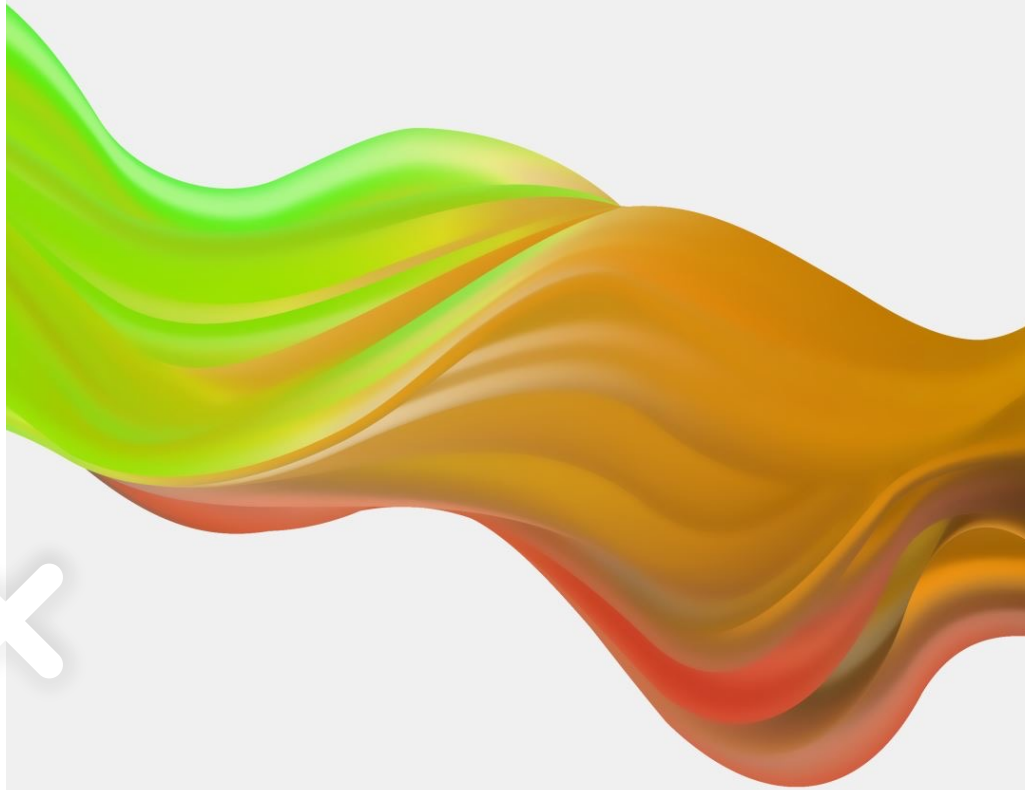
Computer Programming

MENG2020



REVIEW

Matrix Indexing

- The index argument can be a matrix. In this case, each element is looked up individually, and returned as a matrix of the same size as the index matrix.

```
>> a = [10   20   30   40   50   60];
>> b = a([1 2 3 ; 4 5 6])
   b = 10   20  30
       40   50  60
```

This is a matrix of indexes. Get it?

Working Whole Rows or Columns

- The colon operator can select entire rows or columns of a matrix.

```
>> c = b (1,:)
```

```
b = 10   20   30
    40   50   60
```

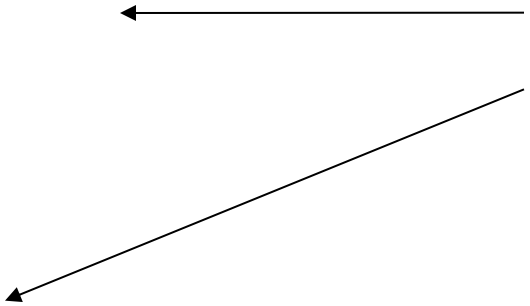
Working Whole Rows or Columns

- The colon operator can select entire rows or columns of a matrix.

```
>> c = b (1,:)
```

```
c = 10  20  30
```

```
b = 10   20  30
    40   50  60
```

