**DISPLAYING VALUES AND TEXT**

There are three ways of displaying numbers and text in MATLAB, which includes
(i) Entering the variable name at the command prompt (the approach used by far)
(ii) Using the MATLAB `disp` command, and
(iii) Using the `fprintf` command.

**disp**

The disp command is used to display the value of a variable without printing the variable. The syntax is

```
disp(x)
```

If x is a string, the string is displayed without the enclosing single quotes, `' '`. If the variable x contains an empty array, `disp` returns without displaying anything.

Examples:

```
>> x = 10;
>> disp(x)
10
```

Note that `disp` takes only a single argument. To display several variables, they must be grouped together, or concatenated.

```
>> x = 10; y = 20; z = 30;
>> x = 10; y = 20; z = 30;
>> disp(x, y, z)
Error using disp
Too many input arguments.

>> disp([x, y, z])
10    20    30

>> A = [10, 20, 30];
>> disp(A)
10    20    30

>> st = 'MATLAB Programming is fun!';
>> disp(st)
MATLAB Programming is fun!
```

To display the name of a variable in the output, one can use the `num2str` function to convert the numeric value of the variable to a string. The concatenation operator `[]` is then used to join the variable name and the string from the num2str function as in the following example.

```
>> x = 45;
```

```
>> disp(['x = ', num2str(x)])
x = 45
```

This next example uses the `num2str` function and the concatenation operator to generate a string out of two variables.

```
>> name = 'Alice';
>> age = 12;
>> X = [name,' will be ',num2str(age),' this year.'];
>> disp(X)
Alice will be 12 this year.
```

To exercise more control over the output, one can use the `sprintf` function to format text before using `disp` to display the text.

**sprintf**

The `sprintf` command is used to format data into string or a character array. The syntax has variants as follows:

(a) `str = sprintf(formatString, A1,A2,...,AN)`
(b) `[str,errormsg] = sprintf(FormatString, A1,A2,...,AN)`
(c) `str = sprintf(literalText)`

`str = sprintf(formatString, A1,A2,...,AN)` formats the data in the list of arrays `A1,A2,...,AN` using formatting operators specified by `FormatString` and returns the resulting string in str.

`[str,errormsg] = sprintf(FormatString, A1,A2,...,AN)` returns an error message as a character vector when the operation is unsuccessful. Otherwise, `errormsg` is empty.

`str = sprintf(literalText)` translates escape-character sequences in `literalText`, such as `\n` and `\t`. It returns all other characters unaltered. If `literalText` contain a formatting operator (such as `%f`), then `str` discards it and all characters after.

**Examples:**

(1) Integers can be formatted using `%d` or `%i` specifiers

```
>> x = 10;
>> str = sprintf('%d', x)
>> disp(str)
10
```

`FormatString` can also include additional text before the percent sign, %, or after the format specifier

```
>> x = 10;
>> str = sprintf('x = %d', x)
>> disp(str)
x = 10

>> a = 10; b = 30;
>> str = sprintf('%d plus %d is %i',a,b,a+b);
>> disp(str)
10 plus 30 is 40

>> A = rand(4,5);
>> [m, n] = size(A);
>>disp(sprintf('The array A is a %d x %d matrix', m,n))
The array A is a 4 x 5 matrix
```

(2) Floating-point numbers can be formatted using `%e`, `%f, and %g` specifiers

```
>> x = 6.2831857072;
>> str = sprintf('%f', x)
>> disp(str)
6.283186

>> disp(sprintf('%e',x))
6.283186e+00

>> disp(sprintf('%g',x))
6.28319

>> disp(sprintf('%g', 6283.1857072))
6283.19

>> disp(sprintf('%e', 6283.1857072))
6.283186e+03
```

Types of format specifiers

| Value Type | Format Specifier | Interpretation |
|---|---|---|
| Signed Integers | `%d or %i` | Format as a base 10 signed integer |
| Unsigned Integer | `%u` | Format as a base 10 unsigned integer |
| | `%o` | Format as base 8 unsigned integer |
| | `%x` | base 16 (hexadecimal) lowercase letters, `a - f` |
| | `%X` | base 16 (hexadecimal) uppercase letters, `A - F` |
| Floating-Point Numbers | `%f` | Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.) |
| | `%e` | Exponential notation, such as 3.141593e+00 |
| | `%E` | Same as %e, but uppercase, such as 3.141593E+00 |
| | `%g` | The more compact of %e or %f, with no trailing zeros |
| | `%G` | The more compact of %E or %f |
| Characters or strings | `%c` | Simple character |
| | `%s` | Character vector or string array |

**Field Width and Precision**
Syntax is %w.pSpecifier. where w is the field with and p is the precision.
Examples: %12d, %0.4f, %5.2f, etc.

