

Functions

Chapter - 10

What are functions ?

- As you call the text blocks in your essays a paragraph, you call the text blocks in Python: **functions**.
- Paragraphs and functions have the same purpose. A paragraph and a function both breaks the text into sub-topics.

Example

I made a game, in which you've to raise a dog

I wrote a book, in which I describe how my dog behaves

In this game the user can click:

- feed
- pet
- walk the dog

I write these different points in different methods

In my book I described:

- how I feed my dog
- how I pet my dog
- how I walk the dog

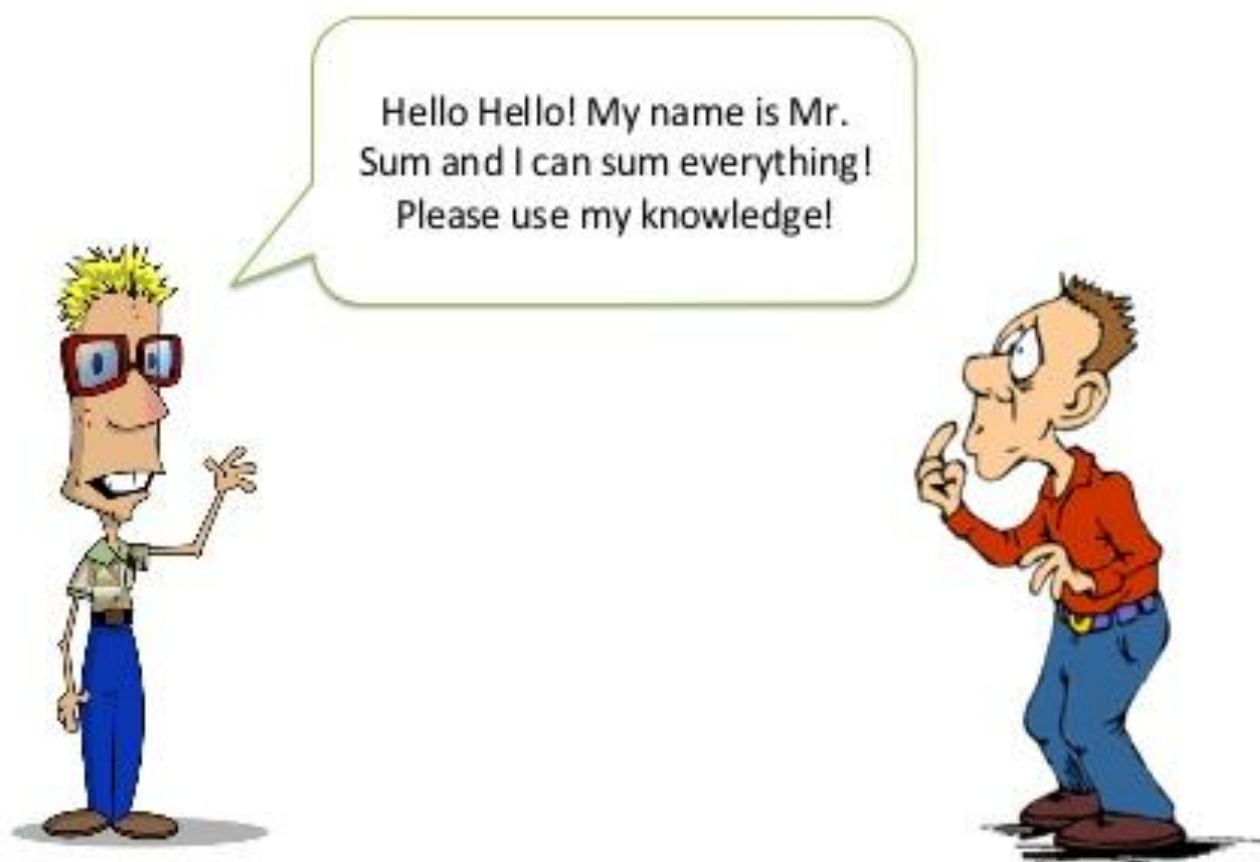
I write these different points in different paragraphs

Real Life Example

I really want to buy christmas gifts for my children. The doll costs 5 euros and the toy car costs 6 euros. I have 12 euros, is that enough? Why didn't I listen to the math teacher at school? Why didn't I learn sum – calculations? Why oh why?



