Matlab functions: The syntax

Typically, each function should be placed in its own M-file, which must have the same name as the function. Matlab functions have the following basic syntax:

function *retvar* = *funcname*(*argvarlist*)
% `Some code can go here`
% ...
retvar = `<some expression>`;
% `some more code might go here`

The italicized terms have the following meaning:

retvar	The variable whose value is returned by the function. Note that *retvar* must be assigned a value somewhere within the body of the function.
funcname	The function name. The M-file containing the function must be named *funcname*.m.
argvarlist	The list of argument variables (also called parameters), separated by commas if there is more than one argument.

Below is an example of a simple function which takes a single (numeric) argument, and returns the value of that argument incremented by 1.

```
function m = inc(n)
m = n+1
```

And here is the function that takes two arguments, adds them, and returns the result.

```
function c = add(a, b)
c = a + b;
```

Function calls

Whenever we type in the name of a function in the command window, or use it anywhere in our code, we are making a *function call*. The code responsible for making the function call is referred to as the *caller*, and the function can be referred to as the *callee*.

The following sequence of events must take place whenever a function call is made:

1. The caller supplies the function arguments. The caller should supply as many arguments as are required by the function, and in the same order. All arguments are evaluated, and the values are passed along to the callee function.

2. Matlab switches into the scope of the function, getting ready to execute the code inside it. All variables *outside* the function become invisible to Matlab, so there can be no conflict between the caller's and callee's variables, even if they share the same name. Despite being invisible, all of the caller's variables persist in Matlab's memory.

3. The variables in the function's argument list are assigned the values supplied by the caller.

4. The code in the body of the function is executed. At some point, the return variable should be assigned a value.

5. Matlab switches out of the scope of the function, and passes the value of the return variable to the caller. All variables *inside* the function become invisible to Matlab and their values disappear from Matlab's memory. Matlab once again begins to see all of the caller's variables.

Note that functions return a value to the caller, rather than just printing it out. The caller is then free to do as it pleases with the returned value: it can store it in a variable, use it in an expression, pass it on to another function, or, of course, print it out. However, if we make a function call without putting a semi-colon at the end of the line, Matlab will output the value returned by the function - as it does with every statement that has a value whenever we omit the semi-colon.

Exercise 8.1: Save the functions `inc` and `add` into their respective M-files. Then, type the following lines of code into the command window:

```
>> inc(3)           % Output, if any?  _____________________
>> inc(4);          % Output, if any?  _____________________
>> k = inc(5);
>> disp(k);         % Output?  _____________________________
>> a = 4;
>> m = inc(a)       % Output?  _____________________________
>> b = 7;
>> add(a, b);       % Output?  _____________________________
>> c = add(a, b);   %
>> disp(c);         % Output?  _____________________________
>> add(a, c)        % Output?  _____________________________
>> disp(c);         % Output?  _____________________________
>> fprintf('%d\n', add(inc(6), add(7,3)));  % Output? _____
```

Exercise 8.2: Write a Matlab function `powersum` which takes two arguments, a vector `v` and an integer `p`, and returns the sum of the elements of `v` raised to the power `p`. That is, the function `powersum` should compute $v_1^p + v_2^p + ...v_n^p$, where n is the number of elements in `v`. When finished, try out your function by entering the following commands:

```
>> v = [2, 3, 5, 7];
>> p = 4;
>> powersum(v, p)
>> w = [6, 7, 8, 9, 10, 11];
>> c = 2;
>> s = powersum(v, p);
>> n = 10;
>> fprintf('The sum of cubes of the first %d integers is %d\n', n, powersum(1:n, 3));
```

M-files, functions, and scripts

By this point, you may have noticed that there are two different types of M-files. The M-files that contain functions are called *function M-files*. The M-files that do not define any functions, but only contain code to execute, are called *script M-files*, or simply *scripts*. Mixed M-files, that start off as scripts, and then try to define a function, are not allowed in Matlab. If you want to define a function, your M-file must start with the **function** header.

Scripts and functions are similar in some ways. Both are contained within M-files, and both contain executable code. Any Matlab statement that can appear inside a script can also appear inside a function, and vice versa (with the exception of the **function** header, and at least one other statement, which is only allowed for functions). Functions can even be called from within scripts, and scripts from within functions. However, there are a couple of major differences:

- Unlike functions, scripts cannot take arguments, and cannot return values. So, it would be doubly syntactically incorrect to write `x = scr(5)` if `scr` is the name of some script.

