4.4 Variable Scope in Matlab

We now know the general structure and use of a function. In this section we'll look closely at the *scope of a variable*, which is loosely defined as the range of functions that have access to the variable in order to set, acquire, or modify its value.

The Base Workspace

Matlab stores variables in parts of memory called workspaces. The *base workspace* is the part of memory that is used when you are entering commands at the Matlab prompt in the command window. You can clear all variables from your base workspace by entering the command **clear**[2] at the Matlab prompt in the command window.

```
>> clear
```

The command **whos** will list all variables in the current workspace. When entered at the prompt in the Matlab command window, the usual behavior is to list all variables in the *base workspace*.

```
>> whos
```

Because we've cleared all variables from the base workspace, there is no output from the whos command.

Create a new variable in the base workspace by entering the following command at the Matlab prompt in the command window.

```
>> M=magic(3);
```

Now when we enter the **whos** command, this new variable is listed as present in the base workspace.

[1] Copyrighted material. See: http://msenux.redwoods.edu/Math4Textbook/

[2] The **clear** command is much more extensive than described in this narrative. For example, **clear functions** will clear all compiled M-functions in memory and the command **clear global** will clear all global variables. For a complete description of the **clear** command and its capabilities, type **help clear** at the Matlab command prompt.

```
>> whos
  Name       Size                        Bytes  Class

  M          3x3                            72  double array
```

Scripts and the Base Workspace

When you execute a script file from the Matlab prompt in the command window, the script uses the base workspace. It can access any variables that are present in the base workspace, and any variables created by the script remain in the base workspace when the script finishes. As an example, open the Matlab editor and enter the following lines.

```
x=1:5
s=sum(x)
p=prod(x)
```

Save the script as **sumprod.m** and return to the Matlab command window. Execute the script by typing its name at the command prompt. The script finds the sum and product of the integers from 1 to 5.

```
>> sumprod
x =
     1     2     3     4     5
s =
    15
p =
   120
```

Examine the current state of the base workspace by typing the **whos** command at the command prompt.

```
>> whos
  Name      Size                    Bytes  Class

  M         3x3                        72  double array
  p         1x1                         8  double array
  s         1x1                         8  double array
  x         1x5                        40  double array
```

Note that the variables x, s, and p, created in the script, remain present in the base workspace when the script completes execution.

If a variable exists in the base workspace, then a script executed in the base workspace has access to that variable. As an example, enter the following lines in the Matlab editor and save the script as **usebase.m**.

```
fprintf('Changing the base variable p.\n')
p=1000;
fprintf('The base variable p is now %d.\n',p)
```

Execute the script.

```
>> usebase
Changing the base variable p.
The base variable p is now 1000.
```

Now, at the Matlab prompt, examine the contents of the variable p in the base workspace.

```
>> p
p =
        1000
```

Note that the script had both access to and the ability to change the base workspace variable p.

Function Workspaces

Functions do not use the base workspace. Each function has its own workspace where it stores its variables. This workspace is kept separate from the base workspace and all other workspaces to protect the integrity of the data used by the function.

As an example, let's start by clearing the base workspace.

```
>> clear
>> whos
```

Note that the base workspace is now empty.

Next, open the Matlab editor and enter the following lines. Save the function as **localvar.m**. Note that the function expects no input, nor does it return any output to the caller.

```
function localvar
p=12;
fprintf('In the function localvar, the value of p is: %d\n',p)
```

Execute the function by typing its name at the command prompt.

```
>> localvar
In the function localvar, the value of p is: 12
```

Now, examine the base workspace with the **whos** command.

```
>> whos
```

No variable p! There is no variable p in the base workspace because the variable p was created in the *function workspace*. Variables that are created in function workspace exist only until the function completes execution. Once the function completes execution, the variables in the function workspace are gone.

