<https://www.w3schools.com/python/python_functions.asp>

<https://www.tutorialspoint.com/python/python_functions.htm>

<https://www.geeksforgeeks.org/function-decorators-in-python-set-1-introduction/>

<https://intellipaat.com/blog/tutorial/python-tutorial/python-functions/>

<https://www.techbeamers.com/python-function/>

<https://python.maateen.me/docs/function/>

<http://w3programmers.com/bangla/what-is-function-in-python/>

# Python Functions

[❮ Previous](https://www.w3schools.com/python/python_for_loops.asp)[Next ❯](https://www.w3schools.com/python/python_lambda.asp)

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

## Creating a Function

In Python a function is defined using the def keyword:

### Example

def my\_function():  
  print("Hello from a function")

## Calling a Function

To call a function, use the function name followed by parenthesis:

### Example

def my\_function():  
  print("Hello from a function")  
  
**my\_function()**

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function)

## Parameters

Information can be passed to functions as parameter.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

### Example

def my\_function(**fname**):  
  print(fname + " Refsnes")  
  
my\_function(**"Emil"**)  
my\_function(**"Tobias"**)  
my\_function(**"Linus"**)

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_param)

## Default Parameter Value

The following example shows how to use a default parameter value.

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

### Example

def my\_function(**country = "Norway"**):  
  print("I am from " + country)  
  
my\_function("Sweden")  
my\_function("India")  
my\_function()  
my\_function("Brazil")

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_param2)

## Passing a List as a Parameter

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

E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

### Example

def my\_function(food):  
  for x in food:  
    print(x)  
  
fruits = ["apple", "banana", "cherry"]  
  
my\_function(fruits)

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_param3)

## Return Values

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

### Example

def my\_function(x):  
  **return 5 \* x**  
print(my\_function(3))  
print(my\_function(5))  
print(my\_function(9))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_return)

## Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

### Example

def my\_function(child3, child2, child1):  
  print("The youngest child is " + child3)  
  
my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_kwargs)

The phrase Keyword Arguments are often shortened to kwargs in Python documentations.

## Arbitrary Arguments

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.

This way the function will receive a tuple of arguments, and can access the items accordingly:

### Example

If the number of arguments are unknown, add a \* before the parameter name:

def my\_function(\*kids):  
  print("The youngest child is " + kids[2])  
  
my\_function("Emil", "Tobias", "Linus")

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_args)

## The pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

### Example

def myfunction:  
  pass

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_pass)

## Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, tri\_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

### Example

Recursion Example

def tri\_recursion(k):  
  if(k>0):  
    result = k+tri\_recursion(k-1)  
    print(result)  
  else:  
    result = 0  
  return result  
  
print("\n\nRecursion Example Results")  
tri\_recursion(6)

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_recursion)

# Python Lambda

[❮ Previous](https://www.w3schools.com/python/python_functions.asp)[Next ❯](https://www.w3schools.com/python/python_arrays.asp)

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

## Syntax

lambda arguments : expression

The expression is executed and the result is returned:

### Example

A lambda function that adds 10 to the number passed in as an argument, and print the result:

x = lambda a : a + 10  
print(x(5))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda)

Lambda functions can take any number of arguments:

### Example

A lambda function that multiplies argument a with argument b and print the result:

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

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda2)

### Example

A lambda function that sums argument a, b, and c and print the result:

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

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda3)

## Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

def myfunc(n):  
  return lambda a : a \* n

Use that function definition to make a function that always doubles the number you send in:

### Example

def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
  
print(mydoubler(11))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda_double)

Or, use the same function definition to make a function that always triples the number you send in:

### Example

def myfunc(n):  
  return lambda a : a \* n  
  
mytripler = myfunc(3)  
  
print(mytripler(11))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda_triple)

Or, use the same function definition to make both functions, in the same program:

### Example

def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
mytripler = myfunc(3)  
  
print(mydoubler(11))  
print(mytripler(11))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda_both)

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

## Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

* 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.
* The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
* The code block within every function starts with a colon (:) and is indented.
* The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

## Syntax

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

## Example

The following function takes a string as input parameter and prints it on standard screen.

def printme( str ):

"This prints a passed string into this function"

print str

return

## Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

[Live Demo](http://tpcg.io/3mrnYY)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme("I'm first call to user defined function!")

printme("Again second call to the same function")

When the above code is executed, it produces the following result −

I'm first call to user defined function!

Again second call to the same function

## Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

[Live Demo](http://tpcg.io/ZwGczd)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist.append([1,2,3,4]);

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

[Live Demo](http://tpcg.io/xgrZcs)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist = [1,2,3,4]; # This would assig new reference in mylist

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

## Function Arguments

You can call a function by using the following types of formal arguments −

* Required arguments
* Keyword arguments
* Default arguments
* Variable-length arguments

## Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

[Live Demo](http://tpcg.io/KIgYav)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme()

When the above code is executed, it produces the following result −

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)

## Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

[Live Demo](http://tpcg.io/A0FJD1)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme( str = "My string")

When the above code is executed, it produces the following result −

My string

The following example gives more clear picture. Note that the order of parameters does not matter.

[Live Demo](http://tpcg.io/hdONo3)

#!/usr/bin/python

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

## Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

[Live Demo](http://tpcg.io/JWyo6q)

#!/usr/bin/python

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Name: miki

Age 35

## Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

[Live Demo](http://tpcg.io/qofSxL)

#!/usr/bin/python

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

print "Output is: "

print arg1

for var in vartuple:

print var

return;

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

When the above code is executed, it produces the following result −

Output is:

10

Output is:

70

60

50

## The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

* Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
* An anonymous function cannot be a direct call to print because lambda requires an expression
* Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
* Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

## Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how *lambda* form of function works −

[Live Demo](http://tpcg.io/TGEvCu)

#!/usr/bin/python

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )

When the above code is executed, it produces the following result −

Value of total : 30

Value of total : 40

## The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

[Live Demo](http://tpcg.io/FHnLKA)

#!/usr/bin/python

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print "Inside the function : ", total

return total;

# Now you can call sum function

total = sum( 10, 20 );

print "Outside the function : ", total

When the above code is executed, it produces the following result −

Inside the function : 30

Outside the function : 30

## Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

* Global variables
* Local variables

## Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

[Live Demo](http://tpcg.io/LFZtHG)

#!/usr/bin/python

total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2; # Here total is local variable.

print "Inside the function local total : ", total

return total;

# Now you can call sum function

sum( 10, 20 );

print "Outside the function global total : ", total

When the above code is executed, it produces the following result −

Inside the function local total : 30

Outside the function global total : 0

## ****Table of Built in Functions in Python****

The print() function is just one of many built in functions in Python. Let’s learn about more built in functions and their descriptions through the following table.

|  |  |
| --- | --- |
| **Python Built in Functions** | **Description** |
| Python abs() | It returns the absolute value of a number, and the returned value is always positive. |
| Python all() | It returns true when all elements in an iterable are evaluated to be true orif there are no elements in an iterable. It returns false if any element is evaluated to be false. |
| Python any() | It checks if any element of an iterable is true. Unlike all(), it returns false if there are no elements in the iterable. |
| Python ascii() | This method returns a string containing printable representation. |
| Python bin() | This built in function in Python is used for converting an integer to a binary string. |
| Python bool() | This built in function in Python is used for converting a value to Boolean. |
| Python bytearray() | This built in function in python returns an array of the given byte size. |
| Python bytes() | This built in function in Python returns immutable bytes object. |
| Python callable() | This built in function Python is used for checking if the object is callable. |
| Python chr() | This built in python function returns a character (a string) from an integer. |
| Python classmethod() | This built in python function returns the class method for a given function. |
| Python compile() | This built in python function returns a Python code object. |
| Python complex() | This built in function in Python is used for creating a complex number. |
| Python delattr() | This built in function in Python is used for deleting an attribute from an object. |
| Python dict() | This built in function is used for creating a [Python dictionary](https://intellipaat.com/blog/tutorial/python-tutorial/python-dictionary/). |
| Python dir() | This built in python function tries to return the attributes of an object. |
| Python divmod() | This built in python function returns a tuple of quotient and remainder. |
| Python enumerate() | This built in python function returns an enumerate object. |
| Python eval() | This built in python function runs the Python code within a program |
| Python exec() | This built in python function is used for executing a dynamically created program. |
| Python filter() | This built in function in Python is used for constructing an iterator from the elements which are true. |
| Python float() | This built in function in Python is used for returning the floating-point number from a number or a string. |
| Python format() | This built in python function returns the formatted representation of a value. |
| Python frozenset() | This built in python function returns an immutable frozenset object. |
| Python getattr() | This built in python function returns the value of the named attribute of an object. |
| Python globals() | This built in python function returns the dictionary of a current global symbol table. |
| Python hasattr() | This built in python function returns a value that tells whether an object has a named attribute. |
| Python hash() | This built in python function returns the hash value of an object. |
| Python help() | This built in python function Invokes the built in help system |
| Python hex() | This built in function in Python is used for converting an integer to its hexadecimal form. |
| Python id() | This built in python function returns the identify of an object. |
| Python input() | This built in python function usually reads and returns a line of string. |
| Python int() | This built in python function returns an integer from a number or a string. |
| Python isinstance() | This function checks if an object is an instance of a class. |
| Python issubclass() | This checks if an object is a subclass of a class. |
| Python iter() | This function returns an iterator for an object. |
| Python len() | This function returns the length of an object. |
| Python list() | This function is used for creating a [Python list](https://intellipaat.com/blog/tutorial/python-tutorial/python-lists/). |
| Python locals() | This function returns the dictionary of a current local symbol table |
| Python map() | This function applies functions and returns a list. |
| Python max() | This function returns the largest element. |
| Python memoryview() | This function returns the memory view of an argument. |
| Python min() | This function returns the smallest element. |
| Python next() | This function retrieves the next element from an iterator. |
| Python object() | This function creates a featureless object. |
| Python oct() | This function converts an integer to its octal form. |
| Python open() | This function returns a file object. |
| Python ord() | This function returns a Unicode point for a Unicode character. |
| Python pow() | This function evaluates and returns x to the power of y. |
| Python print() | It is used for printing a given object. |
| Python property() | This function returns a property attribute. |
| Python range() | This function returns the sequence of integers between start and stop. |
| Python repr() | This function returns the printable representation of an object. |
| Python reversed() | This function returns the reversed iterator of a sequence. |
| Python round() | This function rounds a floating-point number to n digits places. |
| Python set() | This function returns a [Python set](https://intellipaat.com/blog/tutorial/python-tutorial/python-sets/). |
| Python setattr() | This function sets the value of an attribute of an object. |
| Python slice() | It is used for creating a slice object specified by the range() function. |
| Python sorted() | This function returns the sorted list from a given iterable. |
| Python staticmethod() | This function creates a static method from a function. |
| Python str() | This function returns an informal representation of an object. |
| Python sum() | This function is used for adding items of an Iterable. |
| Python super() | This function allows us to refer the parent class by ‘super’. |
| Python tuple() Function | This function creates a tuple. |
| Python type() | This function returns the type of an object. |
| Python vars() | This function returns the \_\_dict\_\_ attribute of a class. |
| Python zip() | This function returns an iterator of tuples. |
| Python \_\_import\_\_() | This function is an advanced function called by ‘import’. |

## What Is A Python Function?

A function in Python is a logical unit of code containing a sequence of statements indented under a name given using the “def” keyword.

Functions allow you to create a logical division of a big project into smaller modules. They make your code more manageable and extensible.

While programming, it prevents you from adding duplicate code and promotes reusability.

### How To Create A Function – Syntax

The syntax of a Python function is as follows.

**Single line function:**

def single\_line(): statement

**Python function with docstring:**

def fn(arg1, arg2,...):

"""docstring"""

statement1

statement2

**Nested Python function:**

def fn(arg1, arg2,...):

"""docstring"""

statement1

statement2

def fn\_new(arg1, arg2,...):

statement1

statement2

...

...

[**TOC**](https://www.techbeamers.com/python-function/#toc)

### Def Statement

Please read the below notes before creating your first Python function.

* The “def” keyword is a statement for defining a function in Python.
* You start a function with the def keyword, specify a name followed by a colon (:) sign.
* The “def” call creates the function object and assigns it to the name given.
* You can further re-assign the same function object to other names.
* Give a unique name to your function and follow the same rules as naming the [**identifiers**](https://www.techbeamers.com/python-keywords-identifiers-variables/).
* Add a meaningful docstring to explain what the function does. However, it is an optional step.
* Now, start the function body by adding valid Python statements each indented with four spaces.
* You can also add a statement to return a value at the end of a function. However, this step is optional.
* Just press enter and remove the indentation to end a function.
* Since def is a statement, so you can use it anywhere a statement can appear – such as nested in an if clause or within another function.

**Example :**

if test:

def test(): # First definition

...

else:

def test(): # Alternate definition

...

...

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## How To Call A Function In Python?

By using the def keyword, you learned to create the blueprint of a function which has a name, parameters to pass and a body with valid [**Python statements**](https://www.techbeamers.com/understand-python-statement-indentation/).

The next step is to execute it. You can do so by calling it from the Python script or inside a function or directly from the Python shell.

To call a function, you need to specify the function name with relevant parameters, and that’s it.

Follow the below example to learn how to call a function in Python.

### Example Of A Function Call

It’s a simple example where a function “typeOfNum()” has nested functions to decide on a number is either odd or even.

def typeOfNum(num): # Function header

# Function body

if num % 2 == 0:

def message():

print("You entered an even number.")

else:

def message():

print("You entered an odd number.")

message()

# End of function

typeOfNum(2) # call the function

typeOfNum(3) # call the function again

[**TOC**](https://www.techbeamers.com/python-function/#toc)

### Polymorphism In Python

In Python, functions polymorphism is possible as we don’t specify the argument types while creating functions.

* The behavior of a function may vary depending upon the arguments passed to it.
* The same function can accept arguments of different object types.
* If the objects find a matching interface, the function can process them.

**Example :**

def product(x, y) : return x \* y

print(product(4, 9)) # function returns 36

print(product('Python!', 2)) # function returns

# Python!Python!

print(product('Python 2 or 3?', '3')) # TypeError occurs

The above example clarifies that we can pass any two objects to the product() function which supports the ‘\*’ operator.

The concept above we’ve explained is known as polymorphism. Some points which you should remember are as follows.

* Python is a dynamically typed language which means the types correlate with values, not with variables. Hence, the polymorphism runs unrestricted.
* That’s one of the primary differences between Python and other statically typed languages such as C++ or Java.
* In Python, you don’t have to mention the specific data types while coding.
* However, if you do, then the code limits to the types anticipated at the time of coding.
* Such code won’t allow other compatible types that may require in the future.
* Python doesn’t support any form of function overloading.

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## Parameters In A Function

We often use the terms **parameters and arguments** interchangeably. However, there is a **slight difference between them**.

**Parameters** are the variables used in the function definition whereas **arguments** are the values we pass to the function parameters.

Python supports different variations of passing parameters to a function. Before we discuss each of them, you should read the following notes.

* The argument gets assigned to the local variable name once passed to the function.
* Changing the value of an argument inside a function doesn’t affect the caller.
* If the argument holds a mutable object, then changing it in a function impacts the caller.
* We call the passing of immutable arguments as Pass by Value because Python doesn’t allow them to change in place.
* The passing of mutable arguments happens to be Pass by Pointer in Python because they are likely to be affected by the changes inside a function.

### Example: Immutable Vs. Mutable

def test1(a, b) :

a = 'Garbage' # 'a' receives an immutable object

b[0] = 'Python' # 'b' receives a list object

# list is mutable

# it can undergo an in place change

def test2(a, b) :

a = 'Garbage 2'

b = 'Python 3' # 'b' now is made to refer to new

# object and therefore argument 'y'

# is not changed

arg1 = 10

arg2 = [1, 2, 3, 4]

test1(arg1, arg2)

print("After executing test 1 =>", arg1, arg2)

test2(arg1, arg2)

print("After executing test 2 =>", arg1, arg2)

After execution, the above code prints the following.

