




# Full Stack Development

## Sesi 3

---



# **PYTHON: FUNCTIONS, MODULE & PACKAGES**

+



# Python Function

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. Function blocks begin with the keyword `def` followed by the function name and parentheses `( )`. For example:

```
def function_name( parameters ):
```

```
    "docstring"
```

```
    statement(s)
```

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

the docstring is short for documentation string. It is used to explain in brief, what a function does.

# 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. Following is the example to call printme() function:

# 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")
```

## 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:

# 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()
```

# 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. You can also make keyword calls to the `printme()` function in the following ways:

# 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 = "Hacktiv8")
```

# 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. For example:

# Function definition is here

```
def printinfo( name, age = 26 ):
```

```
    "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="hacktiv8" )
```

```
printinfo( name="hacktiv" )
```



# 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. here is the syntax:

```
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.

# 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.

Following is the example to show how `lambda` form of function work:

# Function definition is here

```
sum = lambda arg1, arg2: arg1 + arg2;
```

```
def sum(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 ))
```

# The return Statement

A return statement with no arguments is the same as return None. You can return a value from a function as follows:

```
# Function definition is here
```

```
def sum(arg1, arg2):
```

```
    # Add both the parameters and return them."
```

```
    total = arg1 + arg2
```

```
    total2 = total + arg1
```

```
    print("Inside the function : ", total)
```

```
    return total2
```

```
# Now you can call sum function
```

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

```
print("Outside the function : ", total)
```

# 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.

```
total = 0 #This is global variable
```

```
def sum( arg1, arg2 ):
```

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

```
    print("Inside the function local total : ", total)
```

```
    return total
```

```
sum( 10, 20 );
```

```
print("Outside the function global total : ", total)
```

## External References

Function - [Visit Here](#)

Module & Packages - [Visit Here](#)