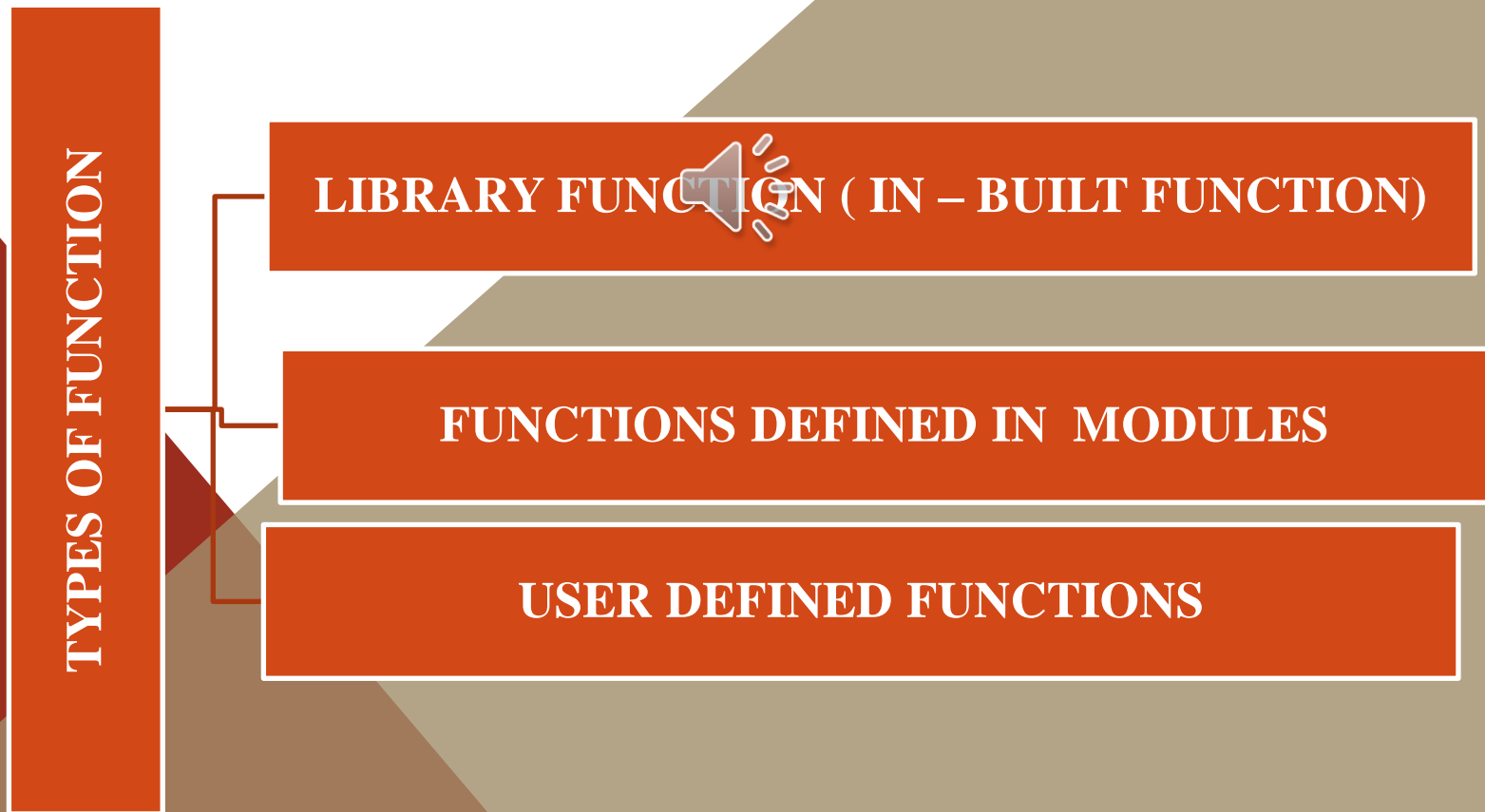


CHAPTER-2 FUNCTIONS IN PYTHON

FUNCTIONS: - A group of statements in a program to perform a particular task is known as a function. A function name in python ends with parenthesis or round brackets.