After executing test 1 => 10 ['Python', 2, 3, 4]

After executing test 2 => 10 ['Python', 2, 3, 4]

### Example: How To Avoid Changing The Mutable Argument

def test1(a, b) :

a = 'Garbage'

b[0] = 'Python'

arg1 = 10

arg2 = [1, 2, 3, 4]

print("Before test 1 =>", arg1, arg2)

test1(arg1, arg2[:]) # Create an explicit copy of mutable object

# 'y' in the function.

# Now 'b' in test1() refers to a

# different object which was initially a

# copy of 'arg2'

print("After test 1 =>", arg1, arg2)

After execution, the above code prints the following.

Before test 1 => 10 [1, 2, 3, 4]

After test 1 => 10 [1, 2, 3, 4]

[**TOC**](https://www.techbeamers.com/python-function/#toc)

### Standard Arguments

The standard arguments are those which you pass as specified in a Python function definition. It means without changing their order and without skipping any of them.

def fn(value):

print(value)

return

fn()

Executing the above code throws the below error as we’ve not passed the single argument required.

TypeError: fn() missing 1 required positional argument: 'value'

### Keyword-Based Arguments

When you assign a value to the parameter (such as param=value) and pass to the function (like fn(param=value)), then it turns into a keyword argument.

If you pass the keyword arguments to a function, then Python determines it through the parameter name used in the assignment.

See the below example.

def fn(value):

print(value)

return

fn(value=123) # output => 123

fn(value="Python!") # output => Python!

While using keyword arguments, you should make sure that the name in the assignment should match with the one in the function definition. Otherwise, Python throws the TypeError as shown below.

fn(value1="Python!") # wrong name used in the keyword argument

The above function call causes the following error.

TypeError: fn() got an unexpected keyword argument 'value1'

### Arguments With Default Values

Python functions allow setting the default values for parameters in the function definition. We refer them as the default arguments.

The callee uses these default values when the caller doesn’t pass them in the function call.

The below example will help you clearly understand the concept of default arguments.

def daysInYear(is\_leap\_year=False):

if not is\_leap\_year:

print("365 days")

else:

print("366 days")

return

daysInYear()

daysInYear(True)

Here, the parameter “is\_leap\_year” is working as a default argument. If you don’t pass any value, then it assumes the default which is False.

The output of the above code is:

365 days

366 days

### Variable Arguments

You may encounter situations when you have to pass additional arguments to a Python function. We refer them as variable-length arguments.

The Python’s print() is itself an example of such a function which supports variable arguments.

To define a function with variable arguments, you need to prefix the parameter with an asterisk (\*) sign. Follow the below syntax.

def fn([std\_args,] \*var\_args\_tuple ):

"""docstring"""

function\_body

return\_statement

Check out the below example for better clarity.

def inventory(category, \*items):

print("%s [items=%d]:" % (category, len(items)), items)

for item in items:

print("-", item)

return

inventory('Electronics', 'tv', 'lcd', 'ac', 'refrigerator', 'heater')

inventory('Books', 'python', 'java', 'c', 'c++')

The output of the above code goes like this.

Electronics [items=5]: ('tv', 'lcd', 'ac', 'refrigerator', 'heater')

- tv

- lcd

- ac

- refrigerator

- heater

Books [items=4]: ('python', 'java', 'c', 'c++')

- python

- java

- c

- c++

Please note that you can choose to have a formal argument or not in the function definition along with the variable arguments.

You may choose to skip the variable arguments while calling the function. In such a case, the tuple would remain empty.

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## Local Variables Inside A Function

A local variable has visibility only inside a code block such as the function def.

They are available only while the function is executing.

Check out the below example of using local variables.

def fn(a, b) :

temp = 1

for iter in range(b) :

temp = temp\*a

return temp

print(fn(2, 4))

print(temp) # error : can not access 'temp' out of scope of function 'fn'

print(iter) # error : can not access 'iter' out of scope of function 'fn'

In this example, we try to access local variables outside the function body which results in the NameError.

Function’s local variables don’t retain values between calls. The names used inside a def do not conflict with variables outside the def, even if you’ve used the same names elsewhere.

In Python, the variables assignment can occur at three different places.

* Inside a def – it is local to that function
* In an enclosing def – it is nonlocal to the nested functions
* Outside all def(s) – it is global to the entire file

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## Global Variables In A Function

The global keyword is a statement in Python. It enables variable (names) to retain changes that live outside a def, at the top level of a module file.

In a single global statement, you can specify one or more names separated by commas.

All the listed names attach to the enclosing module’s scope when assigned or referenced within the function body.

Check the below example.

x = 5

y = 55

def fn() :

global x

x = [3, 7]

y = [1, 33, 55]

# a local 'y' is assigned and created here

# whereas, 'x' refers to the global name

fn()

print(x, y)

In the above code, ‘x’ is a global variable which will retain any change in its value made in the function. Another variable ‘y’ has local scope and won’t carry forward the change.

Let’s now see how a globally declared name behaves in two different Python functions.

foo = 99

def fn1() :

foo = 'new' # new local foo created

def fn2() :

global foo

foo = 'update' # value of global foo changes

In the next example, let’s see how global behaves with the import statement.

Here, we have got the following three scripts:

* **mod\_global.py:** It contains the global definition and a function changing and displaying values.
* **test1.py:** It imports the first file and accesses the global variable.
* **test2.py:** It is using the “from” clause to import the first file and accessing the global variable.

# mod\_global.py

def fn1() :

global x

x = [1,2] ; y = [20, 200]

# a local 'y' is created – availableonly within 'f1'

# 'x' can be accessed anywhere after a call to 'f1'

fn1()

try :

print(x, y) # name 'y' is not defined – error

except Exception as ex:

print('y ->', ex)

print('x ->', x)

# test1.py

import mod\_global

print('test1 ->', mod\_global.x)

# test2.py

from mod\_global import \*

print('test2 ->', x)

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## Name Resolution In A Python Function

It is essential to understand how name resolution works in case of a def statement.

Here are a few points you should keep in mind.

* The name assignments create or change local names.
* The LEGB rule comes in the picture for searching the name reference.
  + local – L
  + then enclosing functions (if any) – E
  + next comes the global – G
  + and the last one is the built-in – B

To gain more understanding, run through the below example.

#var = 5

def fn1() :

#var = [3, 5, 7, 9]

def fn2() :

#var = (21, 31, 41)

print(var)

fn2()

fn1() # uncomment var assignments one-by-one and check the output

print(var)

After uncommenting the first “var” assignment, the output is:

5

5

Next, after uncommenting the second “var” assignment as well, the output is:

[3, 5, 7, 9]

5

Finally, if we uncomment the last “var” assignment, then the result is as follows.

(21, 31, 41)

5

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## Scope Lookup In Functions

Python functions can access names in all available enclosing def statements.

Check the below example.

X = 101 # global scope name - unused

def fn1():

X = 102 # Enclosing def local

def fn2():

print(X) # Reference made in nested def

fn2() # Prints 102: enclosing def local

fn1()

The scope lookup remains in action even if the enclosing function has already returned.

def fn1():

print('In fn1')

X = 100

def fn2():

print('In fn2')

print(X) # Remembers X in enclosing def scope

return fn2 # Return fn2 but don't call it

action = fn1() # Make, return function

action() # Call fn2() now: prints 100

The output is as follows.

In fn1

In fn2

100

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## Return Values From A Python Function

In Python functions, you can add the “return” statement to return a value.

Usually, the functions return a single value. But if required, Python allows returning multiple values by using the collection types such as using a tuple or list.

This feature works like the call-by-reference by returning tuples and assigning the results back to the original argument names in the caller.

def returnDemo(val1, val2) :

val1 = 'Windows'

val2 = 'OS X'

return val1, val2 # return multiple values in a tuple

var1 = 4

var2 = [2, 4, 6, 8]

print("before return =>", var1, var2)

var1, var2 = returnDemo(var1, var2)

print("after return =>", var1, var2)

The above code gives the following output.

before return => 4 [2, 4, 6, 8]

after return => Windows OS X

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## Function Examples

### General Function

Check out a general function call example.

def getMin(\*varArgs) :

min = varArgs[0]

for i in varArgs[1:] :

if i < min :

min = i

return min

min = getMin(21, -11, 17, -23, 6, 5, -89, 4, 9)

print(min)

The output is as follows.

-89

### Recursive Function

Next is an example of the recursive function.

def calcFact(num) :

if(num != 1) :

return num \* calcFact(num-1)

else :

return 1

print(calcFact(4))

The output is as follows.

24

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## Python Functions As Objects

Yes, Python treats everything as an object and functions are no different.

**You can assign a function object to any other names.**

def testFunc(a, b) : print('testFunc called')

fn = testFunc

fn(22, 'bb')

The output is:

testFunc called

**You can even pass the function object to other functions.**

def fn1(a, b) : print('fn1 called')

def fn2(fun, x, y) : fun(x, y)

fn2(fn1, 22, 'bb')

The output is:

fn1 called

**You can also embed a function object in data structures.**

def fn1(a) : print('fn1', a)

def fn2(a) : print('fn2', a)

listOfFuncs = [(fn1, "First function"), (fn2, "Second function")]

for (f, arg) in listOfFuncs : f(arg)

The output is:

fn1 First function

fn2 Second function

**You can return a function object from another function.**

def FuncLair(produce) :

def fn1() : print('fn1 called')

def fn2() : print('fn2 called')

def fn3() : print('fn3 called')

if produce == 1 : return fn1

elif produce == 2 : return fn2

else : return fn3

f = FuncLair(2) ; f()

The output is:

fn2 called

[**TOC**](https://www.techbeamers.com/python-function/#toc)

## Function Attributes

Python functions also have attributes.

* You can list them via the dir() built-in function.
* The attributes can be system-defined.
* Some of them can be user-defined as well.
* The dir() function also lists the user-defined attributes.

def testFunc():

print("I'm just a test function.")

testFunc.attr1 = "Hello"

testFunc.attr2 = 5

testFunc()

print(dir(testFunc))

The output is:

I'm just a test function.

['\_\_annotations\_\_', '\_\_call\_\_', '\_\_class\_\_', '\_\_closure\_\_', '\_\_code\_\_', '\_\_defaults\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_get\_\_', '\_\_getattribute\_\_', '\_\_globals\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_kwdefaults\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_name\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_qualname\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'attr1', 'attr2']

You can utilize the function attributes to archive state information instead of using any of the globals or nonlocals names.

Unlike the nonlocals, attributes are accessible anywhere the function itself is, even from outside its code.

# Python Lambda – How to Create Lambda Functions

[Python Tutorials](https://www.techbeamers.com/python-programming-tutorials/) | By Meenakshi Agarwal

This tutorial walks you through the Python lambda operator a.k.a anonymous function. You’ll learn how to create a lambda function, use it with lists, dictionaries, map() and filter().

In Python, you have a couple of ways to make [**functions**](https://www.techbeamers.com/python-function/):

**a)** **Use Def keyword:** It creates a function object and assigns it to a name.

**b)** **Use lambda:** It creates an inline function and returns it as a result.

A lambda function is a lightweight anonymous function. It can accept any number of arguments but can only have a single expression.

Let’s learn more about the Python lambda.

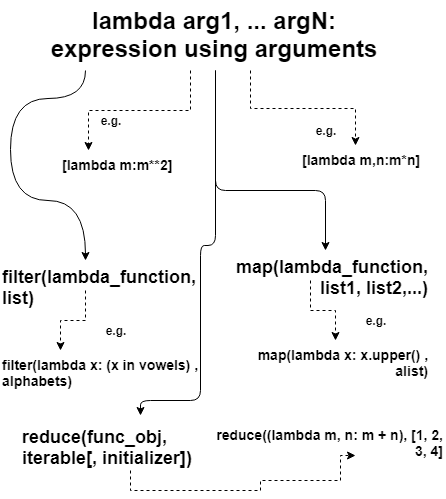
**Table of Content**

* [**What is lambda in Python?**](https://www.techbeamers.com/python-lambda/#what)
* [**How to create a lambda function?**](https://www.techbeamers.com/python-lambda/#how-to-create)
  + [**Syntax**](https://www.techbeamers.com/python-lambda/#syntax)
  + [**Example**](https://www.techbeamers.com/python-lambda/#example)
* [**Extending lambda functions**](https://www.techbeamers.com/python-lambda/#extend)
  + - [**Map functions over iterables – map()**](https://www.techbeamers.com/python-lambda/#lambda-map)
    - [**Select items in iterables – filter()**](https://www.techbeamers.com/python-lambda/#lambda-filter)
    - [**Aggregate items in iterables – reduce()**](https://www.techbeamers.com/python-lambda/#lambda-reduce)

## What Is Lambda In Python?

Lambda is an unnamed function. It provides an expression form that generates function objects.

This expression form creates a function and returns its object for calling it later.



[**TOC**](https://www.techbeamers.com/python-lambda/#toc)

## How To Create A Lambda Function?

### Syntax

It has the following signature:

lambda arg1, arg2, ... argN: expression using arguments

The body of a lambda function is akin to what you put in a def body’s return statement. The difference here is that the result is a typed-expression, instead of explicitly returning it.

Please note that a lambda function can’t include any statements. It only returns a function object which you can assign to any variable.

The lambda statement can appear in places where the def is not allowed. For example – inside a list literal or a function call’s arguments, etc.

### Example

**lambda inside a list :**

alist = [lambda m:m\*\*2, lambda m,n:m\*n, lambda m:m\*\*4]

print(alist[0](10), alist[1](2, 20), alist[2](3)) # Output: 100 40 81

**lambda inside a dictionary :**

key = 'm'

aDict = {'m': lambda x:2\*x, 'n': lambda x:3\*x}

print(aDict[key](9)) # Output: 18

[**TOC**](https://www.techbeamers.com/python-lambda/#toc)

## Extending Python Lambda Functions

We can extend the utility of lambda functions by using it with the filter and map functions.

It is possible by passing the lambda expression as an argument to another function. We refer to these methods as higher-order functions as they accept function objects as arguments.

Python provides two built-in functions like filter(), map() which can receive lambda functions as arguments.

### Map Functions Over Iterables – Map()

The map() function lets us call a function on a collection or group of iterables.

We can also specify a Python lambda function in the map call as the function object.

The map() function has the following signature.

map(function\_object, iterable1, iterable2,...)

**It expects variable-length arguments:** first is the **lambda function object**, and rest are the**iterables such a list, dictionary**, etc.

**What does the map() function do?**

The map function iterates all the lists (or dictionaries etc.) and calls the lambda function for each of their element.

**What does the map() function return?**

The output of map() is a list which contains the result returned by the lambda function for each item it gets called.

Below is a simple example illustrating the use of map() function to convert elements of lists into uppercase.

# Python lambda demo to use map() for adding elements of two lists

alist = ['learn', 'python', 'step', 'by', 'step']

output = list(map(lambda x: x.upper() , alist))

# Output: ['LEARN', 'PYTHON', 'STEP', 'BY', 'STEP']

print(output)

Let’s have another example illustrating the use of map() function for adding elements of two lists.

# Python lambda demo to use map() for adding elements of two lists

list1 = [1, 2, 3, 4]

list2 = [100, 200, 300, 400]

output = list(map(lambda x, y: x+y , list1, list2))

# Output: [101, 202, 303, 404]

print(output)

[**TOC**](https://www.techbeamers.com/python-lambda/#toc)

### Select Items In Iterables – Filter()

The filter() function selects an iterable’s (a list, dictionary, etc.) items based on a test function.

We can also filter a list by using the Python lambda function as the function object.

