SIMPLE INPUT and OUTPUT:

(A) Printing to the screen. The disp( ) command.

If you want to print out the values of a variable to the screen, you simply can type the variable at the command line.

```
> x = 5
x =  5
```

Same thing applies for character strings.

```
> z = ‘hello’
z = hello
```

You can write character and numerical data on the same line, although it won’t print out that way.

```
> 'The value of x is ', x
ans = The value of x is
x =  5
```

However, this doesn’t look very good. We would like to have more control with the output. Specifically, we would like to get rid of that annoying ‘x =’, ‘g =’, and ‘ans =’ after every statement is executed.

The **disp( )** command is useful if you want to print something to the screen without fancy formatting.

```
> score = 90.2;
> disp('Your score is')
Your score is
> disp(score)
 90.200
```

If you want to print different types of data (for example, character and floating-point) on the same line, you must convert the floating-point data to a string using the **num2str( )** command.

```
> score = 90.2;
> disp(['Your test score is ' num2str(score)])
Your test score is 90.2
```

Notice the use of brackets **[ ]** around everything you want printed to the screen.

(B) Getting input from user. The input( ) function.

Sometimes you want a program to read in data provided by the user and then use that data to perform some calculations. For example, the user inputs the number of hours and the program converts that number to minutes and seconds. To read in data from the screen, use the **input**( ) function.

variable or array = input(some instructions for the user)

In mfile.m:

```
hours = input('Enter the number of hours:  ')     % Matlab will wait for a value to be entered
minutes = hours * 60.;
seconds = minutes * 60.;
disp(['This is ' num2str(minutes) ' minutes'])
disp(['This is ' num2str(seconds) ' seconds'])
```

At command line:

```
> mfile
Enter the number of hours:  5
hours =  5
This is 300 minutes
This is 18000 seconds
```

Inputting character data is a little trickier. If you use the same syntax for inputting character data as you use for inputting numerical data, the user must put single quotes around the text they enter.

In mfile.m:

```
name = input('What is your name?  ');
disp(['You have entered: ' name])
```

At command line:

```
> mfile
What is your name?  'Paul'     (Notice that I had to use single quotes when typing ‘Paul’)
You have entered: Paul
```

If you don’t want the user to have to enter character data in single quotes, you can use a slightly different form of the **input**( ) command.

In mfile.m:

```
% The ‘s’ tells Matlab/Octave that a string of characters is expected
name = input('What is your name?  ' , 's');
disp(['You have entered: ' name])
```

At command line:

```
> mfile
What is your name?  Paul
You have entered: Paul
```

I will discuss how to enter in multiple pieces of information at once later.

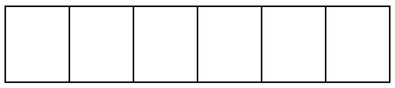
ARRAYS:

Matlab/Octave is designed for handling Matrices (arrays). You can think of arrays as a series of boxes that contain similar items. We use arrays when we want to store a lot of similar data. For example, if we want to find the mean and standard deviation of student test scores, we first need to store all the scores in memory. It would be quite lengthy and complicated to make a new variable for each test score.

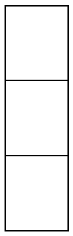
Although in theory arrays can have many different dimensions, we usually only deal with 1D (vector) or 2D (Matrix) arrays.

For 2D arrays, the size of the array is expresses as #ROWS x #COLUMNS. A 1D array is a 2D array with one row.

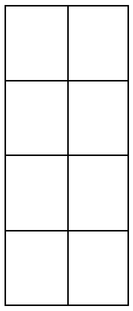
1x6 :



3x1:



4x2:



An array is made of elements. A 1x6 array is 1-dimensional and has 6 elements. A 3x1 array is 1-dimensional and has 3 elements. A 4x2 array is 2-dimensional and has 8 elements. Extending this concept into 3+ dimensions, a 4x2x3 array has 24 elements and a 2x10x3x2 array has 120 elements.

1D ARRAYS:

First we'll discuss 1D arrays since they are simpler to understand than 2D arrays. Every time you create a variable, you create a 1D array that is 1x1 in size.

```
> a = 52.4
a =  52.400
> whos
Variables in the current scope:

  Attr   Name       Size                     Bytes  Class
  ==== ====         ====                     =====  =====
             a          1x1                      8       double

Total is 1 element using 8 bytes
```

The variable **a** is a 1x1 array. Let's create a bigger array. You can define arrays by enclosing numbers in brackets. The value of each element is separated by a comma or a space.

```
> cat = [0.1 , 0.2, 0.4]
cat =
   0.10000   0.20000   0.40000
> dog = [1.1  1.2  2.4]
dog =
   1.10000   1.20000   2.40000
> whos
Variables in the current scope:
    Attr Name           Size                Bytes  Class
  ==== ====          ====                 =====  =====
       cat              1x3                  24     double
       dog              1x3                  24     double
```

You have created two 1x3 arrays that contain double precision data. Here are pictures of the cat and dog arrays. You refer to a particular element in the array using parentheses.

