

Module 01:

Basic MATLAB Programming

Chul Min Yeum

Assistant Professor

Civil and Environmental Engineering

University of Waterloo, Canada



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING



Module 1: Intended Learning Outcomes

- Write an assignment statement to define a variable and assign a value(s).
- Define a valid variable name acceptable in MATLAB
- Describe data types and a type casting process.
- Explain numeric, logical operator, relational operators and compose expression using these operators
- Evaluate the expression with multiple operators following to operator precedence
- Assess some basic math functions (e.g., `round(x)`, `sign(x)`, or `abs(x)`)
- Generate random number(s) using a built-in random number function

Variables and Assignments

- To store a value, use a **variable**
- A way to put a value in a variable is with an **assignment statement**
- General form:

variable = expression

- The order is important:
 - variable name on the left
 - the assignment operator “=”
 - expression on the right

⚠: Here, = is not meant to an equality.

⚠: Make sure that the variable name is always on the left.

```
% variable = expression  
a1 = 3  
a2 = 5
```

Name	Value
a1	3
a2	5

Variable Names

- Names must begin with a letter of the alphabet.
- After that names can only contain letters, digits, and **the underscore character**(_).
- Variable names must not have a space.
- You cannot use other characters except for ' _'.
- MATLAB is **case-sensitive**.
- Names should be **mnemonic**: You and others know what are stored in your variable through its name.
- **clear** clears out variables and also functions.

Example: Variable Names

```
8val = 10; % error: must begin with a letter of the alphabet  
_col = 0 ; % error: must begin with a letter of the alphabet  
row_3_ = 1; % no error  
row@3 = 10; % error: cannot contain characters other than underscore  
col-03 = 10; % error: cannot contain characters other than underscore  
% Following scripts have no error but the names should be mnemonic  
asdf1 = 100; % no error  
love = 10; % no error  
aaaa3 = 10; % no error
```

⚠: Recommend mnemonic variable names

```
% define a variable of 'gal'  
gal = 100
```

Q. What value is in 'Ga1' ?

Example: Question in S19 Midterm



Q. Which of the following scripts have errors?

(1)	<code>Val1 = 10</code>
(2)	<code>4val = 5</code>
(3)	<code>new@data = 8</code>
(4)	<code>val*2 = 3</code>
(5)	<code>Jason = 10+2</code>

1. (1), (2), and (4)
2. (2), (3), and (5)
3. (2), (3), and (4)
4. (1), (2), and (5)

Constants

- In programming, variables are used for values that **could change**, or **are not known in advance**
- **Constants** are used when the value is known, is **pre-defined** and not updated in the program.
- Examples in MATLAB (these are actually functions that return constant values)
 - **pi** 3.14159....
 - **i, j** imaginary number
 - **inf** infinity
 - **NaN** stands for “not a number”; e.g. the result of 0/0

⚠: You can overwrite values to the constants but I do not recommend the use of constants as variables. For example: `pi = 3 % no error`

Modifying Variables

```
myvar = 10; % initialize a variable  
myvar = myvar + 3; % increment by 3
```

Name	Value
myvar	13

```
myvar = 10; % initialize a variable  
val = 3; % initialize a variable
```

```
% identical operations  
myvar1 = myvar + val;  
myvar2 = 10 + 3;
```

Name	Value
myvar	10
val	3
myvar1	13
myvar2	13

Swap two values

```
myvar1 = 10; % initialize a var.  
myvar2 = 5; % initialize a var.
```

```
tmp = myvar1;  
myvar1 = myvar2;  
myvar2 = tmp;
```