**Optional Specifiers or Operators**
(a) Use `'-'` to left justify the output text: E.g., `%-7.4f`

(b) Use `'+'` to always print a sign character (+ or -) for any numeric value or to right-justify text, e.g., `%+10s`.

(c) Use `'0'` to pad output field width with zeros before the value. Example: %05.2f

(d) Use `'#'` to modify selected numeric conversions: For %o, %x, or %X, print 0, 0x, or 0X prefix. For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. For `%g` or `%G`, do not remove trailing zeros or decimal point.
Example: `%#5.0f`

**Escape Characters**
(a) New line: `'\n'`
(b) Carriage return: `'\r'`
(c) Backspace: `'\b'`
(d) Horizontal tab character: `'\t'`
(e) Vertical tab: `'\v'`

(f) Single quote: ''
(g) Percent character: %%
(d) Backslash: \\

## 3.4 STRUCTURED PROGRAMMING WITH MATLAB

### 3.4.1 PROGRAMMING WITH M-FILES

An M-file consists of a series of MATLAB statements saved to a file so that they can be run all at once by the interpreter. The word "M-file" originates from the fact that such files are stored with a `.m` extension. There are two types of M-files: script files and function files.

**Script M-files**

A script file is an M-file that contains a series of MATLAB commands or statements. They are useful for automating repetitive tasks in MATLAB.

You can create a new script in the following ways:

(i) By clicking the **New Script** button on the Home tab.

(ii) Using the `edit` function at the command prompt. For example,

edit NewFileName

creates (if the file does not exist) and opens the file `'NewFileName'`. If `'NewFileName'` is unspecified, MATLAB opens a new file called Untitled.

(iii) By highlighting commands from the Command History, right-clicking, and selecting **Create Script**.

After a script is created, you can add code to the script and save it.

Example 1: Given the set, $x = (0.1, 0.01, 0.001)$, develop a script M-file that calculates the values of $y = \sin(x)/x$. The script must print values of $y$ with full double precision.

Example 2: The velocity of a bungee jumper is given by

$$v = \sqrt{\frac{gm}{m}}\tanh\left(\sqrt{\frac{gc_d}{m}}t\right)$$

where $v$ is velocity (m/s), $g$ is the acceleration due to gravity (m/s$^2$), m is mass (kg), $C_d$ is the drag coefficient (kg/m), and $t$ is time (s).

(a) Develop a script file that computes the velocity of the free-falling jumper for values of $t$ = (0:2:20). Take g = 9.81, m = 68.1, and cd = 0.25.

(b) Create a matrix $m$ of values of $t$ and $v$ with $t$ in the first column and $v$ in the second column.

Example 3: The volume V of liquid in a hollow cylinder of radius r and length L is related to the depth by of the liquid h by

$$V = \left[r^2\cos^{-1}\left(\frac{r-h}{r}\right) - (r-h)\sqrt{2rh-h^2}\right]L$$

Develop an M-file to create a plot of volume versus depth with $r = 2$, $L = 5$ and $h$ as a column vector with values 0:0.04:4.

**PROGRAMMING WITH FUNCTION M-FILES**

A function is a program or sub-program that accepts one or more input arguments, executes a series of MATLAB statements and returns outputs. MATLAB functions are written in a function file. The basic syntax for a MATLAB function is

```
function y = demofun(x)
statements to execute
end
```

which declares a function named demofun that accepts one input x and returns one output y. The function declaration line

```
function y = demofun(x)
```

must be the first executable line in the function file and it must start with the function keyword.

