1. Introduction

Functions are programmer-created code blocks that do a specific task. Functions can be used multiple times within a program, simply by calling the function's name. If you don't use functions, but you want to repeat an operation multiple times, you will have to copy and paste the necessary code multiple times to get your program to work. If you ever have to revise the program, you would have to ensure that all applicable copied and pasted code is updated. With functions, a single code block can be updated. Using functions has the following benefits:

- Increase code readability
- Increase code reusability

2. Creating a Function

In Python a function is created using **def** keyword followed by function name with arguments (if any). For example,

```
def display_hello():
   print("Hello World")
```

3. Calling a Function

A function is called by using function name followed by parenthesis with arguments (if any). For example,

```
display_hello()
```

4. Passing Arguments

Information can be passed into functions as arguments. When we pass arguments to functions, they are passed by reference not by value. Arguments are specified after the function name, inside the parentheses. We can add as many arguments as we want, just by separating them with a comma. For example,

```
def add(a, b):
   print(f"Sum = {a + b}")
add(5, 7)
```

If we do not know how many arguments that will be passed into your function, we add a * before the parameter name in the function definition. In this case, the function will receive a tuple of arguments, and can access the items accordingly. For example,

```
def add(*a):
   print(f"Sum = {a[0] + a[1]}")
add(5, 7)
```

we can also send arguments with the ***key* = *value*** syntax. In this case the order of arguments does not matter. For example,

```
def add(a, b):
   print(f"Sum = {a + b}")
add(a = 5, b = 7)
```

If we do not know how many keyword arguments that will be passed into our function, we add two asterisks (**) before the parameter name in the function definition. In this case, the function will receive a dictionary of arguments, and can access the items accordingly. For example,

```
def add(**arg):
    print(f"Sum = {arg['a'] + arg['b']}")
add(a = 5, b = 7)
```

A function can also be defined with default arguments. If we call such function without argument, it uses the default value. For example,

```
def add(a = 5, b = 7):
    print(f"Sum = {a + b}")
add(15, 8)
```

Remember: You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function. For example,

```
def display(v):
    for x in v:
        print(x)
val = [1, 6, 88, 34, 7]
display(val)
```

5. Packing and Unpacking Arguments

Packing refers to aggregating multiple arguments in a single collection so that the operations can be applied to the arguments combined. It is helpful when the same operation is to be called with a different number of parameters.
Unpacking refers to deconstructing a collection into multiple arguments so that the operations can be applied to the arguments individually. It is helpful when the arguments are required separately to perform operations.

5.1. Packing and Unpacking Arguments Using Tuples

While packing, we send a dynamic number of arguments as parameters to the function and access them through one tuple collection parameter by adding the * operator before it. For example,

```
def add(*args):
    print(f"Sum = { args[0] + args[1] + args[2]}")
add(5, 7, 9)
```

While unpacking, we create a tuple and send it as a parameter to the function by adding the * operator before it. The function receives it as individual parameters and deconstructs it to map each argument on a parameter. For example,

```
def add(a, b, c):
    print(f"Sum = {a + b + c}")
tuple1 = (5, 7, 9)
add(*tuple1)
```

5.2. Packing and Unpacking Arguments Using Dictionaries

While packing, we send a dynamic number of arguments as parameters to the function and access them through one tuple collection parameter by adding the ** operator before it. For example,

```
def add(**keyargs):
   print(f"Sum = {keyargs['a'] + keyargs['b'] + keyargs['c']}")
sum(a = 5, b = 7, c = 9)
```

While unpacking, we create a dictionary and send it as a parameter to the function by adding the ** operator before it. The function receives it as individual parameters and deconstructs it to map each argument on a parameter. For example,

```
def add(a, b, c):
   print(f"Sum = {a + b + c}")
dict1 = {'a' : 5, 'b' : 7, 'c' : 9}
sum(**dict1)
```

6. Return Values

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. If there is no expression in the return statement or the return statement itself is not present inside a function, then the function will return the *None* object. For example,

```
def sum(v):
   s = 0
   for x in v:
      s += x
   return s
val = [1, 6, 88, 34, 7]
print(sum(val))
```

6.1. Return Multiple Values

We return multiple values from a function by simply separating them by commas. When commas are used to return multiple values, they are returned in the form of a tuple. For example,

```
def display():
   return "John", 29
print(display())
```

We can also use list, dictionary, set etc. to return multiple values from the function. For example,

```
# Using list to return multiple values
def display():
   return ["John", 29]
print(display())
```

```
# Using dictionary to return multiple values
def display():
   return {"name" : "John","age" : 29}
```

```
print(display())
```

7. Recursive Function

A recursive function is a function that calls itself. To prevent a function from repeating itself indefinitely, it must contain at least one selection statement. This statement examines a condition called a base case to determine whether to stop or to continue with another recursive step. For example

```
def facto(n):
    if n == 0:
        return 1
    else:
        return n * facto(n-1)
n = int(input("Enter n:"))
print(f"{n}! = {facto(5)}")
```

8. Lambda Function

A lambda function is an anonymous function that is defined without a name. While normal functions are defined using the **def** keyword, anonymous functions are defined using **lambda** keyword. A lambda function in python has the following syntax.

```
lambda arguments : expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. For example,

```
sqrt = lambda x : x * 2
print(sqrt(5))
```

We can also define a lambda and call it immediately. This is something called an IIFE (**immediately invoked function execution).** It means that a lambda function is callable as soon as it is defined. For example,

```
print((lambda x : x * 2)(5))
```

The power of lambda is better shown when you use them as an anonymous function inside another function. Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number. For example,

```
def myfunc(n):
    return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))
```