Real Life Example

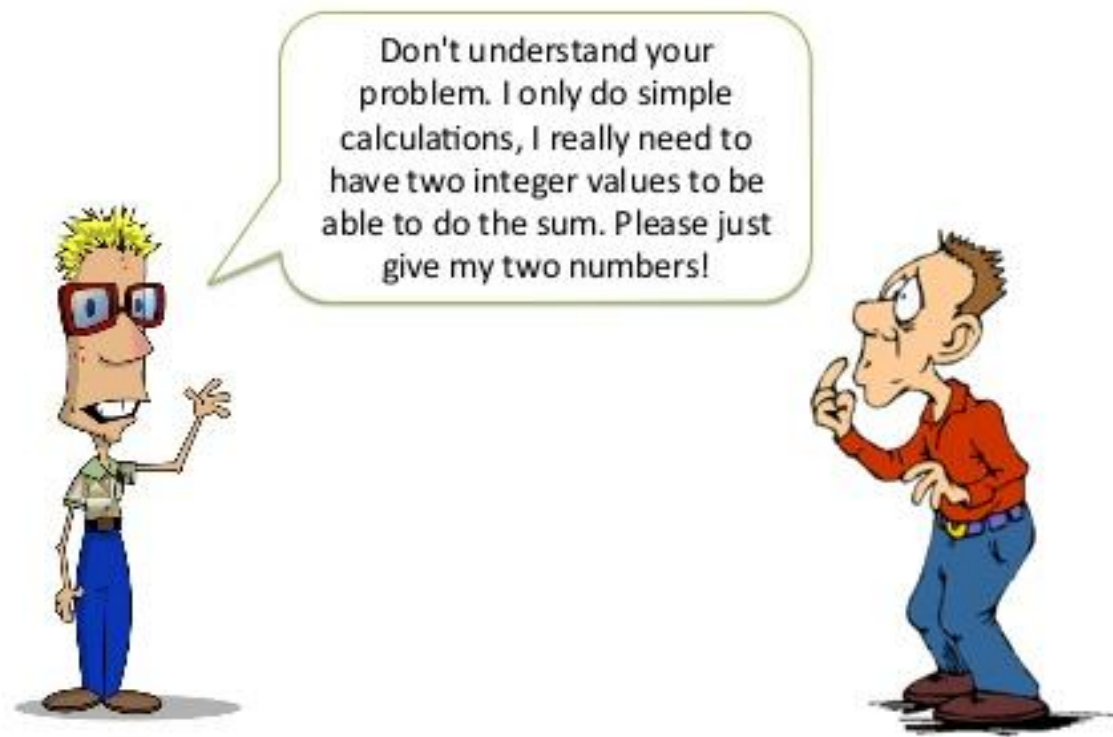


Real Life Example

Well Mr Sum. I have a problem I
don't know what is 5 plus 6.



Real Life Example



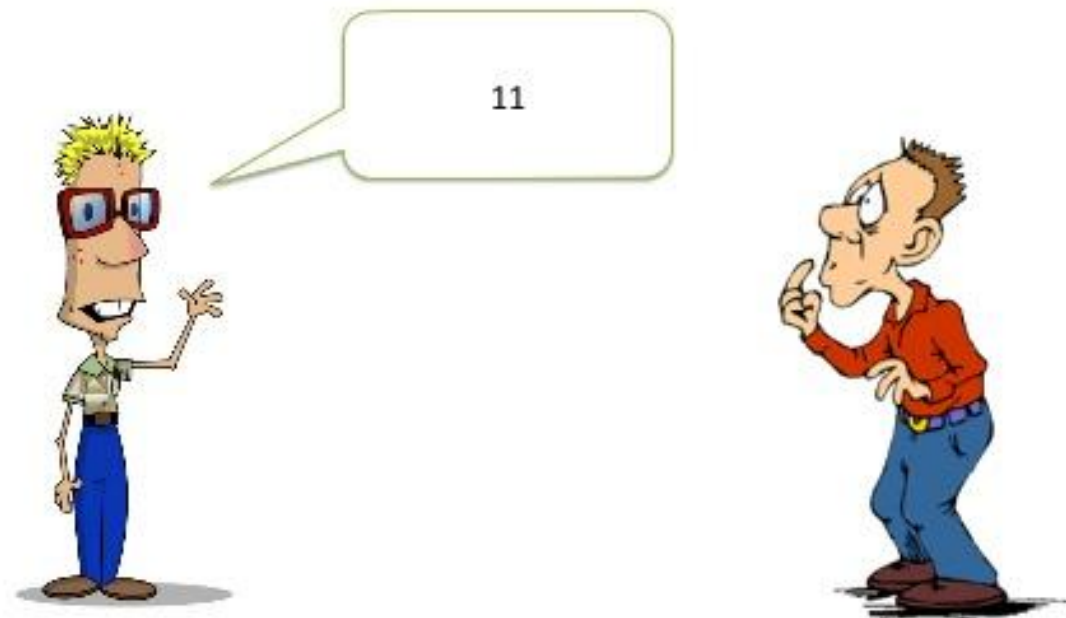
Parameters!