The function workspace is completely separate from the base workspace. As an example, set a value for p in the base workspace by entering the following command at the command prompt.

```
>> p=24
p =
    24
```

Examine the base workspace.

```
>> whos
  Name      Size                    Bytes  Class

  p         1x1                         8  double array
```

Now, execute **localvar** again.

```
>> localvar
In the function localvar, the value of p is: 12
```

At the command prompt, examine the contents of the variable p in the base workspace.

```
>> p
p =
    24
```

Still 24! This is clear evidence that the base workspace and the function workspace are separate creatures. While the function is executing, it uses the value of p in its *function workspace* and prints a message that p is 12. However, the value of p in the function workspace is completely separate from the value of p in the base workspace. Once the function completes execution, the value of p in the function workspace no longer exists. Furthermore, note that when the function sets **p=12** in its function workspace, this has absolutely no effect on the value of p in the base workspace.

Passing by Value. Most of today's modern computer languages allow the user at least two distinct ways to pass a variable as an argument to a function or subroutine: *by reference* or *by value.* Passing by reference means that the existing memory address for the variable is passed to the function or subroutine. Because it now has direct access to the memory location of the variable, the function or subroutine is free to make changes to the value of the variable in that particular memory location.

On the other hand, Matlab passes variables **by value** to function arguments. This means that the function that receives this value has no idea where the variable is actually located in memory and is unable to make changes to the variable in the caller's workspace. In MatlabSpeak, the variable that is passed to the function is *local* to the function, it exists in the function's workspace. If the function makes an attempt to set or modify this variable, it does so to the variable in its workspace, not in the caller's workspace.

For eample, make the following adjustments to the function **localvar** and save as **localvar.m**.

```
function localvar(p)
fprintf('Function localvar received this value of p: %d\n',p)
p=12;
fprintf('Function localvar changed the value of p to: %d\n',p)
```

Examine the base workspace variable p. If needed, reset this variable with the command **p=24**.

```
>> p
p =
    24
```

Execute the function **localvar** by passing it the variable p from the base workspace.

```
>> localvar(p)
Function localvar received this value of p: 24
Function localvar changed the value of p to: 12
```

Now, examine the variable p in the base workspace.

```
>> p
p =
    24
```

Still 24! Again, the base workspace and function workspace are separate animals! The **value** of the base workspace variable p is passed to the function workspace of **localvar** with the command **localvar(p)**. The value of p is changed in the function workspace, but because only the **value** of the workspace variable was passed to **localvar**, the value of the base workspace variable p remains unchanged.

Separate Workspaces. The idea of separate workspaces is Matlab's strategy to protect users from unintentionally changing the value of a variable in a workspace.

Forcing a Change of Variable in the Caller's Workspace. It's possible to have a function change the value of a variable in the caller's workspace, but you have to go to some lengths to do so. For example, adjust the function **localvar** so that it returns the value of p as output.

```
function x=localvar(p)
fprintf('Function localvar received this value of p: %d\n',p)
p=12;
fprintf('Function localvar changed the value of p to: %d\n',p)
x=p;
```

Note that the value of p is changed in the workspace of function **localvar** and returned to the caller via the output variable x. Execute the function at the command prompt as follows.

```
>> p=localvar(p);
Function localvar received this value of p: 24
Function localvar changed the value of p to: 12
```

Now, examine the contents of the variable p in the base workspace.

```
>> p
p =
    12
```

We've finally succeeded in changing the value of p in the base workspace, but note that we had to work hard to do so. This time we changed the value of p in the function **localvar**, then returned it in the output variable. Because we assigned the output of the function **localvar** to the base workspace variable p with the command **p=localvar(p)**, the value of p in the base workspace changed. Note that the command **localvar(p)** would have no effect on the workspace variable p. It's the fact that we assign the output that causes the change to the base workspace variable p.

Global Variables

One way to allow a function access to variables in a caller's workspace is to declare them as *global variables*. You must do this in the caller's workspace and in the function that wants access to these variables. As an example, open the editor, enter the following lines, and save the function M-file as **localvar.m**.

```
function localvar
global P
fprintf('In localvar, the value of P is: %d\n', P)
```

Note that this adaption of **localvar** receives no input nor does it output anything. However, it does declare P as a global variable. Consequently, if the caller does the same in its workspace, the function **localvar** should be able to access the global variable P.