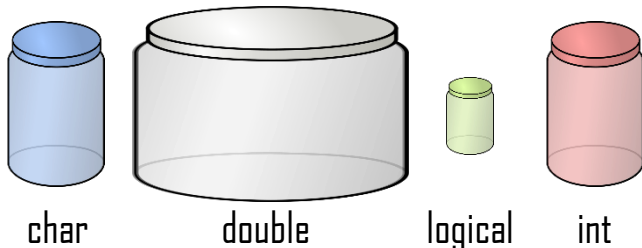
Name	Value
myvar1	5
myvar2	10
tmp	10

Data Types

- Every expression and variable has an associated *type* or *class*
 - Real numbers: *single*, *double* (**default type for numbers**)
 - Integer types: numbers in the names are the number of bits used to store a value of that type
 - Signed integers: *int8*, *int16*, *int32*, *int64*
 - Unsigned integers: *uint8*, *uint16*, *uint32*, *uint64*
 - Single characters and character vectors: *char*
 - Strings of characters: *string*
 - **True/false: *logical* (represented as 1 or 0)**

😊: The same value can be defined as a different type.

Container



Examples

- 2.145, 0.15893, 3.0, 2.45
- 10, 11, 24, 30, 400
- 'a', 'b', 'A', 'c'
- "sam"
- true, false

Data Types (Continue)

Example

- double : 2.145, 0.15893, 3.0,
- integer type: 10, 11, 24, 30
- char: 'a', 'b', 'A', 'c'
- string: "sam"
- logical: true, false

class is a function of determining a class of object.

```
>> class("sam")  
  
ans =  
    'string'
```

```
>> class('a')  
  
ans =  
    'char'  
  
>> class(true)  
  
ans =  
    'logical'  
  
>> class(2.145)  
  
ans =  
    'double'  
  
>> class(10)  
  
ans =  
    'double'
```


⚠: The default type of numbers is 'double'. Integers like 10, 11, .. can be defined as integer types. It does not mean that integer numbers always become integer types.

Type Cast

There are many functions that convert values from one type to another. The names of these functions are the same as the names of the types. The names can be used as functions to convert a value to that type. This is called casting the value to a different type or type casting.

```
>> a = logical(1);  
>> class(a)  
  
ans =  
  
    'logical'  
>> b = double(a);  
>> class(b)  
  
ans =  
  
    'double'
```

```
>> a = logical(1);  
>> b = a + 2;  
>> class(b)  
  
ans =  
  
    'double'  
>> c = logical(b)  
c =  
  
    logical  
  
    1
```

 In the second line in red, the type of `a` is converted to double and arithmetic operation is conducted with 2. In the fourth line, `b` is double but the logical type only carries 0 or 1 so non-zero double (or other numeric type) values becomes 1, otherwise 0. This is a very important concept for logical operations.

Operators

- There are in general two kinds of operators: *unary* operators, which operate on a single value and *binary* operators, which operate on two values.
- Operators include:
 - + addition
 - subtraction or negation
 - * multiplication
 - / division
 - ^ exponentiation (e.g. 5^2 is 25)

```
val1 = 5 + 1;  
val2 = 10*2;  
val3 = 10^2;  
val4 = 100/5;  
val5 = 10 + 2 + 3;
```

Name	Value
val1	6
val2	20
val3	100
val4	20
val5	15

Operator Precedence Rules

- Some operators have precedence over others.
- Within a given precedence, the expressions are evaluated from **left to right**.
- Precedence list (highest to lowest) :
 - () parentheses
 - ^ exponentiation
 - negation
 - *, / , \ all multiplication and division
 - +, - addition and subtraction
- Nested parentheses: expressions in inner parentheses are evaluated first

```
val1 = (5 + 1)*2;  
val2 = 10^2*3;  
val3 = 10^(2+3);  
val4 = 4-3*2;  
val5 = 3*((4+3)*2);  
val6 = -10^2;
```

☺: A good practice is to use parentheses to clarify your operation: $(-10)^2$ or $(-10)^2$.

Name	Value
val1	12
val2	300
val3	100000
val4	-2
val5	42
val6	-100

Example: Operator Precedence Rules



Q. What values are assigned to the variables?

```
val1 = 3*2+3-2*3  
val2 = -10^4*3  
val3 = -(10^4*3)  
val4 = 5-3*2;  
val5 = (3+3)^(2-1);
```

Name	Value
val1	3
val2	-30000
val3	-30000
val4	-1
val5	6

Relational Operator

- The relational operators in MATLAB are:

>	greater than
<	less than
>=	greater than or equals
<=	less than or equals
==	equality
~=	inequality




⚠: Here, == is meant to an equality. Not =.