The filter function has the following signature.

filter(function\_object, list)

**It expects two parameters:** first is the **lambda function object** and the second is a **list**.

**What does the filter() function do?**

The filter function iterates the list and calls the lambda function for each element.

**What does the filter() function return?**

It returns a final list containing items for which the lambda function evaluates to True.

Below is a simple example illustrating the use of filter() function to determine vowels from the list of alphabets.

# Python lambda demo to filter out vowles from a list

alphabets = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

vowels = ['a', 'e', 'i', 'o', 'u']

output = list(filter(lambda x: (x in vowels) , alphabets))

# Output: ['a', 'e', 'i']

print(output)

[**TOC**](https://www.techbeamers.com/python-lambda/#toc)

### Aggregate Items In Iterables – Reduce()

The reduce method continuously applies a function on an iterable (such as a list) until there are no items left in the list. It produces a non-iterable result, i.e., returns a single value.

This method helps in aggregating data from a list and returning the result. It can let us do a rolling calculation over successive pairs of values in a sequence.

We can also pass a Python lambda function as an argument to the reduce method.

The reduce() function has the following syntax.

reduce(func\_obj, iterable[, initializer])

Below is a simple example where the reduce() method is computing the sum of elements in a list.

from functools import reduce

def fn(m, n) : return m + n

print(reduce((lambda m, n: m + n), [1, 2, 3, 4]))

print(reduce(fn, [1, 2, 3, 4]))

After executing the above code, you see the following output.

10

10

**পাইথনে ফাঙ্কশন কি? What is function in Python?**

Python এ function হচ্ছে Program এর এমন একটা code block যা একটা নির্দিষ্ট কার্য সম্পাদনের এবং তা বার বার ব্যবহারের জন্যে ব্যবহৃত হয়। অর্থাৎ function এর মধ্যে এমন কিছু Instruction বা নির্দেশাবলী থাকে যা function ব্যবহারকারী থেকে এক বা একাধিক input নিয়ে এক বা একাধিক output বা result রিটার্ন করার জন্য ব্যবহৃত হয়। তবে অনেক সময় ইনপুট ছাড়াও আমরা আউটপুট রিটার্ন করতে পারি।

ফাঙ্কশনের মাধ্যমে প্রোগ্রামের যে অংশগুলো বার বার আসে সেগুলোকে আমরা পুনরায় ব্যবহারযোগ্য করতে পারি ।

পাইথনে ফাংশনকে অনেকটা আমরা একটা ছোট্ট মেশিনের সাথে তুলনা করতে পারি, অর্থাৎ একবার মেশিনটি তৈরি করবো আর বার বার ব্যবহার করে কিছু একটা কাজ করব বা জিনিষ তৈরি করবো। কিছু তৈরি করতে হলে মেশিনে কিছু ইনপুট দিতে হতে পারে। আবার কিভাবে তৈরি করবে সেটাও মেশিনের মধ্যে যন্ত্রপাতি বসিয়ে সেটআপ করতে হবে। এভাবে বাস্তবে একটা মেশিন তৈরি করাকেই প্রোগ্রামের মধ্যে ফাংশন ডিফাইন করা বলা যেতে পারে।

পাইথনে সাধারণত, program page লোড হওয়ার সাথে সাথেই ডিক্লেয়ার করা function execute হয়না, যতক্ষণ না function কে call করা না হয়।

**পাইথনে function কত প্রকার ?**

দুই প্রকার :

* Python Pre-built function বা Python language কর্তৃক পূর্ব নির্ধারিত বিভিন্ন library function বা library ছাড়া function
* User/Programmer defined function বা Programmer কর্তৃক নির্ধারিত function

**পাইথনে function কিভাবে লিখতে হয়?**

* পাইথনে ফাংশন ডিক্লেয়ার করার জন্য def কি-ওয়ার্ডটি দিয়ে শুরু করতে হবে।
* def কি-ওয়ার্ডটির পর ফাঙ্কশনের নামটি letter অথবা underscore দিয়ে শুরু করতে হয়।
* ফাঙ্কশনের নামের শেষে একজোড়া প্রথম বন্ধনী () এরপর একটি colon (:) দিতে হবে।
* ফাংশনকে কল করার বা ব্যবহার করার আগেই সেই ফাংশনকে প্রোগ্রামে ডিফাইন করতে হবে।
* “””docstring””” অর্থাৎ function এর মধ্যে function এর documentation লেখার জন্য ব্যবহৃত হয়।
* পাইথনে function Case sensitive অর্থাৎ একই নামে কোনো ফাঙ্কশন lowercase এবং uppercase এর কারণে আলাদা function হিসেবে বিবেচিত হবে।

**পাইথনে function লেখার Syntax কি ?**

|  |  |
| --- | --- |
| 1  2  3 | def function\_name(parameters):      """docstring"""      statement(s) |

চলুন syntax অনুসারে একটা function তৈরী করে ফেলি :

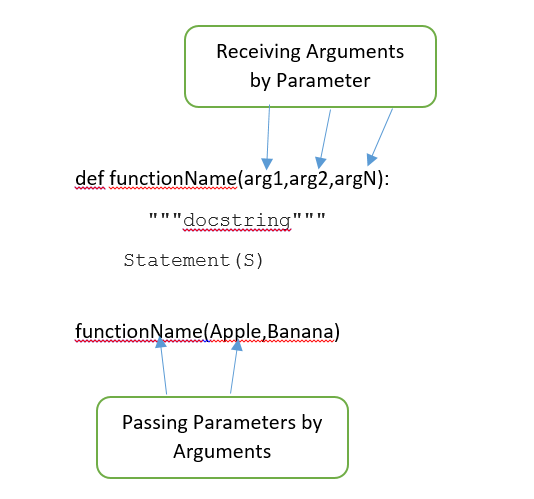
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | def greet(name):      """This function greets to      the person passed in as      parameter"""      print("Hello, " + name + ". Good morning!");    def Greet(name):      """This function Greets to      the person passed in as      parameter"""      print("Hello, " + name + ". Good morning!");    Greet("Masud");  greet("Sohel"); |

**Output**

|  |  |
| --- | --- |
| 1  2 | Hello, Masud. Good morning!  Hello, Sohel. Good morning! |

**পাইথনে function এর মধ্যে Parameter কি এবং argument কি ?**

function declaration অথবা define এর সময় parentheses এর মধ্যে ব্যবহৃত একেকটি variable কে একেকটি function parameter বলা হয়। একই ভাবে function calling এর সময় parentheses এর মধ্যে ব্যবহৃত একেকটি variable কে একেকটি function argument বলা হয়।function parameter এবং arguments কিভাবে কাজ করে তা বাজার জন্য নিচের ছবিটি লক্ষ্য করুন :



*function Parameter and arguments*

চলুন function parameter এবং arguments দিয়ে একটা function define এবং call করি।

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def getSum(num1, num2):    sum = num1 + num2;    print("Sum of the two numbers ",num1," and ",num2,"  is : ", sum);        getSum(10, 20); |

**Output**

|  |  |
| --- | --- |
| 1 | Sum of the two numbers  10  and  20   is :  30 |

**পাইথনে functions এর arguments গুলোকে কয় ভাবে receive বা গ্রহণ করা যায়?**

পাইথনে দুই ভাবে argument গুলোকে receive বা গ্রহণ করা যায় :

* সরাসরি function এর ভিন্ন ভিন্ন Parameter দিয়ে ভিন্ন ভিন্ন argument গুলো রিসিভ করা যায়।
* function এর একটা নির্দিষ্ট Parameter দিয়ে সবগুলো argument গুলো রিসিভ করা যায়।

এবার চলুন একটা উদাহরণের মাধ্যমে দেখা যাক কিভাবে সরাসরি function এর ভিন্ন ভিন্ন Parameter দিয়ে ভিন্ন ভিন্ন argument গুলো রিসিভ করা যায়:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def getSum(num1, num2):    sum = num1 + num2;    print("Sum of the two numbers ",num1," and ",num2,"  is : ", sum);        getSum(10, 20); |

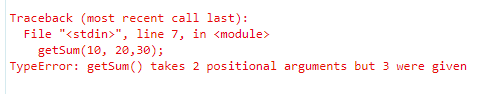
**Output**

|  |  |
| --- | --- |
| 1 | Sum of the two numbers  10  and  20   is :  30 |

লক্ষ্য করুন উপরের উদাহরণে আমরা num1 এবং num2 parameter এর জন্য দুটি আলাদা Argument যথাক্রমে 10 এবং 20 পাঠিয়েছি, যদি কম অথবা বেশি পাঠাই তাহলে নিচের মতো error প্রদর্শন করবে:

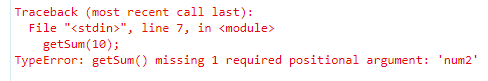
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def getSum(num1, num2):    sum = num1 + num2;    print("Sum of the two numbers ",num1," and ",num2,"  is : ", sum);        getSum(10, 20,30); |

**Output**



*python argument missing error*

আবার যদি কম পাঠান, তাহলে নিচের মতো error আসবে:



*python argument missing error 2*

তবে আপনি চাইলে নিচের মতো করে একটা ডিফল্ট ভ্যালু সেট করে দিতে পারেন :

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def getSum(num1, num2=20):    sum = num1 + num2;    print("Sum of the two numbers ",num1," and ",num2,"  is : ", sum);      getSum(10); |

তবে আপনি চাইলে প্যারামিটার এর নাম ধরেও argument pass করতে পারেন। চলুন একটা উদাহরণে বুঝে নেয়া যাক:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def getSum(num1, num2):    sum = num1 + num2;    print("Sum of the two numbers ",num1," and ",num2,"  is : ", sum);        getSum(num2=30, num1=20); |

**Output**

|  |  |
| --- | --- |
| 1 | Sum of the two numbers  20  and  30   is :  50 |

তবে এক্ষেত্রে সমস্যা হচ্ছে সবগুলো প্যারামিটার এর নাম ধরেই পাঠাতে হয়, আবার সুবিধা হয় আপনি চাইলে প্যারামিটার swaping করতে পারবেন। লক্ষ্য করুন আমরা দ্বিতীয় প্যারামিটার এর ভ্যালু প্রথমে পাঠিয়েছি।

**পাইথনে function এর একটা নির্দিষ্ট Parameter দিয়ে সবগুলো argument গুলো রিসিভ করার নিয়ম :**

পাইথনে function এর একটা নির্দিষ্ট Parameter দিয়ে সবগুলো argument গুলো রিসিভ করতে হলে আপনাকে ফাঙ্কশনের parameter এর সামনে একটা “\*” star দিতে হবে। চলুন একটা উদাহরণে বুঝে নেয়া যাক:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def greet(\*names):     """This function greets all     the person in the names tuple."""       # names is a tuple with arguments     for name in names:         print("Hello",name)    greet("Monir","Minhaz","Iqbal","Sharif") |

**Output**

|  |  |
| --- | --- |
| 1  2  3  4 | Hello Monir  Hello Minhaz  Hello Iqbal  Hello Sharif |

**পাইথনে Variable Scope কি?**

পাইথনে আপনি আপনার সমস্ত কোড জুড়ে যেকোনো জায়গায় Variable Declare বা ঘোষণা করতে পারেন। কোনো Variable যদি আপনি function এর ভিতরে ঘোষণা করেন, তাহলে ঐ Variable এর ব্যবহার Function এর মধ্যেই সীমাবদ্ধ রাখতে হবে। পাইথনে এইটাকে বলা হয় Local Scope, আবার যদি কোনো Variable ফাঙ্কশনের বাহিরে ঘোষণা করেন। তাহলে ঐ Variable সমস্ত কোড জুড়ে ব্যবহার করতে পারবেন এমনকি Funciton এর ভিতরেও ব্যবহার করতে পারবেন।

পাইথনে Variable Scope দুই ধরনের :

* local
* global

প্রথমে চলুন পাইথনে variable এর global scope এর উদহারণ দেখা যাক:

|  |  |
| --- | --- |
| 1  2  3  4  5 | x = 10  def myFunc():      print("Value inside function:",x)    myFunc() |

**Output**

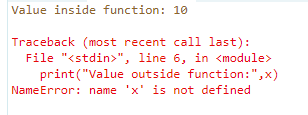
Value inside function: 10

**লক্ষ্য করুন:** আমাদের variable ফাঙ্কশনের বাহিরে ডিক্লেয়ার করা তারপর ও এটা ফাঙ্কশনের ভিতরে ব্যবহার করা গেছে , এবং ফলাফল ও আসছে।

**এবার চলুন পাইথনে variable এর local scope এর উদহারণ দেখা যাক:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def myFunc():      x = 10      print("Value inside function:",x)    myFunc()  print("Value outside function:",x) |

**Output**



*python variable localscope*

তবে আপনি চাইলে variable কে ফাঙ্কশনের মধ্যে global ডিক্লেয়ার করে , ফাঙ্কশনের বাহিরে variable কে ব্যবহার করতে পারেন। চলুন উদাহরণ দিয়ে বুঝে নেয়া যাক:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def myFunc():      global x      x=20      print("Value inside function:",x)    myFunc()  print("Value outside function:",x) |

**output**

|  |  |
| --- | --- |
| 1  2 | Value inside function: 20  Value outside function: 20 |

**পাইথনে Anonymous এবং lambda function কি?**

Anonymous শব্দের অর্থ হচ্ছে নাম বিহীন বা বেনামী, আর anonymous function মানে হচ্ছে নাম বিহীন function বা বেনামী function, সুতরাং পাইথনে যখন কোনো function declare বা ঘোষণা করা হয়, কিন্তু function টির কোনো নাম থাকেনা, পাইথনে সেই function কে anonymous function বলে। আর এই anonymous ফাঙ্কশনকেই আবার lambda function বলে।

সাধারণতঃ পাইথনে function declare করা হয় def কীওয়ার্ড ব্যবহার করে, কিন্তু পাইথনে anonymous ফাংশনগুলি lambda কীওয়ার্ড ব্যবহার করে declare করা হয়।

**পাইথন মধ্যে Lambda/anonymous ফাঙ্কশনের সিনট্যাক্স কি?**

|  |  |
| --- | --- |
| 1 | lambda arguments: expression |

এবার চলুন একটা উদাহরণ দিয়ে বুঝে নেয়া যাক:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # Program to show the use of lambda functions    double = lambda x: x \* 2    # Output: 10  print(double(5)) |

**Output**

10

উপরের ফাঙ্কশনটি নাম যুক্ত অবস্থা ঠিক এই রকম ছিল :

|  |  |
| --- | --- |
| 1  2  3 | def double(x):     return x \* 2  print(double(5)) |

# পাইথন প্রোগ্রামিং এ ফাংশন

[April 13, 2015](https://jakir.me/python-functions/)  [জাকির হোসাইন](https://jakir.me/author/jakir/)  [পাইথন প্রোগ্রামিং](https://jakir.me/category/python/)

এর আগে আমরা বিভিন্ন ফাংশন যেমন print, input ইত্যাদি ফাংশন ব্যবহার করেছি। এগুলো প্রোগ্রামিং ল্যাঙ্গুয়েজের এর সাথে দিয়ে দেওয়া হয়েছে যেন আমরা সহজেই প্রোগ্রাম লিখতে পারি। এ অধ্যায় শিখব কিভাবে নিজের প্রয়োজন মত ফাংশন লিখে ফেলা যায়।  
ফাংশন হচ্ছে পুনরায় ব্যবহার যোগ্য কোড ব্লক। যা একটি নির্দিষ্ট কাজ করতে পারে। ফাংশন ভালো ভাবে জানলেই প্রোগ্রামিং এর একটা বিশাল অংশ শেখা শেষ হয়ে যায়। লেখা যায় নিজের ইচ্ছে মত কোড।