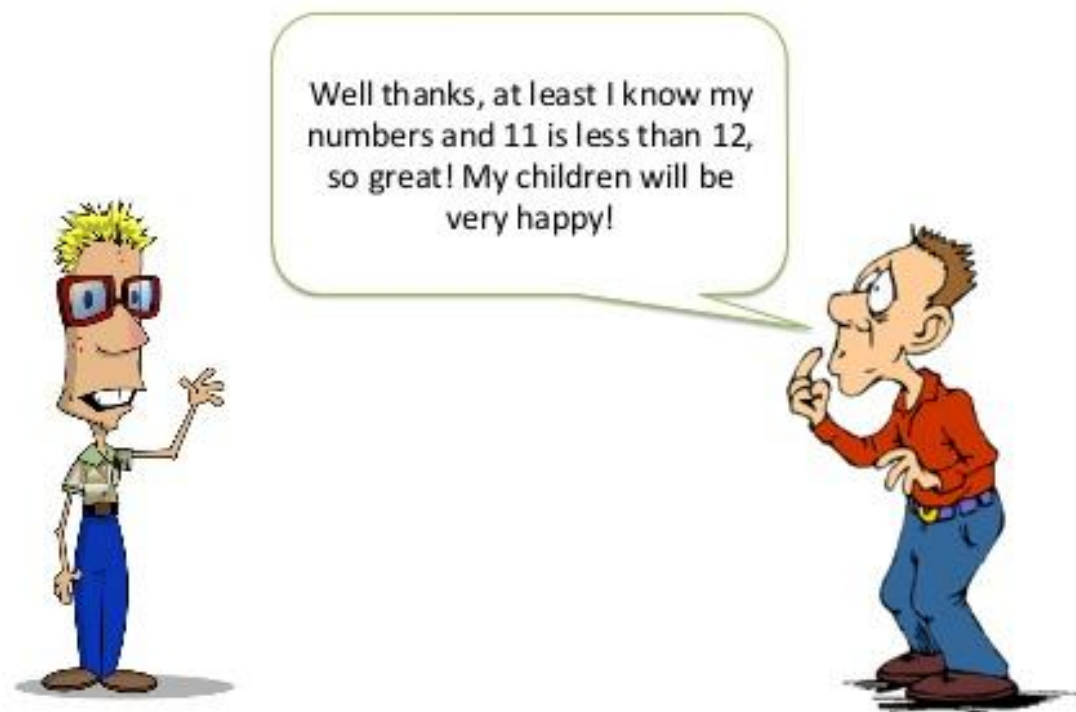
5
6



Return!



Real Life Example



Mr. Sum?



What is a function in Python?

In Python, a function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

In addition, it avoids repetition and makes the code reusable.

Types of functions

There are three types of functions in Python:

- Built-in functions, such as `help()` to ask for help, `min()` to get the minimum value, `print()` to print an object to the terminal,....
- User-Defined Functions (UDFs), which are functions that users create to help them out.
- Anonymous functions, which are also called lambda functions because they are not declared with the standard `def` keyword.

Function name

An identifier by which the function is called

Arguments

Contains a list of values passed to the function

```
def name(arguments):
```

```
    statement
```

```
    statement
```

```
    ...
```

```
    return value
```

Indentation

Function body must be indented

Function body

This is executed each time the function is called

Return value

Ends function call & sends data back to the program

Syntax of Function

- The keyword ***def*** marks the start of the function header. The Python `def` is a true executable statement: when it runs, it creates a new function object and assigns it to a name.
- A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.

Syntax of Function

- A colon (:) to mark the end of the function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
- An optional return statement to return a value from the function.

Example - function

```
# A Basic Function that accepts no and returns nothing.
def hello_world():
    print("Hello, World!")

hello_world()

#A Function that accepts two arguments, and returns the sum of
# those numbers added together.
def add_numbers(x, y):
    return x+y

x=y=1
sum=add_numbers(x, y)
print(sum)
```

Parameters and Arguments

- From a function's perspective:
A **parameter(formal parameter)** is the variable listed inside the parentheses in the function definition.
- An **argument(actual argument)** is the value that are sent to the function when it is called.
- But sometimes, they are used interchangeably

Function Definition

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

Parameters

Function Call

```
add(2, 3)
```

Arguments

Calling a 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.
- If the function accepts parameters, we have to pass them while calling the function.
- Example :*hello_world()* and *add_numbers(10,5)*

```
def functionName():
```

```
    ... ..
```

```
    ... ..
```

```
    ... ..
```

```
    ... ..
```

```
functionName();
```

```
    ... ..
```

```
    ... ..
```



