ENME202 Matlab

LOGICAL COMPARISONS

Logical expressions allow us to compare multiple values, with a resulting return value of either 1 (true) or 0 (false):

```
2 < 3     % ans = 1
```

```
ans = logical
   1
```

```
2 < 1     % ans = 0
```

```
ans = logical
   0
```

```
2 <= 2    % ans = 1
```

```
ans = logical
   1
```

```
2 < 2     % ans = 0
```

```
ans = logical
   0
```

```
3 >= 2    % ans = 1
```

```
ans = logical
   1
```

We can also use the **true** and **false** keywords fortrue both logical comparisons and mathematical operations:

```
true > 0    % ans = 1
```

```
ans = logical
   1
```

```
false > 0   % ans = 0
```

```
ans = logical
   0
```

```
2*true + 1  % ans = 3
```

```
ans = 3
```

The **equality** comparison operator is '==' (entirely different from the assignment operator '=')

```
3 == 3    % ans = 1
```

```
ans = logical
```

```
    1
```

```
3 == 4     % ans = 0
```

```
ans = logical
    0
```

```
a = 3      % This is an assignment, not a comparison
```

```
a = 3
```

```
b = 0      % another assignment...
```

```
b = 0
```

```
a == 3     % ans = 1 (since we just assigned 3 to a)
```

```
ans = logical
    1
```

```
a == 2     % ans = 0
```

```
ans = logical
    0
```

```
a == b     % ans = 0
```

```
ans = logical
    0
```

The **not equal** comparison is '~='. The tilde (~) means 'not' in Matlab (unlike C++ as we will see later)

```
a ~= b     % ans = 1
```

```
ans = logical
    1
```

```
3 ~= 4     % ans = 1
```

```
ans = logical
    1
```

```
3 ~= 3     % ans = 0
```

```
ans = logical
    0
```

BOOLEAN OPERATIONS

Logical comparisons between single true/false (1/0) values can also be performed. These comparisons are known as Boolean operations (named after the 19th century mathemetician George Boole).

A Boolean truth table displays the outputs of a Boolean operation for all possible combinations of inputs. Here are the truth tables for the fundamental Boolean operations (AND, OR, NOT, XOR):

NOT (~):

```
%   X   | ~X
%  ---  | ---
%   0   |  1
%   1   |  0
```

AND (&):

```
%   X   Y   | X&Y
%  --- ---  | ---
%   0   0   |  0
%   0   1   |  0
%   1   0   |  0
%   1   1   |  1
```

OR (|):

```
%   X   Y   | X|Y
%  --- ---  | ---
%   0   0   |  0
%   0   1   |  1
%   1   0   |  1
%   1   1   |  1
```

Exclusive OR (XOR):

```
%   X   Y   | xor(X,Y)
%  --- ---  | ---
%   0   0   |  0
%   0   1   |  1
%   1   0   |  1
%   1   1   |  0   <-- pay attention here!
```

Logical NOT is indicated with the tilde symbol (~), such that ~y is false if y is true, and true if y is false (flip the bit):

```
~1      % ans = 0
```

```
ans = logical
   0
```

```
~0      % ans = 1
```

```
ans = logical
   1
```

Logical AND is indicated with an ampersand (&). x&y is true if both values are true, false otherwise.

```
1 & 1     % ans = 1
```

```
ans = logical
   1
```

```
1 & 0     % ans = 0
```

```
ans = logical
   0
```

```
0 & 1      % ans = 0
```

```
ans = logical
   0
```

```
0 & 0      % ans = 0
```

```
ans = logical
   0
```

Logical OR is indicated with the vertical bar (|). x|y is false if both propositions are false, true otherwise.

```
0 | 0      % ans = 0
```

```
ans = logical
   0
```

```
0 | 1      % ans = 1
```

```
ans = logical
   1
```

```
1 | 0      % ans = 1
```

```
ans = logical
   1
```

```
1 | 1      % ans = 1
```

```
ans = logical
   1
```

XOR is the "exclusive OR" logical operator. The XOR operator yields true if ONE AND ONLY ONE of the two values being compared is true. Matlab does not have a symbol for the XOR operator, so instead use the **xor()** function:

```
xor(0,0)    % ans = logical 0
```

```
ans = logical
   0
```

```
xor(1,0)    % ans = logical 1
```

```
ans = logical
   1
```

```
xor(1,1)    % ans = logical 0
```

```
ans = logical
   0
```

Note that and() and or() functions also exist in Matlab, with the same functionality as & and |

Here is a typical use of the AND/OR operators:

```
x = 2.06;
(x<3) & (x>1)   % ans = 1
```

```
ans = logical
   1
```

```

x = 4;
(x<3) & (x>1)   % ans = 0
```

```
ans = logical
   0
```

```
(x<3) | (x>1)   % ans = 1
```

```
ans = logical
   1
```

