Introduction

Matlab, like most programming languages, gives us many was to make our code more modular and reusable. While not the only way, the simplest way to do this in Matlab is through the use of user defined functions.

Let's assume you want to do something simple in Matlab like add 1.0 to a number. Also assume that you will want to do this more than once (some of the point of programming is to let the computer do tedious tasks, so often that means doing the same thing over-and-over).

```
clear variables; close all; clc;

a = 1;
b = 2;
c = a + 1;
d = b + 1;
```

We are doing the same operation to calculate *c* and *d*, so let's write it once using a function

```
clear variables; close all; clc;

a = 1;
b = 2;
c = plusOne(a)
d = plusOne(b)
```

Note, this code won't run yet, because we have not defined the function plusOne(), but you can hopefully see that this is powerful, in that, we could write something much more complex like doMyTaxes(), and put that in one line of our code. Therefore, we hide the complexity of this function, we can debug it once and use it again and again.

Function Format

The format for functions is

```
function [output1,output2,…] = functionName (input1,input2,…)
     function code;
end
```

Note that everything in italics is user defined, so we choose the name of output1 and functionName, etc. It's generally good form to choose something descriptive. For example a good name for a function that adds one to a number is plusOne(), some bad names would be a(), fun(), dog().

Let write that function

```
clear variables; close all; clc;

a = 1;
b = 2;
c = plusOne(a)
d = plusOne(b)

function ap1 = plusOne(a)
    ap1 = a  + 1;
end
```

The output should be

c =
   2
d =
   3

Internal vs External function

In the above example we wrote the function inside the main file. For reference I will call this an internal function because it is a function defined internal to your main code. Alternatively, we could have written the same function code in a separate file that is located in the same folder as your main code. I will call these external functions because they live in a file external to your main code.

So, your main code would be

mainCode.m

```
clear variables; close all; clc;

a = 1;
b = 2;
c = plusOne(a)
d = plusOne(b)
```

and you would have another file called

plusOne.m

```
function ap1 = plusOne(a)
    ap1 = a  + 1;
end
```

It is good form to name your function and your external function file the same name. In this example the function is called plusOne() and the file is called plusOne.m. If you are having

issues with this, the cause is often that your function file is not in the same folder as your main script.

Also note that you should not put clear variables; close all; clc, in your function file or your function.

The examples here will use internal functions because it is easy to write out, but both have their advantages and disadvantages.

Advantages of internal function

- You don't have to manage a bunch of files
- You can see all your code in one file
- You can have slightly different versions of the same functions in each main code

Advantages of external functions

- They isolate the code for your function so are generally easier to debug
- They make your main code much shorter, simpler to understand, and easier to debug
- Once you debug an external function you can use it over-and-over again in many different main programs

Local Variable Names

In general, all Matlab variables are local, you can define global variables but you should not (its bad form and will cause issues). This does not mean much when you only have your main code, but it is important when you use functions. Let's look at an example.

```
clear variables; close all; clc;

a = 1;
b = 2;
c = plusOne(a)
d = plusOne(b)
a

function ap1 = plusOne(a)
    ap1 = a  + 1;
end
```

In the main code *a* is assigned the value of 1 (lets denote this by a →1). Now when you run `c = plusOne(a)` and you go to your function; the input *a* also has a value of 1. But let's look at *b*. *b* is assigned a value of 2 (b →2), but then you run `d = plusOne(b)` and go into the function and the input is 2, so a→ 2 inside your function. However, *a* is a local variable in the function plusOne(), this means that the value of *a* that your function sees is independent of the value of *a* that your main program sees.

So, the output should be

```
c =
     2
d =
     3
a =
     1
```

So, the value of *a* in the main program does not change even though the value of *a* in the function is given different values each time you call the function. If we made *a* global then the function could see the value of *a* from the main program, and this would mess up a bunch of stuff.

