EXTRA NOTES ABOUT ARRAYS:

You can enter multiple values into an array by putting brackets around two or more numbers.

```
> val = input('Enter some values: ');
Enter some values: [2 5 8]
```

If you use an array as an argument for a built-in function, the function will act on all elements in the array.

```
> d = sqrt(val)
d =
  1.4142   2.2361   2.8284
```

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 always use a period in front of the operation when you want to perform an operation on each element of an array separately.

COMBINING ARRAYS:

You can combine multiple arrays into a single array.

```
> time = [0, 4, 8, 12];
> temps = [55, 52, 58, 63];
> array1 = [time ; temps]
array1 =
   0   4   8  12
  55  52  58  63
> whos
Variables in the current scope:
  Attr   Name       Size                Bytes  Class
  ==== ====       ====                =====  =====
          array1     2x4                    64  double
          temps      1x4                    32  double
          time       1x4                    32  double
```

TRANSPOSE OPERATOR:

If you want to change your array from a MxN array to a NxM, you can use the transpose operator, which is a single quotation mark. There are many times when this operator is useful, including when you want to view data in a more convenient format.

Mathematically, $A^T(i,j) = A(j,i)$

```
> time = [0, 4, 8, 12];
> temps = [55, 52, 58, 63];
> array1 = [time ; temps]
array1 =                          (This style doesn't look that great)
   0    4    8   12
  55   52   58   63
> array1'
ans =                             (This style looks much better)
   0   55
   4   52
   8   58
  12   63
```

There are other ways to do this operation, which you will learn later.

SHORTCUTS FOR DEFINING AND HANDLING ARRAYS:

Matlab/Octave are great for dealing with arrays. It should come as no surprise that there are a lot of nifty commands that will save you a lot of time when manipulating data in arrays. However, we will not be using all these shortcuts in this course because these shortcuts will not be available in most other computer languages.

The colon operator (:)

In general, this operator is used in the following manner,

a : c : b

From a to b in steps of c.

For example,
0 : 0.25 : 2 means go from 0 to 2 in steps of 0.25

If you omit c, the default value of c is 1
0:2 means go from 0 to 2 in steps of 1

If you omit a and b as well, the colon operator means "everything" (see below)

(A) If you want to make an array with element values that are evenly spaced, you can use the colon operator.

```
> time = [0 : 0.25 : 2]                    (this will work with and without the brackets)
time =
   0.00000   0.25000   0.50000   0.75000   1.00000   1.25000   1.50000   1.75000   2.00000
```

(B) You also can use the colon operator to extract part of an array. In this case, the colon operator means "everything." For example, let's extract the second column out of a 3x3 array.

```
> a = [3.5  4.6  7.1 ; 1.1  -0.9  -1. ; 3.3  3.3  2.3]
a =
   3.50000   4.60000   7.10000
   1.10000  -0.90000  -1.00000
   3.30000   3.30000   2.30000
> b = a( : , 2 )                    (all rows, second column only)
b =
   4.60000
  -0.90000
   3.30000
```

The colon operator alone means "all rows" in the second column

(C) If the colon operator is used in an array index and appears between two numbers, it means extract from the first number to the second number. For example,

```
> c = a(2:3 , 2:3)                  (extract rows 2 to 3, columns 2 to 3 from a and put them into c)
c =
  -0.90000  -1.00000
   3.30000   2.30000
```

The end command

If you forget the number of rows or columns in your array, you can use the **end** command.

```
> c(end,end)
ans =  2.3000
> a(1,end)
ans =  7.1000
```

```
> a(end,end)
ans =  2.3000
```

The size( ) and numel( ) commands

If you want to know how many elements are in an array, use the **numel( )** command.
If you want to know the number of elements in each dimension, use the **size( )** command.

```
> a = [2:2:10 ; 3:3:15]
a =
   2   4   6   8  10
   3   6   9  12  15
> numel(a)
ans =  10
> size(a)                    (The size( ) command returns the number of rows and columns)
ans =
  2   5
```

You can store the number of rows and columns in variables.

```
> [row, col] = size(a)
row =  2
col =  5
> numberElements = row * col
numberElements =  10
```

If you assign **size(a)** to a variable, it will create a 1x2 array containing the number of rows and columns.

```
> arrayInfo = size(a)
arrayInfo =
  2   5
> arrayInfo(2)
ans =  5
```

CHARACTER DATA:

Earlier we touched upon character data. Character data is perfect for arrays… each individual character of a word or phrase can be stored in an array element.

```
> food = 'cheese'
food = cheese
> drink = 'pepsi'
drink = pepsi
```

food:

C	H	E	E	S	E

drink:

P	E	P	S	I

```
> food (1)              (we can get just the first element of the array food)
ans = c
> drink(5)
ans = i
> food(1:3)             (we can get a fraction of the array elements)
ans = che
> var23 = food(1:3)
var23 = che
> var23
var23 = che             (the array var23 has been assigned a fraction of food's elements)
> whos
Variables in the current scope:
  Attr   Name        Size                Bytes  Class
  ====   ====        ====                =====  =====
         drink       1x5                     5  char
         food        1x6                     6  char
         var23       1x3                     3  char
```

You can create character arrays as well. Each row will contain a word or phrase.

```
> names = ['Paul' ; 'Bill' ; 'Susan' ; 'Ana']
names =
Paul
Bill
Susan
Ana
> names(1,3)
ans = u
> names(1,5)
ans =                   (there is a blank space here)
> names(2,2)
ans = i
> names(1,6)
error: A(I): Index exceeds matrix dimension.
```

Each row is given 5 characters because the longest name has 5 characters. Blank characters will be assigned to the end of a word/phrase if it is less than the longest word/phrase.

In your textbook, it states that Matlab won't always like the previous method of creating character arrays. Here is an alternative using the **char( )** function.

```
> names2 = char('Paul' , 'Bill' , 'Susan' , 'Ana')
names2 =
Paul
Bill
Susan
Ana
```

You can combine different character strings in a single array to make one long character string.

```
> a = 'hi'
a = hi
> b = 'bye'
b = bye
> c = [a,b]
c = hibye
> d = [a b]
d = hibye
> f = [a ' ' b]
f = hi bye
```

We combined character strings in the past when using the **disp( )** command.

In mfile.m:

```
x = 5;
disp(['The value of x is ' , num2str(x)])
```

At command line:

```
> mfile
The value of x is 5
```