Python and Bioinformatics

Pierre Parutto

September 28, 2016

Contents

Chapter 1

Functions

Most of the computer programs possess the same structure: they receive one (or multiple) input value(s), perform some internal computations and then provide a result in the form of one (or multiple) output value(s).

In lab2, we wrote different pieces of code to perform computations, for example the scalar product between two vectors. In our programs, we distinguished three types of variables:

- Input variable(s);
- Internal variable(s);
- Output variable(s).

From the point of view of the Python interpreter, these variables are all the same, simply variables, the distinction comes from how we use them in the code. The internal variables are the variables that you need to perform the desired computation. Input and output variables are variables with specific names that the user uses to give input values to your code and retrieve the result(s) of the computation.

1.1 The Problem With Input And Output Variables

From the previous classes, we know two ways of providing input values to a program. We will illustrate these two ways through an example. Consider the question of computing the square root of a positive number `a`. A simple method, called the *Babylonian method*, computes the following:

$$u_n = \frac{1}{2}(u_{n-1} + \frac{S}{u_{n-1}})$$

$$\sqrt{a} = \lim_{n \to \infty} u_n$$

where S is a positive number, we will consider $S = a$ and $u_0 = a$. With this method, we need the user to provide two values: the number `a` from which to compute the square root, and the rank `n` at which we shall stop the computation of the sequence. The block of code implementing this method will thus possess two input variables, let's call them `a` and `n` and one output variable containing an approximation of $\sqrt{a}$, let's call it `res`. The implementation of the method is the following:

```
res = a
i = 1
while i <= n:
    res = 0.5 * (res + a/res)
    i = i + 1
```

If we try to run our code, saving it to a file named *my_sqrt.py*, and then pressing the green arrow in Spyder:

```
>>> runfile("/tmp/my_sqrt.py")
NameError: name 'a' is not defined
```

Which is expected because we never defined the name `a` (nor `n`). But `a` and `n` must not be defined by the programmer as they are input values to the program and must be set by the user. The two methods to circumvent this problem of definition of input variables are the following:

1. Define `a` and `n` at the beginning of the program:

   ```
   a = 2
   n = 10
   res = a
   i = 1
   while i <= n:
       res = 0.5 * (res + a/res)
       i = i + 1
   ```

   If we run this code, it works:

   ```
   >>> runfile("/tmp/my_sqrt.py")
   >>> res
   1.414213562373095
   ```

   But what if the user wants to compute $\sqrt{9}$ with 10 steps instead ? He has to go into your code and modify himself the values of `a` and `n`.

2. A second manner is to let the user set the input values beforehand in the interpreter. This is what we did in lab2, it can be done as follows:

```
>>> a = 2
>>> n = 10
>>> runfile("/tmp/my_sqrt.py")
>>> res
1.414213562373095
>>> a = 9
>>> n = 10
>>> runfile("/tmp/my_sqrt.py")
3.0
```

So now it is easier for the user to enter new input values, but it is still tedious to use the code as one has to switch between the interpreter and the code file.

In the end, we would like to be able to easily run a group of instructions depending on some input values and get an output value.

1.2 Functions

Functions allow to solve the previous problem nicely, a function possesses a name, some input values, a block of code and an output value.

Definition 1 *We call* **arguments** *the input values of a function.*

Definition 2 *We call* **return value** *the output value of a function.*

Definition 3 *A function is a name and some arguments associated to a block of code.*

1.2.1 Definition

Python defines the keyword `def` to define functions, it works as follows:

```
def nameFunction(nameArgument1, nameArgument2, ..., nameArgumentn):
    GROUP
    OF
    INSTRUCTIONS
```

where

- `nameFunction` is the name given to the function;
- `nameArgument1, nameArgument2, ..., nameArgumentn` are the names of the arguments;
- `GROUP OF INSTRUCTIONS` is the group of instructions that will be executed when the function is called.

1.2.2 Usage

The syntax for using functions correspond to the one you know from mathematics:

```
nameFunction(valueArgument1, valueArgument2, ..., valueArgumentn)
```

Warning

If there are multiple arguments, you have to respect the order of the arguments when you call the function.

1.2.3 Return Value

The return value is the output of a function, it is specified inside the body of the function using the `return` keyword:

```
return EXPRESSION
```

Where `EXPRESSION` is some expression (some computation that evaluates to a value).

Warning

There can be only one return value for a function.

Warning

The line containing the return keyword is **always the LAST line executed in the function**, even if you have some instructions after the return line, they will not be executed.