- It is also called *Boolean* expressions or *logical* expressions.
- The resulting type is logical 1 for true or 0 for false**


📖: “true” is represented by the logical value 1, and “false” is represented by the logical value 0. A logical type only contain two value, 0 or 1.

Relational Operator (Continue)

```
% relation operator  
ro1 = 3 < 4;  
ro2 = 3 > 5;  
ro3 = 3 == 5;  
ro4 = 3 ~= 7;  
ro5 = 3 <= 3;  
ro6 = 3 >= 3;  
ro7 = 3 > 3;
```

: In the second line, the code is first running the expression ($3 < 4$) and computing its value. This expression means “3 less than 4”, which is a true. Thus, `true` (or logical 1) is assigned to `ro1`.

Name	Value
ro1	1
ro2	0
ro3	0
ro4	1
ro5	1
ro6	1
ro7	0

: Here, the values are all *logical* values, 0 or 1, not *double*.

Logical Operator

- The logical **operators** are:


|| or for scalars
&& and for scalars
~ not

- Note that the logical operators are **commutative**
 - (e.g., `x || y` is equivalent to `y || x`)
- The resulting type is **logical 1** for true or 0 for false

x	y
true	true
true	false
false	false



~x	x y	x && y
false	true	true
false	true	false
true	false	false

: For example, logical (1) && logical (1) becomes 1. This is the same with true && true.

Logical Operator (Continue)

```
% relation operator  
lo1 = true && false  
lo2 = 1 || 0  
lo3 = 0 || 0  
lo4 = true && true  
lo5 = 0 || ~false  
lo6 = ~false && true  
lo7 = ~true || false
```

Name	Value
lo1	0
lo2	1
lo3	0
lo4	1
lo5	1
lo6	1
lo7	0

x	y
true	true
true	false
false	false




~x	x y	x && y
false	true	true
false	true	false
true	false	false

General Operator Precedence

Challenging

Parentheses () Highest
Power: ^
Unary: negation (-), not (~)
Multiplication or division: *, /, \
Addition, subtraction: +, -
Relational: <, <=, >, ==, ~=
And: &&
Or: || Lowest



☺: I know it's bit hard to wrap your head but, you consider relational and logical operators as common operators like +, - and compute the values with this precedence order.

📖: **Remember ! 0 is false otherwise true.**

```
val1 = 3 < (1 + 3)
val2 = (3 < 1) + 3
val3 = 3 < 1 + 3
```

📖: The type of val2 is double and the types of val1 and val3 are logical.

Name	Value
val1	1
val2	3
val3	1

Example: Operator Precedence



```
lg1 = (3 < 4) < 4;  
lg2 = 3 < (4 < 5);  
lg3 = (3 > 5) + 3;  
lg4 = (10 > 4) && (4 > 1);  
lg5 = (10 < 4) && (4 < 1);  
lg6 = ~((10 < 4) && (4 < 1));  
lg7 = 2 < 3 + 4;
```

Parentheses ()

Power: ^

Unary: negation (-), not (~)

Multiplication or division: *, /, \

Addition, subtraction: +, -

Relational: <, <=, >, ==, ~=

And: &&

Or: ||

Highest



Lowest

Q. What values are assigned to the variables?

Name	Value
lg1	1
lg2	0
lg3	3
lg4	1
lg5	0
lg6	1
lg7	1

😊: Regardless of the operator precedence, you could use parentheses to clarify the operation order

