

CHAPTER 6

Functions

As we saw in the previous chapter, the use of script files to implement new computations allows us to automate repetitive tasks and avoid and fix typographical errors, but it involves a serious and substantial complication: Seemingly irrelevant details of the existing computations—such as variable names—profoundly can influence the outcome of the overall computation. This means that to write a successful program using scripts you must either be lucky enough to avoid all name conflicts or you must be fully knowledgeable about each of the existing computations you are using. This is an unreasonably burdensome responsibility for a programmer.

Taking the point of view that a computation is a transformation from inputs to an output, it seems reasonable to expect that the output of a computation should depend only on its inputs and not on internal details such as variable names. The knowledge we should require of the user of a computation should be quite limited: what are the inputs and what are the outputs. Knowing exactly how the transformation from input to output is accomplished (what algorithm is being used, how the algorithm is implemented, what variables are named, etc.) should not be a prerequisite to using the computation.

6.1 Computations without Effects

In order to avoid any possibility of a computation having an undesired effect, many computer languages, including MATLAB, adopt a simple but highly effective strategy: Computations can be packaged so that they have

absolutely no effect at all. For example, the logarithm operator works this way.

```
>> log10(1)
ans: 0
>> log10(20)
ans: 1.30102999566398
```

Although these two commands have produced outputs, no variables have been changed from the values they had before the computation.

Of course, we might decide to use the output of the logarithm computation in an assignment

```
>> a = log10(20);
```

Now something has changed, namely the value of the variable `a`. But the logarithm computation hasn't brought about the change; rather, the assignment statement has. `log10` is an innocent bystander to the assignment.

One consequence of the no-effect nature of computations is that an operator serves the same purpose as a lookup table. Younger readers may have limited experience with such tables because, for mathematical purposes, they have been almost completely replaced by calculators. Figure 6.1, for example, shows part of a table of logarithms. To use it to compute the logarithm of a number, you look up the number in the left-hand column labeled “Num.”: The logarithm is the number in the column labelled “Logarithm.” (This table can also be used in reverse, to compute the antilogarithm of a number.)

An obvious feature of such tables is that they are printed and therefore cannot change as a result of your using them. The table in Figure 6.1 is almost 400 years old but gives the same values that we compute today. Every time you look up a given number, you will find the same answer; the table describes a static, unchanging relationship between quantities. In the case of Figure 6.1, this is the relationship between numbers and their logarithms.

Essentially, all modern computer languages provide ways to package sequences of commands in a way that implements a computation as an ideal transformation of inputs to outputs, one that has no effect on the world other than providing the output. A computation packaged in this way is called a *function*. (The term is drawn from the mathematical term; in mathematics, “function” refers to a *single-valued relationship* between an input and an output. “Single-valued” means that for any given input there is one, and only one, output.)

Functions can implement a no-effect computation: a static, unchanging description of the relationship between quantities. Every time you provide a given input, you will get exactly the same output.

Key Term

On the computer, the logarithm table is implemented as the function named `log10`. Using the table is a matter of invoking the function:

```
>> y = log10(x);
```

`x` represents the quantity in the “Num.” column of the table, `y` is the corresponding quantity from the “Logarithmi.” column of the table. The computerized version of the table has the advantage of ease of use and the ability to hold a great many more numbers than can be printed even in a book-sized table. For instance, we can check the values in the table

```
>> log10(9)
ans: 0.95424250943932
```

but also compute logarithms not in the table

```
>> y = log10(9.5);
ans: 0.97772360528885
```

Such between-entry problems would have been solved in earlier generations by interpolation, an operation that is still useful.

6.2 Creating Functions

In MATLAB, a function can be created in two distinct ways. The first and most common way is the subject of the rest of this section. It involves creating an m-file and writing commands in the same way a script is written (see Section 5.4) but with one more step: adding a declaration line that identifies the inputs and outputs. The second way to create a function, to be covered in Section 6.5, involves using a special-purpose function-creation operator named `inline`.