TYPES OF FUNCTIONS: - There are basically three types of functions in python:



1. LIBRARY OR IN-BUILT FUNCTIONS-

- * **PREDEFINED.**
- * **ALWAYS PRESENT IN THE STANDARD LIBRARY.**
- * **WE DON'T HAVE TO IMPORT ANY MODULE FOR USING THEM.**
- * **PROVIDE EFFICIENCY AND STRUCTURE TO THE PROGRAM THUS MAKING PROGRAMMING EASIER, FASTER AND MORE POWERFUL.**
- * **FOR EXAMPLE: len(), input(), eval() etc.**

2. FUNCTIONS DEFINED IN MODULE-

- * **CONTAINS A COLLECTION OF RELATED FUNCTIONS.**
- * **THE CODE CAN BE REUSED IN MORE THAN ONE PROGRAM.**
- * **TO USE THE FUNCTION THE MODULE HAS TO BE IMPORTED IN THE PROGRAM.**

THERE ARE TWO METHODS TO IMPORT THE MODULE

- * **import statement - to import entire module.**
- * **from - to import all functions or selected ones.**

import math

S. No.	Function	Description	Example
1.	sqrt()	Returns the square root of a number	>>>math.sqrt(49) ie 7.0
2.	pow()	Calculate the power of a number	>>>math.pow(2,3) ie. 8.0
3.	fabs()	Returns the absolute value of a number	>>>math.fabs(-5.6) ie.5.6

EXAMPLE PROGRAM USING IMPORT

```
import random
import math
for i in range(5):
    print(random.randint(1, 25))
print(math.pi)
```

Now, when we execute our program, we'll receive output that looks like this, with an approximation of pi as our last line of output:

Output

```
18
10
7
13
10
3.141592653589793
```



Using from ... import

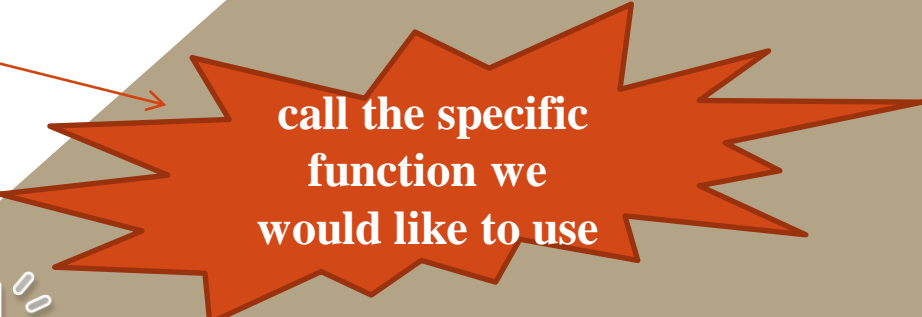
- To refer to items from a module within your program's namespace, you can use the from ... import statement.
- When you import modules this way, you can refer to the functions by name rather than through dot notation.
- In this construction, you can specify which definitions to reference directly.

For example ;-

```
from random import randint
```



first call
the from
keyword



call the specific
function we
would like to use



Now, when we implement this function within our program, we will no longer write the function in dot notation as `random.randint()` but instead will just write `randint()`:

```
my_rand_int.py
from random import randint
for i in range(10):
    print(randint(1, 25))
```

When you execute the program, you'll receive output similar to what we received earlier. Using the from ... import construction allows us to reference the defined elements of a module within our program's namespace, letting us avoid dot notation.

USER DEFINED FUNCTIONS:

The functions those are defined by the user are called user defined functions.

The syntax to define a function is:

```
def function-name ( parameters ) :  
    #statement(s)
```

Where: Keyword def marks the start of function header.

A function name to uniquely identify it.

Function naming follows the same rules of writing identifiers in Python.

Parameters (arguments) through which we pass values to a function.

They are optional.



A colon (:) to mark the end of function header.

One or more valid python statements that make up the function body.

Statements must have same indentation level. An optional return statement to return a value from the function.

Example:

```
def display(name):  
    print("Hello " + name + " How are you?")
```

FUNCTION PARAMETERS AND ARGUMENTS

THE TERMS PARAMETERS AND ARGUMENTS ARE USED INTERCHANGABLY

Formal Parameter: Formal parameters are written in the function prototype and function header of the definition. Formal parameters are local variables which are assigned values from the arguments when the function is called.

Actual Parameter: When a function is called, the values that are passed in the call are called actual parameters. At the time of the call each actual parameter is assigned to the corresponding formal parameter in the function definition.

Default Parameters: Python allows function arguments to have default values. If the function is called without the argument, the argument gets its default value.