Example: Operator Precedence



Q. How to write a code to check if x lies in between 5 and 10. If yes, 1 and otherwise 0.

```
x1 = 6;  
x2 = 11;  
  
lg1 = (5 < x1) && (x1 < 10)  
lg2 = 5 < x1 < 10; % incorrect!  
lg3 = (5 < x1) < 10; % incorrect!  
  
lg4 = (5 < x2) && (x2 < 10)  
lg5 = 5 < x2 < 10; % incorrect!  
lg6 = (5 < x2) < 10; % incorrect!
```

Name	Value
x1	6
x2	11
lg1	1
lg2	1
lg3	1
lg4	0
lg5	1
lg6	1

⚠: For `lg2`, the first expression `5 < x` will be evaluated. It gives a logical value 1. Then, the rest of the expression will be evaluated, `1 < 10`. So, the final value to be assigned to `lg2` become logical 1, true.

Using Functions: Terminology

- To use a function, you **call** it
- To call a function, give its name followed by the **argument(s)** that are **passed** to it in parentheses




out = fun (arg1, arg2, ...)

- Many functions calculate values and **return** the results
- For example, to find the absolute value of -4

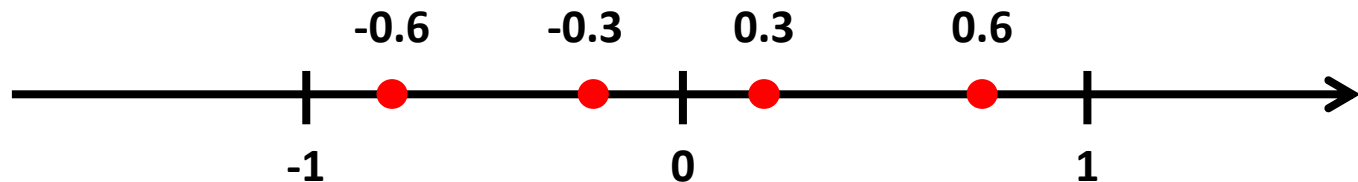
```
>> abs(-4)
ans =
     4
```

- The name of the function is “abs”
- One argument, -4 , is passed to the `abs` function
- The `abs` function finds the absolute value of -4 and returns the result, 4 .

Rounding Functions

Function	Description	Note
<code>round(x)</code>	Rounds x to the nearest integer	
<code>fix(x)</code>	Truncates x to the nearest integer toward zero.	
<code>floor(x)</code>	Rounds x to the nearest integer toward negative infinity.	
<code>ceil(x)</code>	Rounds x to the nearest integer toward positive infinity.	

Rounding Functions (Continue)



Function	Description
<code>round(x)</code>	Rounds x to the nearest integer
<code>fix(x)</code>	Truncates x to the nearest integer toward zero.
<code>floor(x)</code>	Rounds x to the nearest integer toward negative infinity.
<code>ceil(x)</code>	Rounds x to the nearest integer toward positive infinity.

	round	ceil	fix	floor
0.3	0	1	0	0
0.6	1	1	0	0
-0.3	0	0	0	-1
-0.6	-1	0	0	-1

Rounding Functions (Continue)

```
x1 = 10.3;  
x2 = 12.7;  
x3 = -1.3;
```

```
x1_ce = ceil(x1);  
x1_fi = fix(x1);  
x1_fl = floor(x1);
```

```
x2_ce = ceil(x2);  
x2_fi = fix(x2);  
x2_fl = floor(x2);
```

```
x3_ce = ceil(x3);  
x3_fi = fix(x3);  
x3_fl = floor(x3);  
x3_ro = round(x3);
```

Name	Value
x1	10.3
x2	12.7
x3	-1.3
x1_ce	11
x1_fi	10
x1_fl	10
x2_ce	13
x2_fi	12
x2_fl	12
x3_ce	-1
x3_fi	-1
x3_fl	-2
x3_ro	-1

Function	Description
round(x)	Rounds x to the nearest integer
fix(x)	Truncates x to the nearest integer toward zero.
floor(x)	Rounds x to the nearest integer toward negative infinity.
ceil(x)	Rounds x to the nearest integer toward positive infinity.