A more complete syntax of a function that accepts multiple input arguments `x1,x2,...,xN`, and returns multiple outputs, `y1,y2,...,yN`, is as follows:

```
function [y1,y2,...,yN] = testfun(x1,x2,...,xN)
```

A function can be saved in either of two ways (traditionally): (i) in a file containing only the function definition, or (ii) in a file containing other function definitions. The name of the file should match the name of the first function in the file.

Note that beginning from MATLAB R2016B functions can also be defined in script M-files. The rule is that when defined in a script file, the function or functions must be put at the end of the file.

Examples:

(i) Write a function that adds three numbers.

```
function result = add(x, y, z)
result = x + y + z;
end
```

(ii) Write a function that converts temperature in Fahrenheit to Celsius.

```
function Celsius = fah2cel(f)
Celsius = (f - 32)*5/9;
end
```

(iii) Functions with multiple outputs:

(a) Write a MATLAB function that calculates the square and cube of a number.

```
function [s c] = squarecube(x)
s = x.^2;
c = x.^3;
end
```

(b) Write a function that accepts a 1D array of values as input and returns the mean and standard deviation of the array.

```
function [m,s] = statistics(x)
N = length(x);
m = sum(x);
xm = (x - m).^2;
ss = sum(xm);
s = sqrt(ss/n);
end
```

It should be noted that the end keyword is optional when the function is the only function in the file. However, when the function contains a nested function, or is a local function within a function file or a script file, the end keyword is mandatory.

**Other Function Syntax**

(i) If a function returns no output, the output can be omitted in the function declaration.

```
function functionName(x)
```

or use empty square brackets: `function [ ] = functionName(x)`

(ii) If the function accepts no inputs, you can omit the parenthesis, ().

```
function functionName
```

**How to Name MATLAB Functions**

A function name must begin with an alphabetic character, and can contain letters, numbers, or underscores. A space character is not allowed in a function file name. Note that these rules also apply to naming of script M-files. Thus, the following are considered as valid function names: `demofun21`, `my_fun66`, `_testfun`, etc.

On the contrary, the following are not considered as valid function file names: `21myfun`, `6xyz`, `my_ fun66`, etc.

In MATLAB, some names are considered to be reserved keywords and are therefore not allowed as names for functions. For example, the following keywords are reserved in MATLAB:

```
break case catch classdef continue else elseif end for function
global if switch otherwise parfor persistent return spmd try
while
```

To check whether or not a word is a reserved keyword, use the MATLAB command iskeyword().

Lastly, **Do Not** use file names that are identical to the names of MATLAB's built-in functions (e.g., sum, sqrt, exp, zeros, etc.) or user-defined functions that are already in use to prevent naming conflicts.

**Multiple Functions within a File**

Multiple functions can be defined in one file. In this case, the first function in the file is called the main function. This function is visible to functions in other files, or it can be called from the command line. Additional functions within the file are called local functions, and they can occur in any order after the main function. Local functions are only visible to other functions in the same file. They are equivalent to subroutines in other programming languages, and are sometimes called subfunctions.

Example: Create a function that uses two local functions to calculate the area and circumference of a circle.

```
function [a, p] = area_and_perimeter(r)
a = area_c(r);
p = perimeter_c(r);
end

function a = area_c(x)
a = pi*x.^2;
end

function p = perimeter_c(x)
p = 2*pi*x;
end
```

**OTHER WAYS OF DEFINING MATLAB FUNCTIONS**

**FUNCTION HANDLES**

A function handle is a variable that allows you to invoke a function indirectly. The handle provides a way of referring to the function. Function handles have various uses, including constructing anonymous functions, specifying call back functions and passing functions to other functions.

