Workshop 6 More about Functions

18 November 2020

Goals: To better understand the utility of *functions* in Python, and how to use them. Specifically:

- To understand the difference between the **point of definition** of a function and the **point of invocation**;
- To have a better understanding of how functions work within a Python program;
- To have practice using multiple functions in a Python program by working with root-solving algorithms

You should ensure that you have completed all exercises before the next workshop.

Python code is a Sequence of Instructions

Python code is a sequence of instructions that is processed one line at a time. The Python language starts in the left-most column of the first line and processes the instruction from left to right. As Python executes the instructions on each line, the **point of execution** normally moves downward, one line at a time. An example of this is in the code below:

```
""" This program adds two numbers together """
x = 2.
y = 2.
z = x + y
print(f'The value is {z}')
```

If Python encounters **key words**, such `def` for functions, it expects indentation to mark the block of code that belongs to the function. This is where the function is defined, but the execution of the function will come later, where it is 'called' or 'invoked'.

Two key concepts

1. There is a **point of definition** where the function is defined.
2. There is a **point of invocation** where the function is invoked (or "called").

These two concepts are demonstrated in figure (1) on the next page.

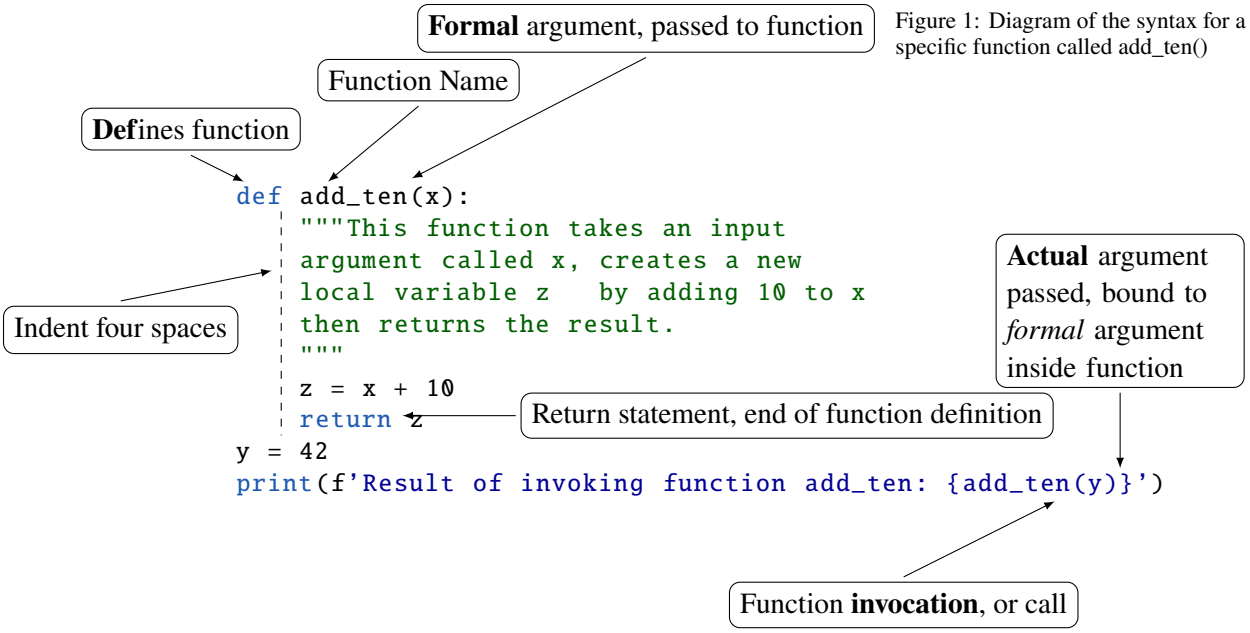

Figure 1: Diagram of the syntax for a specific function called add_ten()

*The **point of definition** of a function:*

The **point of definition** of a function is where the function first appears. More generally, the key world `def` indicates to Python that a function is being defined. When Python encounters this keyword, it expects the block of indented code just beneath the definition statement to be associated with that function. When the function is defined, the arguments that immediately follow the function name are called **FORMAL** arguments. In the example, the formal argument is x. The definition of the function terminates when Python either encounters the key word **RETURN** or there are no more statements in the indented block of code to execute.

