Chapter 5: Functions

Contents

1. Introduction to Subroutines
    1.1. Advantages of Subroutines
    1.2. Basic Syntax
    1.3. Examples of Functions
    1.4. Docstrings

2. Defining Subroutines: Functions and Procedures
    2.1. A procedure that does nothing
    2.2. A procedure without both input parameter and return statement
    2.3. A function with input parameters
    2.4. A function with both input parameter and return statement
    2.5. A function with default arguments
    2.6. A function with a variable number of arguments

3. Variable Scope: Global vs Local
    3.1. Local Variables
    3.2. Global Variables

4. Referenced Readings

Syllabus Learning Outcomes

2.2 Programming Elements and Constructs
    2.2.4 Use functions and procedures to modularise problems into chunks of code.

---

1. Introduction to Subroutines

Subroutines allow us to manage and structure our programs. This allows us to break up programs into smaller chunks of codes. This is called modular design. Such modular design allows you to debug easily, especially when you are dealing with a large program.

There are two types of subroutines.

- **Functions:** Subroutines that allow us to have both input and output. Functions accept inputs and process them into output. They must return at least one value, aka “return value”.

- **Procedures:** Subroutines that do not return values.

1.1. Advantages of Subroutines

The following are some of the advantages of using subroutines.

- **Organization:** Subroutines help break programs into smaller and modular chunks. They are almost like a mini-program that we can write separately from the main program, without having to think about the rest of the program while we write it. This allows us to reduce a complicated program into smaller, more manageable chunks, which reduces the overall complexity of our program.

- **Abstraction:** We can use a subroutine at any time, using its unique name and appropriate inputs.

- **Reusability:** It supports code reusability and reduces repetitive code. Subroutines can also be shared with other programs, reducing the need to code from scratch.

- **Modifying:** When we need to extend our program to handle a case it didn't handle before, subroutines allow us to make the change in one place and have that change take effect every time the subroutine is called.

- **Testing:** The program is easier to test and debug since each subroutine is self-contained. Furthermore, once a subroutine is working properly, there is no need to test the subroutine again unless it has been modified

1.2. Basic Syntax

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

- Keyword def marks the start of a function.
- Every function has a name as uniquely identification.
- It may take in optional values, known as parameters, before running the code block.
- It may return vale, as a result, using a return statement.
    - If no return statement, the function will return None at the end of the function.
- Optional documentation string (docstring) describes what the function does. It may be multiple lines of string.
- All statements must be equally identified, which is usually 4 spaces.

Note: Do not indent your program with a mix of spaces and tabs.

1.3. Examples of Functions

We have used a few functions when we learned about Python basic data types. Can you recall the following functions?

- How do you check data type of an object?
- How do you type cast a value from one data type to another?
- How do you print out the value(s) or output of your programs?

1.4. Docstrings

Recall what we get when we run `help(len)` or `?len`. Docstring is actually the documentation that gets returned when we run the help function.

```
[ ]  help(len)

     Help on built-in function len in module builtins:

     len(obj, /)
         Return the number of items in a container.


[ ]  def test():
         """
         Hi I'm the docstring.
         I am supposed to tell you more about this function.
         """

         pass

     help(test)
```

2. Defining Subroutines: Functions and Procedures

Let's get familiar with how to define the following procedures and functions.

- A procedure that does nothing
- A procedure without both input parameter and return statement
- A function with input parameters
- A function with both input parameter and return statement
- A function with default arguments
- A function with a variable number of arguments

2.1. A procedure that does nothing

Let's define the simplest function which does nothing.

- Keyword `pass` is used to indicate nothing to be done in the function body.
- Usually used when we want to define a function to do something, but we are not ready to code it yet.

```
[3]  def does_nothing():
         pass

[4]  type(does_nothing())

     NoneType
```

2.2. A procedure without both input parameter and return statement

Let's define a basic `hello_world` function without input parameter and return statement.

- It simply prints `"Hello World"`.
- This is an example of a procedure.

```
[5]  def hello_world():
         print("Hello World")

     print("Have a good day")
     hello_world()

     Have a good day
     Hello World
```

```
# more about procedure
# when you assign the procedure to a variable, the function is run once. However, what is the value assigned to x?
x = hello_world()

print(x)

Hello World
None
```

2.3. A function with input parameters

Let's use an input parameter to make the hello function more flexible.