(i) To create a handle to a function, the @ symbol is used. This is done by preceding the function with the @ symbol. For example, a handle to the MATLAB square root function, `sqrt` can be defined as

```
fh = @sqrt
```

The function handle `fh` can now be used to evaluate square root of numbers as though it was the `sqrt` function itself.

```
>> fh = @sqrt;
>> fh(16)
ans =
   4
```

The function handle can also be invoked using the `feval` command. E.g.,

```
>> feval(fh, 81)
ans =
   9
```

(ii) A function handle can also be used to invoke user-defined functions. Example:

calcsquare.m

```
function y = calcsquare(x)
y = x.^2;
end
```

```
>> fh = @calcsquare;
>> x = 15;
>> y = fh(x)
b =
   225
```

(iii) Function handles can be passed as variables to other functions. For example, calculate the integral of $x^2$ on the range [0, 5], using the MATLAB `integral` command.

```
>> fh = @calcsquare;
>> y = integral(fh, 0, 5)
y =
   41.6667
```

**ANONYMOUS FUNCTIONS**

An anonymous function is a one-line expression-based MATLAB function that does not require an M-file for its definition. It can be created using the function handle notation. The syntax is `fh = @(arglist) expression.`

Examples: (a) Create a handle to an anonymous function that computes the square of a number.

```
>> fh = @(x) x.^2;
>> fh(7)
ans =
   49
```

**Anonymous Functions with No Inputs**: If the anonymous function does not require any inputs, use empty parentheses when defining and calling the anonymous function. Eg.,

```
>> fh = @() datestr(now);
>> d = fh()
d =
   '26-Jun-2021 16:18:11'
```

**Anonymous Functions with Multiple Inputs and Outputs**: Multiple input arguments must be specified within the parenthesis, separated by commas. For example, the following anonymous function accepts two inputs, x and y:

```
>> testfun = @(x,y) (x^2 + y^2 + x*y);
>> x = 1;
>> y = 10;
>> z = testfun(x,y)
```

### 3.4.2 PROGRAM CONTROL STATEMENTS

These are statements used to control program execution. There are two types of such statements: Conditional Statements and Loops.

#### 3.4.2.1 Conditional Statements

Conditional statements allow MATLAB to select at run time which block of code to execute. In programming, such statements are also known as Selection Statements. An example of a selection or conditional statement is an **if** statement. Three variants of the if statement are used in MATLAB: if, if/else and if/elseif/else. There syntaxes are as follows:

(a) Selection without alternatives:

if <condition>
    *Statements to be executed*
end

Two types of operators are generally used in the if condition: comparison and logical operators.

Comparison Operators

| MATLAB Operator | Name |
|---|---|
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| == | equal to |
| ~= | not equal to |

Logical Operators (Boolean Operators)

| MATLAB Operator | Name |
|---|---|
| & | logical AND |
| \| | logical OR |
| && | logical AND (with short-circuiting) |
| \|\| | logical OR (with short-circuiting) |

| | |
|---|---|
| ~ | logical NOT |

The logical operators are used when it is necessary to test multiple conditions.

(b) Selection with two alternatives:

```
if <condition>
    Statements to be executed
else
    Alternative statements to be executed
end
```

(c) Selection with multiple alternatives:

```
if <condition 1>
    Statements to be executed
elseif <condition 2>
    Alternative statements to be executed
elseif <condition 3>
    Alternative statements to be executed
end
```

**Examples**

(a) Testing for an even number: Write a code that prompts for a number and tests whether or not the number is an even number.

```
% Choose some number and assign it to the variable, number.
number = 4;

% Check whether the number is even
remainder = rem(number, 2);
if remainder == 0
    disp('number is even')
end
```

(b) (i) The above code has one limitation which is that when the remainder is non-zero, there will be no output to the screen. An improved version of the code is given below, using the if/else structure.

```
% Prompt user to enter a number
number = input('Please, enter any number: ');

% Check whether the number is even
```

```
remainder = rem(number, 2);
if remainder == 0
    disp('number is even')
else
    disp('number is not even')
end
```

(ii) Write a program that calculates the area of a circle. Let the program prompt the user for the value of the radius of the circle; it should reject radius values less than zero.

```
r = input('Please, enter the radius of the circle: ');

if r < 0
    disp('Invalid input')
else
    area = pi*r*r
    disp(['The area of the circle is ', num2str(area)])
end
```

(c) Write a code that receives the score of a student as input and displays the letter grade.

```
score = input('Please, enter the student's score: ');

if score >= 70
    disp(['The grade is ', 'A'])
elseif score >= 60
    disp(['The grade is ', 'B'])
elseif score >= 50
    disp(['The grade is ', 'C'])
elseif score >= 40
    disp(['The grade is ', 'D'])
else
    disp(['The grade is ', 'F'])
end
```