Working of functions in Python

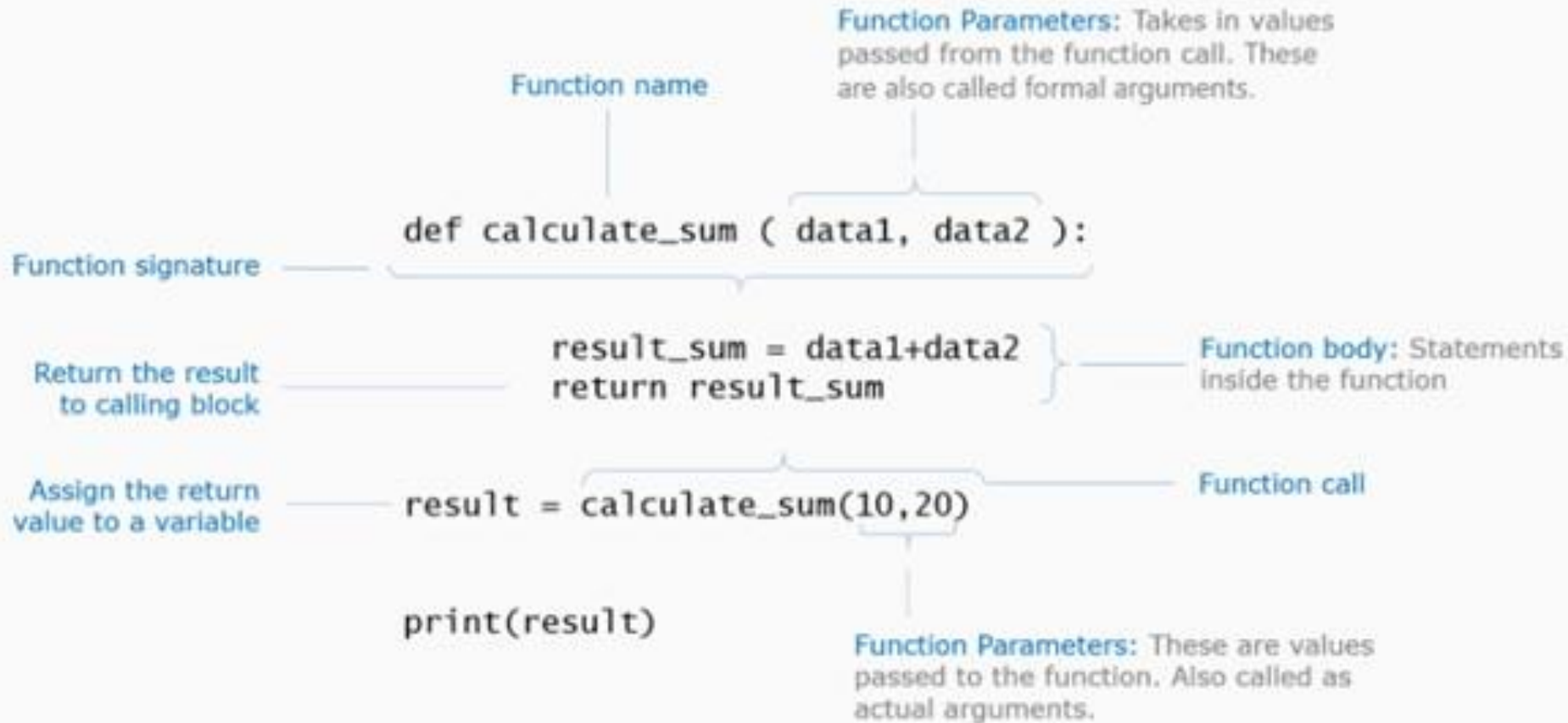
The return statement

- The return statement is used to exit a function and go back to the place from where it was called.
- This statement can contain an expression that gets evaluated and the value is returned, and it can return a tuple/list/dict/set or a single value.
- If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the **None** object.

Syntax of return

```
return [expression_list]
```

return type	'return' statement
Return single value	return a
Return a tuple	return a,b
Return a list	return [a,b]
Return a dictionary	return {1:a,2:b}
Return a set	return {a,b}



Function and function call

None as a return from a function

```
def foo():  
    pass  
  
print(foo())
```

None

>>> █

Returning None Explicitly

There are situations in which you can add an explicit return None to your functions. In other situations, however, you can rely on Python's default behaviour:

- If your function performs actions but doesn't have a clear and useful return value, but you want to have a bare return without a return value just to make clear your intention of returning from the function.
- If your function has multiple return statements and returning None is a valid option, then you should consider the explicit use of return None instead of relying on the Python's default behavior.

