

Interactive Exercise: Validating Variable Names

For each variable name, decide if it is a valid and good variable name in MATLAB.

Numbers and Data Types in MATLAB

MATLAB supports a variety of data types for different kinds of data. Let's explore the most common ones:





What do you think this does?

```
A = zeros(5);
```

```
result = (3 + 5) * 2^2 / (1 + 1) - 1
```

Week 2: MATLAB Fundamentals

Objectives:

- Get familiar with variables and how they're stored in the workspace 
- Learn to work with arrays, vectors, and matrices – the building blocks of MATLAB 
- Learn operators, expressions, and statements to perform calculations 
- Get a sneak peek into basic plotting for data visualization 

Topics Covered:

1. Variables and the workspace
2. Arrays, vectors, and matrices
3. Operators, expressions, and statements
4. Basic plotting for data visualization
5. Vertical motion under gravity (example)

Variables and the Workspace

What are Variables?

- **Definition:** Variables are like labeled storage containers where you store information (like numbers or text) that you can use and modify later. They hold data that can be used in your MATLAB session.
- **Case Sensitivity:** MATLAB treats `var`, `Var`, and `VAR` as three different variables. Be careful with how you capitalize variable names!

Rules for Naming Variables:

- **Must Start with a Letter:** The first character of a variable name must be a letter (`a-z`, `A-Z`).
- **Can Contain Letters, Numbers, and Underscores:** After the first letter, you can use numbers (`0-9`) and underscores (`_`), but no spaces or special characters.
- **Case Sensitivity:** MATLAB is case-sensitive, meaning `MyVar` and `myvar` are different variables.
- **Reserved Words:** Avoid using MATLAB keywords (like `pi`, `clear`, `end`) as variable names. MATLAB will allow this in some cases, but it's good practice to be aware.

Naming Conventions:

- **camelCase**: Start with a lowercase letter, and capitalize the first letter of each subsequent word.
 - Example: `speedOfLight`
- **snake_case**: Use all lowercase letters, with words separated by underscores.
 - Example: `speed_of_light`
- **ALL_CAPS**: Use all uppercase letters, often used for constants.
 - Example: `SPEED_OF_LIGHT`

Example Code:

```
x = 5; % Assigns 5 to variable x  
y = 10; % Assigns 10 to variable y  
z = x * y; % Multiplies x and y, stores result in z  
GRAVITY = 9.8; % Descriptive variable name
```

1 mile

1 mile

1mile

clear

1mile

clear

1 mile

clear

MyVariable

1mile

clear

MyVariable

1mile

clear

MyVariable

x

1mile

clear

MyVariable

x

1mile

clear

MyVariable

x

DEBUG_ENABLED

1mile

clear

MyVariable

x

DEBUG_ENABLED

1. Double (Default Numerical Data Type)

- **Double-precision:** floating-point numbers, which store numbers with about 15 digits of precision.
- **Usage:** Suitable for most numerical computations due to their precision and range.

```
a = 3.14; % Example of a double
```

2. Integer Types (signed and unsigned)

- **Definition:** MATLAB supports signed and unsigned integer types of various sizes (8, 16, 32, 64 bits).
- **Usage:** Useful for precise control over memory usage and when working with hardware or file I/O operations.

```
c = int32(10); % Example of a 32-bit signed integer  
d = uint8(255); % Example of an 8-bit unsigned integer
```

4. Character Arrays (char)

- **Definition:** Character arrays represent text in MATLAB.
- **Usage:** Useful for storing and manipulating text data.

```
e = 'Hello, MATLAB!'; % Example of a char array
```

5. String Arrays

- **Definition:** A newer, more flexible way to handle text than char arrays, allowing for easier text manipulation.
- **Usage:** Preferred over char arrays for modern text handling.

```
f = "Hello, MATLAB!"; % Example of a string array
```

6. Logical (Boolean Values)

- **Definition:** Represents true **1** or false **0** values.
- **Usage:** Used in logical operations and conditional statements.

```
g = true; % Example of a logical value
```

6. Logical (Boolean Values)

- **Definition:** Represents true **1** or false **0** values.
- **Usage:** Used in logical operations and conditional statements.

```
g = true; % Example of a logical value
```

