Topic10
Variable Scope

Persistent Variables

- Normally when a function finishes its task, all of the variables used within the function go out of scope.
- The local variables have no legitimate reason for existing any more as the function no longer needs them.
- Typically the memory allocated to these variables is freed for reuse.
- The next time the function is called, a new local workspace is created with new local variables.
- There is no connection between any newly created local variables and the ones previously destroyed.
- However, there are occasions where you want the value of a variable within a function to persist and be "remembered" from function call to function call.
- We can declare a variable as one to be remembered between function calls with the **persistent** statement.

Example

```
%  RUNNINGAVERAGE:  A function that maintains a running
%                   average of values entered by the user.
%
%  Usage:      runningAverage('reset')
%              initialise the average.
%
%              av = runningAverage(x)
%              enter a new value into the running average.
%
%  Arguments:  x  - The new value to be entered.
%
%  Returns:    av - The running average of the values
%                   entered so far.
%
%  Author:     PK
%  Date:       August 2009
```

Example (cont.)

```
function av = runningAverage(x)

  % The number of values entered so far - declared persistent.
  persistent n;

  % The sum of values entered so far - declared persistent.
  persistent sumOfX;

  if x == 'reset'       % Initialise the persistent variables.
    n = 0;
    sumOfX = 0;
    av = 0;

      else                  % A data value has been added.
    n = n + 1;
    sumOfX = sumOfX + x;
    av = sumOfX / n;     % Update the running average.
  end
```

- Information about the running average remains between successive function calls of runningAverage…

Calling the function

```
>> runningAverage('reset')
ans = 0

>> runningAverage(5)
ans = 5

>> runningAverage(10)
ans = 7.5000

>> runningAverage(3)
ans = 6

>> runningAverage('reset')
ans = 0

>> runningAverage(8)
ans = 8
```

- When using persistent variables, you typically need to provide some mechanism for initialising and resetting them.

Global variables

- So far we have been concerned with *information hiding* to prevent unwanted interaction between functions.
- There are (very rare) occasions when Matlab's pass-by-value approach for communicating values to functions becomes cumbersome.
- There are two main reasons why this might be the case:
  1. The data object to be shared is so large that the time and memory cost in creating a copy may to too large. For example, there may only be enough memory to store one copy of the data object.
  2. There may be some data that needs to be used by a very large number of functions. Communicating this data via an argument or specifying the value in each one of these functions may be excessively cumbersome.

Examples

- For example, gravitational acceleration is 9.81 ms^{-2}. Rather than defining a variable:

```
G = 9.81;
```

- in every function that uses this value, you could declare it as a global variable.
- Variables are declared to be global with the global statement.
- For example:

```
global G;    % Declare G to be global.
G = 9.81;    % Then specify its value (in that order).
```

- Declaring a variable to be global means that it is stored in the global workspace and not in a local workspace.
- Any function wanting to use this global variable must also include the statement:

```
global G;      % This tells Matlab to use the global
               % variable called G and not create a
               % new local variable.
```

- This statement must be placed at the beginning of the function code before any attempt to use the variable *G* is made.

Example

```
%  FALLINGVELOCITY:  A function to calculate the
%                    velocity of an object falling
%                     under the influence of gravity.
%
%  Usage:      vel = fallingVelocity(t)
%
%  Arguments:  t   - The time in seconds.
%
%  Returns:    vel - The velocity of the object.
%
%  Author:     Lectures
%  Date:      August 2007

function vel = fallingVelocity(t)

    global G;
    vel = G*t;

end % function fallingVelocity
```

The very big danger

- If any function accidentally (through programming error) corrupts the value of *G*, the resulting mess spreads through the rest of your code.
- Always avoid global variables where possible.
- Reserve global variables for cases where the data object is so huge that passing it as an argument becomes difficult.
- If possible only share global variables between a function and subfunctions within the same file. Note however, even passing matrices of size 500x500 is no problem.
- Global variables should be fully capitalised to signify their status.

Another way of avoiding global variables

- One way of avoiding global variables for values that are kept constant is to define a function that sets the value for you.
- For example, in the case of the gravitational constant G, you would define a function called G in a file called G.m as follows:

```
% G is a function that does not take any arguments
function value = G
    value = 9.81;
 end %function G
```

- Now if you have a piece of code such as

```
vel = G * t;
```

- The G now refers to a function. MATLAB will
  1. invoke the function G.m,
  2. receive back the value 9.81, and then
  3. multiply that by t.
- Note you still have the potential problem of shadowing should you inadvertently create a variable also called G.

