Interactive Exercise: Validating Variable Names 🧠

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

Numbers and Data Types in MATLAB 🧮

MATLAB supports various data types. Let's explore the most common ones for now:

```
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.
- **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 must be a letter (`a-z`, `A-Z`).
- **Can Contain Letters, Numbers, and Underscores** after the first letter. No spaces or other special characters.
- **Case Sensitivity**: `MyVar` ≠ `myvar`.
- **Reserved Words**: Avoid using MATLAB keywords (like `pi`, `clear`, `end`) as variable names.

Naming Conventions:

- **camelCase**: `speedOfLight`
- **snake_case**: `speed_of_light`
- **ALL_CAPS**: `SPEED_OF_LIGHT` (often used for constants)

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
```

1mile

1mile

1mile

clear

1mile

clear

1mile

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:** ~15 digits of precision.
- Suitable for most numerical computations.

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

2. Integer Types (signed and unsigned)

- MATLAB supports `int8`, `int16`, `int32`, etc.
- Useful for memory-efficient operations and hardware/file I/O scenarios.

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

3. Character Arrays (char)

- Represents text in MATLAB (older style).
- Useful for storing/manipulating short text data.

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

4. String Arrays

- Newer, more flexible for handling text.
- **Preferred** for modern text handling in current MATLAB versions.

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

5. Logical (Boolean Values)

- Represents true (`1`) or false (`0`).
- Used in conditional statements, if/else logic, etc.

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

5. Logical (Boolean Values)

- Represents true (`1`) or false (`0`).
- Used in conditional statements, if/else logic, etc.

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

⚠️ 0 is `false`, ANYTHING else is `true`.

6. Complex Numbers

- Numbers with real and imaginary parts (e.g. 3 + 4i).
- Essential for many engineering or scientific calculations.

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

7. Structures (struct)

- Used to store collections of variables of different types.
- Similar to records in other languages.

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

Heads-up: We'll explore `struct` in more detail around Week 8.

8. Cell Arrays

- Arrays that can hold different data types in each cell.
- Useful for variable-sized arrays or mixed types.

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

Heads-up: We'll discuss `cell` arrays when we do tables/import in a later week.

Summary of Common Data Types

- **Double** — the default for numbers.
- **Integer types** — memory-efficient integer usage.
- **Char** / **String** — text handling.
- **Logical** — boolean values (true/false).
- **Complex numbers** — handle imaginary parts.
- We’ll discuss **Struct** and **Cell Arrays** in future weeks.

The Workspace

- **Workspace**: Contains current variables in the environment.
- Use `who` to list variable names, `whos` for detailed listing.
- Use `clear` to remove variables from the workspace. You can specify certain variables or clear them all.

Example:

```
who          % Lists all active variables
whos         % Detailed view
clear x      % Clears variable x
clear        % Clears everything
```

Arrays, Vectors, and Matrices 📐

Understanding Arrays, Vectors, and Matrices

Arrays: A collection of data arranged in rows/columns. They can be 1D (vectors) or 2D+ (matrices).

Vectors: 1D arrays (either row or column).

Matrices: 2D arrays with multiple rows and columns.

Creating Arrays, Vectors, and Matrices:

Explicit Lists: Use square brackets with commas/semicolons.

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

Colon Operator: Creates evenly spaced elements.

```
v = 1:5;          % [1, 2, 3, 4, 5]
v = 1:2:10;       % [1, 3, 5, 7, 9]
```

linspace and logspace:

- `linspace(start, end, n)` → n linearly spaced points.
- `logspace(start_exp, end_exp, n)` → n log-spaced points between 10^(start_exp) and 10^(end_exp).

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

Creating Arrays of Zeros & Ones

- `zeros(r, c)` → r-by-c array of zeros.
- `ones(r, c)` → r-by-c array of ones.

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

What if we do `A = zeros(5)`? → Creates a 5x5 by default.

Transposing Vectors

Transpose Operator (`'`): flips row→column or column→row.