0.1	0.2	0.4
cat(1)	cat(2)	cat(3)

1.1	1.2	2.4
dog(1)	dog(2)	dog(3)

```
> cat(3)
ans =  0.40000
> dog(1)
ans =  1.10000
> dog(4)
error: A(I): Index exceeds matrix dimension.
```

Notice that we get an error message if we try to access a non-existent array element.

You can use elements of an array to do mathematical operations.

```
> dog(1) + cat(2)
ans =  1.3000
> rat = cat(1)**2
rat =  0.010000
```

You can change the values stored in an array.

```
> dog(1) = 5
dog =
   5.0000   1.2000   2.4000
> dog(1) = dog(3)
dog =
   2.4000   1.2000   2.4000
```

We can add elements to arrays very easily. Note: In many other languages (such as Fortran), this is not allowed… you would need to declare the size of your array ahead of time.

```
> dog(4) = 10
dog =
    2.4000    1.2000    2.4000   10.0000
> whos
Variables in the current scope:
   Attr  Name        Size        Bytes   Class
  ==== ====          ====        =====  =====
        cat          1x3            24  double
```

```
      dog              1x4              32  double
      rat              1x1               8  double
```

The array **dog** is now a 1x4 array.

Continuing from previous line:

If you want to define a long 1D array and do not have enough room on the command line, you can use an ellipsis (…)

```
> cat45 = [1 3 5 6 3 5 3.3 ...
> 55 66 77 888]
cat45 =
   1.0000  3.0000  5.0000  6.0000  3.0000  5.0000  3.3000  55.0000  66.0000 77.0000  888.0000
```

2D ARRAYS:

2D arrays are very similar to 1D arrays, except you need to worry about both rows and columns. You separate rows by using a semicolon (;) or hitting the return key.

```
> temp = [2, 4, 5 ; 6, 3, 2]
temp =
   2   4   5
   6   3   2
> temp2 = [2 4 5          (when I hit return, no command is executed)
> 6 3 2]
temp2 =
   2   4   5
   6   3   2              (we get the same 2D array using different methods of entering data)
> temp(1,2)
ans =  4                  (4 is now the first row, second column of the array temp)
> temp2(2,3)
ans =  2
> whos
Variables in the current scope:
   Attr  Name       Size                 Bytes  Class
  ==== ====         ====                 =====  =====
         temp       2x3                  48     double
         temp2      2x3                  48     double
```

Both temp and temp2 are 2x3 arrays. As with 1D arrays, we can manipulate specific elements in the 2D arrays.

```
> temp(1,1) = 300
temp =
  300   4   5
    6   3   2
> temp(2,1) = temp(1,1)
temp =
  300   4   5
  300   3   2
> a = temp(2,2) * temp(2,3)
a =  6
```

MATRIX OPERATIONS:

There will be times where you will want to perform an operation on all the elements of an array. For example, multiply all the elements of an array by a constant.

```
> a = [2, 3, 9 ; 4, 4, 7]            (a is a 2x3 array)
a =
  2  3  9
  4  4  7
> d = 3
d =  3
> b = a.*d
b =
   6   9  27
  12  12  21
```

Notice I used the **.*** operator instead of just the ***** operator. Adding a period in front of the operator means "do this to every element of the array." When we only are multiplying, dividing, adding, or subtracting by a constant (1x1 array) it doesn't matter if we use the period, but if **d** was not a constant we may run into trouble.

```
> b = a*d      (We get the same result using .* and * in this case)
b =
   6   9  27
  12  12  21
> a = a + 1
a =
   3   4  10
   5   5   8
> a = a .+ 1   (again, we get the same result using .* and * since 1 is a constant)
a =
   4   5  11
   6   6   9
```

```
> a = a/10
a =
   0.40000   0.50000   1.10000
   0.60000   0.60000   0.90000
```

However, when we try to raise the array **a** to the power of 2, we get an error.

```
> a = a^2
error: for A^b, A must be square
```

This is because Matlab/Octave is trying to multiply the array **a** by itself (a 2x3 array multiplied by another 2x3 array), which is forbidden by the rules of matrix multiplication. If you want to square each element of the array **a**, use **.^**

```
> a = a.^2
a =
   0.16000   0.25000   1.21000
   0.36000   0.36000   0.81000
```

In general, it is safest to <u>always</u> use a period in front of the operation when you want to perform an operation on each element of an array separately.