Functions

In this lesson, we will learn about functions in Python.

WE'LL COVER THE FOLLOWING

A **function** is a reusable set of operations. We don't need to write the set of instructions again for different inputs, we can just call the function again.

There are two basic types of functions in Python:

1. Built-in functions
2. User-defined functions

> 💡 It is good practice to define all our functions first and then begin the main code. Defining them first ensures that they can be used anywhere in the program safely.

Declaration

In Python, a function is defined using the `def` keyword in the following format:

```
def functionName(parameters):
    operations
```

The `functionName` is simply the name we'll use to identify the function. The `parameters` of a function are the inputs for that function.

Parameters

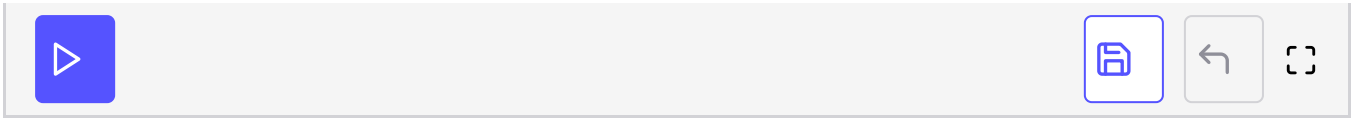
Parameters are the means of passing data into the function. When creating a function, we must define the number of parameters and their names. These names are only relevant to the function and won't affect variable names elsewhere in the code. Parameters are enclosed in parentheses and separated by commas.

The actual values/variables passed from the driver code when invoking the function are known as **arguments**.

```
def minimum (first, second):     # first and second are parameters
  if (first < second):
    print (first)
  else:
    print (second)

num1 = 10
num2 = 20

minimum (num1, num2)    # num1 and num2 are arguments
```



The positions of the parameters are important. In the case above, the value of `num1` will be assigned to `first` because it was the first parameter. Similarly, the value of `num2` assigned to `second`.

> If we call a function with more or less arguments than originally required, Python will throw an error.

The function `minimum()` ends on line 5 and merely defining a function does not cause any statement to run. It is only when we place function calls at the global indentation level (line 10) that a function actually gets called.

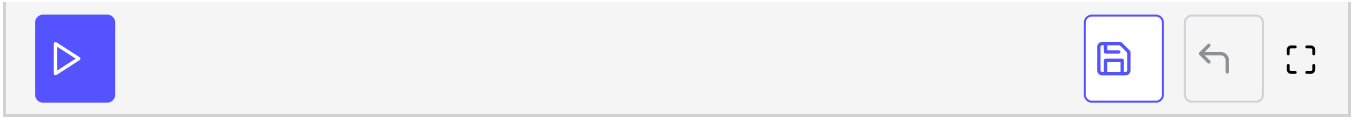
Using default values of parameters

If we specify default values to parameters in the function declaration, the function will use them when called unless we specify the values of these arguments.

We can say that these arguments are *optional*.

```
def add(num1, num2, num3, num4=4, num5=5):   # num4 and num5 are optional
  return num1 + num2 + num3 + num4 + num5

print(add(1, 2, 3))   # specifying values of num1, num2 and num3
print(add(1, 2, 3, -4))   # specifying values of num1, num2, num3 and num4
print(add(1, 2, 3, -4, -5))   # specifying values of num1, num2, num3, num4 and num5
```



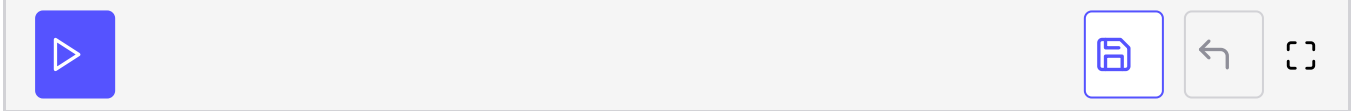
The `return` statement

To return something from a function, we must use the `return` keyword. Once the `return` statement is executed, the compiler returns control to the statement after the calling function. Any remaining lines of code after the `return` statement will not be executed.

```
def minimum (first, second):
  if (first < second):
    return first
  else:
    return second

num1 = 10
num2 = 20

result = minimum (num1, num2) # Storing the value returned by the function
print (result)
```



Scope

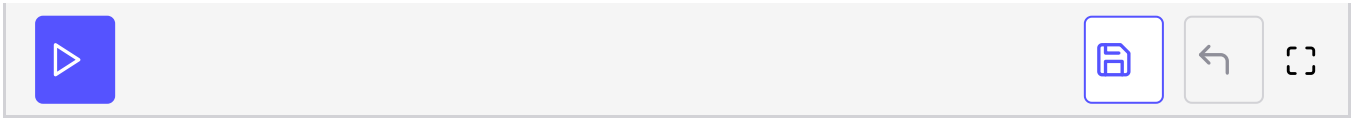
The scope of a function is the extent to which the variables and other data items made inside the function are accessible in the code. In Python, the function scope is the function's body.

Whenever a function executes, the program moves into the function scope. It moves back to the driver code when the scope of the called function ends.

In Python, data created inside the function cannot be used from the outside unless it is being returned from the function. Variables in a function have a local scope and they are released from memory once the function ends. This is why the code below fails to execute:

```
# this code will not work
def func():
    name = "Educative"

func()
print (name) # Accessing 'name' outside the function
```



Changing parameter values

When **mutable** data is passed to a function, the function can modify or alter it and these modifications will stay in effect outside the function scope as well. An example of mutable data is a list.

In the case of **immutable** data, the function can modify it, but the data will remain unchanged outside the function's scope. Examples of immutable data are numbers, strings, etc.

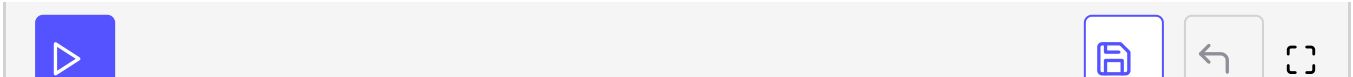
Let's see an example of both of these below:

mutable immutable

```
num_list = [10, 20, 30, 40]
print (num_list)

def multiply_by_10(myList):
    for i in range(len(myList)):
        num_list[i] *= 10

multiply_by_10(num_list)
print (num_list) # The contents of the list have been changed
```



For immutable data, a separate copy of the argument is created in the called function, which is why changes made to such data in the called function do not reflect back in the calling code.

We'll learn about lambdas in the next lesson.