In the command window, we'll assign a value to P. Global variables should first be declared before they are assigned a value, after which we assign P a value of 50.

```
>> global P
>> P=50
P =
    50
```

Execute the function **localvar** by entering the command **localvar** at the Matlab prompt in the command window.

```
>> localvar
In localvar, the value of P is: 50
```

This is a clear indication that the variable P is shared by the base workspace and the workspace of the function **localvar**.

Global Variables.

- Global variables names are usually longer and more descriptive than local variables. Although not required, most Matlab programmers will use all uppercase letters in naming global variables, such as **WIDTH** or **HEIGHT**.
- There is a certain amount of risk involved when using global variables and it is recommended that you use them sparingly. For example, declaring a variable global in a function might override the use of a global variable of the same name in another function. This error can be very hard to track down. Another difficulty arises when the programmer wishes to change the name of a global variable and has to track down and change its implementation in a number of functions.

In a later section, we will discuss the use of *Nested Functions* in Matlab, which allows the nested functions to see any variables declared in the function in which they find themselves nested. In many cases, the use of nested functions is far superior to employing numerous global variables.

Persistent Variables

We probably will have little use for *persistent* variables, but we'll mention them for the sake of completeness. Sometimes you might want a local variable defined in a function to retain its value between repeated calls to the function.

As an example, open the editor, enter the following line, then save the function M-file as **addTerm.m**.

```
function addTerm(term)
persistent SUM_T
if isempty(SUM_T)
    SUM_T=0;
end
SUM_T=SUM_T+term
```

Some comments are in order.

1. You can declare and use persistent variables in a function M-file only.
2. Only the function in which the persistent variable is declared is allowed access to the variable.
3. When the function exits, Matlab does not clear the persistent variable from memory, so it retains its value from one function call to the next.

When the persistent variable is first declared, it is set to the empty matrix. On the first call of the function, we need a different reaction than on ensuing function calls. If the variable **SUM_T** is empty, we initialize it to zero. On the next call of the function, the variable **SUM_T** is no longer empty, so it is updated by incrementing **SUM_T** by **term**.

To get a sense of how this function will perform, we intentionally left off a suppressing semicolon in the command **SUM_T=SUM_T+term** so that we can track progress of the persistent variable between repeated function calls. Enter the following command at the Matlab prompt in the command window.

```
>> for k=1:5,addTerm(1/k),end
SUM_T =
     1
SUM_T =
    1.5000
SUM_T =
    1.8333
SUM_T =
    2.0833
SUM_T =
    2.2833
```

Although the above should clearly demonstrate that **SUM_T** retains its value between successive function calls, another run of the loop should cement the idea of a persistent variable firmly in our minds.

```
>> for k=1:5,addTerm(1/k),end
SUM_T =
    3.2833
SUM_T =
    3.7833
SUM_T =
    4.1167
SUM_T =
    4.3667
SUM_T =
    4.5667
```

Readers should carefully check (with calculator in hand) that these values are correct.

To clear a persistent variable from memory, you can use either clear the function from memory **clear addTerm** or you can use the command **clear all**.

4.4 Exercises

1. Clear the workspace by entering the command **clear all** at the Matlab prompt in the command window. Type **whos** to examine the base workspace and insure that it is empty. Open the editor, enter the following lines, then save the file as **VariableScope.m**. Execute the cell in cell-enabled mode.

```
%% Exercise #1
clc
P=1000;
r=0.06;
t=10;
I=P*r*t;
fprintf('Simple interest gained is %.2f.\n',I)
```

Examine the base workspace by entering **whos** at the command window prompt. Explain the output.

2. Clear the workspace by entering the command **clear all** at the Matlab prompt in the command window. Type **whos** to examine the base workspace and insure that it is empty. At the command prompt, enter and execute the following command.

```
>> P=1000; r=0.06; t=10;
```

Examine the workspace:

```
>> whos
  Name      Size                    Bytes  Class

  P         1x1                         8  double array
  r         1x1                         8  double array
  t         1x1                         8  double array
```

Add the following cell to the file **VariableScope.m**.

```
%% Exercise #2
clc
I=P*r*t;
fprintf('Simple interest gained is %.2f.\n',I)
```

State the output of the script. Use **whos** to examine the base workspace variables. Anything new? Explain.