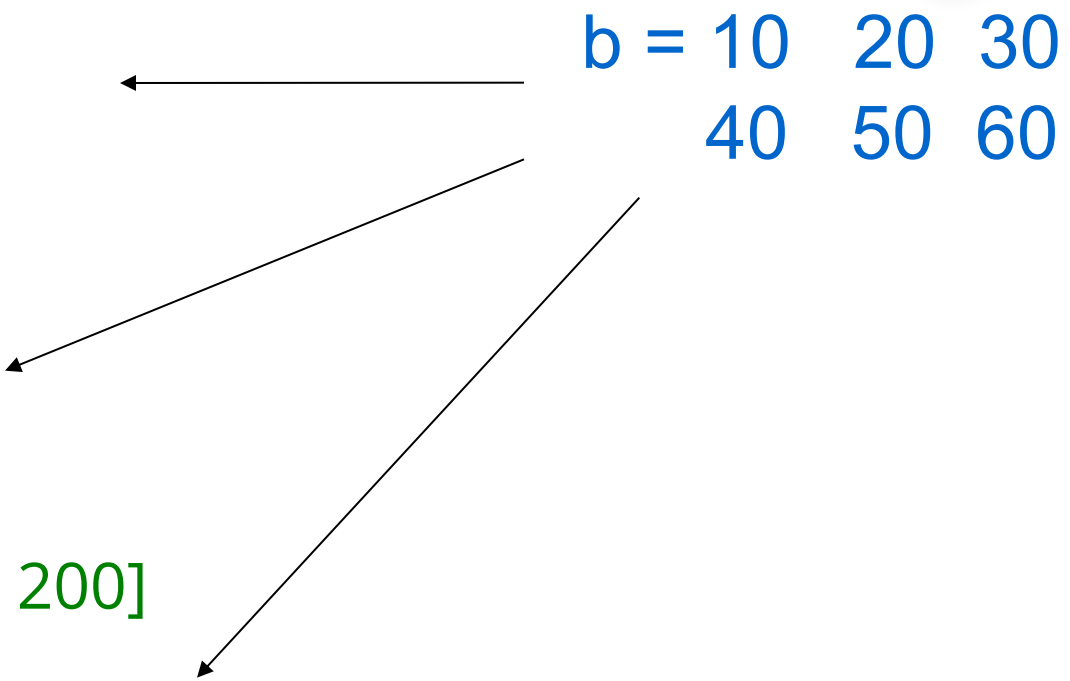


```
>> d = b (:,2)
```

Working Whole Rows or Columns

- The colon operator can select entire rows or columns of a matrix.

```
>> c = b (1,:)
   c = 10  20  30

>> d = b (:,2)
   d = 20
       50

>> b (:, 3) = [100   200]
```

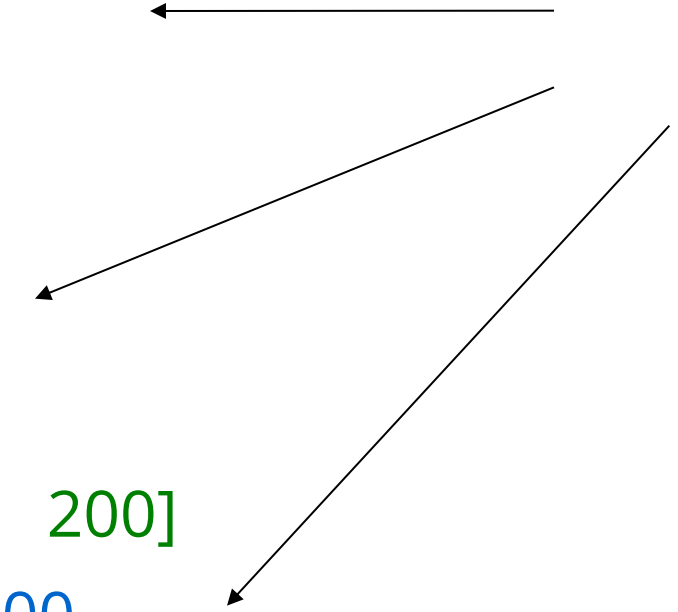


Working Whole Rows or Columns

- The colon operator can select entire rows or columns of a matrix.

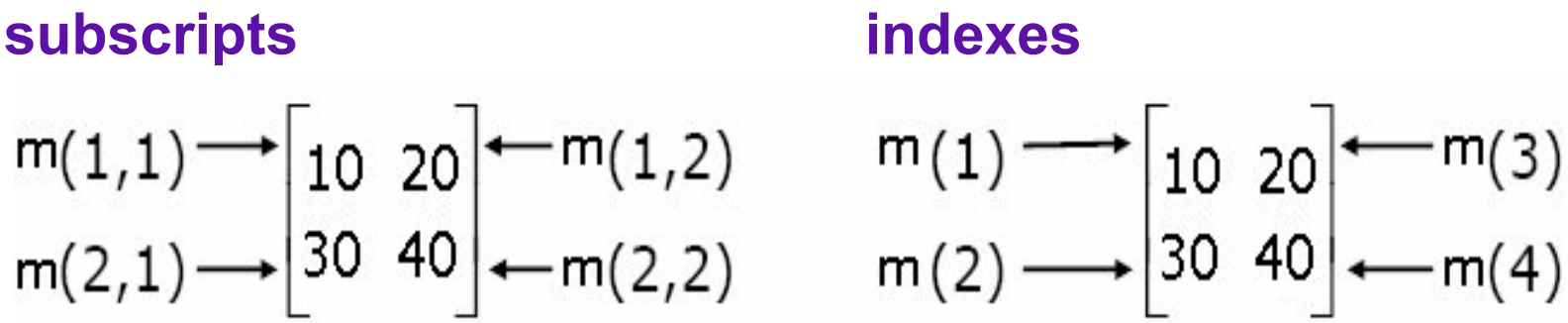
```
>> c = b (1,:)
   c = 10  20  30

>> d = b (:,2)
   d = 20
       50

>> b (:, 3) = [100   200]
   b = 10   20   100
       40   50   200
```

```
b = 10   20  30
    40   50  60
```



Index vs. Subscript

- You remember that indexes are a way to access elements. Remember that indexes start at 1 in MATLAB. Now let's do that with matrices. For a matrix you can access an element using an index or a pair of subscripts.
- For matrices, subscripts have the row and column numbers separated by a comma. Indexes, on the other hand, indicate the linear position from the start of the matrix. See the figure below for the difference.




NEWS!