⚠ In reality, **0** is **false**, and EVERYTHING else is **true**

7. Complex Numbers

- **Definition:** Numbers with both real and imaginary parts.
- **Usage:** Essential for many engineering and scientific calculations.

```
h = 3 + 4i; % Example of a complex number
```


8. Structures (struct)

- **Definition:** Data type for storing collections of variables that can be different types.
- **Usage:** Useful for organizing data with different types, similar to records or structs in other languages.

```
student.name = 'Alice';  
student.age = 20;
```

9. Cell Arrays

- **Definition:** Arrays that can hold different types of data in each cell.
- **Usage:** Useful for storing mixed types of data or variable-sized arrays.

```
cellArray = {1, 'text', [1, 2, 3]};
```

Summary

- **Double** is the default for numbers.
- **Integer types** are for memory-efficient storage of integers.
- **Char** and **String** are for text handling.
- **Logical** is for boolean values.
- **Complex numbers** are for calculations involving imaginary components.
- **Structures** and **Cell Arrays** allow you to store multiple data types.

The Workspace

- **Workspace:** Contains current variables in the "environment".
- Use **who** to display a list of current variables.
- Use **whos** for a detailed list including data types and sizes.
- Use **clear** to remove variables from the workspace. You can specify certain variables or clear them all.

Example:

```
who          % Lists all active variables in the workspace
whos         % Detailed view of all variables
clear x      % Clears variable x from the workspace
clear        % Clears all variables from the workspace
```

Arrays, Vectors, and Matrices

Understanding Arrays, Vectors, and Matrices

Arrays: A collection of numbers or data arranged in rows and columns. Arrays can be one-dimensional (vectors) or multi-dimensional (matrices).

Vectors: A special case of arrays with only one row or one column.

Matrices: Two-dimensional arrays with multiple rows and columns.

Creating Arrays, Vectors, and Matrices:

Explicit Lists: You can create arrays, vectors, or matrices by explicitly listing their elements inside square brackets.

```
v = [1, 2, 3, 4, 5];    % Creates a row vector  
m = [1, 2; 3, 4];      % Creates a 2x2 matrix
```

Colon Operator: The colon operator (:) allows you to create vectors with evenly spaced elements.

```
v = 1:5;           % Creates vector [1, 2, 3, 4, 5]
v = 1:2:10;        % Creates vector [1, 3, 5, 7, 9] (start:step:end)
```

linspace and logspace Functions:

- **`linspace(start, end, n)`**: Generates n linearly spaced points between start and end.
- **`logspace(start_exp, end_exp, n)`**: Generates n logarithmically spaced points between $10^{\text{start_exp}}$ and $10^{\text{end_exp}}$.

```
v = linspace(1, 10, 5); % Creates vector [1, 3.25, 5.5, 7.75, 10]
v = logspace(1, 3, 3);  % Creates vector [10, 100, 1000]
```

Creating Arrays of Zeros & Ones

- **Zeros:** Use `zeros(numRows, numCols)` to create an array of zeros.
- **Ones:** Use `ones(numRows, numCols)` to create an array of ones.

Examples:

```
E = zeros(3, 5); % 3x5 array of zeros  
F = ones(2, 3);  % 2x3 array of ones
```


Without explicit row, col values, MATLAB assumes this value for both dimensions

Without explicit row, col values, MATLAB assumes this value for both dimensions

Without explicit row, col values, MATLAB assumes this value for both dimensions

Transposing Vectors

Transpose Operator ($'$): Converts a row vector to a column vector (or vice versa) by flipping it along its diagonal.

```
v = [1, 2, 3]; % Row vector  
vt = v';      % Column vector (transposed)
```