Warning

If a function does not end by a `return` keyword then its return value is `None`. This value corresponds to an absence of value.

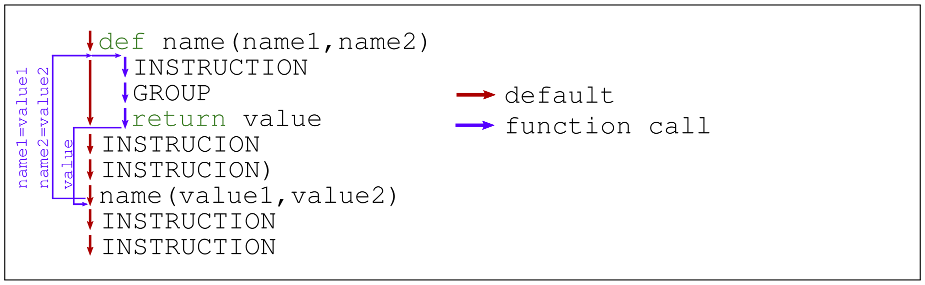
1.2.4 Vocabulary

Definition 4 *We call **definition** the line containing the `def` keyword plus the body of the function.*

Definition 5 *We call **function call** or **call** the action of using a function.*

1.3 Instruction Flow Of Functions

Functions modify the instruction flow as follows:



Basically:

- When Python encounters a `def` keyword it remembers the function name with its arguments. The instruction flow **does not go** through the instruction group of the function.
- When Python encounters a function call it performs the following actions in this order:
  1. it jumps to the first line of the instruction group corresponding to the function name.
  2. The argument variables are defined their corresponding values are in the corresponding order: the first argument value is associated to the first name, the second argument value to the second name, etc.
  3. Python goes through the body of the function and returns the value specified by the `return` keyword.
  4. The function call is then replaced by the return value.

Remark

The arguments are nothing more than variables that are defined **ONLY** in the body of a function. Their values are automatically set by python during the function call.

Remark

In the body of a function you can manipulate the arguments as any other variable: you can assign them values, delete them, etc.

1.4 Prototype Of A Function

When using input and output variables, the programmer has to give the names of these variables to the user so he knows what are the variables to modify and the variables to read. It is still true when using functions, you have to give the user the name of the function and the meaning and type of each argument and of the return value.

Definition 6 *We call* **prototype** *of a function a line providing to the user all the information to call a function. It has the following form:*

```
functionName(Name1: type, Name2: type, ..., Namen: type) -> type
```

Where to each argument (Name1, ..., Namen) is associated a type and `-> type` *is the type of the returned value.*

Warning

The prototype of a function, although very close to the syntax used to define functions in Python, cannot be directly copy/pasted in a code. In Python you do not specify the types of the arguments nor the type of the returned value. This prototype is only used to inform the user about how to use your function.

1.5 Input And Ouput Variables With Functions

Let's go back to the problem of computing $\sqrt{a}$, we are going to create a function called `my_sqrt` that has the following prototype:

```
my_sqrt(a: int, n: int) -> float
```

It reads as follows: the function `my_sqrt` possesses two arguments: `a` of type `int` and `n` also of type `int` and returns a value of type `float`.

The implementation is the following:

```python
def my_sqrt(a,n):
    res = a
    i = 1
    while i <= n:
        res = 0.5 * (res + a/res)
        i = i + 1
    return res
```

And now, saving this code into a file named *my_sqrt.py* and executing it by pressing the green triangle in Spyder we get:

```
>>> runfile("/tmp/my_sqrt.py")
>>> my_sqrt(2, 10)
1.414213562373095
>>> my_sqrt(9, 10)
3.0
```

We can now use the function as much as we want without running our code file as before (except if we change the code of the function). This is the best possible method to provide input values and retrieve output values from a block of code.

1.6 Examples Of Function Calls

In the following I provide more example of what you can do with function calls. Remember, a function call returns a value, so Python simply consider it as a value. You can thus use a function call wherever you can use a value. Consider the following function:

```
def my_add(a,b):
    return a + b
```

All the following calls are valid:

```
>>> my_add(3,2)
5
>>> my_add(3,2) + 5
10
>>> my_add(3,2) - my_add(4, 5)
-4
>>> my_add(my_add(4,5), 5*6)
39
>>> a = my_add(2,3)
>>> a
5
>>> my_add(a, my_add(1, 2)) ** 2
64
```