Example :

```
def ADD(x, y):                #Defining a function and x and y are formal parameters
    z=x+y
    print("Sum = ", z)
```

```
a=float(input("Enter first number: " ))
```

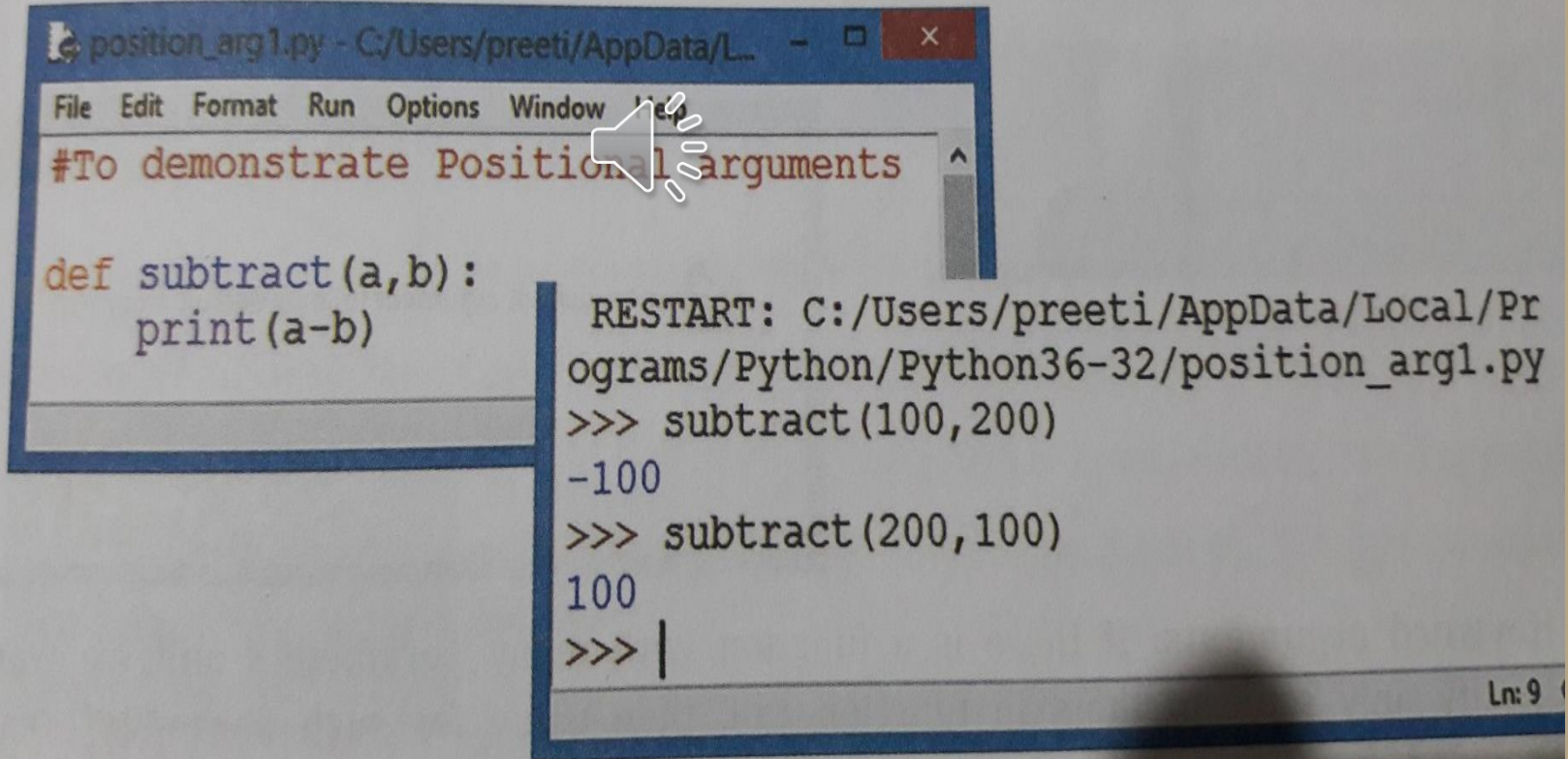
```
b=float(input("Enter second number: " ))
```

```
ADD(a,b)    #Calling the function by passing actual parameters In the above example, x
            and y are formal parameters. a and b are actual parameters.
```

TYPES OF ACTUAL ARGUMENT

- POSITIONAL ARGUMENT

1) **Positional arguments:** Positional arguments are arguments passed to a function in correct positional order.