Any combination of logical operators can be combined into a single Boolean expression:

```
a=1; b=0; c=1;
(a & ~b) | ((c & a) | xor(a,b))
```

```
ans = logical
   1
```

Here is the truth table for the above Boolean statement, evaluated term-by-term:

```
%  a   b   c    ~b   a&~b  c&a  xor(a,b)  (c&a)|xor(a,b)  (a&~b)|((c&a)|xor(a,b))
% --- --- ---  ---  ----  ---  --------  --------------  -----------------------
%  0   0   0    1     0    0      0            0                   0
%  0   0   1    1     0    0      0            0                   0
%  0   1   0    0     0    0      1            1                   1
%  0   1   1    0     0    0      1            1                   1
%  1   0   0    1     1    0      1            1                   1
%  1   0   1    1     1    1      1            1                   1
%  1   1   0    0     0    0      0            0                   0
%  1   1   1    0     0    1      0            1                   1
```

Logical operator precedence: NOT > AND > OR

```
% A | B & C       -->  A | (B & C)
% A & B | C & D   --> (A & B) | (C & D)
% A & B & C | D   --> ((A & B) & C) | D
% ~A & B | C      --> ((~A) & B) | C
```

Logical operators can also be applied to arrays, allowing multiple bits to be compared at once. In this case we can think of the arrays as representing "binary words" (bit sequences of defined length).

```
[1 1 0 0] & [0 1 0 0]
```

```
ans = 1×4 logical array
```

```
    0   1   0   0
```

```
xor( [1 1 0 0], [0 1 0 0] )
```

```
ans = 1×4 logical array
    1   0   0   0
```

```
~[1 0 0 1]
```

```
ans = 1×4 logical array
    0   1   1   0
```

Consider the previous example of (a & ~b) I ((c & a) I xor(a,b)). We can build the full truth table using arrays:

```
a = [0 0 0 0 1 1 1 1];
b = [0 0 1 1 0 0 1 1];
c = [0 1 0 1 0 1 0 1];
(a & ~b) | ((c & a) | xor(a,b))   % matches above result, ok
```

```
ans = 1×8 logical array
    0   0   1   1   1   1   0   1
```

Logical operators are useful for a broad range of applications. For example, consider how to find all of the elements of an array x that are NOT equal to either 1 or 3.

```
x = [-2, 5, 1.5, 3, 1, 29, 1, 1];

find(x~=1 & x ~=3)      % ans =  1  2  3  6
```

```
ans = 1×4
      1     2     3     6
```

Short Circuiting

The AND and OR operators can be used with "short circuiting", meaning that if the first expression in the comparison has a value that ensures a particular output, regardless of the second expression's value, Matlab will not bother evaluating the second expression. In this way short circuiting can reduce the computation time. The short circuiting operators are:

AND --> &&

OR --> II

ANY, ALL

any(x) returns 1 iff any element of x is true

all(x) returns 1 iff all elements of x are true

```
x = [1 1 0 0];
y = [0 0 0 0];
z = [1 1 1 1];
```

```
any(x)  % true
```

```
ans = logical
   1
```

```
all(x)  % false
```

```
ans = logical
   0
```

```
any(y)  % false
```

```
ans = logical
   0
```

```
all(z)  % true
```

```
ans = logical
   1
```

How would you write custom functions that have the same behavior as any() and all() ??

Binary Math

A **binary word** is a sequence of binary digits of some defined length. A binary word can be interpreted as a binary (base-2) number. For example, 1101 is a 4-digit binary word that is equal to 13 in base-10. The built-in functions **bin2dec()** and **dec2bin()** can be used to convert back and forth between the binary and decimal number systems, using binary words represented using strings:

```
bin2dec('101')
```

```
ans = 5
```

```
dec2bin(9)
```

```
ans = '1001'
```

```
% Add b01010 + b00110
dec2bin(bin2dec('01010') + bin2dec('00110'))
```

```
ans = '10000'
```

Matlab also supports **bitwise** functions **bitand()**, **bitor()**, **and bitxor()** that operate on base-10 numbers for arguments:

```
x = bitand(3,6);   % 3 -> '011'   6 -> '110'
dec2bin(x)         % ans = '10'
```

```
ans = '10'
```

```
x = bitor(3,6);    % 3 -> '011'   6 -> '110'
dec2bin(x)         % ans = '111'
```

```
ans = '111'
```

```
x = bitxor(3,6);    % 3 -> '011'   6 -> '110'
dec2bin(x)          % ans = '101'
```

```
ans = '101'
```