Review: Variables in MATLAB

Remember in a previous video, we stated that every variable in MATLAB has three key features:

1. a user-define symbolic name
2. an assigned value
3. a specific storage location

The actual bitstring stored in memory depends on the type of data we store. MATLAB features 16 fundamental types of data, known as *classes*, each of which is stored in the form of a matrix or an array. These are grouped into 7 larger categories including

1. logical
2. string
3. character

   numeric (has 10 different data classes within this category)

14. table
15. cell
16. struct

There is one additional data class, called a function handle, which is always stored as a scalar (a 1-by-1 matrix). In our work together, we explore all of these classes to get a general idea of we can leverage these constructions to solve programming problems. In this tutorial, we focus on the `logical` class.

Logical data class

We can use variables stored in the `logical` data class to test conditions in our programs. Each entry in a logical variable can take precisely one of two values:

Truth value	Assigned variable value
`true`	`logical 1`
`false`	`logical 0`

By default, logical values are stored as arrays. Each entry of a logical array is stored using a single byte of memory (8 bits). Let's take a look at some code that saves two separate logical variables in memory:

```
1  T = logical(1);
2  F = logical(0);
```

We can also use the `true` and `false` functions to achieve the same results:

```
1  T = true;
2  F = false;
```

Let's use the workspace and the format debug command to highlight the difference between variable declaration whose assigned value is encoded using the logical data class and a variable declaration with an assigned valued based on a numeric data class. By default, MATLAB stores data using the `double` class.

```
1  zero = 0;
2  one = 1;
```

Logical variables can be used for many purposes:

A. To perform a logical operation

   We use logical operations to test conditions on one or more logical variables. We can use these operations to transform the logical foundations of mathematical reasoning into computer code that executes based on these conditions.

B. To perform a relational operation

   We use relational operators, sometimes called relational conditions, to compare elements in two arrays. For example, we might want to test if two variables are equal or not equal. We can also test if one variable is greater than (less than) or equal to another variable. For each of these tests, we return a `true` or `false` value to indicate if the relation holds using a `logical` variable to store the result of each test.

C. To test the *state* of MATLAB variables

   We sometimes might find it helpful to test variables for specific characteristics. We might want to test if a given input is stored using a specified data type. Or we might want to test if a given variable meets certain criteria. To do this, we can use logical variables to report the results of such a test.

D. To address or find special entries in an array

   When coding in MATLAB, we sometimes find it helpful to test variables for specific characteristics. We might want to test if a given input is stored using a specified data type. Or we might want to test if a given variable meets certain criteria. To do this, we can use logical variables to report the results of such a test.

Let's go through each of these together to get a sense of what we mean. In this first video, we'll focus on the using logical values to perform logical operations.

Logical scalars

We begin with a theoretic discussion. So much of modern day mathematics (and hence any STEM field that relies on mathematical modeling) is stated in the language of set theory. Set theory, at its core, provides the ability to define inclusion and exclusion from a collection of object using *propositions*.

We say that a **proposition** is a statement that has exactly one truth value: `true` which we denote at T, or `false` which we denote as F. We can assess the specific truth value for a single statement in mathematics and report the result accordingly. For example, let's consider the following statement:

$$\text{Proposition P: } 2^4 = 8$$

Under the assumptions of standard algebra, this proposition is false since

$$2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16 \neq 8.$$

On the other hand, we see

$$\text{Proposition Q: } 2^4 = 16$$

is true. One of the great accomplishments of the 20th century is to build electronic computation machines that encode abstract mathematical logic using electronic signals. The foundational idea of that work is to encode truth values using physical electronic devices. In this encoding, we use the following maps between ideas:

Truth value	Assigned variable value	Voltage Level (Positive Logic)
`true`	`logical 1`	High
`false`	`logical 0`	Low

MATLAB's `logical` class empowers us to encode and leverage this concept. When we are reporting that a *logical* variable takes a value of `1`, that is equivalent to the idea that the corresponding proposition we're testing is `true`. If that logical variable has an assigned value of `0`, this implies the corresponding proposition we are testing is false. Notice that in our code above, we focused our work on *logical scalars*, also know as scalar-valued logical expressions. Below we provide an explicit definition of what this means:

logical scalar (noun)

> An expression in MATLAB that produces an output in the form of a 1-by-1 logical array (a scalar). There is exactly one assigned value for this output out of a choice of two possible values. This output can either be `true` with logical value `1` or `false` with logical value `0`. When the expression is evaluated, the single scalar-valued logical output is assigned based on the truth value of the expression.