- It takes in an input value `who`, and prints `"Hello {who}"`.

```
def hello_who(who):
    print("Hello,", who)

hello_who("Singapore")
```

```
Hello, Singapore
```

2.4. A function with both input parameter and return statement

Let's define a function `my_max` which returns the greater value of 2 input values.

```
def my_max(x, y):
    if x > y:
        return x
    else:
        return y

print(my_max(10, 20))


def my_max2(x, y):
    return x if x > y else y

print(my_max2(23, 18))
```

```
20
23
```

The return statement will exit the function. Consider the following function.

```
def my_max3(x, y):
    return x if x > y else y
    print("We should see this line if we did not exit the function")

my_max3(103, 320)
```

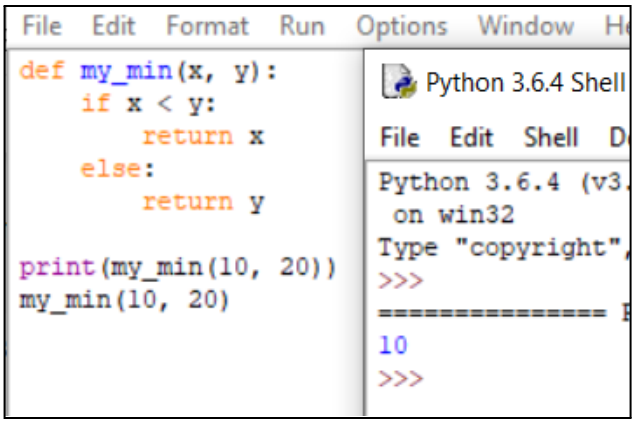
```
320
```

Let's define a function `my_min` which returns the smaller value of 2 input values.

```
def my_min(x, y):
    if x < y:
        return x
    else:
        return y

print(my_min(10, 20))
my_min(10, 20)
```

```
10
10
```



Note the difference between invoking the function in Python Notebook (left image) and Python IDLE (right image). For the latter, you need to call the print function to display the output of the function.

Let us create a function `add_them()` that will take two input parameters `a` and `b` and:

- return the sum of numbers if the input parameters are integers or floats,
- return a joint string if the input parameters are strings,
- return a joint list if the input parameters are lists,
- return an error message if otherwise

```
def add_them(a, b):
    num_types=[int, float]
    #check type of input, then float
    if (type(a) in num_types and type(b) in num_types) or (type(a)==str and type(b)==str):
        return a+b
    elif type(a)==list and type(b)==list:
        a.extend(b)
        return a
    else:
        return "Different/Invalid data types for parameters."

# both num
print(add_them(3,-1.5))

# both strings
print(add_them("3","=1.5"))

# both lists
print(add_them([3],["=1.5"]))

# invalid data types
print(add_them(3,["=1.5"]))
```

```
1.5
3=1.5
[3, '=1.5']
Different/Invalid data types for parameters.
```

Now, consider a function `mul_them(a, b)` that will take 2 input parameters `a` and integer `b` and:

- return the product if the input parameter a is an integer or float
- return the string `a` for `b` times if a is a string

```
def mul_them(a, b):
    return a * b

print(mul_them(3, 4))
print(mul_them('hello', 4))
```

```
12
hellohellohellohello
```

2.5. A function with default arguments

Python allows function arguments to have default values. If the function is called without the argument, the argument gets its default value.

A parameter is a variable defined in the function definition, while an argument is an actual value passed to the function.

```
[20] def mul_them2(a, b=1):
         return a * b

     print(mul_them2(3, 4))
     print(mul_them2('hello'))

     12
     hello
```

```
[21] #The default arguments must always be placed after the parameters without default values.

     def mul_them3(a=1, b):
         return a * b

       File "<ipython-input-21-88591bd12d03>", line 3
         def mul_them3(a=1, b):
                       ^
     SyntaxError: non-default argument follows default argument

     SEARCH STACK OVERFLOW
```

```
[22] # What happens if we have many default arguments and we only want to change the second one?
     def mul_them4(a=1, b=3):
         return a * b

     # how should we call mul_them 4 if we want b to be 5 instead?
```

2.6. A function with a variable number of arguments

We can pass a variable number of arguments to a function using:

- `*args` (Non-keyword arguments)
- **kwargs (Keyword arguments)

We use `*args` and `**kwargs` as an argument when we are unsure about the number of arguments to pass in the functions.

