ENME202 Matlab

VARIABLES AND THE ASSIGNMENT OPERATOR

VARIABLE DEFINITIONS:

So far we have used Matlab as a calculator, with functions or operators applied to numerical values to generate a numerical result.

A computer program needs to be able to not just perform calculations, but store the results in memory so that they can be accessed for later use.

We use variables to reference these stored values in our code. Think of a variable name as simply a reference to a memory location where the value is stored. When we use the variable in our code, we are telling the computer to retrieve or modify the value from the associated memory location.

In the statement below, a is the name of a variable, and a = 2 "assigns" the value of 2 to the variable called a. In other words, this command "stores" the value 2 in the variable called a.

```
a = 2;
```

Note: the semicolon suppresses the output response from Matlab (except for error messages), but the assignment still takes place.

Leaving the semicolon off an assignment will result in Matlab simply confirming that the desired value has been associated with the specified variable

```
b = 3;
```

% vs.

```
b = 3
```

```
b = 3
```

Note that variable names MUST be to the left of "=" in an assignment. A statement like "3 = b" is INVALID and will cause an error!

```
% 3 = b   % Error: The expression to the left of the equals sign is not a valid target
          % for an assignment.
```

Type in the name of a variable to display its assigned value

```
a
```

```
a = 2
```

Once defined, variables can be used with any allowed function or operation

```
sqrt(a)   % find the square root of the number stored in variable a
```

```
ans = 1.4142
```

Can define new variables in terms of existing ones, or functions of existing variables.

```
c = sqrt(a)   % the value stored in "a" is used to compute the value which
```

```
c = 1.4142
```

```
              % is then stored in new variable "c".
```

Example: quadratic formula

Set up variables with the coefficients of the quadratic:

```
a = 2; b = -8; c = 6;   % We can define multiple variables on one line
```

Implement the quadratic formula in Matlab notation

```
x1 = (-b+sqrt(b^2-4*a*c))/(2*a)
```

```
x1 = 3
```

```
x2 = (-b-sqrt(b^2-4*a*c))/(2*a)
```

```
x2 = 1
```

Result is real since the discriminant (b^2-4*a*c) is >= 0. We will see later what happens if the roots are complex.

Variable names must be on the left side of the assignment operator (=). The right side must be something that evaluates to a *value*

```
c = sqrt(37)
```

```
c = 6.0828
```

```
d = exp(-sqrt(b))   % This would be an error if b did not already
```

```
d = -0.9514 - 0.3081i
```

```
                    % have a number stored in it!
e = log(d);
```

Reminder: Matlab prints out nothing for e, but it HAS stored the result of log(d) in the variable e. The semicolon told it to "suppress" display of the answer. This is particularly useful if we have a long chain of calculations, using several variables, but we only really care about the final result (see several of the examples below).

Note that "ans" is a special variable that holds the result of last calculation not explicitly assigned to another variable.

```
sqrt(49)
```

```
ans = 7
```

```
ans
```

```
ans = 7
```

```
x = 10;
ans
```

```
ans = 7
```

You can change the value stored in a variable at any time

```
a = pi
```

```
a = 3.1416
```

```
a = cosd(45)
```

```
a = 0.7071
```

Using descriptive variable names can help make your code readable

```
radius = 5.2;
volume = 4*pi*radius^3/3
```

```
volume = 588.9774
```

But keeping variable names short is also helpful. Using short (but relevant) variable names, *combined with comments*, is often the best choice:

```
r = 5.2;          % r = sphere radius
v = 4*pi*r^3/3    % v = sphere volume
```

```
v = 588.9774
```

Variable names are CASE SENSITIVE

```
% V    % ??? Undefined function or variable 'V'.
```

Matlab variable names can be up to 64 characters but must start with a letter (this is true for Matlab script names as well):

```
thisisaverylongvariablename = 1
```

```
thisisaverylongvariablename = 1
```

Use "snake case" (underscores) to structure a variable name:

```
this_is_a_very_long_variable_name = 2
```

```
this_is_a_very_long_variable_name = 2
```

Or use "camel case":

```
thisIsAVeryLongVariableName = 3
```

```
thisIsAVeryLongVariableName = 3
```

The above is an example of "lower camel case", since the 1st letter is lower case. Upper camel case (also called Pascal case because it was widely used in the Pascal programming language) capitalizes the first letter:

```
ThisIsPascalCase = 4
```

```
ThisIsPascalCase = 4
```

Pascal case if generally reserved for naming functions and classes (we'll talk about classes in C++).