Logical vectors

The `logical` function in MATLAB is quite interesting. This function transform numerical input values into logical output variables. Specifically, if we look at the help documentation for the logical function, we see that the syntax for this function is as described below. Suppose we have a 1×10 row vector $\mathbf{x}$ whose entry-by-entry definition is given by

$$\mathbf{x} = \begin{bmatrix} 0 & 1 & 0 & 2 & 0 & 4 & 0 & 8 & 0 & 16 \end{bmatrix}$$

As we've seen in previous videos, we can create this vector in MATLAB using the following code:

```
x = zeros(1, 10);
twos = 2*ones(1,5);
x(1, 2:2:10) = twos.^(0:4);
lx = logical(x);
```

Notice that each entry of variable `x` is stored using the `double` class, which is one of MATLAB's 10 native numeric data classes. In the last line of this code, we place the variable `x` as input to the `logical` function and store the output as a vector `lx`. The reported value for vector `lx` is a 1×10 row vector whose individual entries are logical scalars. Each nonzero entry of `x` is replaced with a logical `1` while every entry equal to zero is replaces with logical `0`.

Let's take a look at a second example of how this function works, this time using column vectors:

```
y = zeros(12, 1);
one = ones(4,1);
y( 1:3:10 , 1) = one;
y(3:3:12, 1) = -1*one;
ly = logical(y);
```

Notice we have the same pattern, but this time our output is stored as a column vector. In both case, the logical function takes in a one-dimensional numerical array and outputs and *logical vector*, also called a *vector-valued logical expression*, of the same dimension. Below is a more formal definition of a logical vector.

logical vector (noun)

> An expression in MATLAB that produces a vector-valued output each of whose entries are stored as logical scalars. Specifically, this type of logical expression outputs either an $m \times 1$ column vector or a $1 \times n$ row vector, where $m, n \geq 1$. These vector-valued expressions produces outputs with one dimension larger than 1 and the other dimension exactly equal to one. Each entry of this type of array stores a scalar-valued logical expression that has exactly one value, either `true` (stored as logical `1`) or `false` (stored as logical `0`). When the expression is evaluated, the output organizes the collection of such expressions into a vector.

Logical matrices

The `logical` function not only works with scalars and vector but also allows us to transform numeric matrices into logical matrices. For example, suppose variable `A` stores a matrix in MATLAB's workspace with numeric entires. Then, if we write

```
L = logical(A)
```

the matrix L has a `logical 1` (`true`) value in every position where A has a nonzero element and a `logical 0` (`false`) value in every zero position. Suppose we want to test this functionality. Consider the following example

```
1      A = [1 , 0 , -2 ; 0 , 2, -1 ; 1 , 0 , 0];
2      L = logical(A)
```

Notice that the zero elements of L are in the exact positions of the zero elements of A while the values of L equal to `logical 1` are in the locations of nonzero elements of A. We can set up a more general experiment to do this this type of test more quickly using the following code.

```
1  B = randi(4,4,4)-2*ones(4,4);
2  LB = logical(AB);
```

In both of these examples, the `logical` function inputs a 2D numeric array (a matrix) and outputs a 2D logical array of the same size. In other words, this is a *logical matrix*, also called a *matrix-valued logical expression*. Below is a formal definition of this construction.

logical matrix

> The output of a matrix-valued logical expression is an $m \times n$ matrix where both $m \geq 1$ and $n \geq 1$. These matrix-valued expressions produces outputs with both dimensions larger than 1. Each individual entry of this type of array stores a logical scalar that has exactly one value, either `true` or `false`. But when the expression is evaluated, the output organizes the collection of these logical scalars into a matrix having rows and columns.

Important notes on terminology for `logical` data

Of course, it is always true that all scalars are vectors and all vectors are matrices. However, not all matrices are vectors. Nor are all vectors are scalars. The point of this language is to help us speak precisely about the type of data structures we are producing in our work in MATLAB. One of the very powerful yet subtle features of MATLAB is that, because this is an array-based language, almost all operators and functions are designed to work on matrices of any size. But it is sometimes helpful and illustrative to distinguish between scalars (1-by-1), vectors (m-by-1 or 1-by-n), and matrices (m-by-n). For this purpose, we will use the following terminology:

A. scalar: encodes 1×1 data

B. nonscalar column vector: encodes $m \times 1$ data where $m > 1$

C. nonscalar row vector: encodes $1 \times n$ data where $n > 1$

D. nonscalar vector: row or column vector having a dimension not equal to 1

E. nonscalar matrix: an $m \times n$ matrix with at least one dimension larger than 1

F. nonvector matrix: encode $m \times n$ data where $m > 1$ and $n > 1$

As we analyze the logical operators in MATLAB, we will pay special attention to the use of these operators on scalar-valued logical components versus vector-valued logical components. Remember that one of the major features of MATLAB is that this is an array-based language. In other words, all of the data types are assumed to be arrays and all operators are designed to function on array-valued inputs.