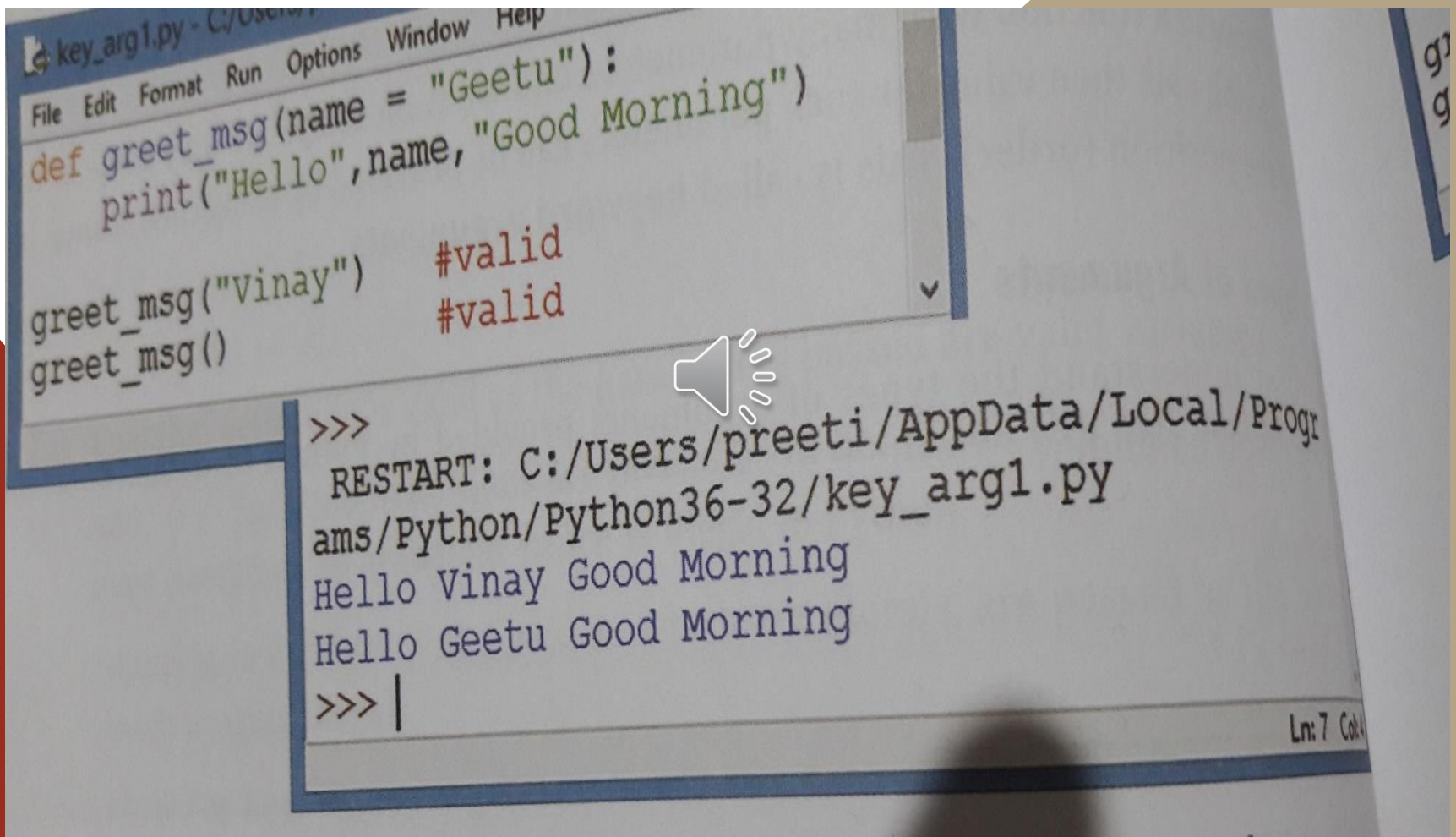
```
position_arg1.py - C:/Users/preeti/AppData/L... - [x]
File Edit Format Run Options Window Help
#To demonstrate Positional arguments

def subtract(a,b):
    print(a-b)

RESTART: C:/Users/preeti/AppData/Local/Pr
ograms/Python/Python36-32/position_arg1.py
>>> subtract(100,200)
-100
>>> subtract(200,100)
100
>>> |
```