3. Clear the workspace by entering the command **clear all** at the Matlab prompt in the command window. Type **whos** to examine the base workspace and insure that it is empty. At the command prompt, enter and execute the following command.

```
>> P=1000; r=0.06; t=10;
```

Examine the workspace:

```
>> whos
  Name      Size                    Bytes  Class

  P         1x1                         8  double array
  r         1x1                         8  double array
  t         1x1                         8  double array
```

Open the editor, enter the following lines, and save the function M-file as **simpleInterest1.m**.

```
function I=simpleInterest1
I=P*r*t;
fprintf('Simple interest gained is %.2f.\n',I)
```

Add the following cell to **VariableScope.m**.

```
%% Exercise #3
clc
I=simpleInterest1;
fprintf('Simple interest gained is %.2f.\n',I)
```

Execute the cell and then state what happens and why.

4. Let's try an approach different to the attempt in **Exercise 3**. Open the editor, enter the following lines, and save the file as **simpleInterest2.m**.

```
function I=simpleInterest2
global P R T
I=P*R*T;
```

Add the following cell to **VariableScope.m**.

```
%% Exercise #4
clc
clear all
global P R T
P=1000; R=0.06; T=10;
I=simpleInterest2;
fprintf('Simple interest gained is %.2f.\n',I)
```

Execute the cell and state the resulting output. Explain the difference between the result in **Exercise 3** and the current exercise.

5. Enter the following lines in the editor and save the file as **clearFunction**.

```
function clearFunction
clear all
```

Add the following cell to **VariableScope.m**.

```
%% Exercise #5
clc
clear all
P=1000; R=0.06; T=10;
clearFunction;
whos
```

Execute the cell and explain the resulting output. Why do the variables P, R, and T still remain in the base workspace? Didn't we clear them with **clearFunction**?

6. Enter the following lines in the editor and save the file as **simpleInterest3.m**.

```
function I=simpleInterest3
P=5000;
R=0.10;
T=8;
I=P*R*T
```

Add the following cell to **VariableScope.m**.

```
%% Exercise #6
clc
clear all
P=1000; R=0.06; T=10;
dbstop simpleInterest3 5
simpleInterest3
whos
```

The command **dbstop simpleInterest3 5** puts a *breakpoint* at line 5 of the function **simpleInterest3**. When the script executes the function on the next line, the Matlab Debugger halts execution at line 5 of the function **simpleInterest3** and displays the debug prompt and the current line.

```
5   I=P*R*T
K>>
```

To see where you are, enter the following command at the debug prompt.

```
K>> dbstack
> In simpleInterest3 at 5
```

This tells us we are in **simpleInterest3** at line 5. Note that the variables P, R, and T are declared in lines previous to line 5. Examine their contents in the workspace of function **simpleInterest3**.

```
K>> P, R, T
P =
        5000
R =
    0.1000
T =
     8
```

Notice that we are in the function workspace, not the base workspace. Now end the debugging session and allow the function to complete its execution with the following command.

```
K>> dbcont
```

When execution of **simpleInterst3** completes, we are returned to the caller, in this case the base workspace. Examine the contents of P, R, and T in the base workspace.

```
>> P, R, T
P =
        1000
R =
    0.0600
T =
    10
```

This is clear evidence that the base workspace and the function workspace are separate entities.

4.4 Answers

1. Script output:

```
Simple interest gained is 600.00.
```

Worspace variables:

```
>> whos
  Name      Size                   Bytes  Class

  I         1x1                        8  double array
  P         1x1                        8  double array
  r         1x1                        8  double array
  t         1x1                        8  double array
```

3. We obtain an error:

```
??? Undefined function or variable 'P'.

Error in ==> simpleInterest1 at 2
I=P*r*t;
```

This happens because the variable P is in the base workspace and the function **simpleInterest1** executes in its own workspace where there exists no instance of the variable P. In short, functions cannot see the variables in the base workspace unless you pass them to the function as arguments or make them global in the base workspace and function.

5. No. The **clear all** in **clearFunction** clears the function workspace, not the base workspace where the cell enable script is operating.