Functions

Scientific Computing
Fall, 2019
Paul Gribble

Functions are one of the most useful programming facilities that you will run into because they allow you to make your code more modular.

1 Encapsulation

We have seen functions already, for example functions to print stuff to the screen like `disp()` in MATLAB, functions for mathematics like `sqrt()`, and so on. The advantage of putting things like these into functions is that we avoid the need to write everything from scratch each time we want to repeat a common operation. Imagine if every time we wanted to take the square root of a number, we had to write an entire algorithm (like Newton's method) to do so. It would be ridiculous. By encapsulating this operation into a function, all we need to do is write `sqrt()` and the program goes and looks up the definition of that function, and executes that code, without us having to type it in again, and again.

Here we will be going through how to write our own functions to encapsulate our own operations, whatever they may be.

You can even imagine a series of functions that you write to perform the various steps of your data analysis, so that each time you collect more data in your experiment, you simply have to type:

```
>> go_my_analysis();
```

and all of your data analysis will be repeated, including the incorporation of any new data that may be residing in your data directory. Of course you have to define what happens inside of `go_my_analysis()` for this to be useful. Maybe inside your function you have defined other functions like:

```
>> load_all_data();
>> filter_data();
>> average_across_subjects();
>> perform_statistics();
>> generate_plots();
>> save_processed_data();
```

You get the idea.

The other situation where modularity in your code is useful, is for when you want to share code with other people (or use someone else's code). If you (or someone else) has a specific input/output functionality in mind, then you can swap in and out one of many potential functions that claim to achieve the desired functionality, as long as it preserves the input/output relationship(s) that you specify.

For example, let's say you discover that as part of your data analysis you will need to compute the square root of a number, and let's pretend that you don't have a function to do so built in to your language. (In fact you do of course—but for the purpose of this thought experiment let's imagine you don't.) The input/output requirements for your square root function are that it takes as input a single floating-point number and returns a single floating point number. Now you can go shopping, among your friends, colleagues, or on the internet, for an implementation of the square root function, and you can simply plug it in to your program and use it, as long as it takes a single floating-point number as input, and returns a single floating-point number as output. This specification is sometimes called the *function prototype*. You might find that several functions that you have found work equally well in terms of returning the correct answer, but that one in particular is way faster than the rest (perhaps it was written by a mathematician who knows some clever numerical tricks).

2 Function specification

In MATLAB usually sits in its own `.m` file, and starts with the word `function`, such as in this file called `mydeg2rad.m`:

```
function radsOut = mydeg2rad(degIn)

% radsOut = mydeg2rad(degIn)
%
% accepts an angle in degrees and returns
% the equivalent in radians
%

conversion = pi / 180;
radsOut = degIn * conversion;
```

The first line of the file begins with the word `function`, followed by (1) output variable(s), (2) the function name, and (3) input variables. In the above case there is a single output variable which we have named `radsOut`, and a single input variable named `degIn`. A function in MATLAB can have zero, one, or more output variables, and zero, one or more intput variables.

Following the function protype on line 1, is a series of commented lines—lines beginning with the `%` symbol. This tells MATLAB not to execute those lines as MATLAB code but that rather, these are human-readable comments. In addition now when you type `help mydeg2rad` on the MATLAB command line, MATLAB will display those comments as help for your function:

```
>> help mydeg2rad
  radsOut = mydeg2rad(degIn)

  accepts an angle in degrees and returns
  the equivalent in radians
```

The last two lines of the file is where the work happens in the function. In this case our function has just two lines of code that do the work—but in general your function can have many lines of code. The thing that your function has to have, is a definition of the value of the output variable(s)—the names of which are

defined on line 1, in the function prototoype—so in our case this is a variable called `radsOut`. This is how a function in MATLAB sends it output back to the code that called the function.

We can use our function like this:

```
>> mydeg2rad(180)

ans =

    3.1416
```

3 Variable scope

In MATLAB, we named the output variable(s) of our function and we also named the input variable(s) of our function. In the `mydeg2rad` function above, the output variable is named `radsOut` and the input variable is named `degIn`. The *scope* of these variable definitions is only within the function itself. Outside of the function, MATLAB does not know about these variables.

Similarly, the `conversion` variable defined on the second-to-last line of the `mydeg2rad` function also has a scope within the function itself, and outside of the function, MATLAB doesn't know about it.

You can think of a function like a black box, and to the outside user, the innards are unknown and inaccessible.

We can show this by trying to access the within-function variables:

```
>> mydeg2rad(180)

ans =

    3.1416

>> degIn
Undefined function or variable 'degIn'.

>> radsOut
Undefined function or variable 'radsOut'.

>> conversion
Undefined function or variable 'conversion'.
```

If we want to collect the output of the function we have to define a new variable to hold it:

```
>> myValue = mydeg2rad(180)

myValue =

    3.1416
```

Similarly, within a function, MATLAB does not know about variables outside of the function. So if we try to access, from within a function, variables defined outside of the function, we get an error message:

```
function out = newFun(in)

out = (myValue * 2) + 10;
```

```
>> newFun(10)
Undefined function or variable 'myValue'.

Error in newFun (line 3)
out = (myValue * 2) + in;
```

MATLAB functions *do* know about scripts and functions defined elsewhere however. It's only *variables* that have limited scope in MATLAB.

4 Multiple Inputs and Outputs

In the example above, we defined a function `mydeg2rad()` that takes a single input parameter and returns a single output value. In MATLAB you can define functions that take any number of parameters as inputs and return any number of values as outputs.

4.1 Multiple Inputs

To define a function that takes more than one input parameter, just list them in the round brackets after the function name, and separate them using commas. For example to define a function called `myAdd()` that takes two parameters, adds them together, and returns the result,

```
function result = myAdd(a,b)
result = a + b;
```

Remember you can name the input parameters anything you want, they do not have to correspond to any named variables in your workspace. They only "live" within the function you define. They have local scope, they are only defined within your function.

4.2 Multiple Outputs

To define a function that returns more than one value as an output, use square brackets in the function definition. For example to define a function `myAddAndSubtract()` that takes two input parameters and return two output parameters,

```
function [apb, amb] = myAddAndSubtract(a,b)
apb = a + b;
amb = a - b;
```

Call the function like this:

```
[r1,r2] = myAddAndSubtract(5,8);
```

5 Variable-length input argument lists

In MATLAB as in some other programming languages, it is possible to define a variadic function, that is, a function that accepts a variable number of function arguments as input.

If you're interested in implementing this in your own work, see the MATLAB documentation for varargin for information about how to do this, and for examples.

6 Anonymous functions

I have said that usually functions in MATLAB are located in their own `.m` file. However there is a way to define a function that is not stored in a file, but is associated with a variable defined in the MATLAB workspace. The hitch is that functions defined in this way can only contain a single executable statement. Here is an example of an *anonymous function* that returns the square of an input value:

```
>> sqr = @(x) x.^2;
>> a = sqr(5)

a =

    25
```

Line 1 says define an anonymous function, named `sqr()` that is a function of one input variable which we shall call `x` (within the function—remember variable scope), and the output of that function ought to be `x.^2`.

If you're interested in doing this, see the MATLAB documentation on anonyous functions for more details and examples:

Anonymous Functions