**ফাংশন লেখার নিয়মঃ**

একটা ফাংশন নিচের মত করে লেখা হয়ঃ

|  |  |
| --- | --- |
| 1  2  3 | def function-name(Parameters):      statements      return |

এখানে  
• def দিয়ে ফাংশন শুরু করা হয়।  
• ফাংশনের তো একটা নাম থাকতে হবে তাই না? যে নাম দিয়ে ফাংশনটিকে ডাকতে হবে। function-name হচ্ছে ফাংশনের নাম।  
• Parameters হচ্ছে ফাংশন দিয়ে কি কি ডেটা পাস করবে। এখানে এক বা একাদিক Parameter পাস করা যেতে পারে। কোন কোন ফাংশনে কোন Parameter নাও থাকতে পারে। এটা নির্ভর করে কি ধরনের ফাংশন লিখা হচ্ছে তার উপর। একের অধিক Parameter থাকলে তাদেরকে কমা দিয়ে লিখতে হয়।  
• কাজ শেষে ফাংশনটি কি রিটার্ন করবে তাই return দিয়ে পাস করা হয়। ফাংশন যদি কোন কিছু রিটার্ন না করে, তাহলে return 0 বা না লিখলেও হবে।  
যেমন আমরা hello নামে একটা ফাংশন লিখব, যেটাকে কল করলে Hello World! প্রিন্ট করবে।

|  |  |
| --- | --- |
| 1  2 | def hello():      print ("Hello world!") |

উপরের ফাংশনে কোন প্যারামিটার নেই, এবং ফাংশনটি কোন কিছু রিটার্ণ ও করবে না। শুধু Hello World! প্রিন্ট করবে। ফাংশন লেখার পর তা ব্যবহার করার জন্য কল করতে হয়।

|  |  |
| --- | --- |
| 1 | hello() |

কল করলে ফাংশনটি এক্সিকিউট হবে। সম্পুর্ণ প্রোগ্রামঃ

|  |  |
| --- | --- |
| 1  2  3  4 | def hello():      print ("Hello world!")    hello() |

আমরা যত বার ইচ্ছে ততবার ফাংশটি কল করতে পারি। এটাই হচ্ছে ফাংশনের মূল সুবিধে। কোড গুলো বার বার না লেখে শুধু আমরা কল করব। আর তখন ফাংশনটি এক্সিকিউট হয়ে আমরা যা চাই, তা করে দিবেঃ

|  |  |
| --- | --- |
| 1  2  3  4  5 | def hello():   print ("Hello world!")  hello()  hello()  hello() |

প্রোগ্রামটি রান করলে তিনবার Hello World! প্রিন্ট করবে।

**ফাংশনের প্যারামিটার**

আমাদের এমন ফাংশন লিখতে হতে পারে, যেখানে ফাংশনে কিছু ডেটা পাঠালে ঐ ডেটা আমাদের প্রসেস করে দিবে। আর এই ডেটা আমরা পাঠাবো প্যারামিটার হিসেবে।  
আমরা add নামে একটা ফাংশন লিখব। যেটা প্যারামিটার হিসেবে দুইটা নাম্বার নিবে এবং রিটার্ন করবে নাম্বার দুটির যোগফলঃ

|  |  |
| --- | --- |
| 1  2  3 | def add(a,b):      return (a+b)  print (add(4,7)) |

এখানে add ফাংশনে আমরা দুইটা সংখ্যা প্যারামিটার হিসেবে দিচ্ছি। ফাংশনটি আমাদের ফেরত দিচ্ছে সংখ্যা দুইটির যোগফল।

যখন আমরা ফাংশনটি কল করেছি, তখন সংখ্যা দুটি পাস করে দিয়েছি, যে সংখ্যা দুইটির যোগ ফল আমরা পেতে চাই, add(4,7) দিয়ে। ফাংশনটি আমাদের ঐ সংখ্যা দুটি নিয়ে তা যোগ করে তা ফেরত দিয়েছে, return (a+b) দিয়ে। পরে তা প্রিন্ট করেছি।

এখন আমরা যতবার ইচ্ছে ততবার add ফাংশনটি কল করতে পারি। যাতে দুইটি সংখ্যা প্যারামিটার হিসেবে দিলে তা যোগ করে আমাদের পাঠিয়ে দিবে। যেন আমরা তা প্রিন্ট করতে পারি। আলাধা করে যোগ করতে হচ্ছে না।

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def add(a,b):      return (a+b)    print (add(4,7))  print (add(89,732))  print (add(55,999)) |

যা আমাদের আউটপুট দিবেঃ  
11  
821  
1054  
**ডিফল্ট প্যারামিটারঃ**  
কোন ফাংশন লেখার সময় আমরা তার ডিফল্ট প্যারামিটার সেট করে দিতে পারি। ডিফল্ট প্যারামিটার সেট করে দেওয়া হলে যদি ফাংশন কল করার সময় ঐ প্যারামিটারের আইটেমের জন্য কোন ভ্যালু সেট করে দেওয়া না হয়, তাহলে ফাংশন ডিফল্ট ভ্যালুটি ব্যবহার করবে।

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def max(a,b=0):        if(a>b):          return format(a) +" is greater than " + format(b)      elif (a<b):          return format(a) + " is less than " + format(b)      elif(a==b):          return format(a) + " and " + format(b) + " both are same"    print(max(6,7))  print(max(9))  print(max(5,5)) |

যা আউটপুট দিবেঃ  
6 is less than 7  
9 is greater than 0  
5 and 5 both are same  
উপরের প্রোগ্রামটি দেখি। আমরা দুইটা সংখ্যার মধ্যে বড় ছোট নির্ণয়ের জন্য একটা ফাংশন লিখেছি। যখন ফাংশনে দ্বিতীয় প্যারামিটার হিসেবে কোন ভ্যালু পাঠাই নি, তখন ডিফল্ট ভ্যালু 0 ব্যবহার করেছে।

**ফাংশন থেকে একাধিক ভ্যালু রিটার্ন**  
এর আগের প্রোগ্রামে দেখেছি ফাংশন থেকে কিভাবে একটা ভ্যালু রিটার্ন করা যায়। পাইথনে ফাংশন থেকে একাধিক ভ্যালু রিটার্ন করা যায়। নিচের প্রোগ্রামটি দেখিঃ

|  |  |
| --- | --- |
| 1  2  3  4 | def binary(a, b):      return bin(a), bin(b)    print(binary(12, 14)) |

এখানে একটি ফাংশন লিখেছি, যেখানে দুইটা সংখ্যা পাঠালে ঐ সংখ্যা দুইটির বাইনারি ভ্যালু রিটার্ণ করবে। যদিও দুইটা একসাথে টাপল আকারে প্রিন্ট করবে। আমরা চাইলে প্রতিটা রিটার্ণ ভ্যালু আলাদা আলাদা এক্সেস করতে পারিঃ

|  |  |
| --- | --- |
| 1  2  3  4 | def binary(a, b):      return bin(a), bin(b)  x,y = (binary(12, 14))  print(x) |

এখানে রিটার্ণ ভ্যালু গুলো x এবং y তে আলাদা আলাদা ভাবে এসাইন হবে। আমরা এরপর যে কোন ভ্যালু প্রিন্ট করে দেখতে পারি।

আরো দুই একটা ফাংশন দেখিঃ

আমরা একটা ফাংশন লিখব। যেটা ব্যবহারকারী কে তার নাম জিজ্ঞেস করবে। এবং তা ফাংশনে পাস করব। ফাংশন পরে ব্যবহার কারীকে হ্যালো জানাবেঃ

|  |  |
| --- | --- |
| 1  2  3  4  5 | def hello(name):      print ("Hello " + name)    your\_name = input("Enter  your Name: ")  hello(your\_name) |

উপরের প্রোগ্রামে আমরা hello নামে একটা ফাংশন লিখেছি। ব্যবহারকারী থেকে আমরা তার নাম ইনপুট নিয়েছি। এরপর hello ফাংশনে তা পাস করেছি, hello(your\_name) দিয়ে। hello ফাংশন আমাদের দেওয়া নামকে হ্যালো জানিয়েছে।  
এবার আমরা আরেকটা ফাংশন লিখব। যেটা একটি নাম্বার জোড় না বিজোড় তা বলে দিবে। ব্যবহারকারী থেকে একটা নাম্বার ইনপুট নিব। তারপর তা ফাংশনে পাস করব। ফাংশন আমাদের বলে দিবে নাম্বারটি জোড় না বিজোড়ঃ

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def checkNumber(n):      if n%2 == 0:          print ("It's Even number")      else: print ("It's Odd number")    number = int (input("Enter  a  number to check: "))  checkNumber(number) |

এভাবে আমরা আমাদের প্রয়োজন মত যে কোন ফাংশন লিখে ফেলতে পারি।

# ফাংশন

আমাদের প্রোগ্রামের যে অংশগুলো বার বার আসে সেগুলোকে আমরা পুনরায় ব্যবহারযোগ্য একক (reusable unit) হিসেবে ব্যবহার করতে পারি ফাংশনের সাহায্যে । গনিতে যেমন দেখেছি কোন ফাংশন একটি ইনপুট নিয়ে সেটার উপর বিভিন্ন ধরনের ম্যাথ করে আউটপুট দেয়, প্রোগ্রামিংএও সেই একই ব্যাপার ঘটে । আপনি এক বা একাধিক প্যারামিটার পাস করবেন একটি ফাংশনে, ফাংশনটি প্রসেস করে আপনাকে আউটপুট “রিটার্ন করবে” । তবে প্রোগ্রামিং এর ক্ষেত্রে সবসময় যে ইনপুট থাকতে হবে বা আউটপুট দিতে হবে এমন কোন কথা নেই ।

একটি ফাংশন আসলে কিছু স্টেটমেন্টের সংকলন । যখনই কোন ফাংশন কল করা হয় তখন এই ফাংশনের ভিতরে থাকা স্টেটমেন্টগুলো এক্সিকিউট করা হয় । পাইথনে আমরা ফাংশন ডিক্লেয়ার করার জন্য def কি-ওয়ার্ডটি ব্যবহার করি । আসুন দেখে নেই একটি ফাংশন:

def hello():

print("Hello World!")

প্রথমে আমরা def কি-ওয়ার্ডটি লিখেছি । তারপর ফাংশনের একটা নাম দিয়েছি – hello, এবং তারপর একজোড়া প্রথম বন্ধনী (). এরপর একটি কোলন তথা : চিহ্ন দিয়ে এর নিচে ফাংশনের আওতাভুক্ত কোড ব্লক বা ফাংশনের কাজ ডিফাইন (নির্ধারণ) করেছি। ফাংশনটির কাজ হচ্ছে “Hello world!” বাক্যটি প্রিন্ট করা।

কিন্তু এভাবে একটা ফাংশনকে প্রোগ্রামের মধ্যে শুধু ডিফাইন করে রেখে দিলে তার মধ্যের কোড বা ইন্সট্রাকশন গুলো এমনি এমনি কাজ করবে না। এর জন্য ওই ফাংশনটিকে তার নাম ধরে কল করতে হবে। কল কিভাবে করতে হয়? কিছুই না ওই ফাংশনের নামটি সাধারণ ভাবে স্টেটমেন্ট আকারে লিখলেই হয়। যেমন - আমরা print ফাংশন ব্যবহার করি যখন দরকার হয় তখন। উপরের ফাংশনটিকে কল করতে হবে নিচের মত করে,

hello()

তাই পুরো প্রোগ্রামটি যদি নিচের মত হয়,

# Defining the function named hello

def hello():

print("Hello World!")

​

# Calling the function to use it

hello()

তাহলে এই প্রোগ্রামটির আউটপুট হবে,

Hello World!

একটা ফাংশনকে প্রোগ্রামে একবার ডিফাইন করলেও সেটাকে বার বার কল বা ব্যবহার করা যাবে। অর্থাৎ নিচের মত -

# Defining the function named hello

def hello():

print("Hello World!")

​

# Calling the function to use it

hello()

​

# Again calling the function

hello()

আউটপুট,

Hello World!

Hello World!

একটি কথা মনে রাখা জরুরি। কোন ফাংশনকে কল করার বা ব্যবহার করার আগেই সেই ফাংশনকে প্রোগ্রামে ডিফাইন বা প্রোগ্রামের কাছে চিহ্নিত করতে হবে। যেমন - উপরের প্রোগ্রামটিকে যদি আমরা নিচের মত করে লিখি,

# Calling the function to use it

hello()

​

# Defining the function named hello

def hello():

print("Hello World!")

তাহলে আউটপুট আসবে,

NameError: name 'hello' is not defined

এটা সহজ ভাবে চিন্তা করলেই যৌক্তিক মনে হবে। অর্থাৎ, যদি প্রোগ্রামের শুরুতেই একটি ফাংশনকে কল বা ব্যবহার করতে চাই, তাহলে কিভাবে আমার প্রোগ্রাম জানবে যে তার মধ্যে hello নামের একটা ফাংশন আছে? বরং এটাই কি যৌক্তিক নয় যে - আগে ফাংশনটাকে প্রোগ্রামে ডিফাইন করবো এবং তার পর প্রোগ্রামের কোন এক বা একাধিক যায়গায় সেটাকে কল করবো।?

কাস্টম ফাংশনকে অনেকটা আপনার দ্বারা তৈরি একটা ছোট্ট মেশিনের সাথে তুলনা করা যায়। অর্থাৎ একবার মেশিনটি তৈরি করবো আর বার বার ব্যবহার করে কিছু একটা কাজ করব বা জিনিষ তৈরি করবো। কিছু তৈরি করতে হলে মেশিনে কিছু ইনপুট দিতে হতে পারে। আবার কিভাবে তৈরি করবে সেটাও মেশিনের মধ্যে যন্ত্রপাতি বসিয়ে সেটআপ করতে হবে। এভাবে বাস্তবে একটা মেশিন তৈরি করাকেই প্রোগ্রামের মধ্যে ফাংশন ডিফাইন করা বলা যেতে পারে।

# ফাংশন আর্গুমেন্ট

মনে আছে আমরা আগের চ্যাপ্টারে ফাংশনকে একটি ছোট্ট মেশিন হিসেবে কল্পনা করেছিলাম। যেকোনো মেশিন বা যন্ত্র যখন বানানো হয় তখন তার কাজের জন্য যেমন কিছু যন্ত্রপাতির সেটআপ দরকার হয় তেমনি সেই মেশিনে ইনপুট হিসেবে কিছু কাঁচামাল দিতে হয় যেগুলো প্রক্রিয়াজাত করে মেশিন আমাদের চাহিদা মোতাবেক জিনিষ তৈরি করে দেয় বা এর থেকে আউটপুট পাওয়া যায়। ধরে নিচ্ছি আমাদের বানানো মেশিনটির এক পাশ দিয়ে ময়দা, চিনি, দুধ, ক্রিম এসব দিলে আরেক পাশ দিয়ে সুন্দর কেক তৈরি হয়ে বের হয়। তাহলে সেই ময়দা, চিনি, দুধ, ক্রিম এসব হচ্ছে সেই মেশিনের **আর্গুমেন্ট** আর কেক বানানোর জন্য মেশিনের মধ্যে বিভিন্ন যন্ত্রের যে সেটআপ আছে সেটাকে বলা যেতে পারে **ফাংশন বডি**। আর শেষে যে সুস্বাদু কেক পাওয়া যায় তাকে বলা যেতে পারে ফাংশনের **রিটার্ন ভ্যালু**। এখন এরকম একটি মেশিন তৈরি হয়ে গেলে এই মেশিনকে যতবার ইচ্ছা ব্যবহার করা যাবে এবং এর থেকে কেক পাওয়া যাবে। কিন্তু অবশ্যই প্রতিবার সঠিকভাবে কেক পেতে হলে এই মেশিনের আর্গুমেন্ট তথা কাঁচামাল গুলো দিতে হবে।

