Lab 9: Function

In the course so far, we have been writing basic statements like selection and iteration, and using Python build-in functions like print(), etc. In this week, we will move on to explore statements that create functions of our own.

There are a lot of benefits that come from using functions.

- Using functions leads to more maintainable programs, because a long program is divided into parts in more maintainable sizes, which simplifies the program creation process at the same time.
- As the program is simplified, the process of debugging also becomes easier.
- By using functions, some variables are moved to be defined locally inside functions, which creates cleaner variable namespace globally.
- A function is a reusable unit. Thus, functions can be used to reduce redundant code and enable code reuse. Using functions allows to group and parameterise chunks of code and to be used arbitrarily many times later on.
- Functions can also be used to modularize code and improve a program's quality.

The general form of a function

```
def <function_name> (<parameters>):
    <statements in function body>
    …
```

Notes on function definition:

- The first line " def <function_name> (<parameter>): " is called ***function header***
- <function_name> must follow the Identifier Naming Rules:
  - Only digits and alphabetic characters (a-z and A-Z) are allowed
  - First character cannot be digit
  - Cannot use Python keywords
- <function_name> needs to be **meaningful**, and reflect what the function actually does.
- <parameters> are optional argument values, separated by comma
- Must have a colon at the end of function header
- <statements in function body> must be indented consistently

The following example defines a function definition that displays the message "Hello, John". The function is then called to have the greeting message displayed to screen:

```
def display_greeting(): # defining a function definition
    print('Hello, John')

display_greeting() # calling the function to display "Hello John" to screen
```

Function definition and invocation

There are two sides when it comes to functions: the definition and the invocation of the definition. In the above example, without the line of calling the function (i.e., *display_greeting()*), there wouldn't be any output.

Functions can be invoked many times. The following script would produce the message "Hello, John" twice.

```
def display_greeting(): # defining a function
    print('Hello, John')

display_greeting() # calling the function
display_greeting() # calling the function again
```

Function calling function

When we have many functions defined, we normally also define a main() function as the entry point of the whole program. In this case, we would invoke ("call") other functions inside the main() function. The previous example can be re-written as follows:

```
def display_greeting(): # defining a function
    print('Hello, John')

def main():
    display_greeting() # calling the display_greeting() function inside main()
    display_greeting() # calling the function again

main() #calling the main() function
```

Functions taking parameters

Functions are meant to be reused. The ability of functions being able to take parameters really makes functions a powerful tool. In the example of the previous program, the display_greeting() function would be more useful if we could change the person's name the output message is greeting to every time we call the function. Function parameters makes that possible. As follows:

```
# defining a function with a string value as parameter
def display_greeting(person_name):
    print('Hello,', person_name)

def main():
    # call the function and pass the name "John" as argument value to it
    display_greeting('John') #output: Hello, John

    # call the function again, with the value "Alan" instead
    display_greeting('Alan') #output: Hello, Alan

main() #calling the main() function
```

In this example, a parameter variable with name "person_name" is created on the function header. The parameter is then used by the print() statement as part of the greeting message. In the main(), when the display_greeting() function is invoked the first time, the string value “John” is passed to the function call, the parameter variable “person_name” takes the value “John” and used to compose the greeting message. When the function is invoked the second time, a different value is passed and given to the parameter “person_name” resulting in the message “Hello, Alan” instead.

Function signature

When we are calling a function, we use the function’s signature to correctly identify which function we meant to call. A function’s signature consists of the function name and the function’s parameters (if any).

The following example defines a function with the name “test”, and two parameters, the first parameter is a number and the second is a string:

```
# defining a function with two parameters: a number and a string
def test(num, word):
    square = num * num
    print(square)
    print(word)

def main():
    # Correctly calling the function by its name, with required argument values,
    # i.e., test(<number>, <string>)
    test(5, ‘hello’)

    # Incorrectly calling the function: requires two values, only one is given
    test(5)

    # Incorrectly calling the function: requires two values, only one is given
    test(‘hello’)

    # Incorrectly calling the function: first argument value needs a number
    test(‘hello’, ‘world’)

    # Incorrectly calling the function: second argument value needs a string
    test(10, 5)


main()
```

When the function “test” is called in the main(), two argument values need to be given – the first has to be a number, and the second needs to be a string – as illustrated in the code above. The code also shows some incorrect invocations of the function, where the argument values were passed to the function incorrectly.