*The **point of invocation** of a function:*

The point of invocation of a function is anywhere in the program where a function is used. In our example, the point of invocation is actually in the print statement! When Python encounters the function being called, the **ACTUAL** arguments are bound to the **FORMAL** arguments. In the example, the actual argument is y, and has been initialised to have the value 42.

Examples

There are two types of function in common usage in Python: a lambda function and a regular function, or just a function. Lambda functions are one line functions and so easy to use and understand. Regular functions are more powerful and, in fact, much more common.

Here is an example with a lambda function:

```
# First we define the function:
#
add_ten = lambda x: x + 10  # This is the point of definition!
#
# And now we invoke it:
y = 10
z = add_ten(y)   # This is the point of invocation!
#
# The value of z should be 20, right?
print('z = ', z)
# or, printing a different way:
print(f'z = {z}')
# It works!
```

Here is an example with a regular function (see also figure 1)

```
# First we define the function:
def add_five(x):
    """This function takes an input argument
    called x, creates a new local variable z
    by adding 5 to x then returns the result.
    """
    z = x + 5
    return z
pass
# And now we invoke it:
y = 10
z = add_five(y)   # This is the point of invocation!
#
# The value of z should be 15.
print(f'z = {z}' )
# It works!
```

The structure of a Python Code: how to handle complex codes

Python codes can use multiple functions, and the functions can be called in a loop:

```
#
# EXAMPLE CODE STRUCTURE 1
#
# All your import statements go at the top -- and they apply to the entire code!
#
import ...

# Function definitions
def myfun_1(x1, y1):
    """
    Describe what the first function does.
    """
    statements
    return x1*y1


def myfun_2(x2):
    """
    Describe what the second function does.
    """
    This statement is within the function

# Notice the following statements of the code are NOT in a function

# Initialisations
x1 = 1.
x2 = 32.4
a = numpy.arange(10.)

# Loop calling function
for i in a:
   print(myfun_1(i, 2), myfun_2(a[2]))
```

Working with and comparing Root-finding algorithms

We will now work with the root-finding algorithms we studied last week.

Exercises

These exercises should be completed before the next workshop. For each program that you write you should make sure you test each aspect e.g., each branch when using conditional statements (*conditional statements* are also called *if statements*) and ensure that you have used *suitable comments and docstrings!*

1. Consider the following block of Python code before answering the questions.

```
import numpy as np

def circumference(d):
    """This function computes the circumference of a input diameter."""
    circum = np.pi * d
    return circum
pass

d1 = 2.
c1 = circumference(d1)
print('The circumference is {c1}')

d2 = 42.
c2 = circumference(d2)
print(f'The circumference is {c2}')

d3 = 12.67
print(f'The circumference is {circumference(d3)}')

print(f'The circumference is {circumferance(67.9)}' )
```

   (a) Identify all the **function definitions** in this block of code. How many are there?[1]

   (b) Identify the formal arguments in the function definition.[2]

   (c) Identify the end of the function definition. Identify whether or not a value will be returned after all the instructions in the function are executed. [3]

   (d) Identify all the places where the function `circumference` is being **invoked**.[4]

[1] There is one function in this block of code.
[2] There variable d is the formal argument in this function definition.

[3] The function definition terminates at the return statement. The value assigned to the variable `circum` will be returned upon completion of the function.
[4] There are 4 places where the function `circumference` is being invoked.

(e) For each occurrence of the function being invoked, write down the **actual** arguments that will be bound to the **formal** arguments at the time of invocation.

Answer: In the first occurrence of the function being invoked, the formal argument *d* is bound to the actual argument *d*1 which is assigned to the value 2. In the second occurrence the formal argument *d* is bound to the variable d2, which is assigned the value 42. In the 3rd invocation of the function, the formal argument *d* is assigned to *d*3 which has the value 12.67. In the last invocation of the function, the formal argument *d* is bound to the value 67.9.

2. Consider the code, above, labelled `EXAMPLE CODE STRUCTURE 1`

   (a) Identify all the **function definitions** in this block of code. How many are there?[5]

   (b) Identify the formal arguments in each of the function definition.