😊: You do not have to memorize the functions and their usage. You can simply search for their usage in google or type `doc round` in command window (`doc fun_name`).

Common Math Functions

Function	Description	Script	Value
abs (x)	Finds the absolute value of x	abs (-3) abs (2)	3 2
sqrt (x)	Finds the square root of x	sqrt (4) sqrt (1.75)	2 1.5
sign (x)	Return -1 if x is less than zero, a value of 0 if x equals zero, and a value of 1 if x is greater than zero.	sign (-5) sign (3) sign (0)	-1 1 0
rem (x,y)	Computes the remainder of x/y	rem (25,4) rem (4,2) rem (9,5)	1 0 4

Common Math Functions (Continue)

Function	Description
abs (x)	Finds the absolute value of x
sqrt (x)	Finds the square root of x
sign (x)	Return -1 if x is less than zero, a value of 0 if x equals zero, and a value of 1 if x is greater than zero.
rem (x,y)	Computes the remainder of x/y

```
x = -4;  
y = 9;  
z = 2;  
  
x_ab = abs(x)  
x_si = sign(x)  
  
xz_r = rem(x, z)  
  
y_ab = abs(y)  
y_sq = sqrt(y)  
y_si = sign(y)  
  
yz_r = rem(y, z)
```

Name	Value
x	-4
y	9
z	2
x_ab	4
x_si	-1
xz_r	0
y_ab	9
y_sq	3
y_si	1
yz_r	1

Logical Operation Functions

Function	Operator
and(A, B)	A && B
or(A, B)	A B
not(A)	~A

☺: These functions improve readability.

For instance:

```
log1 = 3<x && x<10
```

```
log2 = and(3<x, x<10)
```

```
x1 = true;  
x2 = false;  
  
lg1a = and(x1, x2);  
lg1b = x1 && x2;  
  
lg2a = or(x1, x2);  
lg2b = x1 || x2  
  
lg3 = not(x1);
```

Name	Value
x1	1
x2	0
lg1a	0
lg1b	0
lg2a	1
lg2b	1
lg3	0

- MATLAB generates pseudorandom numbers. These numbers are not strictly random and independent in the mathematical sense, but they pass various statistical tests of randomness and independence, and their calculation can be repeated for testing or diagnostic purposes.
- Several built-in functions generate random numbers.
- Random number generators start with a number called the `seed`. This is either a predetermined value or **from the clock**. This is considered as an “index” of a random number lookup table.

