Review: Vector and Array Creation and Subscripting in MATLAB

1. Creating Vectors and Arrays

In MATLAB, vectors and arrays can be created using brackets `[]`.

Row Vector:

```
rowVector = [1, 2, 3, 4, 5];
```

Column Vector:

```
columnVector = [1; 2; 3; 4; 5];
```

2D Array (Matrix):

```
matrix = [1, 2, 3; 4, 5, 6; 7, 8, 9];
```

2. Accessing Elements (Subscripting)

You can access elements in arrays using indices `(row, column)` notation.

Example:

```
element = matrix(2, 3);  % Access element in 2nd row, 3rd column
```

Access Entire Row/Column:

- `matrix(:, 2)` – Access all rows from the 2nd column.
- `matrix(1, :)` – Access all columns from the 1st row.

3. Adding Values in Certain Positions

MATLAB allows you to assign values to specific positions in an array. If you assign a value to a position that does not yet exist, MATLAB will automatically create the necessary elements, filling in missing values with zeros.

Example:

```
array = [1, 2, 3];  % Initial array
array(5) = 10;      % MATLAB fills in positions 4 and 5 with 0 and 10
```

The resulting array would be:

```
[1, 2, 3, 0, 10]
```

4. Using `end` Keyword

The `end` keyword is useful when you want to access the last element of a vector or array, or to access all elements up to the last one.

Examples:

```
lastElement = array(end);  % Access the last element of the array
```

```
lastRow = array(end, :);  % Accesses all columns of the last row
```

```
lastFiveRows = array(end-4:end, :);  % Accesses the last 5 rows
```

```
fromSecondToLastCol = array(:, 2:end);  % Accesses columns from 2 to the last column
```

5. Other Useful Tips

- You can replace an entire row or column by assigning new values using the `:` notation.
- MATLAB automatically adjusts the size of vectors or matrices when new elements are added.

Week 4: Program Design 🛠️ and File Import/Export 💾

Objectives:

- Understand the program design process 🚧.
- Learn to create and use functions effectively 🔧.
- Import and export data in various formats 📂.

Topics Covered:

- Program Design and Algorithm Development 🧩
- MATLAB Functions 📜
- File Import and Export Utilities 📥📤

Program Design and Algorithm Development 🛣️

Program Design Process:

1. **Identify the Problem** 📌: Clearly define what you need to solve. Understand the requirements and constraints.
2. **Develop an Algorithm** 📝: Create a step-by-step plan using pseudocode or flowcharts.
    - *Pseudocode*: Writing down the steps in plain English or structured language.
    - *Flowcharts*: Visual diagrams representing the flow of the algorithm.
3. **Implement the Algorithm** 💻: Translate your algorithm into MATLAB code. Focus on writing clean and commented code.
4. **Test and Debug** 🐞: Run your code with various inputs to ensure it works correctly. Use debugging tools to fix issues.

💡 **Tip**: Break down complex problems into smaller, manageable parts. Think of it like building a LEGO set 🧱—one piece at a time!

Example: Projectile Problem 🎯

Problem:

Calculate the trajectory of a projectile launched at an initial speed and angle.

Example: Projectile Problem 🎯

Problem:

Calculate the trajectory of a projectile launched at an initial speed and angle.

1. **Define variables**:
   - Initial velocity `v0`
   - Launch angle `angle`
   - Acceleration due to gravity `g`

Example: Projectile Problem 🎯

Problem:

Calculate the trajectory of a projectile launched at an initial speed and angle.

2. **Compute components**:
   - Horizontal velocity `vx = v0 * cos(angle)`
   - Vertical velocity `vy = v0 * sin(angle)`

Example: Projectile Problem 🎯

Problem:

Calculate the trajectory of a projectile launched at an initial speed and angle.

3. **Calculate positions over time**:
    - Time array `time`
    - Horizontal position `x = vx * time`
    - Vertical position `y = vy * time - 1/2 * g * time^2`

Example: Projectile Problem 🎯

Problem:

Calculate the trajectory of a projectile launched at an initial speed and angle.

