VARIABLES – Storing numbers:

You may create and use variables in Matlab to store data. There are a few rules on naming variables though:

(1) Variables must begin with a letter and can be followed with any combination of letters, numbers, or underscores only
(2) Variable names must not exceed 63 characters in length
(3) Variables are case sensitive. The variable 'Apple' is not the same as the variable 'apple'

Examples of acceptable variables: a, b, c, apple, Apple, ApPLE89_3, ILIKECHEESE
Examples of unacceptable variables: 89, 8fish, Fis h, pizza!

You should choose variable names that are descriptive. For example, use the variable 'numapple' is a variable that stores information about the number of apples.

You can check if a variable is acceptable with the **isvarname** command. This is just one of many built in commands in Matlab/Octave.

```
> isvarname pizza
ans = 1                 (Note: 0 = no, 1 = yes)
> isvarname pizza!
ans = 0
```

Matlab/Octave, unlike other languages such as Fortran, we do not need to declare the data type of variables (INTEGER, REAL, ETC.). All variables in Matlab/Octave are assumed to store floating point data unless otherwise stated (there will be times where we will want to force a variable to be an integer). In fact, you do not even need to declare variables. They are created when you assign them a value. The equals sign (=) means 'assign the quantity on the right to the variable on the left' not 'equals'.

```
> b = 2
b = 2
```

The value on the right side is assigned to the variable **b**. Think of **b** as a box that can hold numbers. You can use **b** to perform computations.

```
> b + 5
ans = 7
> b * 5
ans = 10
```

You can use the value of **b** to calculate the value of another variable, **c**.

```
> b
```

```
b = 2
> c = b + 5
c = 7
```
(2 + 5 is evaluated, and the resulting answer is assigned to **c**)

You can use the value of multiple variables to get the value of another variable, **cow**.

```
> cow = b * c
```
(2 * 7 is evaluated, and the results answer is assigned to **cow**)
```
cow = 14
```

I emphasize that the right side of the equals sign is evaluated first, then the result is assigned to the variable on the left side.

```
> cow = cow + 1
cow = 15
```

This equation is wrong algebraically, but computationally it is fine. The right side (14+1) is evaluated first (14+1 is 15), then assigned to cow.

You can check out what variables have been created with the **who** command.

```
> who
Variables in the current scope:
 ans  b    c    cow
```

You can get more detailed information about the variables with the **whos** command. This shows the variables in your workspace.

```
> whos
Variables in the current scope:
  Attr   Name   Size          Bytes  Class
  ====   ====   ====          =====  =====
       ans         1x1             8  double
       b           1x1             8  double
       c           1x1             8  double
       cow         1x1             8  double
```

There is a lot of information here. I will explain what it all means as the course goes on. The concept of variable scope will become important when we learn about functions. For now, just notice how all the variables are considered 1x1 arrays that can store double precision data (that is, all variables contain floating-point numbers).

You can suppress the output with a semi-colon.

```
> 2 * c
ans = 14
```

```
> 2 * c;
> 2 * c
ans = 14
```

What if you try to use a variable that hasn't been used yet?

```
> z = y * 2      (y hasn't been used yet)
error: `y' undefined near line 19 column 5
> y = 9
y = 9
> z = y * 2
z = 18
```

Matlab/Octave is case sensitive.

```
> t = 6.6
t = 6.6
> z = T * 2
error: `T' undefined near line 19 column 5
> z = t * 2
z = 13.200
```

Variables will exist until you exit Octave or erase them with the **clear** command.

```
> clear
> whos
```

(nothing appears since all the variables are gone)

If your screen gets too messy, you can use the **clc** command.

Limits on numerical data:

There is a limit to the size of the numbers you can store in memory.
Largest double-precision floating-point number: 1.7977e+308
Smallest double-precision floating-point number: 2.2251e-308
Largest integer: 2147483647
Smallest integer: -2147483648

Sometimes you run into problems when you do mathematical operations with very large and very small numbers

```
> x = 1e200
x =  1.0000e+200
> y = 1e-200
```

```
y =  1.0000e-200
> z = x/y
z = Inf                 (the true answer is 1.0000e+400)
> z = y/x
z = 0                   (the true answer is 1.0000e-400)
> z = (y/x)*x
z = 0                   (The true answer is 1.0000e-200.  Remember statements are evaluated
                         left to right)
```

VARIABLES – Characters:

You can store character data in variables too. Make sure to put single quotes around the text.

```
> a = 'banana'
a = banana      (the variable a stores the word 'banana')
> b = 'chimp'
b = chimp       (the variable b stores the word 'chimp')
> c = chimp
error: `c' undefined near line 29 column 5
```
(Whoops, there is no variable called **chimp**.)
```
> d = 5.6
d = 5.6
> b = 44        (variables can change the type of data stored in them)
b  = 44

> whos
Variables in the current scope:
  Attr   Name        Size                Bytes  Class
  ====   ====        ====                =====  =====
         a           1x6                     6  char
         b           1x2                     2  char
         d           1x1                     8  double
```

Notice that **b** and **d** are character variables. The arrays are bigger than 1x1. Each letter is stored in a separate space in memory. We will discuss this shortly.