Returning None Explicitly

When it comes to returning None, you can use one of three possible approaches:

- Omit the return statement and rely on the default behavior of returning None.
- Use a bare return without a return value, which also returns None.
- Return None explicitly.

```
def omit_return_stmt():  
    # Omit the return statement  
    pass  
  
print(omit_return_stmt())  
  
def bare_return():  
    # Use a bare return  
    return  
  
print(bare_return())  
  
def return_none_explicitly():  
    # Return None explicitly  
    return None  
  
print(return_none_explicitly())
```

MULTIPLE RETURN STATEMENTS

- When we have multiple return statements, the function call is terminated when it reaches the first return statement.

```
def absolute_value(num):  
    """ This Funcyion returns the absolut  
    value of the entered number """  
    if num>=0:  
        return num  
    else:  
        return (num*-1)  
  
print(absolute_value(2))  
print(absolute_value(-4))
```

Output

```
2  
4
```

Multiple return statements

- Here, an explicit value is specified in multiple return statements.

```
def differnt_data(choice):  
  
    if choice ==1:  
        return 21  
    elif choice == 2:  
        return 3.14  
    else:  
        return "Try again!"  
  
answer = differnt_data(1)  
print(answer)  
  
answer = differnt_data(2)  
print(answer)  
  
answer = differnt_data(3)  
print(answer)
```

Returning from a function

Code 1:

```
def func(arg1):  
    ...  
    ...  
    return result
```

func(arg1)

↑
Returned value is not assigned to any variable. Hence the return value from this function invocation is lost

Code 2:

```
def func(arg1):  
    ...  
    ...  
    return result
```

res=func(arg1)

↑
Returned value is assigned to a variable. This can be used in the rest of the program

Code 3:

```
def func(arg1):  
    ...  
    ...  
    return result
```

print(func(arg1))

↑
Returned value is directly printed

Code 4:

```
def func(arg1):  
    ...  
    ...  
    return result
```

if(func(arg1)):

...
...

↑
If the returned value from a function is boolean, then the function invocation can be written directly in a conditional statement (if, while etc)

Recognizing Dead Code

- As soon as a function hits a return statement, it terminates without executing any subsequent code. Consequently, the code that appears after the function's return statement is commonly called dead code.
- The Python interpreter totally ignores dead code when running your functions. So, having that kind of code in a function is useless and confusing.

Recognizing Dead Code

- The statement `print("Hello, World")` in this example will never execute because that statement appears after the function's return statement.
- Identifying dead code and removing it is a good practice that you can apply to write better functions.

```
def dead_code():  
    return 43  
    # Dead code  
    print("Hello, World")  
  
print(dead_code())
```

Recognizing Dead Code

- It's worth noting that if you're using conditional statements to provide multiple return statements, then you can have code after a return statement that won't be dead as long as it's outside the if statement:

```
def no_dead_code(condition):  
    if condition:  
        return 42  
        print("Hello, World")  
  
no_dead_code(True)  
  
no_dead_code(False)
```

Return Multiple Values

- Python can return multiple values, something missing from many other languages. You can do this by separating return values with a comma.

```
# Return addition and subtraction in a tuple
def func(a,b):
    return a+b,a-b

result= func(3,2)

print(result)
```


Return Multiple Values

- When you return multiple values, Python packs them in a **single tuple** and returns it.
- You can then use multiple assignment to unpack the parts of the returned tuple.

```
#Unpack returned tuple
def func(a,b):
    return a+b,a-b

add, sub = func(3,2)

print(add)

print(sub)
```

Returning multiple **types** of values

- Unlike other programming languages, python functions are not restricted to return a single type of value.
- If you look at the function definition, it doesn't have any information about what it can return.

```
def test2():  
    return 'abc', 100, [0,1,2]  
  
a,b,c = test2()  
  
print(a)  
print(b)  
print(c)
```

Types of Arguments

- Python handles function arguments in a very flexible manner, compared to other languages. It supports multiple types of arguments in the function definition. Here's the list:
 - Positional Arguments
 - Keyword Arguments
 - Default Arguments
 - Variable Length Positional Arguments (*args)
 - Variable Length Keyword Arguments (**kwargs)

Positional Arguments

- The most common are positional arguments, whose values are copied to their corresponding parameters in order.
- The only downside of positional arguments is that you need to pass arguments in the order in which they are defined.

```
def func(name, job):  
    print(name, 'is a', job)
```

```
func('Bob', 'developer')
```

```
def func(name, job):  
    print(name, 'is a', job)
```

```
func('developer', 'Bob')
```

Positional Arguments

- Look into the function `greet()` that has two parameters.
- Since we have called this function with two arguments, it runs smoothly, and we do not get any error.

```
def greet(name, msg):  
    """This function greets to  
    the person with the provided message"""  
    print("hello" +name+", " + msg)  
  
greet("Monica","Good morning!")
```

```
def greet(name, msg):  
    """This function greets to  
    the person with the provided message"""  
    print("Hello", name + ', ' + msg)  
  
greet("Monica", "Good morning!")
```

Output

```
Hello Monica, Good morning!
```

Positional Arguments

- If we call it with a different number of arguments, the interpreter will show an error message.
- Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> greet("Monica")    # only one argument  
TypeError: greet() missing 1 required positional argument: 'msg'
```

```
>>> greet()    # no arguments  
TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'
```

Keyword Arguments

- A keyword argument is an argument passed to a function or method which is preceded by a keyword and an equals sign.
- The form of the function call is:
 - `function(keyword=value)`
- Where function is the function name, keyword is the keyword argument and value is the value or object passed to that keyword.

Keyword Arguments

- To avoid positional argument confusion, you can pass arguments using the names of their corresponding parameters.
- In this case, the order of the arguments no longer matters because arguments are matched by name, not by position.
- Like with positional arguments, though, the number of arguments and parameters must still match.