With this in mind, it's useful to analyze the input-output relationships of each call and distinguish between scalar-valued and non-scalar-valued inputs. This habit will be quite helpful as we learn to use logical expressions as control conditions in later lessons (think `if` statements and other branching structures).

The logical NOT operator

Let's explore some basic concepts in the logic of propositions and observe how to implement these ideas in MATLAB. One of the first logical operations we can do is called *negation* and also known as the *not operation*. The **negation** of a proposition P, denoted at ∼P, is the proposition "not P". The proposition ∼P is true exactly when proposition P is false. The table below reviews this in detail.

Truth value of P	Logical value for P	Truth value of ∼P	Logical value for ∼P
`true`	`logical 1`	`false`	`logical 0`
`false`	`logical 0`	`true`	`logical 1`

This truth table is quite detailed and provides a lot of information. Sometimes we might desire a more compact formulation of the same ideas. In that case, we can use a more compact truth table for this type of reporting, as seen below:

P	∼P
1	0
0	1

We can test each individual rows of this table using statements in MATLAB given by the following code:

```
1  P = logical(0);  ~P , not(P)
2  Q = logical(1);  ~Q , not(Q)
```

MATLAB's `not` function produces the same output as the ∼ operator. Notice in both of the lines of code above, we are working with logical scalars.

Using this functionality, remember that MATLAB stores all of its fundamental data types as arrays. In other words, we can define multiple entries for logical variables. For example, consider the following MATLAB code

```
1  P = logical([0; 1])
2  [P, ~P]
```

Here we create a column partition definition of a logical vector where column 1 stores all possible logical values of a proposition P and column 2 stores the corresponding values for ∼P. One really powerful consequent of defining logical variables as arrays is that we can run logical operations on logical matrices. For example, take a look at the following code:

```
1      A = [1 , 0 , -2 ; 0 , 2, -1 ; 1 , 0 , 0];
2      L = logical(A)
3      ~L, not(L), not(logical(A)), ~logical(A)
```

The logical AND operator

One of the delightful possibilities in using propositions in mathematics and coding is the idea of combining propositions together. Suppose we have two propositions P and Q and want to test if both are true simultaneously. To do this, we can use the **AND operation** also known as a **logical conjunction** or a **logical multiplication**. In other words, we can define a new proposition P & Q which is true exactly when both P and Q are true. All possible truth values associated with the and operations are seen in the truth table below.

P	Q	P & Q
0	0	0
0	1	0
1	0	0
1	1	1

We can test individual lines from this truth table using logical scalars. For example, let's test rows 2 and for:

```
%Row 2
P = logical(0); Q = logical(1);
P&Q, and(P,Q)

%Row 4
P = logical(1); Q = logical(1);
P&Q, and(P,Q)
```

Because MATLAB is an array-based language, we can store logical variables in arrays. In doing so, we can confirm every row of the entire truth table in using MATLAB's built-in *logical AND* operator. Below is some code that does this

```
P = logical([0; 0; 1; 1]);
Q = logical([0; 1; 0; 1 ]);
[P, Q, P&Q], [P, Q, and(P, Q)]
```

There are a few interesting features of the code above. First, notice that the `&` operator works on logical vectors (not just logical scalars). For now, we will focus on using the logical operators on arrays having identical sizes. In a later video, we'll study a more general approach focusing on compatible sizes. Second, the `and` function accomplishes the same task as the `&` operator. One line of code relies on operator notation, the other relies on function notation.

Just like we saw when using the ~ operator, we can use the `&` operator on logical matrices. In this video, we focus on combining logical matrices that have identical dimensions. Let's look at the code below:

```
A = zeros(6,6); A(1:2:5, :) = [2*ones(1, 6); 4*ones(1,6); 8*ones(1,6)]
B = zeros(6,6); B(: , 2:2:6) = [16*ones(6, 1), 32*ones(6,1), ...
    64*ones(6,1)]
LA = logical(A); LB = logical(B)
LA & LB, logical(A) & logical(B), and(logical(A), logical(B))
~(A & B), (~A) & B, A & (~B)
```

Propositions versus the components and the form of a proposition

In the work we've done so far, we should distinguish between a specific proposition and the *form* of a proposition. When we say that the form P & Q represents a proposition, this is only true when both P and Q are assigned to be specific propositions with pre-determined truth values. Formally speaking, the statement P & Q *has no truth value on its own* and depends on the assigned values of propositions P and Q.