4. **Plot the trajectory**

Formulas:

- `x = v0 * cos(angle) * time`
- `y = v0 * sin(angle) * time - 1/2 * g * time.^2`

Implementation:

```
% Define variables
v0 = 50; % Initial velocity (m/s)
angle = 45; % Angle of projection (degrees)
g = 9.81; % Acceleration due to gravity (m/s^2)

% Compute components
vx = v0 * cosd(angle); % Horizontal component of velocity
vy = v0 * sind(angle); % Vertical component of velocity

% Time intervals
time_flight = 2 * vy / g; % Total flight time
time = linspace(0, time_flight, 100); % Time array

% Positions
x = vx * time; % Horizontal position
y = vy * time - 0.5 * g * time.^2; % Vertical position

% Plotting
plot(x, y);
xlabel('Distance (m)');
ylabel('Height (m)');
title('Projectile Trajectory');
grid on;
```

🤷 **Try it yourself!** Modify the `v0` and `angle` variables to see how they affect the trajectory. 🚀

Custom Functions 📜

Understanding Functions in MATLAB

Definition:

A function is a block of code designed to perform a specific task. Functions help make your code modular, reusable, and easier to manage.

Why Use Functions?

- **Modularity**: Break down complex programs into smaller, manageable pieces.
- **Reusability**: Write code once and use it multiple times.
- **Maintainability**: Easier to debug and update code.

Function Syntax Components

3. **Function Body**:
    - The code that performs the computations.
    - Uses input arguments to compute output arguments.
4. **End Statement**:

```
end
```

    - Indicates the end of the function.

Variable Scope

- **Local Variables**: Variables defined inside a function are not accessible outside of that function.
- **Input/Output Arguments**: Use these to pass data into and out of functions.
- **Global Variables**: Not recommended; can lead to code that's hard to debug.

  ```
  global myVar
  ```

💡 Tips

- Always manage data flow with input and output arguments to keep functions independent.
  - Avoid using global unless absolutely necessary
- Limit what your function does, to the name you give your function.
  - This keeps your functions small and easily manageable
- Suppress output nearly all of the time.
- Make sure your function can handle vectors where necessary
- If you are copying and pasting, you probably need a function instead

Say we have a function...

```
function greet(petName)
    disp(['Hello, ' petName '!']);
end
```

Now let's pass the following:

```
myDogsName = 'Buddy';
greet(myDogsName);
```

What do we get?

```
Hello, Buddy!
```

💡 **Tip**: It is important to know, the function is completely oblivious to all external code!

"*Write code that is easy to delete, not easy to extend*"

Jean-Paul Sartre

Example Function: Calculate Trajectory 🎯

Objective:

Create a function that calculates the trajectory of a projectile.

Steps:

1. **Create the Function File**
   - Name the file `calculate_trajectory.m`.
   - The function name must match the filename.

2. Write the Function Code

```
function [x, y] = calculate_trajectory(v0, angle, g)
% CALCULATE_TRAJECTORY computes the trajectory of a projectile.
%
%   [x, y] = CALCULATE_TRAJECTORY(v0, angle, g) returns the horizontal (x)
%   and vertical (y) positions of a projectile given:
%       v0    - initial velocity (m/s)
%       angle - launch angle (degrees)
%       g     - acceleration due to gravity (m/s^2)

    % Compute components
    vx = v0 * cosd(angle); % Horizontal component of velocity
    vy = v0 * sind(angle); % Vertical component of velocity

    % Time intervals
    total_time = 2 * vy / g; % Total flight time
    time = linspace(0, total_time, 100);  % Time array

    % Positions
    x = vx * time;  % Horizontal position
    y = vy * time - 0.5 * g * time.^2;  % Vertical position
end
```

3. **Save the Function**

    - Ensure the file is saved in your MATLAB working directory or in a directory that's on the MATLAB path.

Pro Tip!

- Sometimes it's easier to write the whole script, then write your function, watch this... 👀

```
% Define variables
v0 = 50; % Initial velocity (m/s)
angle = 45; % Angle of projection (degrees)
g = 9.81; % Acceleration due to gravity (m/s^2)

% Compute components
vx = v0 * cosd(angle); % Horizontal component of velocity
vy = v0 * sind(angle); % Vertical component of velocity

% Time intervals
time_flight = 2 * vy / g; % Total flight time
time = linspace(0, time_flight, 100); % Time array

% Positions
x = vx * time; % Horizontal position
y = vy * time - 0.5 * g * time.^2; % Vertical position

% Plotting
plot(x, y);
xlabel('Distance (m)');
ylabel('Height (m)');
title('Projectile Trajectory');
grid on;
```

👨🏽‍💻Interactive Presentation 📺

Using the Function:

```
% In your main script or Command Window
v0 = 50; % Initial velocity (m/s)
angle = 45; % Launch angle (degrees)
g = 9.81; % Acceleration due to gravity (m/s^2)

[x, y] = calculate_trajectory(v0, angle, g);

plot(x, y);
xlabel('Distance (m)');
ylabel('Height (m)');
title('Projectile Trajectory');
grid on;
```

⚠️ **Gotcha**: If MATLAB can't find your function, check that:

- The function name matches the filename (`calculate_trajectory.m`).
- The file is in the current directory or on the MATLAB path.