Constant variables and magic numbers

- Often your code will have a variable that really represents something that is constant. For example, `pi` or the speed of light.
- Alternatively, you might have a "magic" number in your code like the number of points to plot, or the number of sides for a polygon, etc.
- Unfortunately Matlab does not provide a mechanism for defining a variable which cannot change value. For example, you can easily reassign any value to pi within Matlab.
- It is good practice to have constants and magic numbers defined as variables at the beginning of your code.
- From then on, your code should just refer to these values via their variable names - the code should never use these numeric values directly.

Example

```
%  BADPLOTSINE:  A poorly written function to
%                plot the sine function
%                up to a specified x-value.
%
%  Note:       This function is an example of a
%              poorly written function and
%              should NOT be used.
%
%  Usage:      badplotsine(xmax)
%
%  Arguments:  xmax - The end value for the range of
%  angles to plot.
%
%  Returns:    Nil.
%
%  Author:     Lecturers
%  Date:       August 2010
```

```
function badplotsine(xmax)

  x = zeros(1, 100);   % Preallocate memory for arrays.
  sinx = zeros(1, 100);

  for ii = 1 : 100
    x(ii) = ii*xmax/100; % Build up the x and y arrays.
    sinx(ii) = sin(x(ii));
  end

  plot(x, sinx);
end % function badplotsine
```

- The problem with the code above is that the value 100 is "hard-wired" into the code in a number of places.

What if …

- If we change our mind about using 100 points, we have to pick our way through the code changing all the occurrences of 100.
- The danger is that you might miss one, or change a value of 100 that is in fact being used for another purpose.
- A better way to write the function starts by stating the "magic number" once at the top of the code, using that "constant" in the main code body:

```
%  BETTERPLOTSINE:  A function to plot the sine function
%                   up to a specified x-value.
%
%  Note:       This function is an example of a poorly written
%              function and should NOT be used.
%
%  Usage:      betterplotsine(xmax)
%
%  Arguments:  xmax - The end value for the range of angles to plot.
%
%  etc...
```

Better one

```
function betterplotsine(xmax)

  Npts = 100;      % The number of points to plot.
                   % Define this magic number up front.

  x = zeros(1, Npts);
  sinx = zeros(1, Npts);

  for ii = 1 : Npts
    x(ii) = ii*xmax/Npts;
    sinx(ii) = sin(x(ii));
  end

  plot(x, sinx);
```

- Now, if we change our mind about how many points to plot, there is only one location in the code that needs to be changed.

Better programming practice

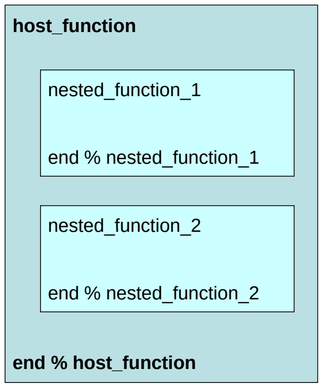
- The consistent use of the variable named *Npts* throughout the code makes it clear what the value represents. This makes the code easier to read.
- Avoid numeric constants in the main body of your code.
- Always define constants and "magic numbers" up front at the top of the code.

Nested Functions and Subfunctions

- It is possible to extend the idea of independent workspaces to allow independent sets of *function spaces.*
- One can have more than one function defined within a file.
- The first function defined in the file is a normal function - a *public* function that anyone can access.
- The public function must have the same name as the M-File.

- Other functions in the file defined *below* the public one are known as internal functions, or *subfunctions*.
- Other functions in the file defined *inside* the public one are known as *nested functions*.

Nested Functions (a diagram illustration)



- If a file contains one or more nested functions, then *every function* in the file must be terminated with an `end` statement.
- This is the only time when the `end` statement is required at the end of a function – at all other times it is optional.

Nested Functions

- Nested functions are functions that are defined *entirely within the body of another function*, called the **host function**. They are only visible to
  - the host function in which they are embedded, and
  - other nested functions embedded at the same level within the same host function
- A nested function has access to any variables defined with it, plus *any variables defined within the host function.*
  - The only exception is if a variable in the nested function has the same name as a variable within the host function. The variable within the host function is *not* accessible.