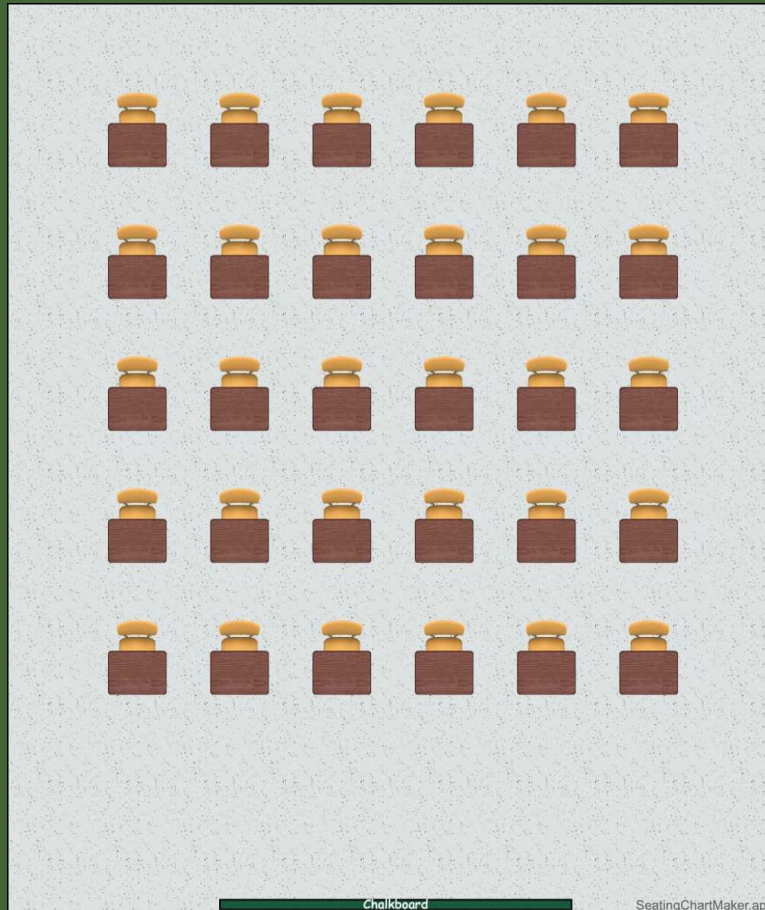
Accessing Elements in Arrays, Vectors, and Matrices

Subscripts: Allows you to access specific elements in an array, vector, or matrix using subscripts (indices).

```
v = [10, 20, 30];  
v(2)           % Accesses the second element (20)  
m = [1, 2; 3, 4];  
m(2, 1)        % Accesses element in the second row, first column (3)
```

Real-World Analogy:

Imagine a seating chart, you have rows and columns of seats.





Seat Finder

Each seat in the classroom has a unique position, just like a subscript in programming!

To find your seat, you need two pieces of information:

- **Row Number:** Tells you the row.
- **Seat Number:** Tells you the exact seat in that row.

In programming, subscripts work the same way. They help you find specific data items in a list or an array!



In Code: Lists and Arrays

Think of a list in programming like a single row of values.

```
% A list of favorite snacks
favoriteSnacks = ['Chips', 'Chocolate', 'Popcorn', 'Soda'];

% Accessing the second snack
secondSnack = favorite_snacks(2); % This is 'Chocolate'
```

The number **2** is the subscript here, just like the seat number example!

A 2D Example: Seat Map

What if the seating chart had both rows and columns? We could use two subscripts!

```
% A 2D seating chart (rows x columns)
seats = [
    'A1', 'A2', 'A3';
    'B1', 'B2', 'B3';
    'C1', 'C2', 'C3'
];

% Accessing seat in 2nd row, 3rd column
selected_seat = seats(2, 3); % This is 'B3'
```

Here, **(2, 3)** are the subscripts indicating the row and column.

Common Pitfall: Off-by-One Error

- Some programming languages start counting at 0, not 1.
- If you're not careful, you might end up in the wrong seat!

? Quick Quiz: Can You Find the Seat?

Look at this seat chart. Which seat is at position (2, 1)?

```
seat_chart = [  
    'VIP1', 'VIP2', 'VIP3';  
    'G1', 'G2', 'G3';  
    'R1', 'R2', 'R3'  
];  
  
% What seat is this?  
quiz_seat = seat_chart(2, 1);
```

? Quick Quiz: Can You Find the Seat?

Look at this seat chart. Which seat is at position (2, 1)?

```
seat_chart = [  
    'VIP1', 'VIP2', 'VIP3';  
    'G1', 'G2', 'G3';  
    'R1', 'R2', 'R3'  
];  
  
% What seat is this?  
quiz_seat = seat_chart(2, 1);
```

Answer: G2

Array Inception

- Arrays can be combined to create more complex structures.
- Use semicolons to stack arrays vertically and commas to combine horizontally.