প্রোগ্রামিং -এও একই ভাবে একটি ফাংশনের কিছু আর্গুমেন্ট থাকতে পারে যেগুলো পক্ষান্তরে ফাংশন বডির মধ্যে ব্যবহৃত হয়ে চাহিদা মোতাবেক প্রসেসড হবে। এই আর্গুমেন্ট গুলো পাঠানোর দায়িত্ব হচ্ছে তার, যে এই ফাংশনকে কল করবে বা ব্যবহার করতে চাইবে। নিচের উদাহরণটি দেখি -

def show\_double(x):

print(x\*2)

​

show\_double(2)

show\_double(100)

আউটপুট,

4

200

উপরে show\_double ফাংশনের আর্গুমেন্ট একটি। আর তাই যখনই আমরা এই ফাংশনকে কল করেছি বা ব্যবহার করতে চেয়েছি তখনি সেই ফাংশনের আর্গুমেন্ট (মেশিনের ক্ষেত্রে ইনপুট) পাঠিয়ে দিয়েছি এভাবে show\_double(2)। একবার কল করার সময় ইনপুট দিয়েছি 2 আবার আরেকবার কল করার সময় ইনপুট দিয়েছি 100 এবং আমাদের ফাংশনের কাজ হচ্ছে এর কাছে আসা যেকোনো আর্গুমেন্টকে দ্বিগুণ করে স্ক্রিনে প্রিন্ট করে। তাই দুইবারই আমাদের ফাংশন কাজটি সঠিক ভাবে করেছে।

আর্গুমেন্টকে ফাংশনের দুটি প্রথম বন্ধনীর মধ্যে ডিফাইন করতে হয়।

একটি ফাংশন কিন্তু একাধিক আর্গুমেন্ট নিয়ে কাজ করতে পারে অর্থাৎ এর একাধিক আর্গুমেন্ট থাকতে পারে। এটাই তো যৌক্তিক, তাই না? কারণ, একটি ফাংশন তথা মেশিনকে জটিল জটিল জিনিষ বানাতে বা আউটপুট দিতে তাকে অনেক গুলো ইনপুট নিয়ে কাজ করতে হতেই পারে। নিচের উদাহরণটি দেখি -

def make\_sum(x, y):

z = x + y

print(z)

​

make\_sum(5, 10)

make\_sum(500, 500)

আউটপুট,

15

1000

একটি বিষয় খেয়াল করুন, ফাংশনের আর্গুমেন্ট গুলোকে তার নিজের বডির মধ্যে একই নামের ভ্যারিয়েবল হিসেবে ব্যবহার করা যায়। যেমন উপরের উদাহরণে, make\_sum ফাংশনের কাছে দুটো আর্গুমেন্ট এসেছে x, এবং y নামে এবং এই দুটি ভ্যালুকে সে নিজের বডির মধ্যে ব্যবহার করেছে যোগ করার জন্য এবং যোগফল জমা করেছে z নামের আরেকটি ভ্যারিয়েবলে। কিন্তু এই x, y বা z কে উক্ত ফাংশনের বাইরে থেকে অ্যাক্সেস করা যাবে না বা ব্যবহার করা যাবে না। যেমন -

def make\_sum(x, y):

z = x + y

print(z)

​

make\_sum(5, 10)

print(z)

আউটপুট,

15

...

NameError: name 'z' is not defined

উপরের উদাহরণে, print(z) স্টেটমেন্টটি এরর দেখাচ্ছে কারণ z ভ্যারিয়েবলের গণ্ডি বা স্কোপ ছিল শুধুমাত্র make\_sum ফাংশনের মধ্যেই। তাই বাইরে থেকে একে অ্যাক্সেস করা যায় নি।

**মাল্টিপল প্যারামিটার হ্যান্ডলিং | আর্বিটরারি আর্গুমেন্ট লিস্ট**

মনে করুন, আপনি make\_sum ফাংশনটিতে অনেকগুলো প্যারামিটার পাঠাতে চাচ্ছেন যেমন, 10, 20, 30 ... ইত্যাদি। যদি আপনি make\_sum (a, b) হিসেবে ডিক্লেয়ার করেন তাহলে দুইটার বেশি প্যারামিটার পাঠাতে পারবেন না। সেক্ষেত্রে কোড হবে এইরকম,

def make\_sum(\*args):

sum = 0

for num in args: # Here, args is like a Tuple which is (10, 20, 30, 40)

sum += num

return sum

​

print(make\_sum(10, 20, 30, 40))

**আউটপুট**

100

# পাইথনে \* এর অর্থ

\* এর আর্গুমেন্টে ভ্যালু Tuple হিসেবে প্যাকড থাকে। এর মানে \* দিয়ে প্যারামিটার ডিক্লেয়ার করলে আমরা যেকোন সংখ্যক পজিশনাল আর্গুমেন্ট পাস করতে পারি। যেমন করলাম make\_sum এর ক্ষেত্রে। শুরুতে make\_sum মাত্র দুইটা আর্গুমেন্ট নিলেও পরবর্তীতে আমরা প্যারামিটারে \* বসিয়ে দিলাম তখন সে অনেকগুলো আর্গুমেন্ট পাস করতে পারছে।

# পাইথনে \*\* এর অর্থ

আমরা চাইলে ফাংশনের প্যারামিটারে ডাবল অ্যাস্টেরিস্কস বসিয়েও ডিক্লেয়ার করতে পারি। ডাবল স্টারের মানে হল যেকোন সংখ্যক named parameter থাকতে পারে। এই মানগুলো ডিকশনারি হিসেবে প্যাকড থাকে। নিচের উদাহরণটি লক্ষ্য করা যাক,

def print\_dict(\*args):

print (args)

​

​

print\_dict(a=1, b=2)

আউটপুট,

TypeError Traceback (most recent call last)

<ipython-input-2-9970453fce76> in <module>()

----> 1 print\_dict(a=1, b=2)

​

TypeError: print\_dict() got an unexpected keyword argument 'a'

সিঙ্গেল অ্যাস্টেরিস্কস ব্যবহার করলে আমরা নেমড আর্গুমেন্ট পাস করতে পারব না। তাই আমাদের এসব ক্ষেত্রে ডাবল অ্যাস্টেরিস্কস ব্যবহার করতে হবে, যেমন

def print\_dict(\*\*kwargs):

print(kwargs)

​

​

print\_dict(a=1, b=2, c=3)

আউটপুট,

{'a': 1, 'c': 3, 'b': 2}

আমরা যদি কোডটা আরেকটু গুছিয়ে লেখি,

def print\_dict(\*\*kwargs):

for args in kwargs:

print("{0} : {1}".format(args, kwargs[args]))

​

​

print\_dict(a=1, b=2, c=3)

আউটপুট,

a : 1

c : 3

b : 2

চাইলে আমরা মিক্সড ভ্যারিয়েডিক আর্গুমেন্ট পাঠাতে পারি। মানে একই ফাংশনে তিন ধরণের আর্গুমেন্ট, তবে খেয়াল রাখতে হবে প্যারামিটারগুলো এমন ভাবে ডিফাইন করা হয় যেন প্রথমে সাধারণ প্যারামিটার তারপরে সিঙ্গেল অ্যাস্টেরিস্কের প্যারামিটার এবং অবশেষে ডাবল অ্যাস্টেরিস্কস এর প্যারামিটার থাকে। মানে আমাদের অবশ্যই ক্রম মানতে হবে এইক্ষেত্রে।

def print\_all(a, \*args, \*\*kwargs):

print(a)

print(args)

print(kwargs)

​

​

print\_all(10, 20, 30, 50, b=5, c=10)

আউটপুট,

10

(20, 30, 50)

{'c': 10, 'b': 5}

**প্যারামিটার ও আর্গুমেন্ট**

যখন একটি ফাংশনকে ডিফাইন করা হয় তখন এর ভ্যারিয়েবল গুলোকে প্যারামিটার বলা হয়। আর যখন একটি ফাংশনকে কল করা হয় তখন সেই ফাংশনের প্যারামিটার হিসেবে যে ভ্যালু পাঠানো হয় তাকে আর্গুমেন্ট বলা হয়।

# ফাংশন রিটার্ন

এ পর্যন্ত আমরা বুঝতে পেরেছি ফাংশন কি (মেশিন), কেনই বা ফাংশন ব্যবহার করবো (কোড এর পুনব্যবহার), ফাংশনের আর্গুমেন্ট (ইনপুট) ইত্যাদি। মনে আছে, আমরা ফাংশনকে কেক বানানোর মেশিন এর সাথে তুলনা করেছিলাম? অর্থাৎ কোন একটি ফাংশন তার বডির মধ্যে কিছু কাজ করে চুপ চাপ থাকতে পারে অথবা কোন কিছু রিটার্নও দিতে পারে। কাকে রিটার্ন দিবে? যে এই ফাংশনকে কল করবে বা ব্যবহার করবে। কোথায় রিটার্ন দিবে? যেখান থেকে কল করা হবে সেখানেই রিটার্ন ভ্যালু পৌঁছে যাবে। নিচের উদাহরণ দেখলে আরও পরিষ্কার বুঝতে পারবো আমরা -

def get\_larger(x, y):

if x > y:

return x

else:

return y

​

larger\_value = get\_larger(23, 32)

print(larger\_value)

আউটপুট,

32

ফাংশনের মধ্যে return কিওয়ার্ড ব্যবহার করে ফাংশন থেকে কোন কিছু রিটার্ন করা হয়। উপরের উদাহরণে get\_larger ফাংশনের কাছে আশা দুটো আর্গুমেন্টের মধ্যে সে তুলনা করে বড়টি বের করে এবং সেটি রিটার্ন করে। আর তাই নিচে যখন = get\_larger(23, 32) স্টেটমেন্টের মাধ্যমে একে কল করা হয়েছে এবং এর চাহিদা মোতাবেক দুটো আর্গুমেন্ট পাঠিয়ে দেয়া হয়েছে তখন বস্তুত সেই ফাংশনের রিটার্ন করা ভ্যালুটি = চিহ্নের ডান পাশে এসে জমা হয়। আর যেহেতু আমরা জানি = চিহ্ন দিয়ে কোন ভ্যালুকে কোন ভ্যারিয়েবলে অ্যাসাইন করা হয়, তাই larger\_value এর মধ্যে সেই রিটার্ন করা ভ্যালু স্টোর হচ্ছে। যা ঘটেছে তা হলঃ

larger\_value = get\_larger(23, 32) # Function is done with working and returned something here

larger\_value = 32

পরিশেষে একটি সাধারণ প্রিন্ট স্টেটমেন্ট।

খুব গুরুত্বপূর্ণ একটি বিষয় হচ্ছে, যখন কোন ফাংশনের মধ্যে একটি return স্টেটমেন্ট থাকে এবং সেটি এক্সিকিউট হয়, তারপর থেকে ফাংশনের মধ্যে থাকা আর কোন কোড এক্সিকিউট হয় না। অর্থাৎ, ফাংশন তার কাজ শেষে কিছু একটা রিটার্ন করেই থেমে যায়। যেমন -

def add\_numbers(x, y):

total = x + y

return total

print("This won't be printed")

​

print(add\_numbers(4, 5))

আউটপুট,

9

উপরের প্রোগ্রামের ফাংশনটির কাজ হচ্ছে দুটো আর্গুমেন্ট ভ্যারিয়েবলকে যোগ করে নতুন একটি ভ্যারিয়েবলে রেখে সেটিকে তার caller এর কাছে রিটার্ন করা । রিটার্ন করেই সে ক্ষান্ত। এরপরে আরেকটি প্রিন্ট স্টেটমেন্ট থাকলেও সেটার কোন এক্সিকিউশন নেই। আর তাই সেই This won't be printed লাইনটিকে স্ক্রিনে দেখা যাচ্ছে না।

# কমেন্ট ও ডক স্ট্রিং

**কমেন্ট** কমেন্ট মানে মন্তব্য। কোড লেখার সময় কোডের সাথে মন্তব্যও লেখা যায়। কিন্তু সেই মন্তব্যের কথা গুলো প্রোগ্রামের মত রান হয় না। তাই এভাবে কোড সাথে কমেন্ট লিখে রাখলে অন্যের সেই কোড বুঝতে সুবিধা হয় এবং ওই কোড দিয়ে কি করা যায় এবং কিভাবে করা যায় সেটাও কোড রান করার আগেই বুঝে নেয়া যায়।

পাইথনে যেকোনো কমেন্ট লাইন লেখার আগে তার আগে # (হ্যাস বা পাউন্ড) চিহ্ন ব্যবহার করতে হয়। নিচের উদাহরণটি দেখি -

# few variables below

x = 10

y = 5

​

# make sum of the above two variables

# and store the result in z

z = x + y

​

print(z) # print the result

# print (x // y)

# another comment

উপরের প্রোগ্রামটিতে বেশ কিছু কমেন্ট আমরা লিখেছি যেগুলো পরে বোঝা যাচ্ছে এই প্রোগ্রামে কি করা হয়েছে।

**ডক স্ট্রিং** কমেন্টের মত আরও একটি জিনিষ হচ্ছে ডক স্ট্রিং বা ডকুমেন্ট স্ট্রিং। কিন্তু এর ব্যবহার একটু ভিন্ন ভাবে আরও স্পেসিফিক ভাবে করা হয়। সাধারণত মাল্টি লাইন স্ট্রিং কে কোন ফাংশনের মধ্যে শুরুতেই লিখে ওই ফাংশন সম্পর্কিত একটি মন্তব্যের ব্লক লেখা হয় যাকে ডক স্ট্রিং বলা হয়।

def greet(word):

"""

Print a word with an

exclamation mark following it.

"""

print(word + "!")

​

greet("Hello World")

সাধারণ কমেন্ট থেকে এটি একটু আলাদা। যেমন - এই স্ট্রিং গুলোকে প্রোগ্রামের রানটাইমের সময়ও অ্যাক্সেস করা যায় নিচের মত করে,

def greet(word):

"""

Print a word with an

exclamation mark following it.

"""

print(word + "!")

​

# What the fucntion does?

print(greet.\_\_doc\_\_)

​

# Make sense, now lets use it

greet("Hello World")

আউটপুট,

Print a word with an

exclamation mark following it.

​

Hello World!

উপরের প্রোগ্রামের print(greet.\_\_doc\_\_) স্টেটমেন্টটির মাধ্যমে আমরা greet ফাংশনের ডকুমেন্টেশন দেখে নিয়েছি এবং তারপর একে কল করে ব্যবহার করেছি।

**স্টাইল গাইড**

​[গুগল স্টাইল গাইড](https://github.com/google/styleguide) মোতাবেক খুব সুন্দর ভাবে একটি ফাংশনের ডক স্ট্রিং লেখা যেতে পারে নিচের মত করে,

def square\_root(n):

"""Calculate the square root of a number.

​

Args:

n: the number to get the square root of.

Returns:

the square root of n.

Raises:

TypeError: if n is not a number.

ValueError: if n is negative.