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

Accessing Elements via Subscripts

Use parentheses `( )` to access specific elements.

```
v = [10, 20, 30];
v(2)            % 20

m = [1, 2; 3, 4];
m(2, 1)         % 3 (2nd row, 1st column)
```

🔄 Real-World Analogy: Seating Chart

Rows and columns = subscript indices.

seat_chart(row, col)

A Quick Example: *Seat Finder*

```
seats = [
    'A1', 'A2', 'A3';
    'B1', 'B2', 'B3';
    'C1', 'C2', 'C3'
];

selected_seat = seats(2, 3);  % 'B3'
```

The pair `(2, 3)` indicates row 2, column 3.

❓ Quick Quiz: Off-by-One Error

MATLAB starts indexing at **1**, not 0. If you forget, you might pick the wrong seat!

Example of *Array Slicing & Reassignment*

You can select multiple elements and replace them at once:

```
v = [5, 6, 7, 8, 9];
v(2:4) = [99, 100, 101];
% Now v = [5, 99, 100, 101, 9]

m = [1, 2; 3, 4];
m(:, 2) = [9; 9];
% Now m = [1, 9; 3, 9]
```

In-Class Challenge: Array Manipulations

1. Create a **4x4 matrix** of random numbers using `rand`.
2. Replace the entire *second row* with zeros.
3. Extract the 1st and 3rd columns into a new matrix.
4. Use `disp` or `fprintf` to show the final result.

Try it out and see if you can navigate indexing carefully!

Common Error: Dimension Mismatch

Let's demonstrate a typical error when array sizes don't align.

```
A = [1, 2, 3];
B = [4, 5];          % Different size
C = A .* B;          % This will trigger an error about dimension mismatch
```

Common Error: Dimension Mismatch

Let's demonstrate a typical error when array sizes don't align.

```
A = [1, 2, 3];
B = [4, 5];        % Different size
C = A .* B;        % This will trigger an error about dimension mismatch
```

Debugging Tip: If you see “Array dimensions must agree,” check that `A` and `B` have compatible sizes for element-wise operations.

Vectorization in MATLAB

Element-wise Operations: Use `.` before `* / ^` to operate on each element independently.

```
A = [1, 2, 3];
B = [4, 5, 6];

C = A .* B;   % [4, 10, 18]
D = A .^ 2;   % [1, 4, 9]

% Compare to matrix multiplication (no dot):
% A * B  -> error unless dimensions allow standard linear algebra multiplication
```

Interactive Exercise:

1. Create vectors `A` and `B` of the same length.
2. Perform **element-wise multiplication** and exponentiation.
3. Observe the differences if you omit the dot.

Arithmetic Operators + − × ÷

Basic Arithmetic:

```
a = 5;
b = 2;
c = a + b;     % 7
d = a - b;     % 3
e = a * b;     % 10
f = a / b;     % 2.5
h = a ^ b;     % 25 (5^2)
```

Operator Precedence (PEMDAS)

1. Parentheses
2. Exponents
3. Multiplication and Division (left→right)
4. Addition and Subtraction (left→right)

Tip 💡: Use parentheses to avoid confusion.

```
result = (a + b) * (c - d);
```

Hierarchy of Operations

Example

```
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:

```
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 📈

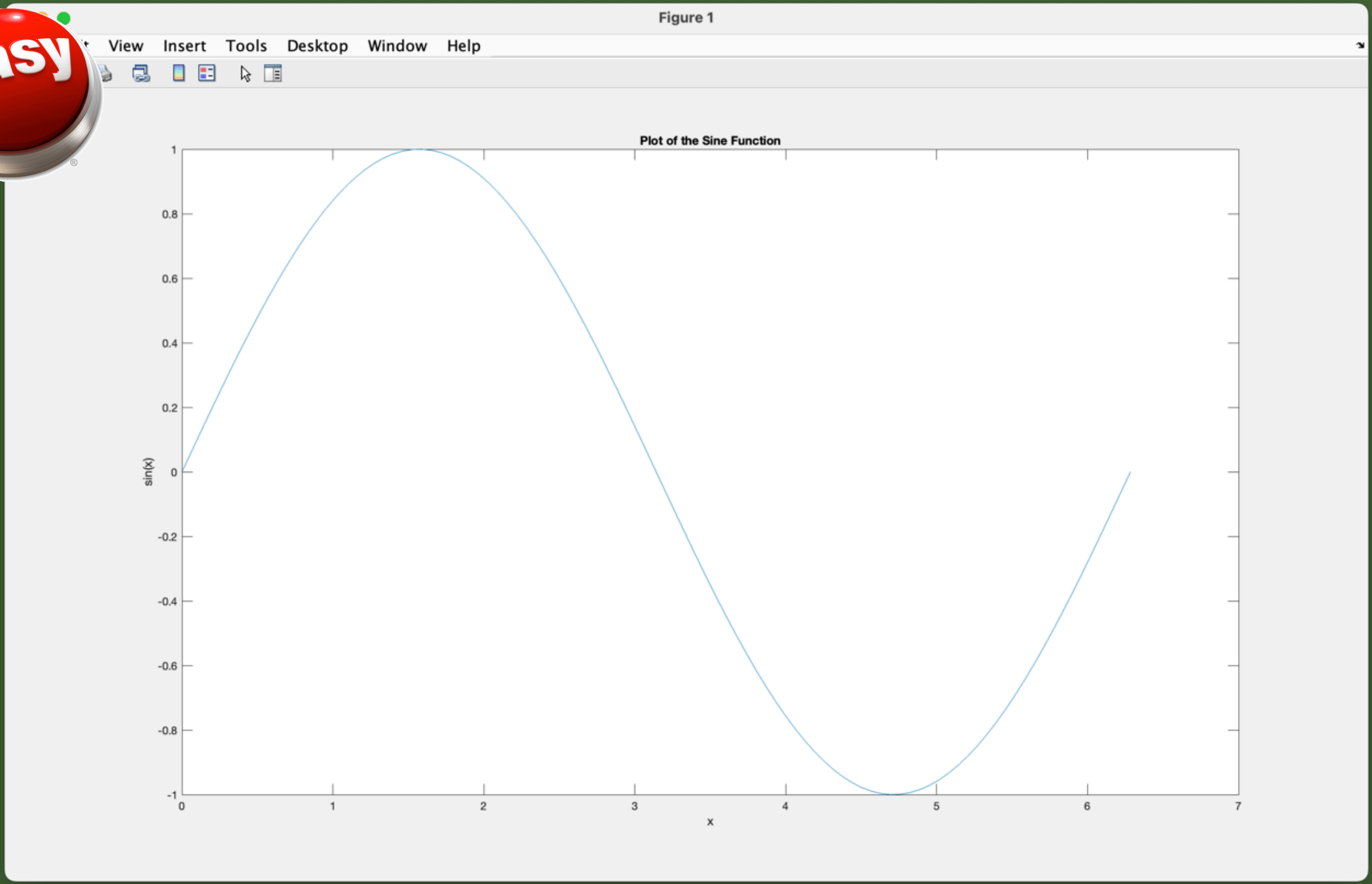
Introduction to Plotting

Why Plot?: Visualizing data helps spot trends, patterns, or outliers. It’s turning numbers into pictures.

- `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');
```

Basic Example



Vertical Motion Under Gravity Example 🌍

The Problem:

Calculate vertical motion of an object under gravity — how things fall.

Approach:

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

```
GRAVITY = 9.81;   % (m/s^2)
time = 0:0.1:10;  % 0 to 10s
v0 = 50;          % initial velocity (m/s)
```

2. **Manipulation - Perform operations to get to your destination**

```
y = v0 * time - 0.5 * GRAVITY * time.^2;
```

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

```
time
y
```

Raw output can be messy. Let's plot for clarity:

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

Gotchas

- Variables **are** case sensitive.
- Remember the difference between `* / ^` and `.* ./ .^`
- 0 is false but anything ≠ 0 is true.
- MATLAB indexing starts at 1.

Key Takeaways 🎓

- Arrays underlie almost everything in MATLAB.
- Vectors and matrices let you handle data in bulk 🧰.
- Be cautious with indexing and dimension mismatches!
- Basic plotting translates raw data into visual insights 📊.
- We'll delve deeper into advanced data types (struct, cell arrays) in later weeks.

Software Engineering