This is not in the A-level syllabus. You may refer to https://www.programiz.com/python-programming/args-and-kwargs for a simple overview.

3. Variable Scope: Global vs Local

The scope of a variable determines the portion of the program where you can access a particular variable. There are two basic variable scopes in Python

- **Global variables:** variables defined outside a function body
- **Local variables:** variables defined inside a function body

Global and local variables are in different scopes. Local variables can be accessed only inside the function in which they are declared, while global variables can be accessed throughout the program body.

It is good practice to use local variables wherever possible. There are several advantages listed below:

- Avoid changing the value that is being stored somewhere else in the program.
- You could use the same variable in different sections of the code, and each could be treated as a separate variable.
- You free up memory each time we are done with a local variable. It is removed from memory.

3.1. Local Variables

Consider the following function. The local variable `msg` is initialised inside a function and therefore, belongs only to that particular function. It cannot be accessed anywhere outside the function.

```
def Hello():
    # local variable
    msg = "Hello World"
    print(msg)

# Driver code
Hello()
```

```
Hello World
```

Analyse what happens when you try to access the variable `msg` outside the `Hello` function.

```
def Hello():
    # local variable
    msg = "Hello World"
    print(msg)

# Driver code
Hello()
print(msg)
```

```
Hello World
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-9c367fc89c7b> in <module>()
      6 # Driver code
      7 Hello()
----> 8 print(msg)

NameError: name 'msg' is not defined
```

3.2. Global Variables

Consider the following function. The global variable msg is defined outside the `Hello` function and therefore, is accessible throughout the program i.e. inside and outside of the function.

```
def Hello():
    print("Inside Function", msg)

# Global scope
msg = "Hello World"
Hello()
print("Outside Function", msg)
```

```
Inside Function Hello World
Outside Function Hello World
```

What if there is a variable with the same name initialised inside a function as well as globally? Consider the following code:

```
def Hello():
    msg = "Hello World"
    print(msg)

# Global scope
msg = "Hello World from me too"
Hello()
print(msg)
```

```
Hello World
Hello World from me too
```

So, we can conclude that if a variable with the same name is defined inside the scope of a function as well as globally, the function will use the value given inside the function only.

But, what if we want to change/edit/assign the value of the global variable?

In such cases, we should use the `global` keyword in the function. This `global` keyword is not needed if you just want to print/access the global variable. Consider the following code:

```
msg = "Hello World"

# Uses global because there is no local 'a'
def Hello1():
    print('Inside Hello1() : ', msg)

# Variable 'a' is redefined as a local
def Hello2():
    msg = "Hello Singapore"
    print('Inside Hello2() : ', msg)

# Uses global keyword to modify global 'a'
def Hello3():
    global msg
    msg = "Hello Universe"
    print('Inside Hello3() : ', msg)

# Global scope
print('global : ', msg)
Hello1()
print('global : ', msg)
Hello2()
print('global : ', msg)
Hello3()
print('global : ', msg)
```

```
global :  Hello World
Inside Hello1() :  Hello World
global :  Hello World
Inside Hello2() :  Hello Singapore
global :  Hello World
Inside Hello3() :  Hello Universe
global :  Hello Universe
```

As you can see in `Hello3()`, using the **`global`** keyword will allow you to edit the global variable.

4. Referenced Readings

- What are functions and procedures for? - Functions, procedures and modules - GCSE Computer Science Revision. (2022). BBC Bitesize. https://www.bbc.co.uk/bitesize/guides/z9hykqt/revision/1

- Tagliaferri, L. (2021, August 20). An Introduction to String Functions in Python 3. DigitalOcean Community. https://www.digitalocean.com/community/tutorials/an-introduction-to-string-functions-in-python-3

- Mester, T. (2021, November 25). Python Built-in Functions and Methods (Python & Data Science - Basics #3). Data36. https://data36.com/python-built-in-functions-methods-python-data-science-basics-3/

- Langfield, S., & Duddell, D. (2019). Cambridge International AS and A Level Computer Science Coursebook with Cambridge Elevate Edition (2 Years) (2nd ed.). Cambridge University Press.

- Rouse, G. (2015). OCR a Level Computer Science (UK ed.). Hodder Education.

- Reeves, B. (2015). Aqa a Level Computer Science (UK ed.). Hodder Education.