   (c) Identify the end of the function definition for each function. Identify whether or not a value will be returned after all the instructions in the function are executed.

   (d) Identify all the places where the functions are being invoked.

   (e) For each occurrence of the function being invoked, write down the **actual** arguments that will be bound to the **formal** arguments at the time of invocation.

[5] Answer: There are TWO function definitions.

3. Consider the code, above, labelled `EXAMPLE CODE STRUCTURE 2`

   (a) Identify all the **function definitions** in this block of code. How many are there?[6]

   (b) Identify the formal arguments in each of the function definition, if any.

   (c) Identify the end of the function definition for each function. Identify whether or not a value will be returned after all the instructions in the function are executed.

   (d) Identify all the places where the functions are being invoked.

   (e) For each occurrence of the function being invoked, write down the **actual** arguments that will be bound to the **formal** arguments at the time of invocation.

[6] Answer: There are THREE function definitions.

4. Write a program that computes the circumference of a circle in the style of `EXAMPLE CODE STRUCTURE 1`. You may wish to consult Exercise 1 of this Workshop. In this case there should be a single function definition. Make sure the function returns the circumference. The code structure should look like the following:

```
#
# All your import statements go at the top
#
import numpy as np

# Functions
def circumference(d):
    """
    This function computes the circumference of a circle given its diameter.
    """
    c = d*np.pi
    return c

# Add code beneath this line that initialises a variable called d, then invokes the
# function circumference and prints out the value it returns.
```

5. In this exercise we will work with the program you wrote last week to use the bisection method. If you do not have this program with you, you can download a copy from ELE.

   (a) Use this program to create a new program, in either the style of `EXAMPLE CODE STRUCTURE 1` or `EXAMPLE CODE STRUCTURE 2`, that uses a Python function (with key word `def find_root(x, power, epsilon)`) for finding the square root of a number x. There are two things to do for this part of the exercise: (1) change your program to be a function and (2) change your program to find the powerth root of x, rather than the square root of x.

   (b) Create a program that will test your new function find_root.

   - **Note**: See example code on page 9 from Figure 4.5 of the text book "Introduction to Computation and Programming using Python" by John Guttag.
   - Use this program as guide to help you with parts (a) and (b) by adding comments to the existing code.
   - Modify the function so that the number of guesses (num_guesses) cannot exceed a certain value (e.g., 50).
   - It may help to draw brackets and arrows on a paper copy of the program so you can easily identify loops, conditionals, and calls to other functions.

6. Use the Save As option in Spyder to create a new program called `CompareRootFinders.py`. Use this as a basis to write a new program that compares the number of iterations it takes to approximate the cube root of x using the Bisection and Newton-Raphson

method, where x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Make sure to write a function that tests the accuracy of both root-finding functions. First, determine how many functions this code will require. You will need at least 2 functions, one that computes the cube root of x using the Bisection method and the other that computes it using the Newton-Raphson iteration. [7]

[7] **Hint:** You will also need to define the list/array of values to test, and you will need a loop. You may also need a function to test the code.

```
def find_root(x, power, epsilon):
    """
    Assumes x and epsilon are int or float, and power is an int
    epsilon > 0, power >= 1
    Returns float y such that y**power is within epsilon of x.
    If such a float does not exist, it returns None
    """
    if x < 0 and power % 2 == 0:
        return None
    num_guesses = 0

    low = min(-1.0, x)
    high = max(1.0, x)
    ans = (high + low)/2.0

    while abs(ans**power - x) >= epsilon:
        num_guesses += 1
        if ans**power < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans


# Now here is a program that uses find_root:

eps = 1e-4          # tolerance
num_list = [0.25, 2.0, 8.0, -0.25, -2.0, -8.0]     # numbers tp find roots of

for num in num_list:
    for power in range(1, 4):
        root = find_root(num, power, eps)
        if root is None:
            print('No root for {} ** 1/{}'.format(num, power))
        else:
            print('{:4.3f}**{} ~= {} ({:5.4f})'.format(root, power, num, root**power))
        pass
pass

print('Program is finished')
```