- All variables used inside a function are *local* to that function, and only exist in memory while the function executes. On the other hand, all variables used in a script exist within the scope of the script's caller, and their values persist even after the script exits. Thus, a script that is called from the command window will effect the variables in the command window, and a script that is called from a function will effect the variables in that function for as long as the function executes. So it is much easier to run into variable name conflicts with scripts than with functions.

Exercise 8.3: Below is code for two scripts and a function that will be calling one another.

```
%%%%%%%% Save as script1.m %%%%%%%%%%

x = 5;
fprintf('In script1, x=%d\n', x);
func(x);
fprintf('In script1, x=%d\n', x);
x = func(x);
fprintf('In script1, x=%d\n', x);

%%%%%%%% Save as func.m %%%%%%%%%%%%%

function x1 = func(x)
x = x+1;
fprintf('In func, x=%d\n', x);
script2;
fprintf('In func, x=%d\n', x);
x1 = x;

%%%%%%%% Save as script2.m %%%%%%%%%%

fprintf('In script2, x=%d\n', x);
x = x+1;
fprintf('In script2, x=%d\n', x);
```

If we run script1, what output will it produce? First, try to predict the output without running the script. Then, run the script, and see if the results agree with your prediction.

Functions that return multiple values

Sometimes, we may want our function to return more than one value at a time. For instance, if we wanted to find the maximum element of a vector, we would probably want to know both the value of the maximum element, and the index in the vector where it is located. Matlab's built-in `max` function can do just that, by returning multiple values. For a demonstration, type in the following:

```
>> v = [9, 3, 7, 13, 10, 16, 12, 6, 4];
>> [val, index] = max(v);
>> disp(val);
>> disp(index);
```

We can write our own functions to return multiple arguments as well. The syntax has the following familiar general form:

function [*retvar1*, *retvar2*, …] = *funcname*(*argvarlist*)
% `Some code can go here`
% …
retvar1 = <`some expression`>;
% `yet some more code might go here`
retvar2 = <`some other expression`>;
% …
% `some more code might go here`

Here, *retvar1*, *retvar2*, etc. are the variables whose values will be returned to the caller. We can have arbitrarily many return variables, just as we can have arbitrarily many argument variables for our function.

Here is an example of a simple function that takes two (integer) arguments a and b, and returns both the result of the whole number division of a by b, and the remainder of that division.

```
function [q, r] = divmod(a,b)
q = floor(a/b);
r = mod(a,b);
```

Note that if we have multiple return variables, they must be enclosed in square brackets in the function header. This is similar to the notation for creating vectors – not coincidentally so! Nonetheless, while there are similarities, it is important to remember that the function returns not a single vector, but multiple separate values. Each of the individual returned values can be anything, from a number, to a character string, or even a separate vector!

Unlike with functions that return only one value, if you would like to capture multiple return values, you must supply a variable to store each return value when you make your function call. For instance, if you would like to obtain both the quotient and the remainder from our `divmod` function, you would need to call it as follows:

```
>> [quot, rem] = divmod(38, 9);
>> fprintf('The quotient is %d, and the remainder is %d\n', quot, rem);
```

However, often we don't care about all the values returned by a function. For instance, we could only care about the first two arguments returned by a function, or only one. In that case, we only need to supply as many variables as we want return values when making our function call. All the other returned values will be simply discarded by Matlab. On the other hand, if we supply more variables in our call than are returned by the function, an error will be generated. For example, type the following:

```
>> q = divmod(38, 9);
>> disp(q);
>> fprintf('The quotient is %d\n', divmod(101, 13));
>> [q, r, junk] = divmod(76, 12);
```

Exercise 8.4:
Recall from high school algebra that the quadratic equation has the general form $ax^2 + bx + c = 0$. Recall also that the solution to this equation is given by the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Write a Matlab function `quadroot` that takes as arguments a, b, and c, returns both roots of the quadratic equation. Then, test your function on the following command-window input

```
>> [x1, x2] = quadroot(1, 0, -16)
>> [x1, x2] = quadroot(9, 12, 4)
>> x1 = quadroot(8, 15, -6)
>> [x1, x2, x3] = quadroot(3, 22, 16)
```

Solutions to selected exercises

Exercise 8.2

```
function s = powersum(v, p)
s = 0;
for k = 1:length(v)
   s = s + v(k)^p;
end
```

Exercise 8.4

```
function [r1, r2] = quadroot(a, b, c)
q = sqrt(b^2 - 4 * a * c);
r1 = (-b + q) / (2 * a);
r2 = (-b - q) / (2 * a);
```