Department of Computer Science and Engineering
PES University, Bangalore, India

# Lecture Notes
# Python for Computational Problem Solving
# UE23CS151A

*Lecture #52*

**Functions: Positional and Keyword Arguments**
**Default Parameters**

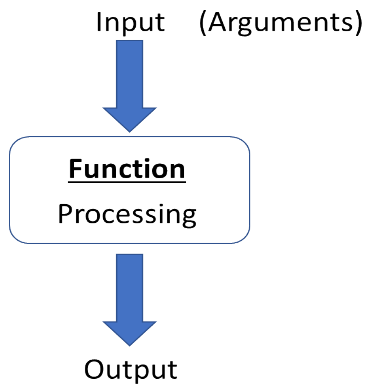
By,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU

Verified by,
PCPS Team - 2023

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)
Prof. Chitra G M(Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

## More on Functions

A function is a self contained block of code that performs a specific task. It can take any **input, performs a set of operations and can return an output** to the program that invoked it.



Two important terms to understand when sending some inputs to function to do processing.

**Arguments:**

Fields used in the function call are known as arguments. **It can be any valid expression in python.**

**Parameters:**

Fields used in the function leader in the function definition are known as parameters. **It should be a variable. And these variables are always local.**

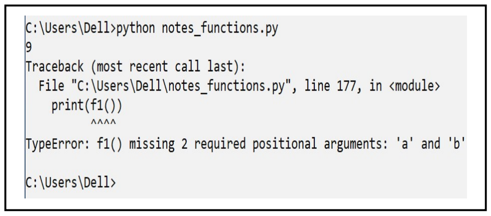
**NOTE:**

Expression in the arguments is evaluated and result is copied to parameters during the function call. Hence the **parameter passing is always by value in python. Another name – call by value.**

**The Number of arguments and the number of parameters must be same if no default parameters. Else results in Error.**

**Let us consider the definition of f1()**

```
def f1():
    print("Hello")
def f1(a,b):
   return a+b
print(f1(4,5))
print(f1())# results in Error
```



```
C:\Users\Dell>python notes_functions.py
9
Traceback (most recent call last):
  File "C:\Users\Dell\notes_functions.py", line 177, in <module>
    print(f1())
          ^^^^
TypeError: f1() missing 2 required positional arguments: 'a' and 'b'

C:\Users\Dell>
```

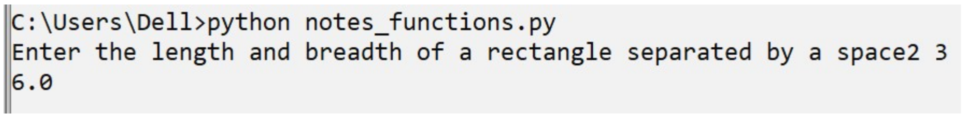
When an interpreter encounters f1() initially, this is added to symbol table. When interpreter encounters the same function name again, entry is made in the symbol table with parameter details such as a and b as 2 parameters. When the function call is f1(4,5), as the function call is matching with the symbol table entry, body executes. But for second call for f1() is invalid as it is not matching with the symbol table entry. Swap the last two statements. This results in error/exception and there is no output of f1(4,5) we can see on the terminal.

**Note: The above code proves that there is NO Overloading of functions in python.**

If two functions have the same name and different arguments, this is known as overloading of functions

**Consider the program to find the area of a rectangle in which the length and breadth is entered by the user.**

```
def find_area(length1,breadth1):
        print(length1* breadth1)
length, breadth = input("Enter the length and breadth of a rectangle separated by aspace").split(" ")
length, breadth = float(length), float(breadth)
find_area(length, breadth)
```



```
C:\Users\Dell>python notes_functions.py
Enter the length and breadth of a rectangle separated by a space2 3
6.0
```

**What happens during the function call?**

On the function call, the activation record of f1 is created with length1 and breadth1 as the parameters. The parameters get a copy of arguments. The parameter is a local variable of the function f1. When the end of the function is reached, the parameter is never copied to the corresponding argument. So, the variable used at the caller side remains unchanged.

When the function call is made, an **Activation Record** is created which will have the following in it.

1. **Parameters** - Are always local.
2. **Local variables** - Variables created within the suite of the function
3. **Return address** - Location to which the control of the program should be transferred once the function terminates.
4. **Temporary variables** - Unnamed variables required by the translator.
5. **Return value** – A value to be passed back to the caller.

An activation record is created every time a function is invoked. It is like a virtual environment in which all the statements in the function suite are executed and the values are stored. At the end of the function execution, when the control returns to the calling program, the activation record is deleted and cannot be accessed again. If we want to access a value that was calculated within the function, we must ensure to return it to the calling program explicitly. If we call the function again in the program, a new activation is created for that instance of the function execution.

