonymous function, without assigning i
(lambda x: x*x)(5)
```

Out[2]:

```
25
```

In [6]:

```
#normal function
def dosomething(fn):
        print('Calling function argument:')
        fn()

#without variable lambda function
dosomething(lambda : print('Hello World')) # passing anonymous function
```

```
Calling function argument:
Hello World
Calling function argument:
Hello World
```

In [ ]:

```
```

In [69]:

```
#Example 1: Program for even numbers without lambda function

def even_numbers(nums):
    even_list = []
    for n in nums:
        if n % 2 == 0:
            even_list.append(n)
    return even_list

num_list = [10, 5, 12, 78, 6, 1, 7, 9]
ans = even_numbers(num_list)
print("Even numbers are:", ans)
```

```
Even numbers are: [10, 12, 78, 6]
```

In [70]:

```
#Example 2: Program for even number with a lambda function

l = [10, 5, 12, 78, 6, 1, 7, 9]
even_nos = list(filter(lambda x: x % 2 == 0, l))
print("Even numbers are: ", even_nos)
```

```
Even numbers are:  [10, 12, 78, 6]
```

Lambda functions are used along with built-in functions like filter(), map() etc.

The filter() function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

In [8]:

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

```
[4, 6, 8, 12]
```

The map() function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

In [9]:

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Recursion

A recursive function is a function that calls itself, again and again.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

In [14]:

```
def factorial(no):
    if no == 0:
        return 1
    else:
        return no * factorial(no - 1)

print("factorial of a number is:", factorial(9))
```

```
factorial of a number is: 362880
```

Exercise Find out advantage and disadvantage of Recursive ?

In [ ]:

```

```