About our lecture

Colon Operator

The colon operator **:** lets you address a range of elements:

- **Vector** (row or column)
  - va(:) - all the elements of the vector
  - va(m:n) – only elements in positions m through n inclusive
- **Matrix**
  - A(:,n) - all the rows of column n
  - A(m,:) - all the columns of row m
  - A(:,m:n) - all the rows of columns m through n
  - A(m:n,:) - all the columns of rows m through n
  - A(m:n,p:q) - columns p through q of rows m through n

Getting Specific Columns

- You can replace vector index or matrix indices by vectors in order to pick out specific elements. For example, for vector `v` and matrix `m`

- `v([a b c:d])` returns elements `a`, `b`, and `c` through `d`

- `m([a b],[c:d e])` returns columns `c` through `d` and column `e` of rows `a` and `b`

Colon Operations on Vectors

```
>> v = 4:3:34
```

Colon Operations on Vectors

```
>> v = 4:3:34

v = 4  7  10  13  16  19  22  25  28  31  34


>> u = v([3, 5, 7:10])
```

Colon Operations on Vectors

```
>> v = 4:3:34

v = 4  7  10  13  16  19  22  25  28  31  34


>> u = v([3, 5, 7:10])

u = 10  16  22  25  28  31


>> u = v([3 5 7:10])
```

Colon Operations on Vectors

```
>> v = 4:3:34

V = 4  7  10  13  16  19  22  25  28  31  34


>> u = v([3, 5, 7:10])

u = 10  16  22  25  28  31


>> u = v([3 5 7:10])

u = 10  16  22  25  28  31
```

Colon Operations on Matrices

```
>> A = [10:-1:4; ones(1,7); 2:2:14; zeros(1,7)]
```

Colon Operations on Matrices

```
>> A = [10:-1:4; ones(1,7); 2:2:14; zeros(1,7)]

A = 10     9     8     7     6     5     4
     1     1     1     1     1     1     1
     2     4     6     8    10    12    14
     0     0     0     0     0     0     0


>> B = A([1 3],[1 3 5:7])
```

Colon Operations on Matrices

```
>> A = [10:-1:4; ones(1,7); 2:2:14; zeros(1,7)]

A = 10     9     8     7     6     5     4
     1     1     1     1     1     1     1
     2     4     6     8    10    12    14
     0     0     0     0     0     0     0


>> B = A([1 3],[1 3 5:7])

B = 10     8     6     5     4
     2     6    10    12    14
```

•Add elements to existing variables

Adding values to ends of variables:

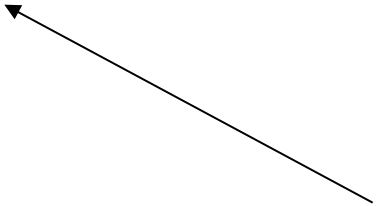
- Adding to ends of variables is called *appending* or *concatenating. Use the keyword* ***end*** *as index to get to the end.*
- The *end* of a vector is the right side of a row vector or bottom of a column vector.
- The *end* of a matrix is the right column of the bottom row.
- That is, MATLAB expands arrays to include indices, puts the specified values in the assigned elements, fills any unassigned new elements with zeros. See next slide.

Assigning to Undefined Indices of Vectors

```
>> v1 = 1:4

v1 = 1  2  3  4

>> v1(5:10) = 10:5:35
```

Assigning to Undefined Indices of Vectors

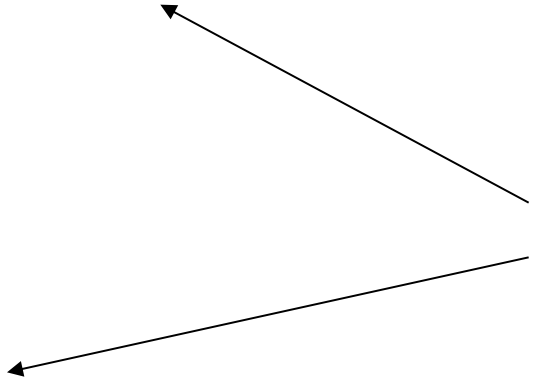
```
>> v1 = 1:4

v1 = 1  2  3  4

>> v1(5:10) = 10:5:35

v1 = 1  2  3  4  10  15  20  25  30  35

>> v2 = [5 7 2]

v2 = 5  7  2

>> v2(8) = 4
```



New elements added

Assigning to Undefined Indices of Vectors

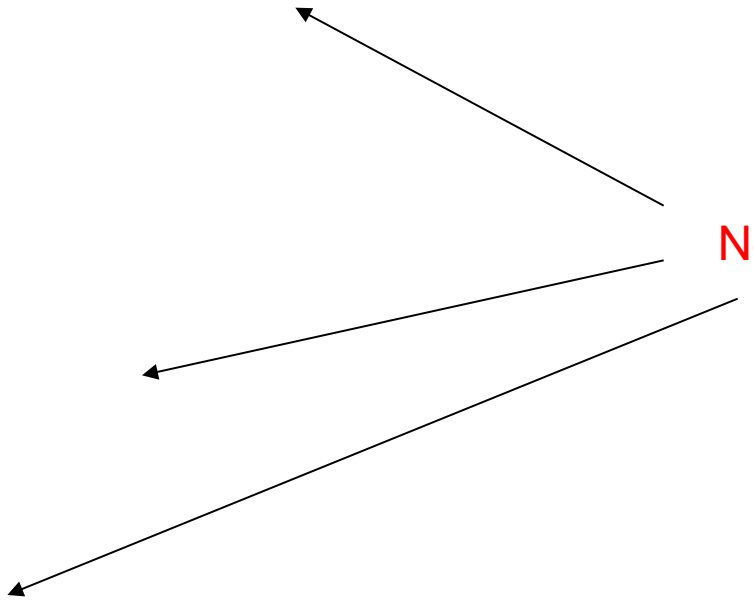
```
>> v1 = 1:4

v1 = 1  2  3  4

>> v1(5:10) = 10:5:35

v1 = 1  2  3  4  10  15  20  25  30  35

>> v2 = [5 7 2]

v2 = 5  7  2

>> v2(8) = 4

v2 = 5  7  2  0  0  0  0  4

>> v3(5) = 24
```