Ln: 9

- **DEFAULT ARGUMENT**



```
key_arg1.py - C:/Users/...
File Edit Format Run Options Window Help
def greet_msg(name = "Geetu"):
    print("Hello", name, "Good Morning")

greet_msg("vinay")      #valid
greet_msg()             #valid

>>>
RESTART: C:/Users/preeti/AppData/Local/Programs/Python/Python36-32/key_arg1.py
Hello Vinay Good Morning
Hello Geetu Good Morning
>>> |
```

Ln: 7 Col: 1

- **KEYWORD ARGUMENT**

3) **Keyword arguments:** If there is a function with many parameters and we want to specify only some of them in function call, then value for such parameters can be provided by using their **name** instead of the position (order). These are called **keyword arguments**.

```
File Edit Format Run Options Window Help
def greet_msg(name, msg) :
    print("Hello", name, msg)

greet_msg(name="Vinay", msg="Good Morning")
greet_msg(msg="Good Morning", name="Sonia")

>>>
RESTART: C:/Users/preeti/AppData/Local/Programs/Python/Python36-32/key_arg1.py
Hello Vinay Good Morning
Hello Sonia Good Morning
>>>
```

```
File Edit Format Run Options Window Help
def greet_msg(name, msg) :
    print("Hello", name, msg)

greet_msg(name="Vinay", msg="Good Morning") #valid
greet_msg(msg="Good Morning", name="Sonia") #valid
greet_msg(name="Radhika", "Good Morning") #invalid
```

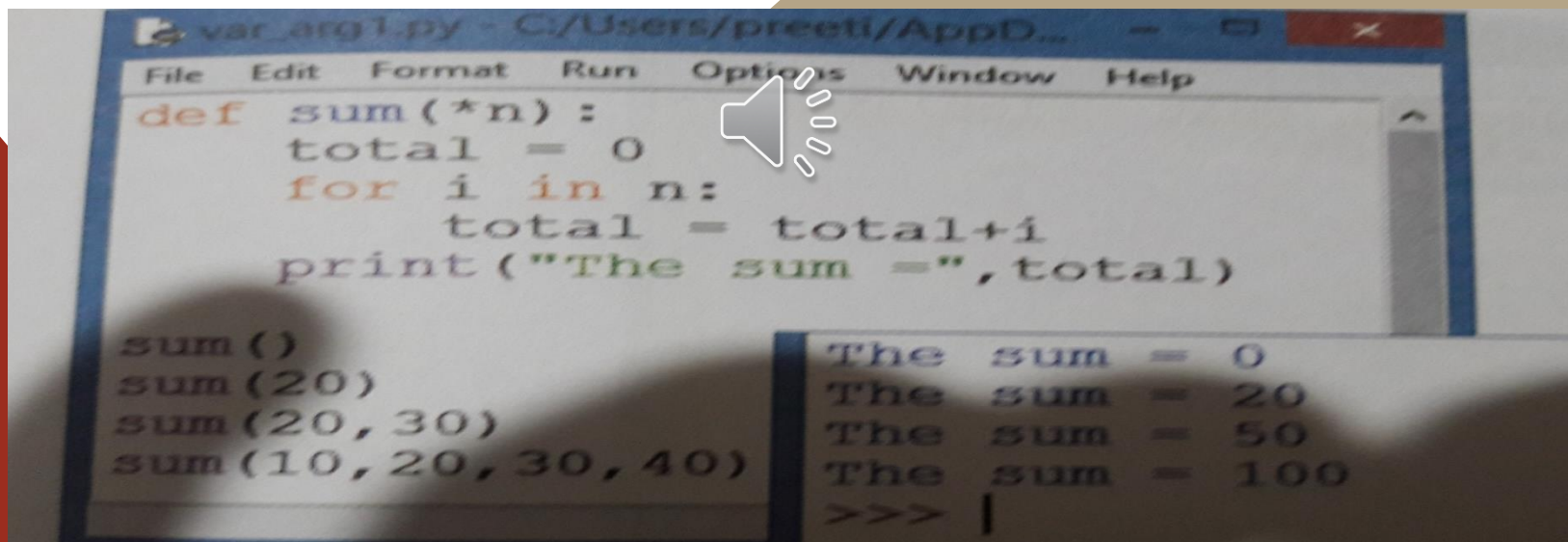
SyntaxError

positional argument follows keyword argument

- **VARIABLE LENGTH ARGUMENT**

) **Variable length arguments:** As the name suggests, in certain situations, we can pass variable number of arguments to a function. Such type of arguments are called variable length arguments.