​

"""

pass

অর্থাৎ প্রথমেই ফাংশনের কাজ। তারপর তার প্যারামিটার গুলোর বর্ণনা। এরপরে সেই ফাংশনের কোন রিটার্ন ভ্যালু থাকলে তার বর্ণনা। অতঃপর এটি কোন এরর বা এক্সেপশন রেইজ করবে কিনা এ ব্যাপারটা লেখা।

ভালো প্রোগ্রামার অবশ্যই ভালো কমেন্ট এবং ডক স্ট্রিং লিখে থাকেন। তাড়াহুড়ায় এড়িয়ে যাওয়া একদম উচিৎ নয়।

# অবজেক্ট হিসেবে ফাংশন

একটা কথা বলা হয় = "পাইথনে সব কিছুই অবজেক্ট"। প্রত্যেকটি অবজেক্টেরই কিছু অ্যাট্রিবিউট ও মেথড থাকে। যেমন, আমরা যখনই একটি স্ট্রিং ভ্যারিয়েবল তৈরি করি a = "Abc" এভাবে, তখন যদি type(a) দেখার চেষ্টা করি তাহলে আউটপুট পাবো <class 'str'> অর্থাৎ এই a অবজেক্টটি str ক্লাসের একটি অবজেক্ট এবং এর কিছু অ্যাট্রিবিউট ও মেথড আছে। যেমন - lower, upper ইত্যাদি, যেগুলো ব্যবহার করে আমরা a কে নিয়ে বিভিন্ন কাজ করতে পারি।

ক্লাস, অবজেক্ট, অ্যাট্রিবিউট, মেথড নিয়ে বিস্তারিত আলোচনা আছে [অবজেক্ট ওরিয়েন্টেড সেকশনে](https://python.howtocode.com.bd/oop-python)​

ফাংশনও একটি অবজেক্ট অর্থাৎ এরও কিছু অ্যাট্রিবিউট ও মেথড আছে। যেমন একটি ফাংশনকে ডিফাইন করার সাথে সাথেই তার \_\_doc\_\_ নামের অ্যাট্রিবিউট তৈরি হয় যার মাধ্যমে একটি ফাংশনের ডক স্ট্রিং অ্যাক্সেস করা যায়। অন্যান্য সাধারণ ভ্যারিয়েবলের ভ্যালুর মত কোন একটি ফাংশনকেও একটি ভ্যারিয়েবলে অ্যাসাইন বা স্টোর করা যায়।

উদাহরণ,

def add\_explanation(line):

return line + '!'

​

update\_line = add\_explanation

​

print(update\_line("Hello World"))

আউটপুট,

Hello World!

উপরের প্রোগ্রামে প্রথমে add\_explanation ফাংশনটিকে update\_line ভ্যারিয়েবলে অ্যাসাইন করা হয়েছে। এরপর, যেখানে add\_explanation ফাংশনের দরকার পরেছে সেখানে তাকে update\_line নামে কল করা হয়েছে। এভাবে বস্তুত add\_explanation -টাই কল হচ্ছে। আরেকটু পরীক্ষা করার জন্য আমরা যদি print(update\_line) স্টেটমেন্টটি এক্সিকিউট করি তাহলে আউটপুট আসবে <function add\_explanation at 0x10dbf5668>

তাহলে একটি প্রশ্ন মাথায় আসতে পারে - যেহেতু ফাংশনকে ভ্যারিয়েবলে অ্যাসাইন করা যায় তাহলে কি ভ্যারিয়েবলের মত করে একটা ফাংশনকেও অন্য ফাংশনের আর্গুমেন্ট হিসেবে পাঠানো যাবে? উত্তর হচ্ছে হ্যাঁ। একটা উদাহরণ দেখি,

def beautify(text):

return text + '!!!'

​

​

def make\_line(func, words):

return "Hello " + func(words)

​

​

print(make\_line(beautify, "world"))

আউটপুট,

Hello world!!!

উপরের প্রোগ্রামটি একটু বিশ্লেষণ করা যাকঃ ধরে নিচ্ছি beautify ফাংশনের কাজ হচ্ছে এর কাছে যাই দেয়া হয় তার সাথে তিনটি বিস্ময় চিহ্ন যুক্ত করে রিটার্ন করে। আবার আমাদের একটি ফাংশন আছে make\_line যা দিয়ে একটি বাক্য তৈরি করা হয়। কিন্তু আমরা চাই এর মধ্যে বাক্য তৈরির সময়ই শেষ শব্দের সাথে কিছু বিস্ময় চিহ্ন জুড়ে দিতে। তো, যেহেতু বিস্ময় চিহ্ন জুড়ে দেয়ার ফাংশন আমাদের বানানোই আছে তাই ওই ফাংশনকে make\_line এর একটি আর্গুমেন্ট বা চাহিদা হিসেবে উল্লেখ করতে পারি। অর্থাৎ make\_line কে কল করতে হলে এর আর্গুমেন্ট হিসেবে একটি ফাংশন এবং একটি ডাটা (ধরে নিচ্ছি একটি শব্দ) পাঠাতে হবে। যাতে করে প্রয়োজনে সে ওই beautify ফাংশনকে তার বডির মধ্যে থেকে কল করে ব্যবহার করতে পারে।

make\_line ফাংশনের ডেফিনেশনে এর কাছে আসা ফাংশনকে func নামে রিসিভ করা হয়েছে এবং এর বডির মধ্যে সেই নামেই ব্যবহার করা হয়েছে সাধারণভাবে পাস করা ভ্যারিয়েবলের মত আর তার মাধ্যমে বস্তুত beautify ফাংশন কল হয়েছে।

# ফাংশন

Published by [subeen](http://pybook.subeen.com/author/subeen/) on [April 16, 2018](http://pybook.subeen.com/2018/04/)

ফাংশনের সঙ্গে পরিচয় আমাদের অনেক আগেই হয়ে গিয়েছে। প্রথমেই যখন Hello world প্রিন্ট করেছিলাম, তখনই আমরা print() ফাংশন ব্যবহার করেছি। এছাড়াও আরো অনেক ফাংশন ব্যবহার করেছি, যেমন input(), len(), type() ইত্যাদি। আবার আমরা turtle মডিউল যখন ব্যবহার করেছি, তখন টার্টলের অনেক ফাংশনও ব্যবহার করেছি, যেমন forward(), left(), right(), dot() ইত্যাদি। ফাংশনগুলো কী কাজ করে, এটি আমাদের জানতে হবে, কিন্তু কিভাবে কাজ করে, সেটি এখন আমাদের জানার প্রয়োজন নেই। এসব ফাংশন তৈরি করে দেওয়া না থাকলে আমাদেরকে অনেক বেশি কষ্ট করতে হতো। যেমন print() ফাংশনটি কিভাবে কাজ করবে, সেটি পাইথনের বিল্টইনস্ মডিউলে লেখা আছে (তবে এই মডিউলটি আমাদের ইমপোর্ট করতে হয় না)। তেমনি dot() কিভাবে কাজ করবে, সেটি লেখা আছে টার্টল মডিউলের ভেতরে। এই অধ্যায়ে আমরা দেখবো, কিভাবে ফাংশন তৈরি করতে হয়, কিভাবে মডিউল তৈরি করতে হয় এবং প্রয়োজনীয় কিছু ফাংশনের ব্যবহার।

আমরা কখন ফাংশন তৈরি করবো? যখন আমাদেরকে একটি নির্দিষ্ট কাজ করতে হবে এবং সেই কাজটি একাধিকবার করার প্রয়োজন হবে, তখন আমরা সেই কাজের জন্য একটি ফাংশন তৈরি করে ফেলবো। আবার কখনও কখনও একটি কাজ একবার করলেও আমরা সেটির জন্য আলাদা ফাংশন তৈরি করতে পারি, যেন কোড বুঝতে সহজ হয়। যেমন ধরা যাক, আমাকে কোনো প্রোগ্রামে অনেকবার দুটি সংখ্যা যোগ করতে হবে। তখন আমরা এই যোগ করার জন্য একটি ফাংশন তৈরি করে ফেলতে পারি এভাবে :

def add(n1, n2):

return n1 + n2

আমরা দেখতে পাচ্ছি যে, ফাংশনের শুরুতে লিখতে হবে def, তাহলে পাইথন বুঝবে যে এখানে একটি ফাংশন তৈরি করা হচ্ছে (বা সংজ্ঞায়িত করা হচ্ছে, definition শব্দের প্রথম তিন অক্ষর def)। তারপর ফাংশনের নাম দিতে হবে। আমরা নাম দিয়েছি add। ফাংশনের নাম দেখে যেন বোঝা যায় যে, ফাংশনটি কী কাজ করবে। তারপরে প্রথম বন্ধনীর ভেতরে ফাংশনের প্যারামিটার লিখতে হবে। সব ফাংশনে প্যারামিটার থাকে না। প্যারামিটার থাকবে কী না এবং কয়টি, সেটি নির্ভর করে আমরা কী ফাংশন তৈরি করছি, তার ওপর। যেমন এখানে আমি তৈরি করছি দুটি সংখ্যার যোগফল বের করার ফাংশন। তাহলে তো আমাকে কোন দুটি সংখ্যা আমি যোগ করবো, সেগুলো জানতে হবে বা ইনপুট নিতে হবে। ফাংশনের ক্ষেত্রে প্যারামিটার হচ্ছে ইনপুট নেওয়ার পদ্ধতি। তারপরে একটি কোলন চিহ্ন দিতে হবে। পরের লাইন থেকে ফাংশনের ভেতরের কোড লিখতে হবে এবং সেগুলো ইনডেনটেশন করা থাকতে হবে, নইলে পাইথনের পক্ষে বোঝা সম্ভব হবে না যে কোন অংশ ফাংশনের ভেতরে আর কোন অংশ বাইরে। ফাংশন থেকে আবার এক বা একাধিক জিনিস ফেরত পাঠানো যায়, যাকে বলে রিটার্ন করা। আমাদের যেমন যোগফল ফেরত পাঠানো দরকার, তাই আমরা n1 + n2 এর মান রিটার্ন করছি। আমরা এখন ফাংশনটির ব্যবহার দেখবো। আমাদের কোনো নিয়মকানুন মুখস্থ করার প্রয়োজন নেই, চর্চা করে ও চিন্তা করে আমরা প্রোগ্রামিং শিখবো।

>>> def add(n1, n2):

... return n1 + n2

...

>>> n = 10

>>> m = 5

>>> result = add(n, m)

>>> print(result)

15

>>>

>>> number1 = 10

>>> number2 = 10

>>> result = add(number1, number2)

>>> print(result)

20

>>>

>>> n1 = 20

>>> n2 = 10

>>> print(add(n1, n2))

30

>>> print(add(2.5, 3.5))

6.0

>>>

আমরা ফাংশনটির বিভিন্ন রকম ব্যবহার দেখলাম। যেসব বিষয় খেয়াল করতে হবে, সেগুলো হচ্ছে –

* যদিও ফাংশনের প্যারামিটার হিসেবে n1 ও n2 ব্যবহার করা হয়েছে, আমরা কিন্তু ফাংশন কল করার সময় যেকোনো নামের আর্গুমেন্ট ব্যবহার করতে পারি।
* আমরা যখন ফাংশনটি সংজ্ঞায়িত বা ডিফাইন করছি বা সহজ বাংলায় ফাংশনটি তৈরি করছি, তখন যে n1, n2 লিখলাম, এগুলোকে বলে ফাংশনের প্যারামিটার (parameter)। আর আমরা যখন ফাংশনটি কল করছি, যেমন add(n, m) বা add(n1, n2) বা add(2, 3) এখানে n, m বা n1, n2 বা 2, 3 হচ্ছে আর্গুমেন্ট (argument)। নামগুলো মুখস্থ করতে হবে না, তবে জেনে রাখা ভালো। ভবিষ্যতে ইংরেজি বই পড়ার সময় কাজে লাগবে।
* আর্গুমেন্ট হিসেবে ভ্যারিয়েবল ব্যবহার করা যায়, আবার সরাসরি বিভিন্ন সংখ্যাও ব্যবহার করা যায়।

ফাংশনের যে সবসময় প্যারামিটার থাকতেই হবে, সেরকম কোনো কথা নেই। আবার ফাংশন থেকে যে কোনো কিছু রিটার্ন করতে হবে, এমন কোনো কথা নেই। আমরা ফাংশন থেকে কোনো কিছু রিটার্ন না করলে পাইথন আপনাআপনি None রিটার্ন করে। এটি পরীক্ষা করে দেখা যেতে পারে। আবার অনেকসময় ফাংশন থেকে একাধিক জিনিসও রিটার্ন করতে হতে পারে। সেক্ষেত্রে যেসব জিনিস রিটার্ন করতে হবে, return স্টেটমেন্টের পর সেগুলো কমা দিয়ে পৃথক করে দিতে হবে। যেমন : return a, b, c।

আমরা টার্টল ব্যবহার করে বর্গক্ষেত্র আঁকার একটি প্রোগ্রাম তৈরি করেছিলাম। প্রোগ্রামটিতে আমরা বর্গক্ষেত্র আঁকার কাজটি একটি ফাংশন তৈরি করে করতে পারি।

def draw\_square(side\_length):

for i in range(4):

turtle.forward(side\_length)

turtle.left(90)

তাহলে একটি সম্পূর্ণ প্রোগ্রাম লিখে দেখি, সেই বৃত্ত তৈরির প্রোগ্রামটি। নিচের প্রোগ্রামটি সাদাকালো, তবে কেউ চাইলে বিভিন্ন রং ব্যবহার করতে পারে :

import turtle

def draw\_square(side\_length):

for i in range(4):

turtle.forward(side\_length)

turtle.left(90)

counter = 0

while counter < 90:

draw\_square(100)

turtle.right(4)

counter += 1

turtle.exitonclick()

অনুশীলনী: একটি ফাংশন তৈরি করতে হবে যা প্যারামিটার হিসেবে একটি বাহুর দৈর্ঘ্য নেবে এবং একটি সমবাহু ত্রিভূজ আঁকবে।

এখন আমরা কিছু উদাহরণের মাধ্যমে ফাংশন সম্পর্কে আরো জানবো।

def myfnc(x):

print("inside myfnc", x)

x = 10

print("inside myfnc", x)

x = 20

myfnc(x)

print(x)

ওপরের প্রোগ্রামটি রান করলে আমরা আউটপুট পাবো এমন :

inside myfnc 20

inside myfnc 10

20

তাহলে আমরা দেখতে পেলাম, ভ্যারিয়েবলের নাম যদিও একই, কিন্তু myfnc-এর ভেতরে x-এর মান পরিবর্তন করে দিলেও ফাংশনের বাইরে x-এর মান পরিবর্তিত হয় নি। এর কারণ হচ্ছে myfnc যখন কল হয়, তখন সে আর্গুমেন্ট হিসেবে যেসব ভ্যারিয়েবল পায়, সেগুলোর একটা কপি তৈরি হয়। তাই myfnc-এর x আর তার বাইরের x আসলে দুটি আলাদা ভ্যারিয়েবল। একটি ফাংশনের ভেতরে যেসব ভ্যারিয়েবল তৈরি করা হয় সেগুলো হচ্ছে ওই ফাংশনের লোকাল (local) ভ্যারিয়েবল। ফাংশনের বাইরে তার অস্তিত্ব থাকে না।

ফাংশনের বাইরে যদি কোনো ভ্যারিয়েবল থাকে, তাহলে ফাংশনের ভেতর থেকে ওই ভ্যারিয়েবল পাওয়া যায়। একে বলে গ্লোবাল (global) ভ্যারিয়েবল। নিচের প্রোগ্রাম রান করলে সেটি আমরা দেখতে পাবো :

def myfnc(y):

print("y =", y)

print("x =", x)

x = 20

myfnc(x)

প্রোগ্রামের আউটপুট হবে এরকম :

y = 20

x = 20

এখন আমরা যদি myfnc এর বাইরে থেকে y-এর মান দেখতে চাই, সেটি কি সম্ভব? নিচের প্রোগ্রাম রান করলেই বুঝতে পারা যাবে।

def myfnc(y):

print("y =", y)

print("x =", x)

x = 20

myfnc(x)

print("y:", y)

আউটপুট হবে এরকম :

y = 20

x = 20

Traceback (most recent call last):

File "fnc\_test.py", line 7, in <module>

print("y:", y)

NameError: name 'y' is not defined

অর্থাৎ y নামে কোনো কিছু পাওয়া যায় নি।

এখন আমরা দেখবো, ফাংশনের প্যারামিটারের ডিফল্ট (default) মান কিভাবে দেওয়া যায়।

def myfnc(y=10):

print("y =", y)

x = 20

myfnc(x)

myfnc()