```
#Keyword arguments can be put in any order
def func(name, job):
    print(name,"is a", job)

func(name="Bob", job= "developer")
func(job= "developer", name="Bob")
```


Mixing positional and keyword arguments

- It is possible to combine positional and keyword arguments in a single call.
- If you do so, specify the **positional arguments before keyword arguments, else you'll get an error.**

```
def func(name, job):  
    print(name, "is a", job)  
  
func("Bob", job= "developer")
```

```
Bob is a developer
```

```
def func(name, job):  
    print(name, "is a", job)  
  
func(job= "developer", "Bob")
```

```
func(job='developer', 'Bob')  
      ^  
SyntaxError: positional argument follows keyword argument
```

Default Parameters

- Functions with **optional arguments** offer more flexibility in how you can use them.
- You can call the function with or without the argument.
- When you define a function, you can specify a default value for each parameter.
- To specify default values for parameters, you use the following syntax:

```
def function_name(param1, param2=value2, param3=value3, ...):
```

- In this syntax, you specify default values (value2, value3, ...) for each parameter using the assignment operator (=).

Default Parameters

- When you call a function and pass an argument to the parameter that has a default value, the function will use that argument instead of the default value.
- However, if you don't pass the argument, the function will use the default value.

```
def popcorn_time(time, genre='action', watch='web series'):  
    print(f"You have {time} minutes!")  
  
    print(f"Let'd watch a {genre} type {watch}")  
  
popcorn_time(120)  
popcorn_time(150, 'thriller')  
popcorn_time(150, 'thriller', 'movie')
```

```
You have 120 minutes!  
Let's watch a action type web series  
You have 150 minutes!  
Let's watch a thriller type web series  
You have 200 minutes!  
Let's watch a horror type movie
```

Default Parameters

This function is called in several ways:

- giving only the mandatory argument: `popcorn_time(120)`
- giving one of the optional arguments: `popcorn_time(150, 'thriller')`
- or even giving all arguments: `popcorn_time(200, 'horror', 'movie')`

Default Values

- Any number of arguments in a function can have a default value, But once we have a default argument, all the arguments to its right must also have default values.
- This means to say, **non-default arguments cannot follow default arguments.** For example, if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```

```
def popcorn_time(time, genre='action', watch='web series'):
    print(f"You have {time} minutes!")
    print(f"Let'd watch a {genre} type {watch}")
```

```
popcorn_time(120)
popcorn_time(150, 'thriller')
popcorn_time(time=180, watch='TV-show')
popcorn_time(watch='Documentary', time=100)
popcorn_time(200, 'horror', 'movie')
```

```
# 1 positional argument
# 1 keyword argument
# 1 positional, 1 keyword
# 2 keyword arguments
# 2 keyword arguments
# 3 positional arguments
```

Mixing all the three types of arguments

► But take a note that, the following function calls would be invalid:

```
popcorn_time()
popcorn_time(time=160, 'horror')
popcorn_time(150, time=150)
popcorn_time(platfrom='Netflix')
```

```
# required argument missing
# non-keyword argument after a keyword argument
# duplicate value for the same argument
# unknown keyword argument
```

Example – positional, keyword and default arguments in a function

```
def check_passwd(username, password, min_length=8, check_username=True):  
    if len(password) < min_length:  
        print('Password is too short')  
        return False  
    elif check_username and username in password:  
        print('Password contains username')  
        return False  
    else:  
        print(f"Password for user {username} has passed all checks")  
        return True  
  
check_passwd('angela', 'angela@22', min_length = 3)  
check_passwd('angela', 'angela@22', min_length = 3, check_username=True)  
check_passwd('angela', 'angela@22', min_length = 3, check_username=False)
```

```
Password contains username  
Password contains username  
Password for user angela has passed all checks
```

Variable Length Arguments (*args and **kwargs)

- Variable length arguments are useful when you want to create functions that take unlimited number of arguments. Unlimited in the sense that you do not know beforehand how many arguments can be passed to your function by the user.
- This feature is often referred to as **var-args**.

*args

```
def print_arguments(*args):  
    print(args)  
  
print_arguments(1, 54, 60, 8, 98, 12)
```

- When you prefix a parameter with an asterisk * , it collects all the unmatched positional arguments into a tuple.
- Because it is a normal tuple object, you can perform any operation that a tuple supports, like indexing, iteration etc.
- The function on the left prints all the arguments passed to the function as a tuple.
- You don't need to call this parameter as args, but it is standard practice.

Example - 2

- Here, we have called the function, greet() with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function.
- Inside the function, we use a for loop to retrieve all the arguments back.