Seed: n
Seed: 2
Seed: 1

Seed: i

Seed: i

4633	83386	7488	15481	6048	4471	3215	5584	4775	4042	1083	1363
31361	1771	6775	2714	9278	1028	6228	3918	3545	3945	4023	8138
4634	83390	7489	1549	6050	4472	3216	5585	4776	4043	1084	1364
7488	2838	2916	7037	7072	7014	7061	7067	6013	8238	2427	8139
4635	83391	7490	1550	6051	4473	3217	5586	4777	4044	1085	1365
8519	1838	6781	1929	1008	1079	1094	1051	1206	1758	1904	1515
2842	3843	2847	7037	7072	7014	7061	7067	6013	8239	2428	8140
4636	83392	7491	1551	6052	4474	3218	5587	4778	4045	1086	1366
3547	8341	6426	7348	6048	4471	3215	5584	4775	4042	1083	1363
4637	83393	7492	1552	6053	4475	3219	5588	4779	4046	1087	1367
1670	4048	2846	2905	4427	4601	4427	3216	5585	4776	4043	1084
1471	4049	2847	2906	4428	4602	4428	3217	5586	4777	4044	1085
4638	83394	7493	1553	6054	4476	3220	5589	4780	4047	1088	1368
7039	8044	1548	6048	4471	3215	5584	4775	4042	1083	1363	1363
8719	3844	2847	7037	7072	7014	7061	7067	6013	8240	2429	8141
4639	83395	7494	1554	6055	4477	3221	5590	4781	4048	1089	1369
4640	83396	7495	1555	6056	4478	3222	5591	4782	4049	1090	1370
4641	83397	7496	1556	6057	4479	3223	5592	4783	4050	1091	1371
4642	83398	7497	1557	6058	4480	3224	5593	4784	4051	1092	1372
4643	83399	7498	1558	6059	4481	3225	5594	4785	4052	1093	1373
4644	83400	7499	1559	6060	4482	3226	5595	4786	4053	1094	1374
4645	83401	7500	1560	6061	4483	3227	5596	4787	4054	1095	1375
4646	83402	7501	1561	6062	4484	3228	5597	4788	4055	1096	1376
4647	83403	7502	1562	6063	4485	3229	5598	4789	4056	1097	1377
4648	83404	7503	1563	6064	4486	3230	5599	4790	4057	1098	1378
4649	83405	7504	1564	6065	4487	3231	5600	4791	4058	1099	1379
4650	83406	7505	1565	6066	4488	3232	5601	4792	4059	1100	1380
4651	83407	7506	1566	6067	4489	3233	5602	4793	4060	1101	1381
4652	83408	7507	1567	6068	4490	3234	5603	4794	4061	1102	1382
4653	83409	7508	1568	6069	4491	3235	5604	4795	4062	1103	1383
4654	83410	7509	1569	6070	4492	3236	5605	4796	4063	1104	1384
4655	83411	7510	1570	6071	4493	3237	5606	4797	4064	1105	1385
4656	83412	7511	1571	6072	4494	3238	5607	4798	4065	1106	1386
4657	83413	7512	1572	6073	4495	3239	5608	4799	4066	1107	1387
4658	83414	7513	1573	6074	4496	3240	5609	4800	4067	1108	1388
4659	83415	7514	1574	6075	4497	3241	561				

Random Number (Continue)

Challenging

- **rand(n)** creates an $n \times n$ matrix of random reals
- **rand(n,m)** create an $n \times m$ matrix of random reals
- **randi([range],n,m)** creates an $n \times m$ matrix of random integers in the specified range
- **rng(seed)** specifies the seed for the random generator

```
>> rand(1)
```

```
ans =
```

```
0.8147
```

```
>> rand(1)
```

```
ans =
```

```
0.9058
```

📖: A seed is changed in each run of your script. Thus, different random numbers are generated

```
>> rng(10); rand(1)
```

```
ans =
```

```
0.7713
```

```
>> rng(10); rand(1)
```

```
ans =
```

```
0.7713
```

📖: You can specify a seed. Then, the same numbers are generated.

Write a Numerical Expression

Suppose that you want to compute y when $x = 10$

$$y = \frac{\{x^2(100x + 10) + x^3(20x^2 + 3)\}}{-x^{-3} + 1}$$

```
x = 10;  
y1 = (x^2*(100*x + 10) + x^3*(20*x^2 + 3))/(-x^(-3)+1)
```

☺: The more terms and parentheses in your equation, the larger the probability that you may be making a mistake. In this case, I usually do it like this

```
x = 10;  
y_nom = (x^2)*(100*x + 10) + (x^3)*(20*(x^2) + 3);  
y_den = -x^(-3)+1;  
y = y_nom/y_den;
```

Arithmetic Operation in MATLAB

Optional

Suppose that you are solving a problem: If the car has a mass of 300kg and you push the car with an acceleration of 5 inch/s², compute the force that is generated from the car in newton

```
% MATLAB as a calculator  
300*5*0.0254
```

Name	Value
ans	38.1

☺: If you are using MATLAB as a programming tool, I would recommend writing the code below. This improves code readability and allows others to understand your code.

```
% MATLAB as programming tool  
  
inch2m = 0.0254; % inch to m  
mass = 300; % kg  
accel = 5; % inch/s/s  
  
% force = mass(kg) * acceleration (m/s/s)  
force = mass * accel * inch2m;
```

Name	Value
inch2m	0.0254
mass	300
accel	5
force	38.1