New elements added

Assigning to Undefined Indices of Vectors

```
>> v1 = 1:4

v1 = 1  2  3  4

>> v1(5:10) = 10:5:35

v1 = 1  2  3  4  10  15  20  25  30  35

>> v2 = [5 7 2]

v2 = 5  7  2

>> v2(8) = 4

v2 = 5  7  2  0  0  0  0  4

>> v3(5) = 24

v3 = 0  0  0  0  24
```



New elements added

Unassigned elements are initialized at zero

Deleting Elements in a Vector or a Matrix

To delete elements in a vector or a matrix, set the index, subscript, or range to be deleted to empty brackets.

```
>> kt=[2 8 40 65 3 55 23 15 75 80]

kt = 2 8 40 65 3 55 23 15 75 80

>> kt(6)=[]
```

Deleting Elements in a Vector or a Matrix

To delete elements in a vector or a matrix, set the index, subscript, or range to be deleted to empty brackets.

```
>> kt=[2 8 40 65 3 55 23 15 75 80]

kt = 2 8 40 65 3 55 23 15 75 80

>> kt(6)=[]

kt = 2 8 40 65 3 23 15 75 80

>> kt(3:6)=[]
```

Remove the 6th element of **kt**

55 is gone!

Deleting Elements in a Vector or a Matrix

To delete elements in a vector or a matrix, set the index, subscript, or range to be deleted to empty brackets.

```
>> kt=[2 8 40 65 3 55 23 15 75 80]

kt = 2 8 40 65 3 55 23 15 75 80

>> kt(6)=[]

kt = 2 8 40 65 3 23 15 75 80

>> kt(3:6)=[]

kt = 2 8 15 75 80
```

Remove the 6th element of **kt**

55 is gone!

Remove elements 3 through 6 of the current **kt**, not the original **kt**

Another Removing Example

```
>> mtr = [5 78 4 24 9; 4 0 36 60 12; 56 13 5
89 3]

mtr = 5    78     4    24     9

      4     0    36    60    12

     56    13     5    89     3

>> mtr(:,2:4)=[]
```

Another Removing Example

```
>> mtr = [5 78 4 24 9; 4 0 36 60 12; 56 13 5
89 3]

mtr = 5    78     4    24     9

      4     0    36    60    12

     56    13     5    89     3

>> mtr(:,2:4)=[]

mtr = 5     9
```


Delete all the rows
for columns 2 to 4

```
      4    12

     56     3
```

Finding Minimum and Maximum Values

```
>> vec = [10  7  8  13  11  29];
```

- To get the minimum value and its index:

```
>> [minVal,minInd] = min(vec)
minVal = 7  minInd = 2
```

- To get the maximum value and its index:

```
>> [maxVal,maxInd] = max(vec)
maxVal = 29  maxInd = 6
>> [a,b] = max(vec)
a = 29  b = 6
```

You can use any variable name you want

Only one variable you get the value only, not the position.

```
x = min (vec)
x = 7
y = max (vec)
y = 29
```

Finding Minimum and Maximum Values

```
>> M = [10 20 ; 30 40];
```

- With entire matrices it works column by column. The index is the row.

```
>> [minVal,minInd] = min(M)
minVal = 10 20
minInd =   1   1
>> [maxVal,maxInd] = max(M)
maxVal = 30 40
maxInd =   2   2
```

Only one variable you get the values only, not positions.

```
a = min (M)
a = 10  20
b = max (M)
b = 30  40
```

Matrix Built-In Functions

MATLAB has many built-in functions for working with arrays. Some common ones are:

- `length(v)` : gives the number of elements in a vector.
- `size(A)` : gives the number of rows and columns in a matrix or a vector.
- `reshape(A,m,n)` : changes the number of rows and columns of a matrix or vector while keeping the total number of elements the same. For example, you can change a 4x4 matrix into a 2x8 matrix.

reshape Example

```
>> a = [1 2 ; 3 4]
a =
 1 2
 3 4

>> a = [a , [5 6 ; 7 8]]
```

reshape Example

```
>> a = [1 2 ; 3 4]
a =
 1 2
 3 4

>> a = [a , [5 6 ; 7 8]]
a =
 1 2 5 6
 3 4 7 8

>> reshape (a, 1, 8)
```

reshape Example

```
>> a = [1 2 ; 3 4]
a =
 1 2
 3 4

>> a = [a , [5 6 ; 7 8]]
a =
 1 2 5 6
 3 4 7 8

>> reshape (a, 1, 8)
ans =
 1 3 2 4 5 7 6 8
```

```
>> reshape (a, 8, 1)
```

reshape Example

```
>> a = [1 2 ; 3 4]
a =
 1 2
 3 4

>> a = [a , [5 6 ; 7 8]]
a =
 1 2 5 6
 3 4 7 8

>> reshape (a, 1, 8)
ans =
 1 3 2 4 5 7 6 8
```

```
>> reshape (a, 8, 1)

ans =
 1
 3
 2
 4
 5
 7
 6
 8
```