If the function call is as below for the above definitions, will the code run? **No.**

```
find_area(2)
find_area([2,3,4],[2,4,1])
find_area()
```

Remember**, function should only do what it is supposed to do. It must not do something extra or lesser.** In above definition of find_area(), small change is made as below.

```
def find_area(length1,breadth1):
	area = length1* breadth1
```

find_area(2,4)   #This function doesn’t print area. It should only find and shouldn’t print area as per the conventional rules.

**Can we print the variable area directly outside the function?**

print(area)  # area is a local variable, cannot access this outside the find_area function

In this situation, **return keyword helps us**. Add return area inside the function definition.

```
def find_area(length1,breadth1):
        area = length1* breadth1
        return area   #If you comment this, None is the output of below function call
print(find_area(2,4))
```

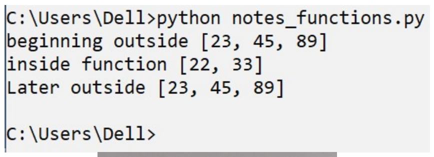
Let us consider **passing the collections** as an argument to a function.

**Few points to note:**

- **Changing the primitive type doesn't affect the argument.**
- **Changing the collection type doesn't affect the argument.**
- **Changing the collection type through the reference changes the argument.**

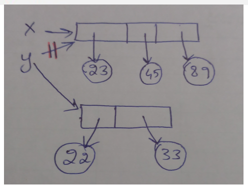
**Example code 1:**

```
x=[23,45,89]
def f1(y):
        y=[22,33]
        print("inside function",y)
print("beginning outside",x)
f1(x)
print("Later outside",x)
```



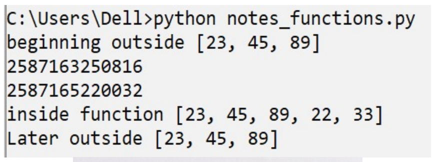
```
C:\Users\Dell>python notes_functions.py
beginning outside [23, 45, 89]
inside function [22, 33]
Later outside [23, 45, 89]

C:\Users\Dell>
```

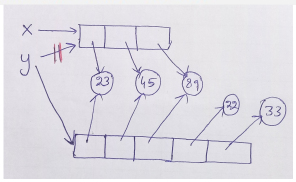


**Example code 2:**

```
x=[23,45,89]
def f1(y):
        print(id(y))
        y=y+[22,33]            # changes the id of y
        print(id(y))
        print("inside function",y)
print("beginning outside",x)
f1(x)
print("Later outside",x)       #x is not changed
```

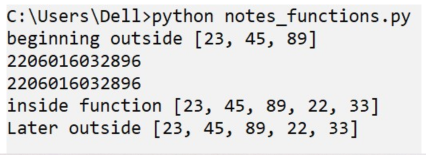


```
C:\Users\Dell>python notes_functions.py
beginning outside [23, 45, 89]
2587163250816
2587165220032
inside function [23, 45, 89, 22, 33]
Later outside [23, 45, 89]
```

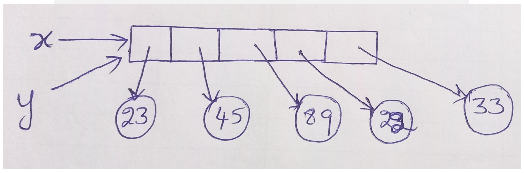


**Example code 3:**

```
x=[23,45,89]
def f1(y):
        print(id(y))
        y+=[22,33]  # change this to y.extend([22,33])
        print(id(y))      #no change in id of y.
                          #same as y above and x outside
        print("inside function",y)
print("beginning outside",x)
f1(x)
print("Later outside",x) #x is affected
```

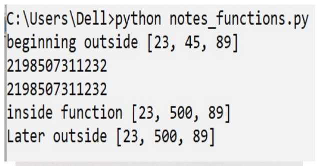


```
C:\Users\Dell>python notes_functions.py
beginning outside [23, 45, 89]
2206016032896
2206016032896
inside function [23, 45, 89, 22, 33]
Later outside [23, 45, 89, 22, 33]
```

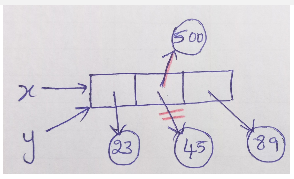


**Example code 4:**

```
x=[23,45,89]
def f1(y):
        print(id(y))
        y[1] = 500
        print(id(y))      #no change in id of y.
                          #same as y above and x outside
        print("inside function",y)
print("beginning outside",x)
f1(x)
print("Later outside",x) #x is affected
```



```
C:\Users\Dell>python notes_functions.py
beginning outside [23, 45, 89]
2198507311232
2198507311232
inside function [23, 500, 89]
Later outside [23, 500, 89]
```

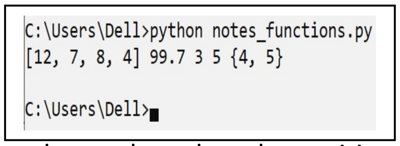


Now, let us try to understand **two types of Arguments: Positional Arguments and Keyword Arguments**