👩‍💻 **Exercise**: Modify the function to return the time array as an additional output.

Rounding Functions

MATLAB provides several functions for rounding numbers:

- `round(x)`: Rounds to the nearest integer.
- `floor(x)`: Rounds towards negative infinity.
- `ceil(x)`: Rounds towards positive infinity.
- `fix(x)`: Rounds towards zero.

Examples with Positive Numbers:

```
x = 16.3;
disp(round(x)); % Outputs 16
disp(floor(x)); % Outputs 16
disp(ceil(x));  % Outputs 17
disp(fix(x));   % Outputs 16
```

Examples with Negative Numbers:

```
x = -16.3;
disp(round(x)); % Outputs -16
disp(floor(x)); % Outputs -17
disp(ceil(x));  % Outputs -16
disp(fix(x));   % Outputs -16
```

💡 **Explanation**:

- `floor`: Rounds down to the nearest integer (towards negative infinity).
- `ceil`: Rounds up to the nearest integer (towards positive infinity).
- `fix`: Rounds towards zero (truncates the decimal part).

Trigonometric Functions

MATLAB can compute trigonometric functions in degrees or radians.

- To convert degrees to radians, you can use `deg2rad()`.
- Alternatively, use degree-specific functions: `sind`, `cosd`, `tand`.

Example using Degrees:

```
theta = 45; % Degrees
sin_theta = sind(theta); % Directly computes sine of 45 degrees
cos_theta = cosd(theta); % Cosine of 45 degrees
tan_theta = tand(theta); % Tangent of 45 degrees
```

💡 **Tip**: Using `sind`, `cosd`, and `tand` can simplify your code and prevent errors from unit conversion.

File Manipulation in MATLAB 📂

Checking if a File Exists

- Use the `exist()` function to check if a file exists in the current directory:

```
if exist('filename.txt', 'file') == 2
    disp('File exists');
else
    disp('File does not exist');
end
```

- `exist('filename.txt', 'file')`: Returns 2 if the file exists.
- The second argument `'file'` ensures that MATLAB is looking for a file, not a variable or folder.

Deleting a file

- Use the `delete()` function to remove a file:

```
if exist('filename.txt', 'file') == 2
    delete('filename.txt');
    disp('File deleted');
else
    disp('File not found');
end
```

Other Useful File Functions

- `dir()` Lists all files and directories in the current folder.

```
files = dir;
disp({files.name});
```

- `moveFile()` Moves or renames a file

```
movefile('oldname.txt', 'newname.txt');
```

- `copyFile()` Copies a file to a new location

```
copyfile('source.txt', 'destination.txt');
```

💡 Tips:

- Always check if a file exists before trying to delete or manipulate it to avoid errors.
- Use these functions for file management in scripts and when handling multiple files in projects.

Data Import and Export Utilities 📥📤

Importing Data:

MATLAB offers several functions to import data from various file types.

Common Functions:

- `readmatrix('filename.ext')`: Reads numeric data into a matrix.
- `readtable('filename.ext')`: Reads data into a table, which can handle mixed data types and includes headers.
- `readcell('filename.ext')`: Reads data into a cell array, useful for mixed data types.
- `textscan(fileID, formatSpec)`: Reads data from text files using format specifiers.

Default Behaviors and Optional Parameters

When importing data, MATLAB uses default settings that can be customized using optional parameters.

Default Behaviors and Optional Parameters

When importing data, MATLAB uses default settings that can be customized using optional parameters.

Default Behaviors:

- **Range**: By default, reads all data from the file.
- **Variable Names (Headers)**:
  - `readtable`: Assumes the first row contains headers (`'ReadVariableNames', true`).
  - `readmatrix` and `readcell`: Do not read headers by default (`'ReadVariableNames', false`).
- **Data Types**: MATLAB automatically detects data types based on the content.