The diag command

- `diag(v)`

The diag command

- `diag(v)` : makes a square matrix of zeroes with the specified vector in the main diagonal.

```
v1 = [33 66 99]; m1 = diag (v1)
```

The diag command

- `diag(v)` : makes a square matrix of zeroes with the specified vector in the main diagonal.

```
v1 = [33 66 99]; m1 = diag (v1)
 m1 = 33   0   0
       0  66   0
       0   0  99
```

The diag command

- `diag(v)` : makes a square matrix of zeroes with the specified vector in the main diagonal.

```
v1 = [33 66 99]; m1 = diag (v1)
 m1 = 33   0   0
       0  66   0
       0   0  99
```

- `diag(A)?`

The diag command

- `diag(v)` : makes a square matrix of zeroes with the specified vector in the main diagonal.

```
v1 = [33 66 99]; m1 = diag (v1)
 m1 = 33   0   0
       0  66   0
       0   0  99
```

- `diag(A)` : creates a vector equal to the main diagonal of the specified matrix.

```
m2 = [42 89 ; 78 31]; v2 = diag (m2)
```

The diag command

- `diag(v)` : makes a square matrix of zeroes with the specified vector in the main diagonal.

```
v1 = [33 66 99]; m1 = diag (v1)
 m1 = 33   0   0
       0  66   0
       0   0  99
```

- `diag(A)` : creates a vector equal to the main diagonal of the specified matrix.

```
m2 = [42 89 ; 78 31]; v2 = diag (m2)
 v2 = 42 31
```

More Built-In Functions for Arrays

- `mean(v)`: calculates the mean (average) of the specified vector.

```
v3 = [45 13 65 62]; average = mean (v3)
average = 46.2500
```

`sum(v)`: gives the sum (total) of all the elements of the specified vector.

```
v3 = [45 13 65 62]; total = sum (v3)
total = 185
```

`sort(v)`: sorts the elements in the specified vector in ascending order.

```
v3 = [45 13 65 62]; v4 = sort (v3)
v4 = 13  45  62  65
```

Strings

A *string* is an array of characters

Strings have many uses in MATLAB

- Display text output
- Specify formatting for plots
- Input arguments for some functions
- Text input from user or data files

Creating Strings

• We create a string by typing characters within single quotes (').

• Many programming languages use the quotation mark (") for strings. Not MATLAB!

•Strings can contain letters, digits, symbols, spaces. Can be words, sentences, anything you want!

This is a string, not a number!

•Examples of strings: `'Boston'`, `'4522'`, `'{Canada1867!}'`, `'Life is good.'`

You can assign string to a variable, just like numbers.

```
>> province = 'Ontario'

province =

Ontario

>> park = 'Canada''s Wonderland'

park =

Canada's Wonderland
```

To have the ' character, you double it.

In a string variable

- Numbers are stored as an array
- A one-line string is a row vector
  - Number of elements in vector is number of characters in string

```
>> place = 'Niagara Falls';

>> size(place)
```

In a string variable

- Numbers are stored as an array
- A one-line string is a row vector
  - Number of elements in vector is number of characters in string

```
>> place = 'Niagara Falls';

>> size(place)

ans =

 1 13
```

String Indexes

Strings are indexed the same way as vectors and matrices

- Can read by index
- Can write by index
- Can delete by index

Ex:

```
>>word(1) = 'd';

>>word(2) = 'a';

>>word(3) = 'l';

>>word(4) = 'e';

>>word

word = dale
```

Example:

word variable is ‘dale’
from previous slide.

```
>> word(1)

>> word(1) = 'v'

>> word(end) = []

>> word(end+1:end+3) = 'ley'
```

Example:

word variable is ‘dale’ from previous slide.

```
>> word(1)

ans = d

>> word(1) = 'v'

word = vale

>> word(end) = []

word = val

>> word(end+1:end+3) = 'ley'

word = valley
```

First letter is now **v**

Last letter is gone

ley added at the position following the end and the two other positions after that

MATLAB stores strings with multiple lines as an array. This means each line must have the same number of columns (characters).

```
>> names = ['Greg'; 'John']

names =

Greg

John

>> size( names )
```

MATLAB stores strings with multiple lines as an array. This means each line must have the same number of columns (characters).

```
>> names = ['Greg'; 'John']

names =

Greg

John

>> size( names )

ans =

2  4
```

Problem

```
>> names = [ 'Greg'; 'Jon' ]
```

MATLAB stores strings with multiple lines as an array. This means each line must have the same number of columns (characters).

```
>> names = ['Greg'; 'John']

>> size( names )
```

Problem

4 characters 3 characters

```
>> names = [ 'Greg'; 'Jon' ]???
Error using ==> vertcat
```

vertcat means vertical concatenation

```
CAT arguments dimensions are not
consistent.
```

Must put in extra characters (usually spaces) by hand so that all rows have same number of characters

```
>> names = [ 'Greg'; 'Jon ' ]

Greg

Jon
```