Because each specific propositional form does not have a single truth value on its own, we instead think about the behavior of the form based on the list of possible values it might achieve. These possible output values depend on the assigned truth values of its individual components. This is exactly the information we capture in our truth tables. Each row of our truth table represents an input-output relation between the truth values assigned to the component part(s) of a propositional form and the corresponding scalar-valued output that form takes with respect to its input(s).

The logical OR operator (aka inclusive-OR operator)

We can combine two propositions together in other interesting ways. Suppose we have two propositions P and Q and want to test if one of the two are true (also known as an inclusive OR operation). To do this, we can use the **OR operation** also known as a **logical disjunction** or a **logical addition**. In other words, we can define a new proposition $P|Q$ which is true exactly when at least one of P or Q is true. All possible truth values associated with the and operations are seen in the truth table below.

P	Q	P | Q
0	0	0
0	1	1
1	0	1
1	1	1

We can test individual lines from this truth table using 1-by-1 logical variables. For example, let's test row 1:

```
P = logical(0);
Q = logical(0);
P|Q, or(P,Q)
```

We can do this more quickly using `logical(0) | logical(0)`. Let's also test row 3 of this table:

```
P = logical(1);
Q = logical(0);
P|Q
or(P,Q)
```

Once agin, we confirm the entire truth table for the (inclusive) logical OR operation using MATLAB's built-in *logical or* operator. Below is some code that does this

```
P = logical([0; 0; 1; 1]);
Q = logical([0; 1; 0; 1 ]);
[P, Q, P|Q]
```

Logical operators and compatible sizes

One unique feature of MATLAB is that we can do logical operations on arrays that have different dimensions. The only requirement is that the arrays must have compatible sizes. We've studied the simplest case of this in which the arrays we analyze have exactly the same size. However, most binary operators and functions in MATLAB support more general operations on arrays that have compatible (but not identical) sizes.

Two inputs have compatible sizes if one of the following is true:

A. Both inputs have identical dimensions (we've seen that in our previous videos)

B. One input is a scalar and the other input is a matrix of any size

```
a = 5; la = logical(a)
x = zeros(8,1);
neg_one = -2*ones(4,1);
x(2:2:8, 1 ) = neg_one.^([0 1 0 1]');
lx = logical(x);
la & lx, ~(la & lx), (~la)& lx
```

C. One input is a matrix and the other input is a column vector with the same number of rows.

D. One input is a matrix and the other input is a row vector with the same number of columns

E. One input is a column vector and the other input is a row vector

The sizes of incompatible inputs cannot be implicitly expanded to be the same. Here are some examples of arrays with incompatible sizes:

A. Both inputs are matrices (not vectors) and at least one of the dimension sizes is not equal

B. Two nonscalar row vectors with different sizes

C. Two nonscalar column vectors with different sizes

These cannot be implicitly expanded to the same size.

The logical XOR operator (aka the exclusive-OR operator)

There is another way to combine propositions together. Suppose we have two propositions P and Q and want to test if either one, but not both, of the propositions is true (also known as an exclusive OR operation). To do this, we can use the **XOR operation**. In other words, we can define a new proposition `xor(P,Q)` which is true when exactly one (but not both) of P or Q is true. All possible truth values associated with the and operations are seen in the truth table below.

P	Q	xor(P, Q)
0	0	0
0	1	1
1	0	1
1	1	0

We can test individual lines from this truth table using 1-by-1 logical variables. For example, let's test row 2:

```
P = logical(0);
Q = logical(1);
xor(P,Q)
```

We can also test row 4 of this table:

```
P = logical(1);
Q = logical(1);
xor(P,Q)
```

To produce the entire truth table for our exclusive-OR operator, we use the following code:

```
P = logical([0; 0; 1; 1]);
Q = logical([0; 1; 0; 1 ]);
[P, Q, xor(P,Q)]
```

Equivalent propositions

We say that two propositions are *equivalent* if and only if they have the same truth table. This definition empowers us to find that two propositions that have different form might actually be equivalent.