Variable length arguments are declared with * (asterisk) symbol in Python as
>>>def f1(*n):



The screenshot shows a Python IDE window titled 'var_arg1.py'. The code defines a function 'sum' that takes a variable number of arguments (*n). The function initializes 'total' to 0, iterates through each element 'i' in 'n', and adds it to 'total'. It then prints the total sum. Below the code, the function is called with four different sets of arguments: an empty list, a single number, two numbers, and four numbers. The output shows the sum for each case: 0, 20, 50, and 100 respectively. A speaker icon is overlaid on the code.

```
def sum(*n):  
    total = 0  
    for i in n:  
        total = total+i  
    print("The sum =",total)  
  
sum()  
sum(20)  
sum(20,30)  
sum(10,20,30,40)
```

```
The sum = 0  
The sum = 20  
The sum = 50  
The sum = 100  
>>> |
```

CALLING THE FUNCTION:

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

Syntax: function-name(parameter)

Example: ADD(10,20)

OUTPUT: Sum = 30.0



THE RETURN STATEMENT:

The return statement is used to exit a function and go back to the place from where it was called.

There are two types of functions according to return statement:

- a. Function returning some value**
- b. Function not returning any value**

A. FUNCTION RETURNING SOME VALUE :

Syntax: return expression/value

Example-1: Function returning one value

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

Example-2 Function returning multiple values:

```
def sum(a,b,c):
```

```
    return a+5, b+4, c+7
```

```
S=sum(2,3,4)          # S will store the returned values as a tuple
```

```
print(S)
```

```
OUTPUT: (7, 7, 11)
```



Example-3: Storing the returned values separately:

```
def sum(a,b,c):
```

```
    return a+5, b+4, c+7
```

```
s1, s2, s3=sum(2, 3, 4)          # storing the values separately
```

```
print(s1, s2, s3)
```

```
OUTPUT: 7 7 11
```

B. FUNCTION NOT RETURNING ANY VALUE :

The function that performs some operations but does not return any value, called void function.

```
def message():  
    print("Hello")  
m=message()  
print(m)
```

OUTPUT: Hello
None

SCOPE AND LIFETIME OF VARIABLES:

Scope of a variable is the portion of a program where the variable is recognized.

There are two types of scope for variables:



- i) Local Scope**
- ii) Global Scope**

Local Scope: Variable used inside the function. It cannot be accessed outside the function. In this scope, the lifetime of variables inside a function is as long as the function executes. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Global Scope: Variable can be accessed outside the function. In this scope, Lifetime of a variable is the period throughout which the program resides in the memory.

Example:

```
def my_func():  
    x = 10  
    print("Value inside function:",x)  
  
x = 20  
my_func()  
print("Value outside function:",x)
```

**OUTPUT: Value inside function: 10
Value outside function: 20**

Here, we can see that the value of x is 20 initially.

Even though the function my_func() changed the value of x to 10, it did not affect the value outside the function. This is because the variable x inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

On the other hand, variables outside of the function are visible from inside. They have a global scope. We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

The basic rules for global keyword in Python are:

- * When we create a variable inside a function, it's local by default.
- * When we define a variable outside of a function, it's global by default.
- * You don't have to use global keyword.
- * We use global keyword to read and write a global variable inside a function.
- * Use of global keyword outside a function has no effect

Example 1: Accessing global Variable From Inside a Function

```
c = 1 # global variable
```

```
def add():
```

```
    print(c)
```

```
add()
```



When we run above program, the output will be:

1

Example 2: Modifying Global Variable From Inside the Function

```
c = 1 # global variable
```

```
def add():
```

```
    c = c + 2 # increment c by 2
```

```
    print(c)
```

```
add()
```


When we execute the above program, the output shows an error:

UnboundLocalError: local variable 'c' referenced before assignment

This is because we can only access the global variable but cannot modify it from inside the function.

The solution for this is to use the global keyword.

Example 3: Changing Global Variable From Inside a Function using global

```
c = 0 # global variable
```

```
def add():
```

```
    global c
```

```
    c = c + 2 # increment by 2
```

```
    print("Inside add():", c)
```

```
add()
```

```
print("In main:", c)
```



When we run above program, the output will be:

```
Inside add(): 2
```

```
In main: 2
```

In the above program, we define `c` as a global keyword inside the `add()` function.

Then, we increment the variable `c` by 2, i.e. `c = c + 2`. After that, we call the `add()` function.

Finally, we print global variable `c`.

As we can see, change also occurred on the global variable outside the function, `c = 2`.