Example:

```
A = [12, 18, -3];
```

```
B = [2, 5, 2;  
     1, 1, 2;  
     0, -2, 6];
```

```
C = [A; B];
```

```
D = [C, C];
```

Vectorization in MATLAB

Element-wise Operations: Use the dot (.) operator before arithmetic operators to perform operations element-by-element on arrays, vectors, or matrices

```
A = [1, 2, 3];  
B = [4, 5, 6];  
C = A .* B; % Element-wise multiplication  
D = A .^ 2; % Element-wise squaring
```

Interactive Exercise:

1. Create vectors A and B.
2. Perform element-wise multiplication and exponentiation.
3. Observe how vectorized operations differ from regular operations.

Arithmetic Operators $+$ $-$ \times \div

Basic Arithmetic:

```
a = 5;  
b = 2;  
c = a + b;    % Addition  
d = a - b;    % Subtraction  
e = a * b;    % Multiplication  
f = a / b;    % Division  
h = a ^ b;    % Exponentiation (a raised to the power b)
```


Operator Precedence (PEMDAS) **Order of Operations:**

- MATLAB follows the standard mathematical order of operations, known as PEMDAS
 1. Parentheses
 2. Exponents
 3. Multiplication and Division (left to right)
 4. Addition and Subtraction (left to right)

Tip 💡 : Always use parentheses to clarify the order in complex expressions.

```
result = (a + b) * (c - d); % Parentheses ensure correct order of operations
```

Hierarchy of Operations

- MATLAB follows PEMDAS for operations.
- **Left-to-right execution** after applying PEMDAS.

Example Breakdown:

```
c = 2 * 3^2 + 1/(1 + 2); % Step-by-step breakdown
c = 2 * 9 + 1/3;
c = 18 + 0.33333;
c = 18.33333;
```

Operator Precedence: PEMDAS Challenge 🧠

Predict the result of the following expression:

```
result = (3 + 5) * 2^2 / (1 + 1) - 1
```



Step 1: Calculate inside the parentheses.

$$(3 + 5) = 8$$

$$(1 + 1) = 2$$

Step 1: Calculate inside the parentheses.

$$(3 + 5) = 8$$

$$(1 + 1) = 2$$

Step 2: Apply exponentiation.

$$2^2 = 4$$

Step 1: Calculate inside the parentheses.

$$(3 + 5) = 8$$

$$(1 + 1) = 2$$

Step 2: Apply exponentiation.

$$2^2 = 4$$

Step 3: Perform multiplication and division (left to right).

$$8 * 4 = 32$$

$$32 / 2 = 16$$

Step 1: Calculate inside the parentheses.

$$(3 + 5) = 8$$

$$(1 + 1) = 2$$

Step 2: Apply exponentiation.

$$2^2 = 4$$

Step 3: Perform multiplication and division (left to right).

$$8 * 4 = 32$$

$$32 / 2 = 16$$

Step 4: Perform subtraction.