P	∼P	∼ (∼P)
1	0	1
0	1	0

We can also see that `~(P & Q)` is equivalent to `( ~ P ) | ( ~Q )` . Let's explore the truth table for each of these variables and confirm this for ourselves.

P	Q	P & Q	~(P & Q)	~P	~Q	(~ P) | (~Q)
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

We can write MATLAB code to confirm the same logical equivalence:

```
1  P = logical([0; 0; 1; 1]);
2  Q = logical([0; 1; 0; 1 ]);
3  [P, Q, P&Q, ~(P&Q), ~P, ~Q, (~P) | (~Q)]
```

Mention operator precedence in these discussions.

Combining logical operators

We can also combine logical operators together to form new logical operators. These include

- NAND operator: NOT AND

NAND operator

P	Q	$\sim$ (P&Q)
0	0	1
0	1	1
1	0	1
1	1	0

- NOR operator: NOT OR

NOR operator

P	Q	$\sim$ (P | Q)
0	0	1
0	1	0
1	0	0
1	1	0

- XNOR operator: NOT XNOR

XNOR operator

P	Q	$\sim$`xor(P,Q)`
0	0	1
0	1	0
1	0	0
1	1	1

These operations for the basics for almost all logical operations implemented in computer science. Below is a truth table that captures all of these operations:

		NOT		AND	OR	XOR	NAND	NOR	XNOR
P	Q	$\sim$ P	$\sim$ Q	P&Q	P | Q	`xor`(P,Q)	$\sim$ (P&Q)	$\sim$ (P | Q)	$\sim$ `xor`(P,Q)
0	0	1	1	0	0	0	1	1	1
0	1	1	0	0	1	1	1	0	0
1	0	0	1	0	1	1	1	0	0
1	1	0	0	1	1	0	0	0	1

We can achieve this truth table using the following code in MATLAB.

```
P = logical([0; 0; 1; 1]);
Q = logical([0; 1; 0; 1 ]);
[P, Q, ~P, ~Q, P&Q, P|Q, xor(P,Q), ~(P&Q), ~(P|Q), ~xor(P,Q)]
```

Relational Operations using Logical Variables

As we discussed previously, logical variables are helpful for comparing elements in two arrays. We can use relational operators to quantitatively compare the values stored in two operands. The table below highlights the different type of relational operations we might use in MATLAB.

Symbol	Description	Equivalent Function
==	Equal to	`eq`
~=	Not equal to	`ne`
<	Strictly less than	`lt`
<=	Less than or equal to	`lt`
>	Strictly greater than	`gt`
>=	Greater than or equal to	`ge`

Just like we saw with logical operators, these relational operators can be used to compare elements in arrays. We're going to analyze the results of these comparisons based on input-output relationships. Specifically, we will see how to compare individual numeric scalar-, vector-, and matrix-valued quantities in MATLAB.

Just like we did with logical operators, we distinguish between four uses of relational operators based on the type of output we generate:

1. Scalar-valued relational operation

   In this use of the relational operator, we compare two scalar-valued quantities to each other. The result is a logical scalar that indicates whether the corresponding comparison is `true` or `false`

2. Vector-valued relational operation

   When we do a vector-valued relational operation, we compare two vector-valued quantities having the same dimension. This results in a logical vector that indicates the result of an element-wise comparison between the two vectors.

3. Matrix-valued relational operation

   These operations compare two matrix-valued inputs with identical dimensions. The result is a logical matrix that stores the element-wise comparison between the two inputs.

4. Compatible (mixed-dimension) relational operation

   This is similar to our video on doing logical operations on compatible arrays.

Equal to and not equal to operators

Let's begin with analyzing the equal to operator by comparing two scalar-valued quantities. Below are example lines of code that run this comparison. Notice the output for each individual line of code is a logical scalar. Moreover, we can store this output into a variable using the logical data class.

```
2 == 3, eq(2, 3)
2 ~= 3, ne(2, 3)

4 == 2^2, eq(4, 2^2)
4 ~= 2^2, ne(4, 2^2)

x = 6; test = (x == 5);
x = 6; test = eq(x, 5);

x = 6; test = (x ~= 5);
x = 6; test = ne(x, 5);

y = 27/9; check = (y == 3);
y = 27/9; check = eq(y, 3);

y = 27/9; check = (y ~= 3);
y = 27/9; check = ne(y, 3);
```

In each case, we have two scalar-valued numerical inputs to our operator (or function). The output is a logical scalar indicating the truth value of this scalar comparison. If the output becomes a logical scalar with the value of logical `1`, the relationship between the two inputs is `true`. If the output gets a logical scalar with value logical `0`, the relationship between the two inputs is `false`.