Extra space

String Padding

Making sure each line of text has the same number of characters is a big pain. MATLAB solves problem with **char** function, which *pads* each line on the right with enough spaces so that all lines have the same number of characters.

```
>> question = char('Romeo, Romeo,','Wherefore art thou', 'Romeo?')
question =

Romeo, Romeo,

Wherefore art thou

Romeo?

>> size (question)
```

String Padding

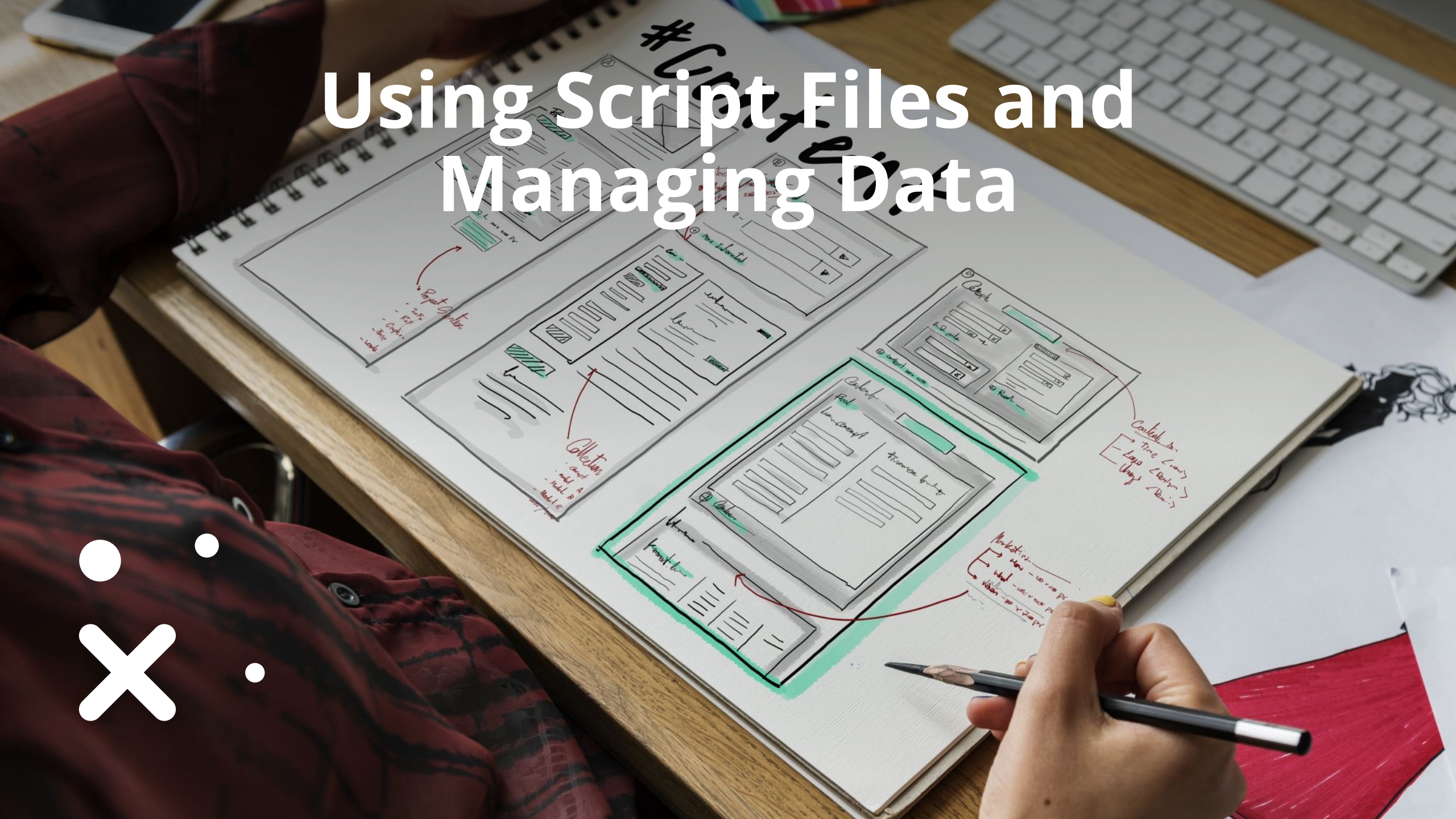
Making sure each line of text has the same number of characters is a big pain. MATLAB solves problem with **char** function, which *pads* each line on the right with enough spaces so that all lines have the same number of characters.

```
>> question = char('Romeo, Romeo,','Wherefore art thou', 'Romeo?')
question =

Romeo, Romeo,

Wherefore art thou

Romeo?

>> size (question)
ans =

3 18
```

```
R o m e o ,    R o m e o ,

W h e r e f o r e   a r t   t h o u

R o m e o ?
```

- Three lines of text stored in a 3x18 array.
- MATLAB makes all rows as long as longest row.
- First and third rows above have enough space characters added on ends to make each row 18 characters long.

Using Script Files and Managing Data



How to program in MATLAB

1. To create a new program, use the **Script** option under the **New** menu or enter the edit command in the command window (like `edit prog1` - prog1 being the name of your program). You now have a blank editor window (script editor).