**Positional Arguments:**

Based on the position of arguments and parameters, arguments are copied to parameters.

```
def f1(x, y, z, a, b):
	print(a, b, x, y, z)
f1(3,5,{4,5},[12,7,8,4], 99.7)
# 3 is copied to x, 5 copied to y,{4,5} is copied to z and so on based on the position.
```



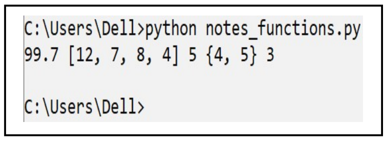
```
C:\Users\Dell>python notes_functions.py
[12, 7, 8, 4] 99.7 3 5 {4, 5}

C:\Users\Dell>
```

**Keyword Arguments:**

Based on the parameter names, arguments are copied to parameters. Parameter names are used in the arguments.

```
def f1(x, y, z, a, b):
	print(a, b, x, y, z)
f1(z = 3,x = 5,y = {4,5},b = [12,7,8,4], a = 99.7)
#observe the use of parameter names in arguments in the function call as the LHS of assignment operator
```



```
C:\Users\Dell>python notes_functions.py
99.7 [12, 7, 8, 4] 5 {4, 5} 3

C:\Users\Dell>
```

**Combination of both Positional and Keyword Arguments:**

Rule: All keyword arguments must follow positional arguments

```
def f1(x, y, z, a, b):
	print(a, b, x, y, z)
f1(3,5,z = {4,5},b = [12,7,8,4], a = 99.7) #    99.7 [12, 7, 8, 4] 3 5 {4, 5}
#f1(3,5,x = {4,5},b = [12,7,8,4], a = 99.7) #TypeError: f1() got multiple values for argument 'x'
#f1(x = 9, y = 99,2,3,b = 22)  #SyntaxError: positional argument follows keyword argument
```

**Note: In all above cases, the number of arguments is equal to the number of parameters.**

If the user is not sure of how many positional arguments and how many keyword arguments to be sent in the function call, how do we handle this situation? Refer to Lecture #53_#54

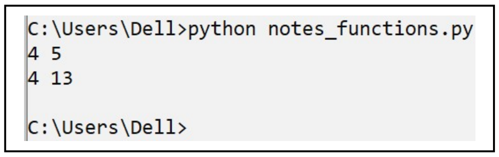
Sometimes, user might not want to send inputs to all parameters. How do we handle this situation? We can have default parameters in the function definition.

**Default Parameters:**

Default parameters **are always a part of the symbol table which is added during the leader processing phase. If the user did not send the argument, then this default parameter is used in the processing.**

**Example code 5:**

```
def f1(a,b=5):
        print(a,b)
f1(4)
f1(4,13)
```

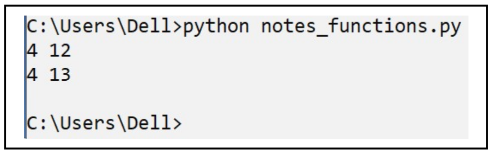


```
C:\Users\Dell>python notes_functions.py
4 5
4 13

C:\Users\Dell>
```

**Example code 6:**

```
x = 12
def f1(a,b=x):
        print(a,b)
f1(4)
f1(4,13)
```

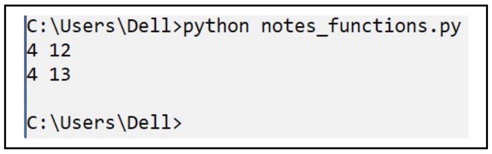


```
C:\Users\Dell>python notes_functions.py
4 12
4 13

C:\Users\Dell>
```

**Example code 7:**

```
x = 12
def f1(a,b=x):    #at the time of leader processing x is 12 and then x is modified
        print(a,b)
x = 5
f1(4)
f1(4,13)
```

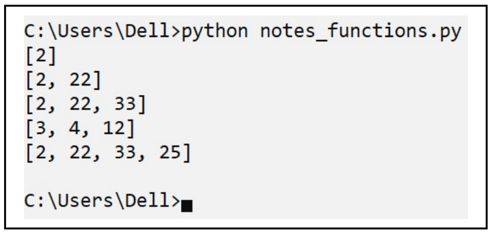


```
C:\Users\Dell>python notes_functions.py
4 12
4 13

C:\Users\Dell>
```

**Example code 8:**

```
def f1(a,b=[]):
        b.append(a)
        print(b)
f1(2)
f1(22)
f1(33)
f1(12,[3,4])
f1(25)
```



```
C:\Users\Dell>python notes_functions.py
[2]
[2, 22]
[2, 22, 33]
[3, 4, 12]
[2, 22, 33, 25]

C:\Users\Dell>
```

**-END-**