```
def greet(*names):  
    """ This function greets all  
    the person in the names tuple. """  
    for name in names:  
        print("Hello", name)  
  
greet("Monic","Luke", "Steve", "John")
```

Output

```
Hello Monica  
Hello Luke  
Hello Steve  
Hello John
```

```
def sum(*n):  
    total=0  
    for n1 in n:  
        total=total+n1  
    print("The Sum=",total)
```

```
sum()  
sum(10)  
sum(10,20)  
sum(10,20,30,40)
```

Output

The Sum= 0

The Sum= 10

The Sum= 30

The Sum= 100

Example - 3

*args

- We can also mix the positional arguments with the variable length arguments.

```
def f1(n1,*s):  
    print("n1 is:",n1)  
    for s1 in s:  
        print("s1 is: ",s1)
```

```
f1(10)
```

```
f1(10,20,30,40)
```

```
f1(10,"A",30,"B")
```

```
>>> f1(10,20,30,40)
```

```
n1 is: 10
```

```
s1 is: 20
```

```
s1 is: 30
```

```
s1 is: 40
```

```
>>> f1(10,"A",30,"B")
```

```
n1 is: 10
```

```
s1 is: A
```

```
s1 is: 30
```

```
s1 is: B
```

*args

- Note: After variable length argument, if we are taking any other arguments then those should be the keyword arguments.

```
>>> def f1(*s,n1):
        print("n1 is:",n1)
        for s1 in s:
            print("s1 is:",s1)

>>> f1(10)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    f1(10)
TypeError: f1() missing 1 required keyword-only argument: 'n1'
>>> f1("A","B",n1=10)
n1 is: 10
s1 is: A
s1 is: B
>>> |

>>> f1(n1=10)
n1 is: 10
>>> |
```

Mixing the positional, default and var-args and keyword parameter

```
def test(a, b=1, *c):  
    print(a,b)  
    for c1 in c:  
        print(c1)  
  
test(10,20,30,40,50)  
test(10,30,40,50)  
test(10)
```

```
10 20  
30  
40  
50  
10 30  
40  
50  
10 1  
PS C:\Users\Angelo
```

```
def test1(*c,b=1):  
    print("test1")  
    print("b:",b)  
    for c1 in c:  
        print("c:",c1)  
  
test(10,20,30,40,50)  
test(10,30,40,50)  
test(10)
```


```
test1  
b: 1  
c: 10  
c: 20  
c: 30  
c: 40  
c: 50  
test1  
b: 1  
c: 10  
c: 30  
c: 40  
c: 50  
test1  
b: 1  
c: 10
```

Mixing the positional, default and var-args and keyword parameter

```
def test(a,*c,b=1):  
    print("test1")  
    print("a:",a)  
    print("b:",b)  
    for c1 in c:  
        print("c:",c1)  
  
test(10,20,30,40,50, a=100)  
test(10,30,40,50, b=20)  
test(10)
```

```
a: 10  
b: 20  
c: 30  
c: 40  
c: 50  
test1  
a: 10  
b: 1  
c: 20  
c: 30  
c: 40  
PS C:\Use
```

```
Traceback (most recent call last):  
  File "c:\Users\Angela\Desktop\PythonScripts\Classwork\Mine\func1.py", line 37, in <module>  
    test1(20,30,40,50,a=10)  
TypeError: test1() got multiple values for argument 'a'
```

Mine >  func1.py > ...

```
65
66 def test1(a,b=1,*c):
67     print("test1")
68     print("a:",a)
69     print("b:",b)
70     for c1 in c:
71         print("c:",c1)
72
73 test1(20,30,40,50,a=10)
74 test1(10,30,40,50,b=20)
75 test1(10,20,30,40)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\Angela\Desktop\PythonScripts\Classwork> 

**kwargs

- The ** syntax is similar to var args, but it **only works for keyword arguments**. It collects them into a new **dictionary**, where the argument names are the keys, and their values are the corresponding dictionary values.

```
def print_arguments(**kwargs):  
    print(kwargs)  
  
print_arguments(names="Bob", age=25, job='dev')
```