2. Enter your code and save your script using the **Save** button.

3. Run your script by pressing the **Run** button or enter the script's name in the command window: `>>prog1`.

Basic MATLAB Programming

We know already that variables are containers of data.

We also know that we can use commands to ask MATLAB to do certain operations.

We know as well that a programming mode exists in MATLAB, permitting us to send multiple instructions being executed one after the other. We can save this program as an **.m** file.

Using Variables in Simple Scripts

Method #1: You assign the values in the program itself (we saw that before remember? the = operator).

The following is an example of such a case. The script file (saved as Chapter4Example2) calculates the average points scored in three games.

```
% This script file calculates the average points scored in three games.
% The assignment of the values of the points is part of the script file.
game1=75;
game2=93;
game3=68;
ave_points=(game1+game2+game3)/3
```

The variables are assigned values within the script file.

The display in the Command Window when the script file is executed is:

```
>> Chapter4Example2

ave_points =
   78.6667
>>
```

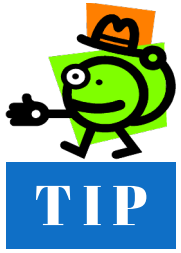
The script file is executed by typing the name of the file.

The variable `ave_points` with its value is displayed in the Command Window.

Using Variables in Simple Scripts

Method #2: Defining the variables in the command window before running the script.

- Advantage: More versatile as the script can produce a different result at every run depending on the variable values.



- *Instead of retyping entire command, use up-arrow to recall command and then edit it.*

For the previous example in which the script file has a program that calculates the average of points scored in three games, the script file (saved as Chapter4Example3) is:

```
% This script file calculates the average points scored in three games.
% The assignment of the values of the points to the variables
% game1, game2, and game3 is done in the Command Window.

ave_points=(game1+game2+game3)/3
```

Example of Method #2

The Command Window for running this file is:

```
>> game1 = 67;
>> game2 = 90;
>> game3 = 81;
```

The variables are assigned values in the Command Window.

```
>> Chapter4Example3

ave_points =
   79.3333

>> game1 = 87;
>> game2 = 70;
>> game3 = 50;
>> Chapter4Example3

ave_points =
    69
>>
```

The script file is executed.

The output from the script file is displayed in the Command Window.

New values are assigned to the variables.

The script file is executed again.

The output from the script file is displayed in the Command Window.

Using Variables in Simple Scripts

Method #3: Assign by prompt in script file (meaning ask the user to fill the variables values).

The program asks the user to enter a value, then the script assigns that value to a variable. The MATLAB **input** command is used for that purpose.

```
variable_name = input('prompt')
```

prompt is the text that the input command will display in the Command Window (must be between single quotes).

The Input Command

```
variable_name = input('prompt')
```

When the script executes the input command it does the following in order:

1. Displays the prompt text in the Command Window.
2. Puts the cursor immediately to the right of prompt.
3. User types the value and presses ENTER.
4. Script assigns the user's entered value to variable and displays the value unless the input command has a semicolon at the end.

Input Command - Example

CODE

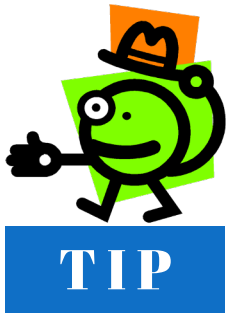
```
% This script file calculates the average of points scored in three games.
% The points from each game are assigned to the variables by
% using the input command.
game1=input('Enter the points scored in the first game ');
game2=input('Enter the points scored in the second game ');
game3=input('Enter the points scored in the third game ');
ave_points=(game1+game2+game3)/3
```

RESULT

```
>> Chapter4Example4
Enter the points scored in the first game   67
Enter the points scored in the second game   91
Enter the points scored in the third game   70

ave_points =
    76
>>
```

The computer displays the message. Then the value of the score is typed by the user and the **Enter** key is pressed.

Input Command - User Fills Variable



It is good form to put a space, or a colon and a space, at the end of the prompt so that the user's entry is separated from the prompt.

For example (if the user enters 2001):

```
age = input('Year of birth');
Year of birth2001
```

Bad!

```
age = input('Year of birth ');
Year of birth 2001
```

Better!

```
age = input('Year of birth: ');
Year of birth: 2001
```

Best!

Input Command - User Fills Variable

But what if you want to fill the variable with a string like a name or a sentence?

```
city = input ('City of birth: ')
```

If you enter just Toronto, you will get an error. You must enter the quotes to indicate that it is a string: 'Toronto'.

```
City of birth: 'Toronto'
```

But there is another way. You can use the 's' qualifier:

```
city = input ('City of birth: ', 's')
City of birth:  Toronto
```

Output Commands: disp and fprintf

We know that when omit the semicolon at end of a statement, MATLAB displays the result on screen. You have no control over the appearance of the result (how many lines, what precision in numbers...)

You can use the MATLAB command **disp** for some control of appearance and **fprintf** for full control.

The disp Command

- **disp (display)**: Displays variable values or text on the screen.
- Each display appears on a separate line. It doesn't show the variable name, only its value. **disp** only offers a limited control of display.
- For the best control of display we can use **fprintf**., a more advanced command.
- **disp(variable_name)** or **disp('text string')**

```
>> abc = [5  9  1;  7  2  4];
>> disp(abc)
     5     9     1
     7     2     4

>> disp('The problem has no solution.')

The problem has no solution.
>>
```

A 2 × 3 array is assigned to variable abc.