Non-value-returning vs. Value-returning functions

In the preceding section, we defined the function display_greeting() and test(). These functions are so-called “non-value-returning” functions, because they do not return a value when it’s being invoked. We could also define functions that return a value. Such a function is commonly known as “value-returning” function. A value-returning function must have a “return” statement that returns a value.

To see the differences between a value-returning function and a non-value-returning function, let’s re-design the non-value-returning function “display_greeting()” to be value-returning, with the name “get_greeting()”. The “display_greeting()” function is left in the program for the sake of comparison.

```
def display_greeting(person_name):
    print(‘Hello,’, person_name)

def get_greeting(person_name):
       message = ‘Hello, ’ + person_name
       return message

def main():
       name = input(“What’s your name? “)
       msg = get_greeting(name) #calling a value-returning function
       print(msg)

       display_greeting(name) #calling a non-value-returning function

main()
```

In the code above, the function “get_greeting” is defined as a value-returning function (which returns a string value). The “return” statement at the end of the function returns the string value back to its “caller”, which is a line of code in the main() function. Because the “get_greeting” function returns a string, the call to the function (i.e., the code “get_greeting(name)”) is used as a string value, which is assigned to the variable “msg”, i.e., the “msg” variable stores the returned value from the “get_greeting” function. The “msg” variable is then printed to the screen.

In the contrast, because the “display_greeting” function is a non-value-returning function, it is therefore invoked as a stand-alone statement in the main().

Function variable scope

A variable has a scope. The ***score of a variable*** is the part of the program where the variable can be referenced (or accessed). In the context of functions, a variable created inside a function is referred to as a ***local variable***. Local variables can only accessed within the function where it’s created. We use the same program code in the previous section to illustrate what a local variable is and the scope of local variables.

```
def get_greeting(person_name):

       # variable “message” is a local variable of “get_greeting” function
       message = ‘Hello, ’ + person_name

       # message = ‘Hello, ’ + name # Error: variable “name” is not available here

       return message
       # Note, the parameter “person_name” is also a local variable

def main():
       # variable “name” is a local variable of “main” function
       name = input(“What’s your name? “)

       # variable “msg” is a local variable of “main” function
       msg = get_greeting(name) #calling a value-returning function
       print(msg)

main()
```

There are two local variables defined in the main() function, “name” and “msg”. Because they are defined in main(), they are only available in main().

Function “get_greeting” defines a local variable “message”. The “message” variable is created using the parameter “person_name”. Function parameters are no different than local variables, except that, they hold the values passed to the function. The “message” variable is used by the “return” statement to return a string value back to the main(), the string value is then assigned to the “msg” variable in the main(). Note that, it’s the value held by variable “message” being passed back to main(), not the “message” variable itself.

Inside the “get_greeting” function, it’s also pointed out that the variable “name” is not available in the function – it’s a local variable of the main() function. In fact, the value held by the “name” variable is passed to the “get_greeting” function via the parameter “person_name”.

Do not override function parameters

A common pitfall is to override function parameters. Overriding parameters means assigning a value to a function parameter. Because parameters hold the data passed from the function caller, overriding a parameter means the data from the caller gets lost, which defeats the purpose of function parameters. The following example illustrates the discussion:

```
def display_greeting(person_name):
    print(‘Hello,’, person_name)

def display_greeting_1(p_name):
    p_name = ‘John Smith’
    print(‘Hello,’, p_name)

def main():
       name = input(“What’s your name? “)

       # Call the function and pass to it the data (held by the variable “name”)
       display_greeting(name)

       # Call the function and pass to it the data (held by the variable “name”)
       display_greeting_1(name)

main()
```

In the function “display_greeting”, the parameter “person_name” holds the data passed from main, and is used to compose a message for printing to screen.

In the function “display_greeting_1”, the parameter “p_name” is assigned the value “John Smith”, which causes the data passed from the main being overridden. Overriding parameters eliminates the reusability of a function which is one of the benefits of using functions in programming.