PSYCH 20A (PROF. FRANE) – LECTURE 10

Defining Functions, Applying Functions to Cell Arrays and Structures

A function is basically machine that gives you an output based on the input you give it. We've discussed several functions (`mean`, `sort`, etc.). We can also define our own functions. There are three basic types of functions we can define: *anonymous functions* (which are stored temporarily like variables, and can only consist of a single line of code), *function programs* (which are saved as stand-alone `.m` files that we can run any time and can be as complex as we want), and *local functions* (which are imbedded into a program for use only within that program and can be as complex as we want).

For any function you write, it's important to check it using many different kinds of inputs (e.g., positive values, negative values, integers, nonintegers) and verify that it gives you the right answer every time. A function should either work or give you an error message explaining why it can't work with the inputs you gave it.

Anonymous functions

Here's how to make an anonymous function called `doublePlusOne` that multiplies the input by 2, and adds 1:

```
doublePlusOne = @(x) 2*x + 1 ;
```

The `@(x)` here means "function of x." Note that using the letter "x" is arbitrary. We could instead use `@(a)` or `@(myInput)` or whatever, and it would work the same. We aren't actually defining `x` as anything here; it's just a placeholder that doesn't exist outside the function. We can use our `doublePlusOne` function just like any other function. For example, `doublePlusOne(5)` outputs 11. And `doublePlusOne([2 4])` outputs [5 9].

Here we define a function called `f2c` that takes a temperature in Fahrenheit and converts it to Centigrade:

```
f2c = @(tempF) (tempF - 32) * 5/9 ;
```

Note that instead of `tempF`, we could have used a generic letter, such as `x`, but using `tempF` makes the formula clearer. Again, `tempF` won't actually be defined in the workspace; it doesn't exist outside the function.

Functions can take multiple inputs. For instance, here we define a function that inputs weight (in kg) and height (in meters), and outputs the body mass index (which is simply the weight in kg divided by the square of the height in meters):

```
bmi = @(kg, m) kg ./ m.^2 ;
```

Now, typing `bmi(100, 2)` outputs 25, which is the body mass index for a 100 kg person who is 2 meters tall. Note that instead of using `kg` and `m` in the formula, we could have used generic letters like `x` and `y`, but using `kg` and `m` makes the formula clearer. Note also that by using the `./` and `.^` operators (instead of `/` and `^`), the function can be applied not only to single values of weight and height, but also to entire arrays of corresponding weights and heights.

Function programs

We write function programs in the edit window, just like any other program. To start writing a new function, click on "Function" in the "New" drop-down menu in the top-left corner. It will automatically give you a template that looks like this:

```
function [ output_args ] = untitled( input_args )
%UNTITLED Summary of this function goes here
%   detailed explanation goes here


end
```

Replace "untitled" with the name of the function; **this should be the same as the filename** (without the `.m`).

Replace `output_args` with placeholder name(s) for the output variable(s), separated by commas if there are more than one. The brackets `[]` are only necessary if there's more than one output variable. Note that these output names are just placeholders to define how the function works; they won't exist outside the function. In fact, none of the variable names you define within the function will exist in the workspace.

Replace `input_args` with placeholder names for the inputs.

The comments that you put **immediately after the first line** (with no skipped lines) should explain what the function does and how to use it. After you save the function, these comments are what will come up when you type `help` followed by the name of your function. **Don't put any code before the function statement.**

Notice that Matlab puts `end` at the bottom of the program. That's just a convention when writing functions and doesn't actually do anything other than mark the end of the program (unless you are "nesting" functions within the function, in which case using `end` is necessary).

The body of the program is where the output is actually computed.
For example, here's what a function program called `bmi` to compute body mass index would look like:

```
function bmiStat = bmi( kg, m )
%Syntax: bmiStat = bmi( kg, m )
%Inputs weight in kg and height in meters, and outputs the body mass index

bmiStat = kg ./ m.^2 ;

end
```

Notice that the comments tell us not only what the function does, but also exactly how to use it. We can see the exact syntax to use and that the inputs should be in kg and meters, respectively. After we save the function as `bmi.m` (just like we would save any program), we can type `bmi(100, 2)` into the command window to output 25, just like before. Or we could type `x = bmi(100, 2)` and thus make `x` equal to 25. Either way, the variables `kg`, `m`, and `bmiStat` would not exist outside the function; again, those are just placeholders within the function.