(e) **Example on Logical Operators:** Write a program that prompts the user for a number and checks whether the number is divisible by 2 and 3, by 2 or 3, and by 2 or 3 but not both.

```
% Receive an input
number = input('Enter an integer: ');
if rem(number, 2) == 0 & rem(number, 3) == 0
    fprintf('%d is divisible by 2 and 3\n', number)
elseif rem(number, 2) == 0 | rem(number, 3) == 0
    fprintf('%d is divisible by 2 or 3\n', number)
```

```
elseif (rem(number, 2) == 0 | rem(number, 3) == 0) & ...
    ~(rem(number, 2) == 0 & rem(number, 3) == 0)
    fprintf('%d is divisible by 2 or 3, but not both\n', number)
else
    fprintf('%d is neither divisible by 2 nor 3\n', number)
end
```

### 3.4.2.2. Loop Control Structures

Loops are used to make programs execute statements repeatedly. Each repeated execution is called an **iteration**. Consider a programming example in which you are required to print the statement "MATLAB Programming is fun!" a hundred times. It would be tedious to type the statement

```
disp('MATLAB Programming is fun!')
```
or
```
fprintf('MATLAB Programming is fun!\n')
```

a 100 times! The solution is to use a *loop* to execute the statement a 100 times. MATLAB provides two types of loops that can be used to accomplish such a task: **for** loops and **while** loops.

#### 3.4.2.2.1. For Loops

A for loop is a count-controlled loop that repeats a specific number of times. The for loop keeps track of each iteration by incrementing an index variable. The syntax of the for loop is as follows:

```
for IndexVariable = 1:TotalNumberOfIterations
    statements to be executed repeatedly
end
```

**For Loop Examples**

(a) To print the statement "MATLAB Programming is fun!" a hundred times, we can write

```
for i = 1:100
    % Use either disp or fprintf to print the statement
    fprintf('MATLAB Programming is fun!\n')
end
```

(b) Write a program that uses the for loop to print the elements of the vector $v$ = [1,2,3,4,5,6].

```
v = [1,2,3,4,5,6];
N = length(v);
for i = 1:N
    % Use either disp or fprintf
    fprintf('%d\n', v(i))
end
```

(c) Use the for loop to find and display the sum of elements of a vector, e.g., *v* = [1,2,3,4,5,6].

```
v = [1,2,3,4,5,6];
N = length(v);
% Define a variable to store the sum or total. Variables
% that store totals are normally initialized to zero
total = 0;
for i = 1:N
    total = total + v(i);
end

% Print the sum to the screen. Let's use fprintf
fprintf('The sum is %d\n', total)
```

(d) Using the for loop to populate an array

(i) Vector or 1D array: For a 10 element array for example, the elements can be filled in as follows:

```
v = zeros(1,10);

for i = 1:10
    v(i) = 2*i + 1;
end
```

(ii) Matrices or 2D array example: Here, we can use a nested for loop as follows.

```
A = zeros(3,4);
for i = 1:3
   for j = 1:4
      A(i,j) = i + j;
   end
end
disp('The 3 by 4 matrix A is')
disp(A)
```

**3.4.2.2.2. While Loops**

A while loop is condition-controlled loop that executes statements repeatedly as long as a condition remains true. The syntax for the while loop is as follows:

```
while <condition to be tested>
    statements to be executed
end
```

**While Loop Examples**

(a) Let's re-implement the original example of printing the statement "MATLAB Programming is fun!" a 100 times using a while loop.

```
% First, initialize a counter variable that will be used
% to control the loop
counter = 0;
while counter < 100
    fprintf('MATLAB Programming is fun!\n')
    counter = counter + 1;
end
```

(b) Write a MATLAB function that takes as input the number of students in a class prompts the user for the grades of the students. The program should calculate the total and average grades for the class.

```
function [total, average] = classgrade(number)
if number <= 0
    disp('Invalid argument')
    total = 0;
    average = 0;
    return
end

% Initialization phase
total = 0;
gradecounter = 1;

while gradecounter <= number
    grade = input('Enter grade: ');
    total = total + grade;
    gradecounter = gradecounter + 1;
end

average = total/number;
end
```

(c)