When to use subfunctions

- Subfunctions differ from public functions in that they are only accessible to other functions within the same file.
- Subfunctions are used when programmers want to:
  1. Hide *special purpose* functions within a file.
  2. Prevent conflicts with other public functions of the same name.
  3. Prevent certain functions from being used accidentally by external functions.
  4. Define a small function that would not have any general use outside of the main function being written. Creating a separate M-File for such a function may not be warranted.

An example

```
%  PLOTSINE:   A function to plot the sine function
%              with a specified frequency.
%
%  Usage:      plotsine(freq, amin, amax)
%
%  Arguments:  freq - The desired frequency.
%              amin - The start value for the range
%                     of angles to plot.
%              amax - The end value for the range of

%                     angles to plot.
%
%  Returns:    Nil.
%
%  Author:     PK
%  Date:       August 2005
%  Modified by:    WL
%  Date:      August 2010
```

Example (cont.)

```
function plotsine(freq, amin, amax)

    % The number of points to plot.
    Npts = 100;

    % Check that amin is less than amax and
    % reverse if necessary.

    [amin, amax] = order(amin, amax);

    % Generate an array of angles to evaluate
    % the sine function over.

    angles = amin : (amax-amin)/Npts : amax;

    plot(angles, sin(freq*angles));
end % function plotsine
```

Example (cont.)

```
%  ORDER:  A function to find the maximum and minimum
%          of two values.
%
%  Usage:     [maxValue, minValue] = order(x, y)
%
%  Arguments: x        - The first value.
%             y        - The second value.
%
%  Returns:   maxValue - The maximum value of x and y.
%             minvalue - The minimum value of x and y.
%
%  Author:    Luigi Barone
%  Date:      August 2005
%  Modified by:    WL
%  Date:      August 2010
```

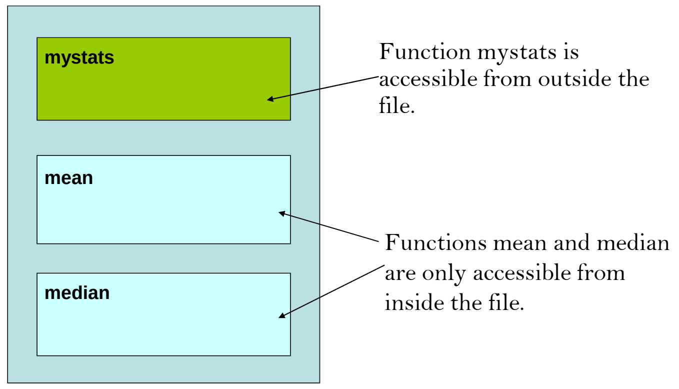
Example (cont.)

```
function [maxValue, minValue] = order(x, y)

    if x < y
        minValue = x;
        maxValue = y;
    else
        minValue = y;
        maxValue = x;
    end
end % function order
```

Sub-Functions (a diagram illustration)

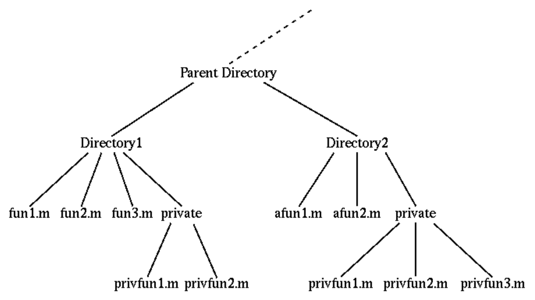
- File mystats.m


mystats
mean
median
Function mystats is accessible from outside the file.
Functions mean and median are only accessible from inside the file.

Private functions

- Private functions are those that reside within a subdirectory specifically named *private*.
- Private functions are only visible to functions in the parent directory and the private directory.
- Private functions present another mechanism by which one can create private function spaces.
- Different private subdirectories can contain functions with the same name, but because these functions can only be accessed by functions in the respective parent directories, no name conflicts can arise.

Directory Structure



Matlab's search procedure for functions

1. First, Matlab checks to see whether there is a nested fucntion with the specified name. If so, it is executed.
2. Matlab checks to see if an internal subfunction of that name. If so, it is executed.
3. Matlab checks for a private function matching that name. If so, it is executed.
4. Matlab checks for a function with the specified name in the current directory. If so, it is executed.
5. If the name is still not found, Matlab finally looks through the standard Matlab search path for a matching function name