Use "who" to get a list of all variables currently defined in the "workspace" (Matlab's term for current "scope" of the system):

```
who
```

```
Your variables are:

ThisIsPascalCase                  e                                 v
a                                 r                                 volume
ans                               radius                            x
b                                 thisIsAVeryLongVariableName       x1
c                                 this_is_a_very_long_variable_name x2
d                                 thisisaverylongvariablename
```

We can "erase" a variable using the clear command:

```
clear a
who
```

```
Your variables are:

ThisIsPascalCase                  r                                 volume
ans                               radius                            x
b                                 thisIsAVeryLongVariableName       x1
c                                 this_is_a_very_long_variable_name x2
d                                 thisisaverylongvariablename
e                                 v
```

Built-in variables like pi can have their values changed too (generally NOT a good idea to do this, so be careful):

```
pi = 3
```

```
pi = 3
```

```
2*pi
```

```
ans = 6
```

Clearing the new value for a built-in variable will reset it back to its original default:

```
clear pi
pi
```

```
ans = 3.1416
```

This can happen with *function* names as well

```
sin = 3;
% sin(pi/4)   % expect 0.7071, but will give an error
```

Ooops -- now Matlab thinks sin is a variable instead of a function!

```
clear sin
sin(pi/4)
```

```
ans = 0.7071
```

Using clear by itself deletes ALL variables (same as "clear all")

```
clear
who
```

Functions can be "nested", with the input to one function depending on the result of another. In such an expression, the "innermost" function is evaluated first, yielding a number which is then used as the input to the next function.

In the above, the numerical result of log(8) is evaluated first, then used as the input to the exp function. Since exp and log are inverse functions, the result is of course just 8.

```
exp(log(8))
```

```
ans = 8.0000
```

```
sin(sqrt(pi))
```

```
ans = 0.9797
```

You can nest functions as deep as you want:

```
sin(sqrt(log(pi)))
```

```
ans = 0.8772
```

```
% It is possible to implement very complicated formulas like this,
% but can sometimes get confusing if the nesting gets too deep.  In such
% a case, you can always use variables to hold intermediate results

a = log(pi);
b = sqrt(a);
sin(b)
```

```
ans = 0.8772
```

"Assignment" vs. "Equals"

When working with variables, only the numerical result of an assignment is stored. Assignment does NOT create a formula that automatically changes value as the argument values change.

```
a = 4; b = 3;
c = a/pi+b   % c = 4.2732
```

```
c = 4.2732
```

Suppose we now change the value stored in a:

```
a = -2;
```

What does c now contain?

```
c            % c = 4.2732
```

```
c = 4.2732
```

Value in c hasn't changed! If we want it to update based on new value of a we have to repeat the assignment using the same formula (use the "up arrow" key, or the "command history" pane, to repeat previous calculations and save a lot of typing!!):

```
c = a/pi+b   % c = 2.3634
```

```
c = 2.3634
```

The key point is that the equal sign does not mean "equal to", but instead means "is assigned the value of" (this is why "=" is called the "assignment" operator instead of the "equals" operator).

Writing mathematical expressions in Matlab (and other programming languages) may look like algebra, but it is not. Each statement is simply a DIRECTIVE to the computer to do a specific calculation, and assign the resulting value to a given variable name. Matlab does NOT remember relationships between variables implied by your previous directives.

This is why it makes sense to write things like:

```
a = 2;
a = a + 3  % a = 5
```

```
a = 5
```

```
a = a * 6  % a = 30
```

```
a = 30
```

Similarly, Matlab is not natively "symbolic", that is it cannot work with expressions containing variables unless that variable has a defined value stored in it.

```
% c = 5*x+2       % Error: Undefined function or variable 'x'.
```

A value has not yet been assigned to x, so Matlab doesn't know what to do here.