When creating a function using an m-file, there is a distinctive first line that is used to indicate that the commands that follow are part of a function and to identify explicitly the inputs and outputs. Here’s an example that computes the length of the hypotenuse of a right triangle given the lengths of the two sides:

```
Function declaration line [1] function c = hypotenuse(a,b)
                        [2] c = sqrt( a.^2 + b.^2 );
```

Putting the preceding two lines¹ into an m-file named `hypotenuse.m` gives us a new operator called `hypotenuse` that takes two inputs. We can

¹Remember that the m-file contains only the text on the right side, displayed here in monospaced typewriter font. The line numbers and comments on the left side are added in this book only to help explain the commands.

invoke this operator (we will sometimes use the terminology “evaluate the function”) using the standard operator-calling syntax:

```
>> hypotenuse(3,4)
ans: 5
```

or, using the explicit function evaluation syntax:

```
>> feval('hypotenuse', 3, 4)
ans: 5
```

Key Term

When the statements in the function have been executed, we say that the function “returns.” The value of the variable identified as the output in the m-file, *c* in this case, is called the *return value*.

If we like, we can assign the return value to a variable:

```
>> a = hypotenuse(3,4);
>> a
ans: 5
```

Note that although we identified the output name as *c* within the function, we are free to call the output by any valid variable name *outside* the function’s m-file. The assignment on line 2 of *hypotenuse.m* to a variable named *c* has absolutely no effect outside of the function. To see this, let’s define a variable *c* and check what effect evaluation of the function has on it:

```
>> c = 0;
```

Evaluate the function

```
>> d = hypotenuse(3,4);
```

and check to see if the value of *c* has changed:

```
>> c
ans: 0
```

It hasn’t.

Line 1 of the m-file, beginning with the word *function*, indicates to the interpreter that there will be two arguments to the function. When the function is evaluated, the value of the first argument will be given the name *a* and the value of the second argument will be given the name *b*. The line also indicates the name of the variable whose value will become the output of the function. Although *c = hypotenuse(a,b)* looks like an expression involving assignment to a variable *c*, it is not; it is a special syntax that says which variables hold input values when the function is invoked and which variables hold output values when the function returns.

Hiding Variables

Functions rigorously sequester their internal variables from the rest of the computational world; the internal variables are completely hidden to any outside command.

Here’s the scene metaphorically: Imagine yourself in an office containing many pieces of paper with important information. Executing a script is like leaving your office and inviting someone else, a stranger, to come in to perform an operation. When the person is done, he or she leaves your office and you return. Unfortunately, in the course of the operation, the stranger might have done just about anything: change the information on a paper, delete some papers, or create new pieces of paper.

Evaluating a function is different. It is as if you write a memorandum on a fresh piece of paper saying what operation you want performed and giving the value of each input to the operation. You mail this memorandum to a distant office where it is processed. By return mail, you receive an envelope containing the output of the operation given the inputs you specified. You can do whatever you want with the output contained in the envelope, but you can be assured that performing the operation has changed nothing else in your office; there can be no surprises.

How does the interpreter prevent the statements inside a function from having an effect on variables outside the function? In Chapter 9, we’ll answer this question more completely, but for now the following picture will suffice. Whenever a function is invoked, the interpreter goes through the following steps:

Key Term

1. Start up a new *environment*. An environment is, essentially, a fresh session of MATLAB with no variables defined.
2. In the new environment, create the variables identified as inputs on the `function` line. Assign these variables the values of the arguments given in the function evaluation. For instance, in `hypotenuse(3,4)` the variable `a` is assigned the value 3 and the variable `b` is assigned the value 4.
3. Execute all of the function’s statements inside the new environment.
4. When the function’s statements have been executed, take the value of the variable identified as the output return that has the value of the evaluated function.
5. Eliminate the environment created for the function invocation.

Multiple Outputs

Sometimes a computation involves more than one output. We have seen this with the `max` function, which returns the largest value in a numeric vector as well as the index of that value.