Default Behaviors and Optional Parameters

When importing data, MATLAB uses default settings that can be customized using optional parameters.

Optional Parameters:

- **'Range'**: Specifies the subset of data to read.
- **'ReadVariableNames'**: Indicates whether to treat the first row as headers.
- **'Sheet'**: Specifies the worksheet in Excel files.
- **'VariableTypes'**: Defines data types for each column (used with import options).
- **'Delimiter'**: Specifies the character(s) that separate fields in text files.
- **'TreatAsMissing'**: Defines representations of missing data.

Specifying a Range:

You can specify a range to read specific rows and columns from your data file.

Syntax:

- `'Range', 'A2:C4'`: Reads data from cells A2 to C4.
- `'Range', [row_start, col_start, row_end, col_end]`: Alternative numeric indexing.

Example: Importing a Specific Range from Excel

```
% Import data from cells A2 to C4 in 'data.xlsx'
data = readmatrix('data.xlsx', 'Range', 'A2:C4');

% Display the imported data
disp(data);
```

💡 **Tip**: Specifying a range can speed up data import by reading only the necessary data.

Handling Headers (Variable Names):

Default Behavior:

- `readtable`: Assumes the first row contains variable names (`'ReadVariableNames', true`).
- `readmatrix`: Does not read variable names by default (`'ReadVariableNames', false`).

Specifying Headers:

- Use `'ReadVariableNames', true` or `false` to control whether the first row is treated as headers.

Example: Importing Data with Headers

```
% Import data including headers from 'data_with_headers.xlsx'
dataTable = readtable('data_with_headers.xlsx', 'Range', 'A1:C4', 'ReadVariableNames', true);

% Display the table
disp(dataTable);

% Access column headers (variable names)
variableNames = dataTable.Properties.VariableNames;
disp('Variable Names:');
disp(variableNames);
```

💡 **Tips**:

- Use `'ReadVariableNames', false` if your data does not include headers in the first row.
- These options have no specific order but must be provided in pairs.

Specifying Data Types for Each Column

By default, MATLAB attempts to automatically detect the data type for each column. However, you can specify data types to ensure correct interpretation.

Using Import Options:

1. **Create Import Options**:

```
opts = detectImportOptions('data.xlsx');
```

2. **Modify Variable Types**:

```
% Set 'Age' as double and 'Name' as string
opts = setvartype(opts, 'Age', 'double');
opts = setvartype(opts, 'Name', 'string');
```

3. **Use the Options with `readtable`**:

```
dataTable = readtable('data.xlsx', opts);
```

Example: Specifying Data Types

```
% Create import options
opts = detectImportOptions('data.xlsx');

% Modify variable types
opts = setvartype(opts, {'ID', 'Age'}, 'double');
opts = setvartype(opts, 'Name', 'string');
opts = setvartype(opts, 'EnrollmentDate', 'datetime');

% Specify date format if necessary
opts = setvaropts(opts, 'EnrollmentDate', 'InputFormat', 'MM/dd/yyyy');

% Read the data using the modified options
dataTable = readtable('data.xlsx', opts);

% Display the imported data
disp(dataTable);
```

💡 **Tip**: Specifying data types prevents errors due to incorrect automatic detection, especially with mixed data types.

Accessing Metadata and Additional Information

After importing data, you might want to know more about it.

Getting the Size of the Data:

- For matrices (`readmatrix`):

```
[numRows, numCols] = size(data);
```

- For tables (`readtable`):

```
numRows = height(dataTable);
numCols = width(dataTable);
```

Accessing Variable Names (Headers):

- For tables:

```
variableNames = dataTable.Properties.VariableNames;
```

Checking Data Types of Variables:

```
varTypes = varfun(@class, dataTable, 'OutputFormat', 'cell');
varNames = dataTable.Properties.VariableNames;

% Display variable names and their types
for i = 1:length(varNames)
    fprintf('Variable: %s, Type: %s\n', varNames{i}, varTypes{i});
end
```

Checking for Missing Data:

```
% For matrices
numMissing = sum(isnan(data), 'all');
fprintf('Number of missing values: %d\n', numMissing);

% For tables
numMissing = sum(sum(ismissing(dataTable)));
fprintf('Number of missing values: %d\n', numMissing);
```

⚠️ **Gotcha**: Be cautious when your data contains missing values (`NaN` or empty cells). They can affect calculations.