Unlike a script, typically a function program should not begin with standard housekeeping tasks like clearing the variables from the workspace or seeding the random number generator. You just want the function to perform its job without messing with anything else. For example, obviously you don't want the `mean` function to erase all your variables every time you use it; you just want it to take an input and give you an output. The same is true for most custom functions that you would write yourself.

Accepting different numbers of inputs
Whenever you run a function program, Matlab automatically defines a temporary variable called `nargin`, which means "number of arguments inputted." You can use that variable to make the function behave differently depending on how many arguments the user inputs.

Here's a function called `mixed2secs` that takes three inputs—hours, minutes, and seconds—and outputs the total duration in seconds; if only 2 inputs are provided, then the 3rd argument is considered as zero; if only 1 input is provided, then the 2nd and 3rd arguments are both considered as zero:

```
function timeInSecs = mixed2secs( inputHrs, inputMins, inputSecs )
%Syntax: timeInSecs = mixed2secs( inputHrs, inputMins, inputSecs )
%Inputs hours, minutes, and seconds, and outputs total seconds.
%Missing 2nd and/or 3rd argument(s) considered as zero.

if nargin == 3 % if 3 inputs, consider as hrs, mins, secs
   timeInSecs = 3600*inputHrs + 60*inputMins + inputSecs ;
elseif nargin == 2 % if 2 inputs, consider as hrs, mins
   timeInSecs = 3600*inputHrs + 60*inputMins ;
else % if 1 input, consider as hrs
   timeInSecs = 3600*inputHrs ;
end

end
```

Here's another example. We can modify the `bmi` function defined earlier so that it accepts an optional third input: `'metric'` or `'imperial'`. This third input tells the function whether to interpret the first two inputs as metric units (specifically, kilograms and meters) or Imperial units (specifically, pounds and inches), respectively; if no third output is given, then metric units are assumed by default:

```
function bmiStat = bmi( weight, ht, units )
%Syntax: bmiStat = bmi( weight, ht, units )
%Inputs weight and height, and outputs the body mass index.
%The units argument is either 'metric' (takes weight & height as kg & m) or
%   'imperial' (takes weight & height as pounds & inches). If no units
%   argument is entered, then metric units are assumed by default.

   if nargin == 2 % if only 2 arguments entered, assume metric inputs
      units = 'metric' ;
   end

   if strcmpi(units, 'metric')     % if metric units
      bmiStat = weight ./ ht.^2 ; % ...use this formula

   elseif strcmpi(units, 'imperial')              % elseif imperial units
   bmiStat = .45359237*weight ./ (.0254*ht).^2 ; % ...use this formula

   else % else throw error
      error('INVALID UNITS ARGUMENT. Must be ''metric'' or ''imperial''.')
   end
end
```

Notice that we use the `error` function above. The error message we specify will be shown in red and accompanied by a beep, and then the program will be aborted (just like with any other error).

Notice also that we are using very precise values in the conversion formulas. If you're going to write a program that computes something, you want it to be as precise as possible!

Accepting an indefinite number of inputs
If you put `varargin` as the only or last input in the function statement, then the user can enter however many inputs they want, and `varargin` will be defined (within the function) as a cell array containing all those inputs. For an example of using `varargin` you can type `edit strcat` to see the code for the `strcat` function.

Local functions

When we want to define a custom function to be used within some other program, we can use an *anonymous function* if the function is only a single line of code. But what if the function needs more than one line of code? In that case, we could define a separate *function program* that is called by our main program, but then our main program isn't self-contained because it relies on the separate function program to work. A solution is to use a *local function*, which is imbedded into our main program.

Unlike anonymous functions, a local function doesn't need to be defined earlier in the code than it is used. Instead, it needs to be defined *at the very end of the code*, after literally everything else in your script. If you need multiple local functions in a program, put them all in a row at the very end of the code.

The syntax for a local function is identical to the syntax for a function program: It starts with a `function` statement and ends with `end`. For example, we could paste our entire `bmi` function into the end of a script.

