1. Introduction

The Matlab commands can be executed in the Command Window. The problem is that the commands entered in the Command Window cannot be saved and executed again for several times. Therefore, a different way of executing repeatedly commands with MATLAB is:

1. To create a file with a list of commands,

2. Save the file, and

3. Run the file.

If needed, corrections or changes can be made to the commands in the file. The files that are used for this purpose are called *script files or scripts* for short.

2 M-File Scripts

A *script file* is an *external* file that contains a *sequence* of MATLAB *statements*. Script files have a *filename extension .m* and are often called *M-files*. M-files can be scripts that simply execute a series of MATLAB statements, or they can be *functions* that can *accept arguments* and can *produce one or more outputs.*

2.1 Examples

1. Consider the system of equations:

$x + 2y + 3z = 1$

$3x + 3y + 4z = 1$

$2x + 3y + 3z = 2$

Find the solution *x* to the system of equations.

Solution:

• Use the MATLAB editor to create a file: *File → New → M-file*.

• Enter the following statements in the file:

```
A = [1 2 3; 3 3 4; 2 3 3];
b = [1; 1; 2];
x = A\b
```

• Save the file, for example, *example1.m*.

• Run the file, in the command line, by typing:

```
>> example1

x =
-0.5000
1.5000
```

```
-0.5000
```

When execution completes, the variables *(A, b, and x)* remain in the workspace. To see a listing of them, enter *whos* at the command prompt.

Note: The *MATLAB editor* is both a text editor specialized for creating *M-files* and a *graphical MATLAB debugger*. The MATLAB editor has numerous menus for tasks such as *saving, viewing, and debugging*. Because it performs some simple checks and also uses *color* to differentiate between various elements of *codes*, this text editor is recommended as the tool of choice for writing and editing M-files.

There is another way to open the editor:

```
>> edit
or
>> edit filename.m   % to open filename.m.
```

2. Plot the following cosine functions, $y_1 = 2\cos(x)$, $y_2 = \cos(x)$, and $y_3 = 0.5 * \cos(x)$, in theinterval $0 \leq x \leq 2\pi$. This example has been presented in previous Chapter. Here we put the commands in a file.
• Create a file, say *example2.m*, which contains the following commands:

```
x = 0 : pi/100 : 2*pi ;
y1 = 2*cos(x);
y2 = cos(x);
y3 = 0.5*cos(x);
plot(x,y1,'--',x,y2,'-',x,y3,':')
xlabel('0 \leq x \leq 2\pi')
ylabel('Cosine functions')
legend('2*cos(x)','cos(x)','0.5*cos(x)')
title('Typical example of multiple plots')
axis([0 2*pi -3 3])
```

• Run the file by typing *example2* in the Command Window.

2.2 Script side-effects

All variables created in a script file are added to the workspace. This may have undesirable effects, because:

• Variables already existing in the workspace may be overwritten.

• The execution of the script can be affected by the state variables in the workspace.

As a result, because scripts have some undesirable side-effects, it is better to code any complicated applications using rather *function M-file*.

3. Inputs and outputs

3.1 Input to a script file

When a script file is executed, the *variables* that are used in the calculations within the file must have *assigned values*. The assignment of a value to a variable can be done in three ways.

1. The variable is defined in the script file.

2. The variable is defined in the command prompt.

3. The variable is entered when the script is executed.

In the third case, the variable is defined in the *script file.* When the file is executed, the user is prompted to assign a value to the variable in the *command prompt*. This is done by using the *input command*. Here is an example.

This script file calculates the average of points scored in three games. The point from each game are assigned to a variable 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 ');
average = (game1+game2+game3)/3
```

When the script file (saved as example3) the command prompt is :

```
>> example3
>> Enter the points scored in the first game 15
>> Enter the points scored in the second game 23
>> Enter the points scored in the third game 10
average =
16
```

The *input command* can also be used to *assign string* to a variable.

3.2. Output commands

MATLAB automatically generates a display when commands are executed. In addition to this automatic display, MATLAB has several commands that can be used to generate displays or outputs.

Two commands that are frequently used to generate output are: *disp and fprintf.*

The main differences between these two commands can be summarized as follows

`disp`	. Simple to use. . Provide limited control over the appearance of output
`fprintf`	. Slightly more complicated than `disp`. . Provide total control over the appearance of output

Table : disp and fprintf commands

4. MATLAB Data Types

4.1 Define data types in MATLAB

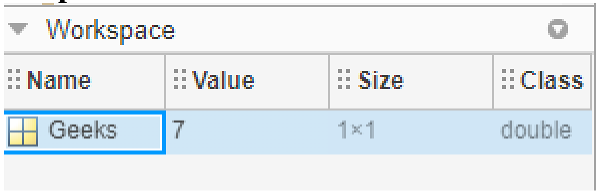
In MATLAB we do not require any type of declaration statement, when it gets any new variable name it creates the variable and allocates appropriate memory space to it but if the variable name already exists it will replace the original content with new content and allocate it to new storage space when required.

Syntax: variable name = a value (or an expression)

Example:

```
>> Geeks = 7;
```

Output:

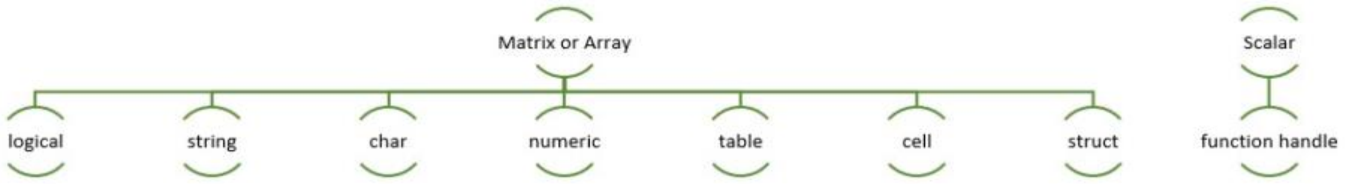


Workspace

Name	Value	Size	Class
Geeks	7	1×1	double

4.1. Data Types in MATLAB

In MATLAB, *data* can be stored in different types, *numeric*, *text, complex number*, etc. To store these data MATLAB has different *classes* which have various characteristics. MATLAB provides a total of 16 fundamental data types.



- **Logical Type**

Logic types are *true and false* values that are represented with the logical value *0 and 1*. Any numerical value (non-complex) can be converted into a logical representation.

Syntax: G = logical (x)

Example:

MATLAB code for random matrix generation

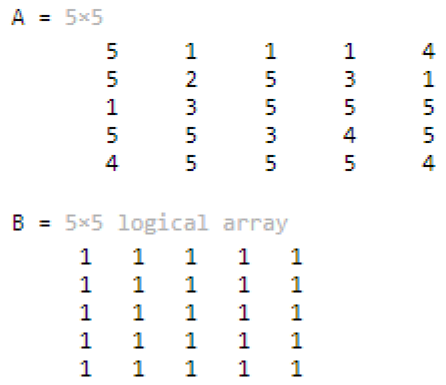
```
>>A = randi(5,5)
```

It will generate random matrix of size 5x5

```
>>B = A < 9
```

The result is a logical matrix.
Each value in B represents a *logical 1 (true),* or *logical 0 (false)* state to indicate whether the corresponding element of A fulfills the *condition A < 9.*
For example, if *A(1,1) is 13, so B(1,1) is logical 0 (false).*
However, if *A(1,2) is 2, so B(1,2) is logical 1 (true).*

Output:

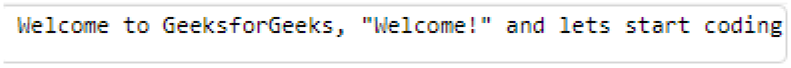


```
A = 5×5

     5     1     1     1     4
     5     2     5     3     1
     1     3     5     5     5
     5     5     3     4     5
     4     5     5     5     4

B = 5×5 logical array

   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
```

- **Char and String type**

In MATLAB character and string array provide storage for text type data. The strings are character array compared with the sequence of numbers called a numeric array.
Syntax: s = 'String'
Example:
```
>>str = "Welcome to GeeksforGeeks, ""Welcome!"" and lets start coding."
>>fprintf(str);
```

Output:



```
Welcome to GeeksforGeeks, "Welcome!" and lets start coding
```

- **Numeric Type**

Integer and floating-point data are in the following descriptions.

Data Type	**Short Description**	**Features**
double	Double-precision arrays	• Default numeric data type (class) in MATLAB • Stored as 64-bit (8-byte) floating-point value • Range: Negative numbers = -1.79769 x 10308 to -2.22507 x 10-308 Positive numbers = 2.22507 x 10-308 to 1.79769 x 10308
single	Single-precision arrays	• Stored as 4-byte (32-bit) floating-point value • Range- Negative numbers = -1.79769 x 10308 to -2.22507 x 10-308 Positive numbers = 2.22507 x 10-308 to 1.79769 x 10308
int8	8-bit signed integer arrays	• Stored as 1-byte (8-bit) signed integers • Range is -27 to 27-1
int16	16-bit signed integer arrays	• Stored as 2-byte (16-bit) signed integers • Range -215 to 215 -1
int32	32-bit signed integer arrays	• Stored as 4-byte (32-bit) signed integers • Range is -231 to 231-1
int64	64-bit signed integer arrays	• Stored as 8-byte (64-bit) signed integers • Range is -263 to 263-1
uint8	8-bit unsigned integer arrays	• Stored as 1-byte (8-bit) unsigned integers • Range is 0 to 28-1
unit16	16-bit unsigned integer arrays	• Stored as 2-byte (16-bit) unsigned integers • Range is 0 to 216 -1

uint32	32-bit unsigned integer arrays	• Stored as 4-byte (32-bit) unsigned integers • Range is 0 to 232-1
uint64	64-bit unsigned integer arrays	• Stored as 8-byte (64-bit) unsigned integers • Range is 0 to 264-1

- **Table type**

The table contains *rows and column variables*. Each variable can be of different *data types* and *different sizes*, but each variable needs to have the same number of rows. Range of functions are used to access data to create, edit, and read the table data.

Syntax: T = table(ColumnName1,ColumnName2);

Example:
T = table(Name,QuestionAttempted,CodingScore);
data = {'Atul Sisodiya',22,100};
Tnew = [Tnew;data];

Output:
```
Table array

2x3

   Name           QuestionAttempted        CodingScore

   Atul Sisodiya                  22              100
```

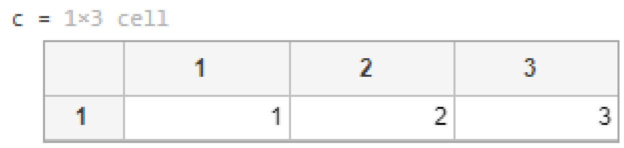
- **Cell**

A *cell array* is a MATLAB data type that contains *indexed* data containers called *cells*. *Cells* can contain any type of data, commonly contain *character vectors* of different lengths, *numbers, an array of numbers* of any size. Sets of cells are enclosed in () and access to the cells is done by using {} which is to create, edit or delete any cell functions.

Syntax: c = { }

Example:
C = {1, 2, 3}

Output:



```
c = 1×3 cell
```

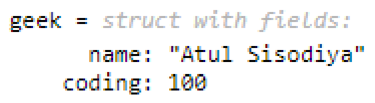
	1	2	3
1	1	2	3

- **Structure**

In *structure* data containers are used to group related data and their type, which are called fields. Fields may contain any type of data. In structures, Data is accessed using the dot notation.

Syntax: structname.fieldName

Example:
>>geek(1).name = ("Atul Sisodiya");
>>geek(1).coding = 100;
>>geek
Output:



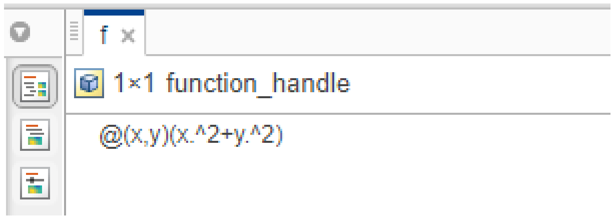
```
geek = struct with fields:
      name: "Atul Sisodiya"
    coding: 100
```

- **Function Handles**

Function Handles is majorly used in MATLAB is to pass a function (numerical or char) to another function. Variables that are used to invoke function indirectly can be named as a function handle.

To create a function handle *'@'* operator is used.

Example: To create a function handle to evaluate x$^{\wedge 2}$ + y$^{\wedge 2}$, a function used is:
Output:



```
1×1 function_handle
@(x,y)(x.^2+y.^2)
```

4.2. Identification of MATLAB Data Types

Features of MATLAB provides a collection of variables for developers to identify and understand the data types of values and variables to ensure correct and efficient programming. Additionally, MATLAB offers several built-in functions for data type identification and conversion.

The *"class" function* is commonly used to determine the data type of a variable. It returns a string representing the *class* of the variable, such as *"double", "char", "logical", or "cell"*. This information helps in verifying the expected data type and performing appropriate operations.