More Complex Example

Consider, that you want to determine if a number is negative, positive, or zero (within some tolerance, so if the tolerance was 1e-12 the number 1.4e-13 would be considered zero).

```
clear variables; close all; clc

a = 1;
b = -1;
c = 0;

tol = 1e-9;
if abs(a) <= tol
    disp('a is zero');
elseif a < 0
    disp('a is negative');
else
    disp('a is positive');
end

tol = 1e-6;
if abs(b) <= tol
    disp('b is zero');
elseif b < 0
    disp('b is negative');
else
    disp('b is positive');
end

if abs(c) <= tol
    disp('c is zero');
elseif a < 0
    disp('c is negative');
else
    disp('c is positive');
end
```

Since you want to check if the number is positive, negative, or zero for 3 numbers, you need to write code for this check three times (if you don't use functions). You can see this takes time, and is prone to error (can you find the error, in the code above?). It would be better to write a function.

```
clear variables; close all; clc

a = 1;
b = -1;
c = 0;

isPosNegZero(a,1e-9);
isPosNegZero(b,1e-6);
isPosNegZero(c,1e-12);

function isPosNegZero(a,tol)
if abs(a) <= tol
    disp([num2str(a) ' is zero']);
elseif a < 0
    disp([num2str(a) ' is negative']);
else
    disp([num2str(a) ' is positive']);
end
end
```

This code is much shorter, is less prone to error, and forces us to check that we specified a tolerance each time (before the previous tolerance was used when checking the value of *c*, because we forgot to specify a new tolerance for *c*).

The output should be
1 is positive
-1 is negative
0 is zero

Multiple Inputs and Outputs

A function can have zero inputs and zero output (although this is uncommon)

```
clear variables; close all; clc;

helloWorld;

function [] = helloWorld()
    disp('Hello World!')
end
```

The output should be
`Hello World!`

A function can have inputs but no output (also somewhat uncommon)

```
clear variables; close all; clc;

helloSomething('Zorld');

function [] = helloSomething(name)
    disp(['Hello ' name '!'])
end
```

The output should be
Hello Zorld!

A function can also have no inputs but have outputs

```
clear variables; close all; clc;

c = helloWorldOutput();
disp(c);

function c = helloWorldOutput()
    c = 'Hello World!';
end
```

The output should be
Hello World!

Note that you can enclose the output in brackets `[c] = helloWorldOutput()`, but it is common to drop the brackets when there is only one output.

It is more common to have multiple outputs, inputs, or both. Two examples are below

```
clear variables; close all; clc;

a = [1 2 3];
b = [4 5 6];
m = 100;
[dotProd, tensorProd] = dotTensorProd(a,b)
magVector = vectorMag(a)

%% internal function that calculates
% dot and tensor products
function [dotProd, tensorProd] = dotTensorProd(a,b)
dotProd = a(1)*b(1) + a(2)*b(2) + a(3)*b(3);
tensorProd = a'*b;
end

%%  internal function that calculate
% magnitude of vector
function m = vectorMag(x)
m = sqrt(x(1)^2 + x(2)^2 + x(3)^2);
end
```

The output should be

dotProd =
   32

tensorProd =
    4   5   6
    8   10   12
   12   15   18

magVector =
   3.7417

User Exercises

1) write a function that calculates the area of a triangle based on two inputs (base and height). Check it with base values of 0.5, 1, 2 and heights of 1,2,3 respectively.
2) write a function called myTrapRule(x,y) that calculates the approximate integral of a function $y(x) = x^2$ from x=0 to x=1 where y is evaluated at 100 values of x. (see https://en.wikipedia.org/wiki/Trapezoidal_rule), and check your results with Matlab's trapz() function.

3) write a function that takes in a character string ‘twosheds’ and returns the first letter, last letter, and length of the string. Also, test it with the character string ‘shrubbery’ and check the results.

4) make a function with no inputs or outputs that prints a smile. The output should be something like this

```
*  *
 l
\_/
```