Let's try an equality check for numerical data stored in nonscalar vectors with identical sizes. The example code below checks element-wise equality between row vector input. Notice in this case that the input vectors are all identical dimensions.

```
0:8 == linspace(0,8,9)
0:8 ~= linspace(0,8,9)

-15:5:5 == linspace(20,0,5)
-15:5:5 ~= linspace(20,0,5)

x = -8:4:20; y = -4:2:10; test = (x == y)
x = -8:4:20; y = -4:2:10; test = eq(x, y)

x = -8:4:20; y = -4:2:10; test = (x ~= y)
x = -8:4:20; y = -4:2:10; test = ne(x, y)
```

For each of these examples, we have two vector-valued numerical inputs with identical dimensions. The output is stored in a logical vector with the same dimensions. Each entry of that vector-valued output is a logical scalar indicating the truth value of the element-wise comparison between the two input values.

We can also test for equality between nonvector matrices with identical dimensions.

```
A = [4, -1; 1 , 4]; B = [-2 , -1; 1 , 4]; A == B
C = [0:4:12 ; 16:4:28; 32:4:44];
twos = 2*ones(1,4); D = [ twos.^(3:-1:0); twos.^(4:-1:1); ...
     twos.^(5:-1:2)];
test = (A == B)
```

Let's try some examples of using relational operators on two inputs with different dimensions, the so called arrays with compatible sizes scenario.

```
x = -8:4:20; y = -4:2:10; x == 8, 8 == y
A = [4, -1; 1 , 4]; B = [-2 , -1; 1 , 4]; 4 == A, B == 4
C = [0:4:12 ; 16:4:28; 32:4:44];
twos = 2*ones(1,4); D = [ twos.^(3:-1:0); twos.^(4:-1:1); ...
    twos.^(5:-1:2)];
D(:,1) == C, D(:, 2) ~= C, D(3,:) == C
```

We can also combine relational operators with logical operators to create more complex logical tasks.

```
x = -8:4:20; y = -4:2:10; x == 8 | 8 == y
A = [4, -1; 1 , 4]; B = [-2 , -1; 1 , 4]; 4 == A  & 4 == B
```

One of the interesting features of the comparison operators is the fact that we are not limited to the case of inputs having identical sizes. In fact, MATLAB only requires that the sizes of the inputs are compatible. There is some good documentation on what this means under the title Compatible Array Sizes for Basic Operations.

Two inputs have compatible sizes if one of the following is true:

A. Both inputs have identical dimensions

B. One input is a scalar and the other input is a matrix of any size

C. One input is a matrix and the other input is a column vector with the same number of rows.

D. One input is a matrix and the other input is a row vector with the same number of columns

E. One input is a column vector and the other input is a row vector

Here are some examples of arrays with incompatible sizes:

- Both inputs are matrices (not vectors) and at least one of the dimension sizes is not equal
- Two nonscalar vectors

Inequality operators

The other relational operators work in a similar fashion by taking entry-by-entry comparison between the two inputs and producing a logical output that stores the results of those comparisons in the address location from each individual entry. Let's take a look at some examples of this code:

```
2 < 3, lt(2, 3)
2 <= 3, le(2, 3)
2 > 3, gt(2, 3)
2 >= 3, ge(2, 3)

4 < 2^2, lt(4, 2^2)
4 <= 2^2, le(4, 2^2)
4 > 2^2, gt(4, 2^2)
4 >= 2^2, gt(4, 2^2)

4 <= 2^2 & 4 >= 2^2, 4 == 2^2

y = 27/9; check = (y == 3);
y = 27/9; check = eq(y, 3);

y = 27/9; check = (y ~= 3);
y = 27/9; check = ne(y, 3);
```

Testing the State of Variables using Logical Variables

1. Video 1: Highlight the `isa` function to determine if the input has specified data type.
2. Video 2: Highlight extended logical operators described in logical operations documentation
   - `all` : determine if all array element are nonzero or true
   - `any` : determine if all array element are nonzero
   - `find` : find indices and values of nonzero elements
   - `islogical` : determine if input is a logical array
   - `true` , `false`, `logical`
   - How to reduce logical arrays into a single value
3. Video 3: Highlight the `is*` documentation to test the state of various MATLAB entities:

Addressing and finding special entries in an array

1. Video 1: How does logical indexing work (i.e. indexing with logical values)
2. Video 2: Find array elements that meet a condition.