Functions

What is Functions?

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

You can pass data, known as parameters,

A function can return data as a result.

```
keyword name ( parameters or arguments ):
        """
            documentation string, Explanation about your function so that
            a different person can use it easily without being confuse.
        """
        BODY

Later in the code, you call the function using its name
```

Creating a Function

It is defined using `def` keyword:

```
def my_fucntion():
        print ('hello')
```

Calling a Function

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

```
def my_fucntion():
        print('hello')

my_function() # hello
```

Arguments

- Information can be passed into functions as arguments.
- Arguments are specified after the function name,

```
def my_fucntion(fname):
        print (fname + 'hello')

my_function('adi') # adihello
my_function('bro') # brohello
```

- You can add as many arguments as you want, just separate them with a comma.
- A function must be called with the correct number of arguments.

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

Arbitrary Arguments - `*args`

If the number of arguments is unknown, add a `*` before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Keyword Arguments - `kwargs`

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

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Arbitrary Keyword Arguments - `**kwargs`

If the number of `kwargs` is unknown, add `**` before the parameter

```
def my_function(**kid):
  print("His last name is " + kid["lname"])
```

```
my_function(fname = "Tobias", lname = "Refsnes")
```

Default Parameter Value

```
def my_function(country= 'Norway'):
        print ( 'I am from ' + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")

#I am from Sweden
#I am from India
#I am from Norway
#I am from Brazil
```

Passing a List as an Argument

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)

#apple
#banana
#cherry
```

Return Values

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

print(my_function(3)) # 15
print(my_function(5)) # 25
print(my_function(9)) # 45
```

Pass Statement

`function` definitions can't 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.

```
def my_fucntion():
    pass
```

Positional-Only Arguments - `, /`

Positional arguments in Python are values passed to a function in the order they are defined. This means that the first value you pass will be assigned to the first parameter, the second value to the second parameter, and so on.

```
def greet(name, age):
    print("Hello,", name, "!")
    print("You are", age, "years old.")

greet("Alice", 30)

# Hello, Alice !
# You are 30 years old.
```

- To specify that a function can have only positional arguments, add `, /` after the arguments:

```
def my_function(x, /):
  print(x)

my_function(3) # 3
```

- Without the `, /` you are actually allowed to use keyword arguments even if the function expects positional arguments:

```
def my_function(x):
  print(x)

my_function(x = 3) # 3
```

- But when adding the `, /` you will get an error if you try to send a keyword argument:

```
def my_function(x, /):
  print(x)

my_function(x = 3)
#typeError: my_function() got some positional-only arguments passed as keyword arguments: '
```

Keyword-Only Arguments - `*,`

```
def my_function(*, x):
  print(x)

my_function(x = 3) # 3
```

Without the `*,` you are allowed to use positionale arguments even if the function expects keyword arguments:

```
def my_function(x):
  print(x)

my_function(3) # 3
```

```
def my_function(*, x):
  print(x)

my_function(3)
# TypeError: my_function() takes 0 positional arguments but 1 was given
```

Combine Positional-only and Keyword-only

Any argument *before* the `/ ,` are positional-only, and any argument *after* the `*,` are keyword-only.

```
def my_function(a, b, /, *, c, d):
  print(a + b + c + d)

my_function(5, 6, c = 7, d = 8)
```

Difference B/w `Print` & `return`

Feature	print	return
Purpose	Display output	Send a value back
Effect on program flow	No effect	Terminates function execution
Value returned	None	Value specified in return statement

If we use print in function instead of return what will happen

- **The function will still execute:** The code within the function will run as normal.
- **The function will print the desired output:** The `print` statement will display the value you want to "return" on the console.
- **The function will not send a value back to the caller:** This is the key difference. Unlike `return`, `print` does not provide a way to store or use the value outside the function.

```
def add(x, y):
    result = x + y
    print(result)  # Using print instead of return

sum = add(3, 4)
print(sum)

# OUPUT for print(result) --> 7
# Ouput for print(sum) --> None
```

Global & Local Variable

```
def f(y):
        x = 1 # this is Local Variable as it is inside a function
        x += 1
        print(x)

x = 5 # this is global variable as it is not inside any function.
f(x)
print(x)

# Output of Local x which is inside a function
2
# Output of Global x
5
```

As a function has return `None` to f(x) and we are not printing it, so Global value of x is printed `5`

💡

- If a function doesn't have local variable, it can use Global variable.
- If u want to change the value of global variable using function, that will throw error `UnboundLocalError`

```
def f(x):
        x = x+1

x = 5
f(x) # this will throw error
#=========================================================#

def f(x):
        x = x+1
        return(x)
x = 5
z = f(x)
print(z) # 6
print(x) # 5
# This will not throw an error.
```