এই প্রোগ্রামে আমি দুইবার myfnc ফাংশন কল করলাম। প্রথমবার আর্গুমেন্ট হিসেবে x পাঠাচ্ছি। দ্বিতীয়বার কিছুই পাঠাচ্ছি না। কিন্তু ফাংশনের প্যারামিটারে আবার বলে দিয়েছি y=10। এর মানে হচ্ছে যদি কোনো আর্গুমেন্ট পাঠানো না হয়, তাহলে y-এর মান হবে 10, আর যদি কোনো আর্গুমেন্ট পাঠানো হয়, তাহলে আর্গুমেন্টে যেই ভ্যারিয়েবলটি পাঠানো হলো, সেই ভ্যারিয়েবলের মান y-তে কপি হবে।

এবারে আরেকটি প্রোগ্রাম লিখবো :

def myfnc(x, y=10, z):

print("x =", x, "y =", y, "z =", z)

x = 5

y = 6

z = 7

myfnc(x, y, z)

ওপরের কোড যদি রান করি, তাহলে আউটপুট পাবো এরকম :

def myfnc(x, y=10, z):

^

SyntaxError: non-default argument follows default argument

এই এররের অর্থ হচ্ছে আমরা যদি ডিফল্ট মান দেই, তাহলে তারপরে সবগুলো আর্গুমেন্টে ডিফল্ট মান থাকতে হবে। মানে আমরা যদি লিখতাম def myfnc(x, y=10, z=0) তাহলে এই এরর আর আসবে না। আমরা প্রোগ্রাম লিখে পরীক্ষা করে দেখি। নিচের প্রোগ্রামটি রান করে আউটপুট দেখে বুঝে নিতে হবে।

def myfnc(x, y=10, z=0):

print("x =", x, "y =", y, "z =", z)

x = 5

y = 6

z = 7

myfnc(x, y, z)

myfnc(x, y)

myfnc(x)

আর আমরা যদি চাই যে, z-এর কোনো ডিফল্ট মান থাকবে না তাহলে এভাবে লিখতে হবে : myfnc(x, z, y = 10)। কিন্তু এভাবে লিখলে তো একটা সমস্যা আছে, দ্বিতীয় আর্গুমেন্ট যেটি পাঠাবো, সেটি তো z-এ আসবে, কারণ প্যারামিটারের নাম তো বিবেচনা করা হয় না, বরং প্রথমটি x-এ, দ্বিতীয়টি z-এ এবং তৃতীয়টি y-এ কপি হবে। কিন্তু পাইথনে একটি উপায় আছে যেখানে নির্দিষ্ট প্যারামিটারে নির্দিষ্ট মান পাঠানো যায়। নিচের উদাহরণে আমরা সেটি দেখবো।

def myfnc(x, z, y=10):

print("x =", x, "y =", y, "z =", z)

myfnc(x = 1, y = 2, z = 5)

a = 5

b = 6

myfnc(x = a, z = b)

a = 1

b = 2

c = 3

myfnc(y = a, z = b, x = c)

প্রোগ্রামটি রান করলে নিচের মতো আউটপুট আসবে :

x = 1 y = 2 z = 5

x = 5 y = 10 z = 6

x = 3 y = 1 z = 2

পাইথনে সুযোগ-সুবিধা অনেক বেশি। তবে এখানে একটি কথা বলা প্রয়োজন। বইটি প্রথমবার পড়ার পরে বেশিরভাগ পাঠকেরই এত কিছু মনে থাকবে না এবং এটি খুবই স্বাভাবিক। এতে চিন্তিত হওয়ার কিছু নেই। বইটি দ্বিতীয় বা তৃতীয়বার পড়ার (এবং সেই সঙ্গে প্রতিবারই বইয়ের সব উদাহরণ ও অনুশীলনীগুলো করতে হবে) পর এগুলো মনে থাকবে।

আমরা চাইলে ফাংশনের ভেতরে লিস্টও পাঠাতে পারি। যেমন, এখন আমি একটি ফাংশন লিখবো, যেটিতে আর্গুমেন্ট হিসেবে একটি লিস্ট পাঠানো যাবে এবং আমরা সেই লিস্টের সংখ্যাগুলো যোগ করে যোগফল ফেরত পাঠাবো। যদিও লিস্টে আমরা যেকোনো কিছুই পাঠাতে পারি; কিন্তু যেহেতু যোগ করবো, তাই আমরা সংখ্যার লিস্ট পাঠাবো।

def add\_numbers(numbers):

result = 0

for number in numbers:

result += number

return result

result = add\_numbers([1, 2, 30, 4, 5, 9])

print(result)

এখন, ফাংশনের ভেতর ভ্যারিয়েবল পাঠানো আর লিস্ট পাঠানোর মধ্যে একটি পার্থক্য আছে। যে ফাংশনের ভেতরে লিস্ট পাঠানো হয়েছে, সেখানে যদি লিস্টটি পরিবর্তন করা হয়, তাহলে আসল লিস্টও পরিবর্তিত হয়ে যায়। এটিও আমরা একটি প্রোগ্রাম লিখে যাচাই করো নেবো। তবে তার আগে আমাদের আরো একটি জিনিস জেনে নিতে হবে। আমরা যদি লিস্টের কোনো নির্দিষ্ট উপাদান পেতে চাই, যেমন প্রথম, দ্বিতীয়, তৃতীয় ইত্যাদি, তাহলে আমরা তৃতীয় বন্ধনীর ভেতরে সেই সংখ্যাটি উল্লেখ করে দিতে পারি। একে বলা হয় ইনডেক্স। তবে পাইথনে লিস্টের ইনডেক্স কিন্তু 1 থেকে নয়, 0 থেকে শুরু হয়। অর্থাৎ, আমরা যদি লিস্টের প্রথম উপাদান একসেস করতে চাই, তাহলে আমরা লিস্টের নামের পর [0] লিখবো। country নামের একটি লিস্টে যদি পৃথিবীর সবগুলো দেশের নাম থাকে, আর সেই লিস্টের 50-তম দেশটি আমরা দেখতে চাই, তাহলে আমাদেরকে country[49] প্রিন্ট করতে হবে।

def test\_fnc(li):

li[0] = 10

my\_list = [1, 2, 3, 4]

print("before function call", my\_list)

test\_fnc(my\_list)

print("after function call", my\_list)

প্রোগ্রামটি রান করলে আমরা আউটপুট পাবো এরকম :

before function call [1, 2, 3, 4]

after function call [10, 2, 3, 4]

এরকম হওয়ার কারণ কী? কারণ আমরা যখন একটি লিস্ট অন্য একটি ভ্যারিয়েবলে অ্যাসাইন করি, তাহলে নতুন কোনো লিস্ট তৈরি হয় না, বরং নতুন ভ্যারিয়েবল আর পুরনো ভ্যারিয়েবলে একই লিস্ট থাকে। সেটিও আমরা পরীক্ষা করে দেখবো পাইথন ইন্টারপ্রেটারে।

>>> list1 = [1, 2, 3, 4]

>>> list2 = list1

>>> print(list1)

[1, 2, 3, 4]

>>> print(list2)

[1, 2, 3, 4]

>>> list2[0] = 100

>>> print(list2)

[100, 2, 3, 4]

>>> print(list1)

[100, 2, 3, 4]

তাই ফাংশনের ভেতরে লিস্ট পাঠানোর সময় কিংবা আরেকটি ভ্যারিয়েবলে লিস্ট অ্যাসাইন করার সময় আমাদের সতর্ক হতে হবে। নইলে প্রোগ্রামে বাগ-এর সৃষ্টি হবে। কম্পিউটার প্রোগ্রামের ত্রুটিকে বলে বাগ (bug)।

আমরা যে একটু আগে লিস্টের উপাদানগুলো যোগ করার জন্য একটি ফাংশন লিখলাম, পাইথনে কিন্তু সেই কাজ করার জন্য একটি বিল্টইন ফাংশন আছে। আমাদের প্রয়োজন হলে আমরা সেই ফাংশন ব্যবহার করবো। তাহলে আমি উদাহরণের প্রোগ্রাম কেন লিখলাম? ফাংশনে কিভাবে লিস্ট পাঠাতে হয়, সেটি দেখানোর জন্য।

>>> li = [1, 2, 3]

>>> result = sum(li)

>>> print(result)

6

### [প্রোগ্রামিং বইঃ অধ্যায় সাত] ফাংশন (Function)।

তোমরা কি একটি মজার ব্যাপার জানো? একজন লেখক সারা জীবনে যতটা সময় লেখেন তার চেয়ে বেশি সময় তিনি অন্যের লেখা পড়েন? ব্যাপারটি প্রোগ্রামারদের বেলাতেও সত্য। একজন প্রোগ্রামার তার প্রোগ্রামিং জীবনে যতটা সময় নিজে কোড লেখে তার চেয়ে বেশি সময় অন্যের লেখা কোড পড়ে! তাই কোড লিখার সময় খেয়াল রাখতে হবে, যেন সেটি পড়াও সুবিধাজনক হয়।

যারা বইটি শুরু থেকে পড়ে এসেছ তারা ইতিমধ্যে অনেকবার ফাংশন শব্দটি দেখেছ। যারা আরও বেশি মনোযোগ দিয়ে পড়েছ তারা এটিও খেয়াল করেছ যে printf, scanf ইত্যাদি, যেগুলো তোমরা ব্যবহার করছ সেগুলো একেকটি ফাংশন। আবার mainও একটি ফাংশন। আমরা এবার দেখব ফাংশন ব্যাপারটি আসলে কী, এর দরকারটাই বা কী। আর তারপর আমরা নিজেদের ফাংশন তৈরি করা শিখব।

ফাংশন ব্যবহার করা হয় কোনো একটি নির্দিষ্ট কাজ করার জন্য। যেমন printf ফাংশনটি দিয়ে আমরা মনিটরে আউটপুট দিই। আবার scanf, getchar এসব ফাংশন দিয়ে আমরা কিবোর্ড থেকে ইনপুট নিই। এখন printf ফাংশনটি যে আমরা লিখলাম, কম্পিউটারের তো আর এটি বোঝার কথা নয়। printf ফাংশনটি কী কাজ করবে, কীভাবে করবে সেটি আসলে বলে দেওয়া আছে stdio.h নামের একটি হেডার (header) ফাইলের মধ্যে। এজন্যই আমরা আমাদের প্রোগ্রামগুলোতে (যেখানে printf, scanf ইত্যাদি ব্যবহার করেছি) ওই হেডার ফাইলটির কথা বলে দিই (#include )। আবার স্ট্রিং-সংক্রান্ত ফাংশনগুলো ব্যবহার করলে string.h – এই হেডার ফাইলটির কথাও বলে দিই। এখন চিন্তা করো, printf ফাংশনের এই কোডটি যদি আমাদের নিজেদের লিখতে হতো, তাহলে ব্যাপারটি কী বিরক্তিকরই না হতো! এরকম অনেক ফাংশন আছে যেগুলোর ব্যবহার তোমরা আস্তে আস্তে জেনে যাবে।  
  
আচ্ছা, main কে ও তো আমি একটি ফাংশন বলেছি, কিন্তু এটি দিয়ে আমরা আবার কী করি? সি ল্যাঙ্গুয়েজে এই ফাংশনটি দিয়েই আসলে আমরা একটি প্রোগ্রাম চালাই। কম্পাইলার জানে যে main ফাংশন যেখানে আছে, সেখান থেকেই কাজ শুরু করতে হবে। তাই একটি প্রোগ্রামে কেবল একটিই main ফাংশন থাকে।  
  
এবারে দেখি, আমরা নিজেরা কীভাবে ফাংশন তৈরি করতে পারি। একটি ফাংশন যখন আমরা তৈরি করব সেটির গঠন হবে মোটামুটি এই রকম:  
  
return\_type function\_name (parameters) {  
function\_body  
return value  
}  
  
return\_type: এখানে বলে দিতে হবে ফাংশনটি কাজ শেষ করে বের হবার সময় কী ধরনের ডাটা রিটার্ন করবে। সেটি, int, double এসব হতে পারে। আবার কিছু রিটার্ন করতে না চাইলে সেটি void হতে পারে। অর্থাৎ সে কিছুই রিটার্ন করবে না। এর মানে দাঁড়াচ্ছে, তুমি আসলে ফাংশনকে দিয়ে কোনো একটি কাজ করাবে, সেজন্য কাজ শেষে সে তোমাকে কী ধরনের ডাটা ফেরত দেবে সেটি বলে দিতে হবে। ফাংশনের কোনো জায়গাতে তুমি যখনই return ব্যবহার করবে, ফাংশনটি সেই জায়গা থেকেই রিটার্ন করবে বা বের হয়ে যাবে। অনেক ফাংশনের ভেতর দেখবে একাধিক রিটার্ন আছে এবং সঙ্গে বিভিন্ন শর্ত দেওয়া আছে। শর্তের উপর নির্ভর করে যখনই প্রোগ্রামটি কোনো রিটার্ন পাবে তখনই ফাংশন থেকে বের হয়ে যাবে।  
  
function\_name: এখানে আমাদের ফাংশনের নাম লিখতে হবে। ফাংশনের নাম হতে হবে অর্থপূর্ণ যাতে নাম দেখেই ধারনা করা যায় যে ফাংশনটি কী কাজ করবে। যেমন কোন সংখ্যার বর্গমূল নির্ণয়ের জন্য যদি আমরা একটি ফাংশন লিখি তবে সেটির নাম আমরা দিতে পারি square\_root বা sqrt। আমরা নিশ্চয়ই সেটির নাম beautiful দিব না, যদিও কম্পাইলার তাতে কোন আপত্তি করবে না।  
  
parameters: এখানে ফাংশনটি কাজ করার জন্য প্রয়োজনীয় ডাটা আমরা দেব। যেমন স্ট্রিং-এর দৈর্ঘ্য নির্ণয়ের জন্য আমরা যখন strlen ফাংশনটি ব্যবহার করি সেখানে কোন স্ট্রিং-এর দৈর্ঘ্য নির্ণয় করতে হবে সেটি বলে দিতে হয় (নইলে সেটি কার দৈর্ঘ্য নির্ণয় করবে?)। আবার বর্গমূল নির্ণয়ের জন্য ফাংশন লিখলে কোন সংখ্যার বর্গমূল বের করতে হবে সেটি বলে দিতে হবে। প্যারামিটারের মাধ্যমে আমরা সেসব ডাটা ওই ফাংশনের কাছ পাঠাতে পারি। আবার কোনো কিছু পাঠাতে না চাইলে সেটি খালিও রাখতে পারি। যেমন, getchar() বা main() ফাংশন। একাধিক প্যারামিটার পাঠানোর সময় প্রতিটি প্যারামিটার কমা (,) দিয়ে আলাদা করতে হবে।  
  
function\_body: ফাংশনটি কীভাবে কী কাজ করবে সেটি বডিতে বলে দিতে হবে। মানে কোড লিখতে হবে আর কি।  
  
return value: ফাংশনটি কাজ শেষ করে, তাকে যে জায়গা থেকে কল করা হয়েছে সে জায়গায় ফিরে যায়। ফেরার সময় আমরা কোনো মান পাঠাতে পারি। যেমন sqrt() ফাংশনে আমরা চাই সে বর্গমূল বের করবে। তো বর্গমূলটি বের করে তো সেটি ফেরত পাঠাবার ব্যবস্থা রাখতে হবে? বর্গমূলটির মান যদি x হয়, তবে আমরা return x; স্টেটমেন্ট দিয়ে সেটির মান ফেরত পাঠাব।  
  
int root = sqrt(25);  
  
এখানে sqrt ফাংশন 25-এর বর্গমূল নির্ণয় করার পর বর্গমূলটি ফেরত পাঠাবে এবং সেটি root নামের একটি ইন্টিজার ভেরিয়েবলে জমা হবে।  
  