Another Example

```
def marvel(**movie):  
    print("Marvel Stuiod prents - ", movie["name"])  
    print("Starrying -", movie["actor"])  
  
marvel(name ="Iran Man", actor= "Robert Downey Jr.")  
marvel(name = "Captain America: The First Aveger", actor = "Chris Evans")  
marvel(name = "Thor", actor = "Chris Hamsworth")
```

```
Marvel Studios presents - Iron Man  
Starring - Robert Downey Jr.  
Marvel Studios presents - Captain America: The First Avenger  
Starring - Chris Evans  
Marvel Studios presents - Thor  
Starring - Chris Hemsworth
```

Ordering Arguments in a Function

When mixing the positional and named arguments, positional arguments must come before keyword arguments. Furthermore, arguments of a fixed length come before arguments with variable length. Therefore, we get an order like this:

1. positional arguments
2. *args
3. default arguments
4. **kwargs

A function with all types of arguments can look like this:

```
def super_function(num1, num2, *args, callback=None, message=[], **kwargs):  
    pass
```

All types of arguments

```
def test1(a,*c,b=1,**d):  
    print("test1")  
    print("a:",a)  
    print("b:",b)  
    for c1 in c:  
        print("c:",c1)  
    print("d:",d)  
  
test1(10,20,30,40,50, b=20)  
test1(10,30,40,dict="hi")  
test(10,20,30,40,50, b=1)
```

```
Python/Python39/py  
test1  
a: 10  
b: 20  
c: 30  
c: 40  
c: 50  
d: {}  
test1  
a: 10  
b: 1  
c: 20  
c: 30  
c: 40  
d: {'dict': 'hi'}  
test1  
a: 10  
b: 1  
c: 20  
c: 30  
c: 40  
d: {}  
PS C:\Users\Angel
```

```
test1(10,30,40,50,b  
test1(10,20,30,40,d  
test1(10,20,30,40,b  
(parameter) b: Any  
Parameter cannot follow "*" parameter Pylance  
View Problem \(Alt+F8\) No quick fixes available  
  
def test1(a,*c,**d,b=1):  
    print("test1")  
    print("a:",a)  
    print("b:",b)  
    for c1 in c:  
        print("c:",c1)  
    print("d:",d)
```

- 1.default arguments should follow non-default arguments
- 2.keyword arguments should follow positional arguments
- 3.All the keyword arguments passed must match one of the arguments accepted by the function and their order is not important.
- 4.No argument should receive a value more than once
- 5.Default arguments are optional arguments.

Unpacking Arguments with *args and **kwargs

- Let's consider a function add3(), that accepts 3 numbers and prints their sum. We can create it like this:

```
def add3(num1, num2, num3):  
    print("The grand total is", num1 + num2 + num3)
```

- If you had a list of numbers, you can use this function by specifying which list item is used as an argument:
- If you run this code, you will see:

```
magic_nums = [32, 1, 7]  
add(magic_nums[0], magic_nums[1], magic_nums[2])
```

```
The grand total is 40
```

Unpacking Arguments

- While this works, we can make this more succinct with `*args` syntax:

```
add3(*magic_nums)
```

- The output is The grand total is 40, just like before.
- When we use `*args` syntax in a function call, we are unpacking the variable. By unpacking, we mean that we are pulling out the individual values of the list. In this case, we pull out each element of the list and place them in the arguments, where position 0 corresponds to the first argument.

Unpacking Arguments

- You can also similarly unpack a tuple:

```
tuple_nums= (32,1,7)
add(*tuple_nums)
```

- If you would like to unpack a dictionary, you must use the `**kwargs` syntax.

```
dict_nums = {
    'num1' : 32,
    'num2' : 1,
    'num3' : 7,
}

add(**dict_nums)
```

- In this case, Python matches the dictionary key with the argument name and sets its value. And that's it! You can easily manage your function calls by unpacking values instead of specifying each argument that needs a value from an object.


```
Code 1: Positional
def func(arg1,arg2):
    ...
    ...
    return result
```

```
res=func(val1,val2)
```

Default way of specifying arguments.
In this, the order, count and type of actual arguments should exactly match that of the formal arguments. Else, it will result in error.

```
Code 3: Default
def func(arg1,arg2=default_value):
    ...
    ...
    return result
```

```
res=func(val1)
```

Allows to specify default value for an argument in the function signature. It is used only when no value is passed for that argument else it works normally. In Python, default arguments should be last in the order.

```
Code 2: Keyword
def func(arg1,arg2):
    ...
    ...
    return result
```

```
res=func(arg2=val2,arg1=val1)
```

Allows flexibility in the order of passing the actual arguments by mentioning the argument name.

```
Code 4: Variable argument count
def func(arg1,arg2,*arg3)
    ...
    ...
    return result
```

```
res=func(val1,val2,val3,val4,val5)
```

Allows a function to have variable number of arguments. In Python, any argument name starting with '*' is considered to be a varying length argument. It should be last in the order. It works by copying all values beyond that position into a tuple