The disp command is used to display the abc array.

The array is displayed without its name.

The disp command is used to display a message.

The fprintf Command

- **fprintf** means file print formatted.
- You can write to the screen or to a file.
- You can mix numbers and text in the output.
- You have the full control of the output display but it is a little bit more complicated to use than **disp**.

The fprintf Command : A Simple Sentence

```
>> fprintf ('The quick brown fox')
The quick brown fox>>
```

Problem - The Command Window displays the command prompt (>>) at end of the text, not at start of next line! Not very pretty.

The solution is to use a `\n` *(new line) character at the end of the sentence. Looks much better!*

```
>> fprintf ('The quick brown fox\n')
The quick brown fox
>>
```

The \n Character

You can use as many \n in your fprintf as you like.

```
>> fprintf('Get\nBusy\nNow!\n\n')

Get
Busy
Now!

>>
```

fprintf Command – Printing a Variable

What if I want to print out the value of a variable? We can do that using a more advanced form of fprintf containing two parts: a string (including placeholders), and the arguments (the variable themselves). Here is a simple example with one numeric variable following a sentence.

```
>>year = 1867;
>>fprintf ('The year is %d\n', year)
The year is 1867
>>
```

This is a placeholder

You can also print without any extra text (aka labeling):

```
>>fprintf ('%d\n', year)
1867
>>
```

fprintf: Printing More Than One Variable

You can print more than one variable in the same fprintf, like this following example:

```
>> day = 25; temperature = 20;

>> fprintf ('It is October %d, and the temperature
  is %d degrees.\n', day, temperature)

It is October 25 and the temperature is 20 degrees.

>>
```

- *The number of variables and placeholders must be the same. The first placeholder formats the first variable, the second placeholder formats the second variable, and so on…*

Integer Placeholders

%d is the default integer placeholder. When used it will simply display the value ***as is*** without any padding. To add padding, to have nicely aligned columns for example, we need formatted placeholders.

%*n*d will reserve *n* places to display the number. Justification will be to the right. The negative sign takes one place.

If the value is 17 and %4d is used, then it will display 2 blank spaces followed by 17 on the screen:

 # is used to represent a blank space but blank spaces are really invisible!

Integer Placeholders

%d is the default integer placeholder. When used it will simply display the value ***as is*** without any padding. To add padding, to have nicely aligned columns for example, we need formatted placeholders.

%*n*d will reserve *n* places to display the number. Justification will be to the right. The negative sign takes one place.

If the value is 17 and %4d is used, then it will display 2 blank spaces followed by 17 on the screen:

##17

 # is used to represent a blank space but blank spaces are really invisible!

Integer Placeholders Rules

The number is always displayed in its entirety, even when the format is too narrow. With -1234 and a %3d placeholder, you would see -1234, therefore using 5 spaces instead of the 3 requested.

A negative number in the placeholder changes the justification to the left of the field. For example, with a value of -1234 and a %-8d placeholder, the display will be

Integer Placeholders Rules

The number is always displayed in its entirety, even when the format is too narrow. With -1234 and a %3d placeholder, you would see -1234, therefore using 5 spaces instead of the 3 requested.

A negative number in the placeholder changes the justification to the left of the field. For example, with a value of -1234 and a %-8d placeholder, the display will be -1234###.

Three trailing blank spaces

Floating Point Placeholders

By default the %f placeholder displays the number with 6 decimal digits and no padding (may vary depending on computer system).

The formatted double placeholder has this format: *%w.df*, where **w** is the total width of the field (including sign and decimal point) and **d** the number of decimal digits.

If the value is 4.56 and the placeholder is %6.3f, then the display will be

Floating Point Placeholders

By default the %f placeholder displays the number with 6 decimal digits and no padding (may vary depending on computer system).

The formatted double placeholder has this format: *%w.df*, where **w** is the total width of the field (including sign and decimal point) and **d** the number of decimal digits.

If the value is 4.56 and the placeholder is %6.3f, then the display will be #4.560

One leading blank space

Floating Point Placeholders Rules

With a floating point formatted print, **you always get the requested number of digits to the right of the decimal point** (even if the field is not wide enough).

You also **always** (like the integer placeholder) **get all the significant numbers** (the numbers to the left of the decimal point).

However, if there are more decimal precision in the value than are requested in the placeholder, the value is truncated to match the placeholder size and rounded up if need be.

Floating Point Placeholder Examples

If the value is -32.4573 and the placeholder is %7.3f

If the value is -3.4578 and the placeholder is %8.3f

If the value is 187.123 and the placeholder is %8.7f

*Note: Internal values in the computer's memory are **unaffected** by placeholders (just look at the workspace).*

Floating Point Placeholder Examples

If the value is -32.4573 and the placeholder is %7.3f, then you will get 3 decimal digits as requested plus the significant numbers: -32.457 for a total of 7 spaces.

If the value is -3.4578 and the placeholder is %8.3f, then you will get 3 decimal digits as requested plus the significant numbers and padding: ##-3.458. Note the rounding-up effect.

Two leading blank spaces

If the value is 187.123 and the placeholder is %8.7f it will display 187.1230000.

*Note: Internal values in the computer's memory are **unaffected** by placeholders (just look at the workspace).*

This concludes our overview of MATLAB and a taste of things to come!

End of lesson