$$16 - 1 = 15$$

The result is: 15

Basic Plotting for Data Visualization

Introduction to Plotting

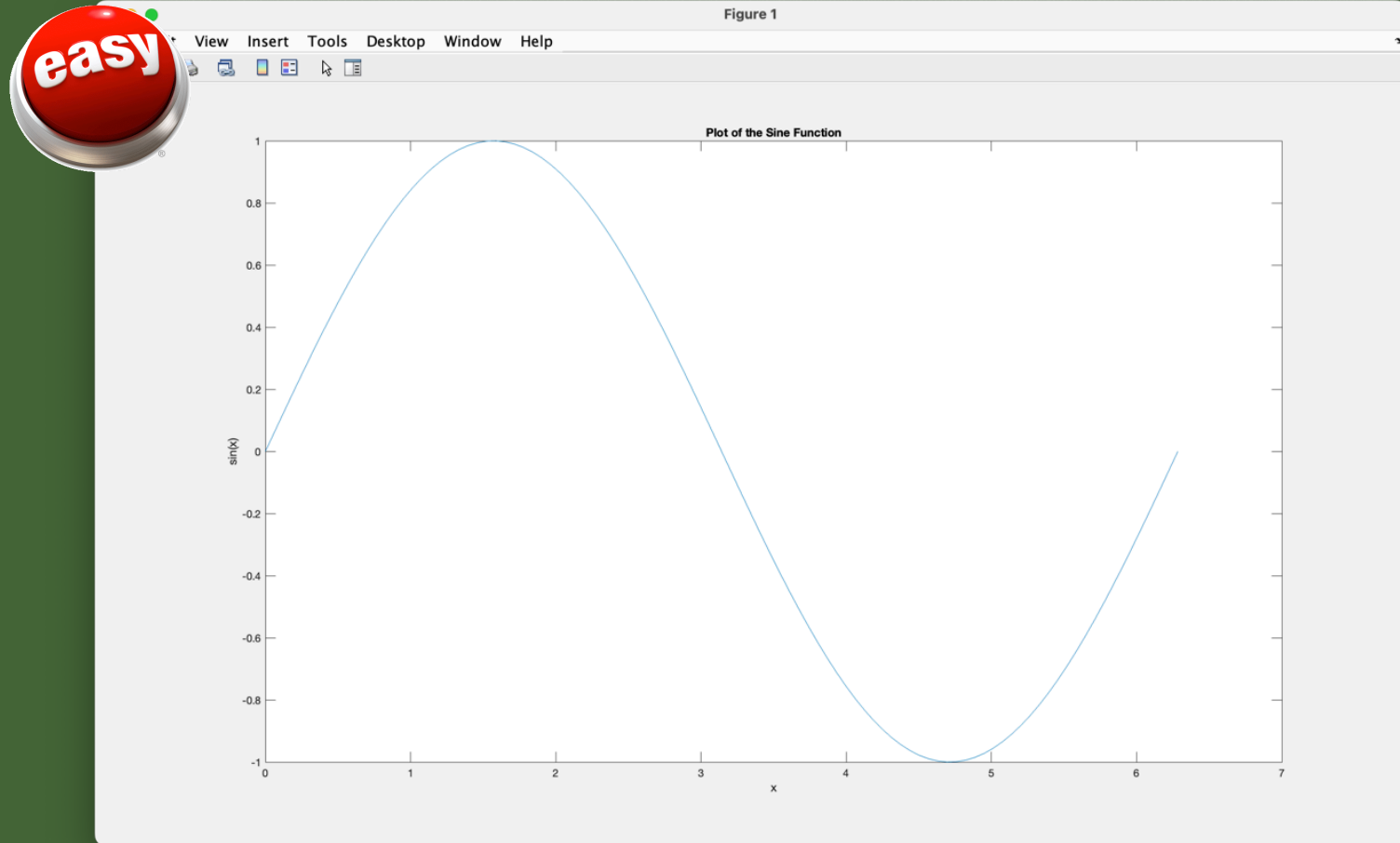
Why Plot?: Plotting helps you visualize your data, making it easier to understand trends, patterns, and outliers. It's like turning numbers into pictures!

Basic Commands:

- `plot(x, y)`: Creates a 2D line plot.
- `xlabel('x')`: Labels the x-axis.
- `ylabel('y')`: Labels the y-axis.
- `title('Title')`: Adds a title to the plot. Example Code:

```
x = linspace(0, 2*pi, 100);  
y = sin(x);  
plot(x, y);  
xlabel('x');  
ylabel('sin(x)');  
title('Plot of the Sine Function');
```

Tada



Vertical Motion Under Gravity Example

The Problem:

Calculate the vertical motion of an object under gravity – it's all about how things fall!

Approach:

1. Inputs - What data do you need to solve this problem

```
GRAVITY = 9.81;      % Acceleration due to gravity (m/s^2)
time = 0:0.1:10;    % Time vector from 0 to 10 seconds
v0 = 50;             % Initial velocity (m/s)
```

2. Manipulation - Perform operations to get to your destination

```
y = v0 * time - 0.5 * GRAVITY * time.^2; % Vertical position over time
```

3. Output - Produce clear and concise output Let's see what the raw output looks like

```
time
y
```

Not very user-friendly, let's plot this for a visual representation.



```
plot(time, y);  
xlabel('Time (s)');  
ylabel('Height (m)');  
title('Vertical Motion Under Gravity');
```



Gotchas

- Variables **are** case sensitive, do not try to use this to your advantage.
- There is a difference between `*` `/` `^` and `.*` `./` `.^`
- 0 is false but anything not zero is true
- MATLAB indexing starts at 1

Key Takeaways 🎓

- Arrays are the basis for all variables in MATLAB
- Arrays, vectors, and matrices are essential tools for handling lots of data at once .
- Though not so important now, the transpose operator will be used often later.
- Basic plotting is your way of turning data into visuals that are easier to understand and analyze .

Software Engineering