Matlab does have a symbolic "toolbox" that allows purely symbolic expressions to be evaluated, but WE WILL NOT USE IT IN THIS COURSE. If you have learned symbolic toolbox functions elsewhere, DO NOT USE THEM IN THIS ENME202.

Any submitted code which attempts to use symbolic calculation will be marked incorrect!

SCRIPTS

Every language has its own terminology to refer to a file containing code for that language. In Matlab we call the code files either "scripts" or "m-files" since the default extension for the scripts is ".m"

Scripts are just text files which contain a list of statements, exactly like you would type them into the command window. After they are saved, typing the name of the script in the command window will execute ALL code in the file.

Example:

Make & run a script called "quadform.m" that will calculate x1 and x2. The script has 3 lines, as follows:

```
a, b, c       % Report the current values of a, b, and c
```

```
a = 30
b = 3
c = 2.3634
```

```
x1 = (-b + sqrt(b^2 - 4*a*c)) / (2*a)   % 1st root
```

```
x1 = -0.0500 + 0.2762i
```

```
x2 = (-b - sqrt(b^2 - 4*a*c)) / (2*a)   % 2nd root
```

```
x2 = -0.0500 - 0.2762i
```

Remember that the file must be saved to the U: drive when running Matlab through VCL to ensure that the file will remain accessible after closing the current Matlab session.

Note that a, b, and c are not defined in the script, so we will need to manually define these variables in the command window before running the code.

```
clear
a=2; b=4; c=10;
quadform
```

```
a = 2
b = 4
c = 10
x1 = -1.0000 + 2.0000i
x2 = -1.0000 - 2.0000i
```

Now try:

```
a = 3;
quadform
```

```
a = 3
b = 4
c = 10
x1 = -0.6667 + 1.6997i
x2 = -0.6667 - 1.6997i
```

Results for new value of a, and old values of b and c since we didn't change those.

Scripts use existing values for any variables already declared in the workspace.

CODING STYLE

You should always strive to make your scripts well organized, easy to read, well commented, and with good coding style with a consistent variable naming convention and selection of meaningful variable names.

It is always a good idea to put comments at the start of scripts to indicate what the script does, and to add comments throughout the code to explain what each line or group of lines is doing.

```
%{
   An entire block of text may be commented out by adding
   curly braces, as shown here.  This is very convenient when you have
   many lines of text that can change over time, and you don't want
   to worry about maintaining the percent symbol at the front of each line.
%}
```

Another thing that can sometimes improve clarity and reduce the risk of mistakes ("bugs") is to introduce intermediate variables. Let's take the quadratic formula as an example again, but now with some intermediate variables:

```
a=2; b=4; c=10;
disc = b^2 - 4*a*c;              % disc = discriminant
x1 = -b/(2*a) + sqrt(disc)/(2*a)
```

```
x1 = -1.0000 + 2.0000i
```

```
x2 = -b/(2*a) - sqrt(disc)/(2*a)
```

```
x2 = -1.0000 - 2.0000i
```

Or we could use even more intermediate variables, like

```
a=2; b=4; c=10;
disc = b^2 - 4*a*c ;
term1 = -b/(2*a) ;
term2 = sqrt(disc)/(2*a) ;
x1 = term1 + term2
```

```
x1 = -1.0000 + 2.0000i
```

```
x2 = term1 - term2
```

```
x2 = -1.0000 - 2.0000i
```

Other times, it might help to re-define the problem using new variables. Here, we could re-define the problem to be x^2 + (b/a)*x + (c/a) = 0, as

```
a=2; b=4; c=10;
bnew = b/a ;
cnew = c/a ;
x1 = -bnew/2 + sqrt( bnew^2-4*cnew )/2
```

```
x1 = -1.0000 + 2.0000i
```

```
x2 = -bnew/2 - sqrt( bnew^2-4*cnew )/2
```

```
x2 = -1.0000 - 2.0000i
```

Finally, note that you can have as many white spaces as you want in Matlab. Sometimes this can make things look cleaner -- Matlab doesn't care, but here are two general rules to follow:

1. Use the same number of spaces for all indentations; don't have some lines indented by 2 spaces and other lines by 5 spaces.

2. Indent blocks of code that are the same "level" equally. This will make more sense when we talk about conditionals, control statements, and custom functions.