একটি উদাহরণ দিই। তোমরা যারা ত্রিকোণমিতি পড়েছ তারা নিশ্চয়ই sin, cos, tan ইত্যাদির সঙ্গে পরিচিত। sin 300-এর মান হচ্ছে 0.5। এখানে sin কিন্তু আসলে একটি ফাংশন, যার প্যারামিটার হিসেবে আমরা কোণের মান দিচ্ছি। আর ফাংশনটি ওই কোণের sine (সংক্ষেপে sin)-এর মান রিটার্ন করছে।  
  
এবারে চলো, আর বকবক না করে প্রোগ্রামিং শুরু করে দিই। তারপর দেখি কী করলে কী হয়।

#include <stdio.h>

int main()

{

double a, b, c;

a = 2.5;

b = 2.5;

c = a + b;

printf("%lf\n" c);

return 0;

}

প্রোগ্রাম: ৭.১

প্রোগ্রামটি চালাও। আউটপুট কী? 5.000000।  
  
এবার আমরা দুটি সংখ্যা যোগ করার জন্য একটি ফাংশন লিখে ফেলি। যোগের কাজটি আর main ফাংশনের ভেতরে করব না।

#include <stdio.h>

int add(int num1, int num2)

{

double sum = num1 + num2;

return sum;

}

int main()

{

double a, b, c;

a = b = 2.5;

c = add(a, b);

printf("%lf\n", c);

return 0;

}

প্রোগ্রাম: ৭.২

প্রোগ্রামটি চালাও। আউটপুট কী? 4.000000! ওহ‍্‍ আমরা তো গাধার মতো একটি ভুল করেছি। num1 ও num2 তো আসলে int টাইপের হবে না, double টাইপের হবে। ওই দুটি ভেরিয়েবল ইন্টিজার হিসেবে ডিক্লেয়ার করার কারণে 2.5 হয়ে গিয়েছে 2 (টাইপ কাস্টিংয়ের কথা মনে আছে তো?)। আমরা ভুল ঠিক করে ফেলি:

int add(double num1, double num2)

{

double sum = num1 + num2;

return sum;

}

এবারে প্রোগ্রামটি রান করলে আউটপুট কী? 5.000000। যাক, সমস্যার সমাধান হয়ে গেল! আচ্ছা, এবারে আমরা a, b-এর মান একটু বদলাই। a = 2.8; b = 2.7; করে দিই। আউটপুট কত হবে? 5.500000? এটিই হওয়া উচিত (2.8 + 2.7 = 5.5) কিন্তু প্রোগ্রামটি রান করে দেখো তো কত হয়? তুমি আউটপুট পাবে 5.000000। কারণ কী?  
  
কারণ, আমাদের ফাংশনের রিটার্ন টাইপ int, যা কিনা একটি ইন্টিজার রিটার্ন করতে সক্ষম। num1 ও num2 যোগ করার পর sum-এর মধ্যে 5.5 ঠিকই থাকবে কিন্তু রিটার্ন করার সময় সেটি ইন্টিজারে বদলে যাবে। সুতরাং রিটার্ন টাইপ আমরা double করে দেব। এবার আমাদের প্রোগ্রাম ঠিকঠাক কাজ করবে:

#include <stdio.h>

double add(double n1, double n2)

{

double sum = n1 + n2;

return sum;

}

int main()

{

double a, b, c;

a = 2.8;

b = 2.7;

c = add(a, b);

printf("%lf\n", c);

return 0;

}

প্রোগ্রাম: ৭.৩

এখন আমরা একটি এক্সপেরিমেন্ট করব। add ফাংশনটি main ফাংশনের পরে লিখব:

#include <stdio.h>

int main()

{

double a = 2.8, b = 2.7, c;

c = add(a, b);

printf("%lf\n", c);

return 0;

}

double add(double n1, double n2)

{

double sum = n1 + n2;

return sum;

}

প্রোগ্রাম: ৭.৪

এবারে প্রোগ্রামটি রান করতে গেলে দেখবে, কম্পাইলার এরর দিচ্ছে: "error: ‘add’ was not declared in this scope", অর্থাৎ সে আর add ফাংশনটিকে চিনতে পারছে না। তবে চিন্তা নেই, এটিকে চিনিয়ে দেওয়ার ব্যবস্থাও আছে। সেটি হচ্ছে main ফাংশনের আগে add ফাংশনের প্রোটোটাইপ (prototype) বলে দেওয়া:  
  
double add(double n1, double n2);  
  
প্রোটোটাইপে পুরা ফাংশনটি লিখতে হয় না। এর অংশগুলো হচ্ছে:  
  
return\_type function\_name (parameters) ;  
  
সেমিকোলন দিতে ভুল করবে না কিন্তু। আর প্রোটোটাইপের প্যারামিটারে যে ভেরিয়েবল ব্যবহার করবে তার সঙ্গে মূল ফাংশনের ভেরিয়েবলের নাম একরকম না হলে কোনো অসুবিধা নেই, তবে ডাটা টাইপ একই হতে হবে। এখন নিচের প্রোগ্রামটি ঠিকঠাক কাজ করবে:

#include <stdio.h>

double add(double x, double y);

int main()

{

double a = 2.8, b = 2.7, c;

c = add(a, b);

printf("%lf\n", c);

return 0;

}

double add(double n1, double n2)

{

double sum = n1 + n2;

return sum;

}

প্রোগ্রাম: ৭.৫

এবার আমরা আরও কিছু পরীক্ষা-নিরীক্ষা করব।

#include <stdio.h>

int test\_function(int x)

{

int y = x;

x = 2 \* y;

return (x \* y);

}

int main()

{

int x = 10, y = 20, z = 30;

z = test\_function(x);

printf("%d %d %d\n", x, y, z);

return 0;

}

প্রোগ্রাম: ৭.৬

প্রোগ্রামটি না চালিয়ে শুধু কোড দেখে বলো তো আউটপুট কী হবে? আমাদের কোনো তাড়া নেই, তাই ধীরেসুস্থে চিন্তা করে বলো।  
  
এবার কে কে আমার সঙ্গে একমত যে আউটপুট হবে:  
20 10 200 (অর্থাৎ x = 20, y = 10, z = 200)?  
  
কারণ x, y-এর মান তো test\_function-এর ভেতরে আমরা বদলে দিয়েছি। প্রথমে x-এর মান 10 যাচ্ছে প্যারামিটার হিসেবে, তারপরে সেই মানটি আমরা y-তে বসাচ্ছি। মানে y-এর মান এখন 10। তারপর x-এর মান বসাচ্ছি 2 \* y মানে 20। তারপর রিটার্ন করছি x \* y (যার মান, 20 \* 10 বা 200)। সুতরাং z-এর মান হবে 200।  
  
এবারে প্রোগ্রামটি চালাও, আউটপুট দেখবে: 10 20 200 (অর্থাৎ x = 10, y = 20, z = 200)। এমন হওয়ার কারণ কী? z-এর মান নিয়ে কোনো আপত্তি নেই, ফাংশনটি 200 রিটার্ন করে আর সেটি আমরা z-এ বসিয়ে দিয়েছি। কথা হচ্ছে, x আর y-এর মান নিয়ে। আসলে test\_function-এর ভেতরে আমরা x, y-এর মান পরিবর্তন করায় main ফাংশনের x, y-এর কিছু আসে-যায় না। প্রত্যেক ফাংশনের ভেরিয়েবলগুলো আলাদা। একে বলে লোকাল ভেরিয়েবল (local variable)। আমরা main ফাংশনের x, y-এর মান প্রিন্ট করেছি test\_function ফাংশনের x, y-এর মান প্রিন্ট করিনি। এক ফাংশনের লোকাল ভেরিয়েবলের অস্তিত্ব অন্য ফাংশনে থাকে না। তুমি এখন কিছু প্রোগ্রাম লিখে আরও পরীক্ষা-নিরীক্ষা করে দেখতে পারো। কী প্রোগ্রাম লিখবে সেটি তোমার ওপর ছেড়ে দিলাম।  
  
আমরা যদি চাই, কোনো ভেরিয়েবলের অস্তিত্ব আমাদের প্রোগ্রামের সব ফাংশনের ভেতরে থাকতে হবে, তবে আমরা সেটি করতে পারি গ্লোবাল (global) ভেরিয়েবল ডিক্লেয়ার করার মাধ্যমে। আমরা প্রোগ্রামের শুরুতে কোনো ফাংশন বা ফাংশনের প্রোটোটাইপ লিখার আগে সেগুলো ডিক্লেয়ার করে দেব। যেমন:

#include <stdio.h>

double pi = 3.14;

void my\_fnc() {

pi = 3.1416; /\* এখানে আমরা pi-এর মান একটু পরিবর্তন করে দিলাম \*/

return; /\* ফাংশনের রিটার্ন টাইপ void হলে এই return; না দিলেও কিন্তু চলে \*/

}

int main() {

printf("%lf\n", pi); /\* এখানে pi-এর মান হবে 3.14 \*/

my\_fnc();

printf("%lf\n", pi); /\* এখানে pi-এর মান হবে 3.1416 কারণ আমরা সেটি my\_fnc ফাংশনে গিয়ে বদলে দিয়েছি। \*/

return 0;

}

আবার আমরা যদি my\_fnc ফাংশনের ভেতরে গিয়ে pi নামে একটি ভেরিয়েবল ডিক্লেয়ার করতাম (double pi;), তবে সেটি একটি লোকাল ভেরিয়েবল হতো এবং গ্লোবাল pi-এর মানের কোন পরিবর্তন হতো না।  
  
এতক্ষণ আমরা ফাংশনের প্যারামিটার হিসেবে কেবল ভেরিয়েবল ব্যবহার করেছি। এবারে আসো আমরা ফাংশনের প্যারামিটার হিসেবে অ্যারে পাঠাই। আমরা একটি প্রোগ্রাম লিখব যেটি কোনো একটি ইন্টিজার অ্যারে থেকে সবচেয়ে বড় সংখ্যাটি খুঁজে বের করবে। অ্যারে থেকে সর্বোচ্চ সংখ্যা খুঁজে বের করার কাজটি করার জন্য একটি ফাংশন লিখে ফেলি, কী বলো?  
  
int find\_max(int ara[], int n) { /\* এখানে আমরা দুটি প্যারামিটার দিচ্ছি। প্রথমটা হচ্ছে একটি অ্যারে, আর তারপর একটি সংখ্যা যেটি নির্দেশ করবে অ্যারেতে কয়টি সংখ্যা আছে। লক্ষ করো, প্যারামিটারে যখন অ্যারের কথাটি বলে দিচ্ছি তখন সেখানে কয়টি উপাদান আছে সেটি না দিলেও চলে, যেমন আমরা int ara[11] ও লিখতে পারতাম। \*/  
  
int max = ara[0]; /\* এখানে একটি ভেরিয়েবলে ধরে নিচ্ছি যে সবচেয়ে বড় সংখ্যাটি হচ্ছে অ্যারের প্রথম সংখ্যা। তারপরে আমরা অ্যারের বাকি উপাদানগুলোর সঙ্গে maxকে তুলনা করব আর যদি অ্যারের কোনো উপাদানের মান max-এর চেয়ে বড় হয় তখন সেই মানটি max-এ রেখে দেব। অর্থাৎ তখন আবার max হয়ে যাবে ওই অ্যারের সর্বোচ্চ সংখ্যা। \*/  
  
int i;  
for(i = 1; i < n; i++) { if (ara[i] > max) {  
max = ara[i]; /\* ara[i] যদি max-এর চেয়ে বড় হয় তবে max-এ ara[i]-এর মানটি অ্যাসাইন করে দিচ্ছি। \*/  
}  
}  
return max; /\* ফাংশন থেকে সর্বোচ্চ মানটি ফেরত পাঠাচ্ছি \*/  
}  
  
এখন কথা হচ্ছে এই ফাংশনকে আমরা কল করব কীভাবে? ভেরিয়েবলের জায়গায় তো এর নাম দিয়ে কল করতে হয়, কিন্তু অ্যারের বেলায় কী দেব? অ্যারের বেলাতেও শুধু নাম দিলেই চলবে। পুরো প্রোগ্রামটি এবারে রান করে দেখো:

#include <stdio.h>

int find\_max(int ara[], int n);

int main()

{

int ara[] = {-100, 0, 53, 22, 83, 23, 89, -132, 201, 3, 85};

int n = 11;

int max = find\_max(ara, n);

printf("%d\n", max);

return 0;

}

int find\_max(int ara[], int n)

{

int max = ara[0];

int i;

for(i = 1; i < n; i++) {

if (ara[i] > max) {

max = ara[i];

}

}

return max;

}

প্রোগ্রাম: ৭.৭

এখন তোমরা find\_min নামে আরেকটি ফাংশন লেখো যার কাজ হবে সবচেয়ে ছোট সংখ্যাটি খুঁজে বের করা। find\_sum, find\_average এসব ফাংশনও লিখে ফেলতে পারো। আর তোমাদের নিশ্চয়ই বলে দিতে হবে না এইসব ফাংশন কী কাজ করবে।  
  
ফাংশনে ভেরিয়েবল পাস করা (pass, পাঠানো অর্থে) আর অ্যারে পাস করার মধ্যে একটি গুরুত্বপূর্ণ পার্থক্য রয়েছে। আমরা ইতিমধ্যে দেখেছি যে ফাংশনের ভেতর ভেরিয়েবল পাস করলে ওই ফাংশনের ভেতরে সেটির আরেকটি কপি তৈরি হয়, সুতরাং সেখানে ওই ভেরিয়েবলের মান পরিবর্তন করলে মূল ফাংশন (যেখান থেকে ফাংশন কল করা হয়েছে) ভেরিয়েবলের মানের কোনো পরিবর্তন হয় না। তবে অ্যারের বেলায় ব্যাপারটি আলাদা। আগে আমরা একটি প্রোগ্রাম লিখে দেখি:

#include <stdio.h>

void test\_function(int ara[])

{

ara[0] = 100;

return;

}

int main()

{

int ara [] = {1, 2, 3, 4, 5};

printf("%d\n", ara[0]);

test\_function(ara);

printf("%d\n", ara[0]);

return 0;

}

প্রোগ্রাম: ৭.৮

এই প্রোগ্রামের আউটপুট কী হবে? প্রথম printf ফাংশনটি 1 প্রিন্ট করবে সেটি নিয়ে তো কোনো সন্দেহ নেই, কিন্তু দ্বিতীয় printf কী প্রিন্ট করবে? test\_function-এর ভেতর আমরা অ্যারের প্রথম উপাদানের মান 100 অ্যাসাইন করেছি। এখন যদি সেটি মূল অ্যারেকে পরিবর্তন করে, তবে ara[0]-এর মান হবে 100, আর পরিবর্তন না হলে মান হবে আগে যা ছিল তা-ই, মানে 1।  
  
আমরা আউটপুট দেখব 100, কারণ অ্যারেটির প্রথম উপাদানের মান পরিবর্তিত হয়েছে। অর্থাৎ আমরা বুঝতে পারলাম ফাংশনের ভেতরে অ্যারে পাস করলে ওই অ্যারের আলাদা কোনো কপি তৈরি হয় না। কারণ হচ্ছে আমরা ফাংশনের ভেতর অ্যারের নামটি কেবল পাঠাই, যেটি কিনা ওই অ্যারেটি মেমোরির কোন জায়গায় আছে তার অ্যাড্রেস। এখন তোমরা বৃত্তের ক্ষেত্রফল নির্ণয়ের জন্য একটি ফাংশন লিখে ফেলো। ক্ষেত্রফল বের করার সূত্রটি মনে আছে তো? মনে না থাকলে জ্যামিতি বই থেকে দেখে নাও।