VARIABLES – Logical:

We will learn about this type of variable later.

BUILT-IN VARIABLES AND FUNCTIONS:

Matlab/Octave contains a large library of commonly used variables and functions. The variable **pi** is reserved for π (3.141592654…), *i* and *j* are reserved for imaginary numbers, and so on.

```
> pi
ans =  3.1416   (many more digits are stored, but only 4 digits right of the decimal are displayed)
> pi*3
ans =  9.4248
> degrees = 90
degrees =  90
> radians = degrees * (pi/180)
radians =  1.5708                    (this is π/2)
```

There are many different types of functions. Almost all functions are given one or more arguments. For example, the trigonometric function **sin(x)** has one argument, **x**, which must store number data. The arguments of trigonometric functions are in radians.

```
> sin(0.)
ans = 0
> cos(0.)
ans =  1
> sin(90.)
ans =  0.89400          (remember that arguments of trigonometric functions must be in radians)
> sin(3.141592654)
ans = -4.1021e-010      (very close to 0)
> sin(pi/2)
ans = 1
```

Some other functions:
sqrt(x) calculates the square root of x
exp(x) calculates e^x
abs(x) calculates |x|

See Chapter 3 for a longer list of functions.

Matlab/Octave has many functions that allow you to do common tasks with a single command. Other languages, such as Fortran and C, may not have these built-in functions and are less 'user-friendly.' For example, in Matlab/Octave **mean(x)** finds the average value of 1 or more numbers stored in **x**. However, in this class we will not be using many of these functions because I want you to understand the thinking behind the functions.

Using function names as variable names:

You may use function names as names for variables. In general this is a bad idea because it may confuse the user.

```
> pi
ans =  3.1416
> 3*pi
ans =  9.4248
> pi = 7
pi =  7
> 3*pi
ans =  21
> sin = 4
sin =  4
> sin(4)
error: A(I): Index exceeds matrix dimension.
```

There are certain keywords that are not allowed to be used for variables. Use the command **iskeyword** to see the keywords

GETTING HELP:

Matlab/Octave have a lot of documentation built-in that can help you if you need more information. For example, if you need help with the **sin( )** function,

```
> help sin
`sin' is a built-in function
 -- Mapping Function:  sin (X)
    Compute the sine for each element of X in radians.
    See also: asin, sind, sinh
```

There is a TON of information and tutorials online too. Google is your friend.

CREATING M-FILES:

So far we have only been executing simple commands at the command line. What if you want to make a small change to a code that has a lot of commands without having to type those commands over again every time you run your code?

You can create an "M-File" – a simple text file that ends with a **.m** extension. You can make simple text files using text editors like Notepad and Wordpad, or at the command line using the edit command.

```
> edit filename.m
```

You should see a text editor on the screen (sometimes there will be a lot of extra text). Erase any extra text and start typing your commands. For example, an M-File file might look like this:

```
x = 5;
y = 7
c = x*y
```

Save your program as **filename.m** (where **filename** could be anything you wish). To run your M-File, type the filename minus the .m extension at the command line.

```
> filename      (notice there is not .m extension)
y =  7        (x = 5 is suppressed due to the use of a semi-colon)
c =  35
```

You may open and edit your M-File later if needed.

It is a good idea to use the clear command at the beginning of your M-File. This will remove all variables from memory.

```
clear;
```

COMMENTARY:

The percent symbol (**%**) is used to create commentary. There are two reasons you should heavily comment your code: (1) Another user may want to use your code and will need to understand what is going on; (2) When your code becomes long, you may need to remember exactly what your code is doing. Anything after the **%** will be ignored by Matlab/Octave.

In mfile.m:

```
% hello          (nothing happens)
% x = 5          (same)
```

```
y = 7
z = 99  % hello
hi = 12%3
```

At command line:

```
> mfile
y = 7
z = 99
hi = 12
```

The variable **x** is never assigned a value, the text ‘hello’ is ignored by the program, and the 3 is cut off of 123 due to the percent sign.