Checking what kind of object something is

In some cases, we want a function to behave differently depending on what kind of object the user inputs. For example, if the input is empty `[]` we might want to use a default. Or if the input is supposed to be numeric, as it is for our `bmi` function, then we might want to throw an error if the input is nonnumeric; we can do that using the `isnumeric` function, which outputs a logical value (`true` or `false`) indicating whether the input is numeric:

```
if ~isnumeric(ht)
   error('Invalid input. ht must be numeric.')
end
```

Here are some other functions that work similarly to `isnumeric`:

`ischar`	indicates whether the input is a character (or character array)
`isstring`	indicates whether the input is a string (or string array)
`iscell`	indicates whether the input is a cell (or cell array)
`isstruct`	indicates whether the input is a struct (or struct array)
`ismatrix`	indicates whether the input has no more than 2 dimensions
`isscalar`	indicates whether the input is a *scalar* (meaning a single value)
`isrow`	indicates whether the input is a nonempty row-vector (or a scalar)
`iscolumn`	indicates whether the input is a nonempty column-vector (or a scalar)
`isvector`	indicates whether the input is a nonempty vector of either orientation (or a scalar)
`istable`	indicates whether the input is a table
`isempty`	indicates whether the input is empty

Some other functions work somewhat similarly to those above, except that they operate on each individual value in the input so they will output an array of logical values if applied to an array. For example:

`isnan`	indicates whether the given value is `NaN`
`isinf`	indicates whether the given value is `Inf` (infinity)
`isprime`	indicates whether the given value is a prime number
`isreal`	indicates whether the given value is a real number

We can use the `isa` function to ask whether the object is of a specific class. For example:

```
if ~isa( ht, 'double')
   error('Invalid input. ht input must have class ''double''.')
end
```

We can use the `ndims` function to check the number of dimensions of an object (but note that a vector or scalar is considered as a matrix, so applying `ndims` in such cases will output a value of 2):

```
if ndims(ht) > 2
   error('Invalid input. ht must not have more than 2 dimensions.')
end
```

Applying a function to the contents of each cell in a cell array

Imagine that you have a 1 × 10 cell array called `arrayOfVectors` that has a vector in each cell. And imagine that you want to get the length of each of those vectors. You could to do that with a for-loop, like this:

```
% initialize vector in which each value will be the length of a vector
vectorLengths = NaN(1, 10) ;

% loop through the vectors in arrayOfVectors, getting the length of each
for iVectorNumber = 1:10
   vectorLengths{iVectorNumber} = numel(arrayOfMatrices{iVectorNumber}) ;
end
```

But a much more efficient method would be to use the `cellfun` ("cell function") function, like this:

```
vectorLengths = cellfun(@numel, arrayOfVectors) ;
```

Note that here the function being applied is `numel`, which is a built-in Matlab function. But we could also use custom functions that we define ourselves.

Note also that the output of `cellfun` (in this case, `vectorLengths`) is a cell array. But we could convert `vectorLengths` to a matrix by using the `cell2mat` function:

```
vectorLengths = cell2mat( cellfun(@numel, arrayOfVectors) ) ;
```

For functions that output more than one value
When applying functions that might output more than one value, we can still use `cellfun`, but we have to tell Matlab to allow different sized outputs by setting the `'UniformOutput'` parameter to zero. For instance, imagine that we have a cell array called `arrayOfMatrices` in which each cell contains a matrix. And imagine that we want to get the size of each matrix in that cell array. In that case, we could say:

```
matrixSizes = cellfun(@size, arrayOfMatrices, 'UniformOutput', 0) ;
```

Applying a function to each field in a structure

The `structfun` ("structure function") function is analogous to the `cellfun` function, but it's used to apply a function to each field in a structure, instead of to the contents of each cell in an array. For instance, if we have a structure called `structOfMatrices` in which each field contains a matrix, then we can get the size of each matrix by typing the following:

```
matrixSizes = structfun(@size, structOfMatrices, 'UniformOutput', 0)
```

The output of `structfun` (in this case, `matrixSizes`) will be a structure with the same field names that `structOfMatrices` has. Each field in `matrixSizes` will give the size of the matrix from the corresponding field of `structOfMatrices`.