💡 **Tip**: Notice `sum(sum())`? It's a quick way to sum across 2D data.

Working with Tables in MATLAB

1. Creating Tables Manually

```
Names = {'Alice'; 'Bob'; 'Charlie'};
Ages = [20; 22; 21];
dataTable = table(Names, Ages, 'VariableNames', {'Name','Age'});
disp(dataTable)
```

- Tables can contain mixed data types in different columns.
- You can specify column (variable) names using `'VariableNames'`.

2. Adding/Removing Columns

```
% Add a new column
dataTable.Height = [1.65; 1.80; 1.72];

% Remove a column
dataTable = removevars(dataTable, 'Height');  % remove 'Height' column
```

3. Indexing Tables

- **Dot-notation**: `dataTable.Age` returns the Age column as an array.
- **Brace-index**: `dataTable{:, 'Age'}` also returns the numeric array for 'Age'.

```
% Dot notation
ageArray = dataTable.Age;

% Brace indexing
ageArray2 = dataTable{:, 'Age'};
```

💡 **Tip**: Dot notation is convenient for referencing columns by their variable name.

4. Renaming Variables

```
dataTable.Properties.VariableNames{'Age'} = 'YearsOld';
```

- Now `dataTable.YearsOld` is valid.

5. Summaries and Basic Operations

- **summary(dataTable)**: Provides a statistical summary (min, max, etc.) for each column.
- You can do **logical indexing** on table columns:

```
adultsOnly = dataTable(dataTable.YearsOld >= 18, :);
```

  - Great for quickly filtering or subsetting table rows.

Why Tables?

- Keep column names attached to data, making code more readable.
- Handy for data analysis or after importing CSV files with headers.
- Mix numeric, text, or categorical data in the same container.

Reading Text Files with `textscan`

`textscan` allows you to read data from text files by specifying a format specifier, similar to `fprintf`.

Syntax:

```
fileID = fopen('filename.txt', 'r');
dataArray = textscan(fileID, formatSpec, 'Delimiter', delimiter, 'HeaderLines', N);
fclose(fileID);
```

- The second argument `'r'` in fopen() says we want to open this as **read only**
- **`formatSpec`**: A string that specifies the format of each column (e.g., `'%f %s %d'`).
- **`delimiter`**: Character(s) that separate fields (e.g., `','` for CSV files).
- **`HeaderLines`**: Number of lines to skip at the beginning of the file.

Example: Reading a Text File with Specified Formats

Suppose you have a text file `data.txt` with the following content:

```
ID,Name,Age,Score
1,Alice,20,85.5
2,Bob,22,90.0
3,Charlie,21,88.0
```

```
% Open the file
fileID = fopen('data.txt', 'r');

% Define the format specifier
formatSpec = '%d %s %d %f';

% Read the data, skipping the header line
dataArray = textscan(fileID, formatSpec, 'Delimiter', ',', 'HeaderLines', 1);

% Close the file
fclose(fileID);

% Access the data
IDs = dataArray{1};
Names = dataArray{2};
Ages = dataArray{3};
Scores = dataArray{4};

% Display the data
disp('IDs:');
disp(IDs);
disp('Names:');
disp(Names);
disp('Ages:');
disp(Ages);
```

💡 **Tip**: Use `%*s` can be used to skip a field.

Handling Delimiters and Headers in `textscan`

Specifying Delimiters:

- Use the `'Delimiter'` parameter to specify the character(s) that separate fields.
- Common delimiters:
  - Comma-separated values: `','`
  - Tab-separated values: `'\t'`
  - Space-separated values: `' '`

Skipping Header Lines:

- Use `'HeaderLines', N` to skip `N` lines at the beginning of the file.
- Useful when your file contains headers or metadata.

💡 **Tip**: Always close the file after reading using `fclose(fileID);` to free system resources.

Exporting Data:

Common Functions:

- `writematrix(data, 'filename.ext')`: Writes matrix data to a file.
- `writetable(dataTable, 'filename.ext')`: Writes table data to a file.
- `writecell(dataCell, 'filename.ext')`: Writes cell array data to a file.
- `fprintf(fileID, formatSpec, variables)`: Writes formatted data to a text file.

Specifying Options When Exporting

- **Default Behavior**:
  - Writes data starting at cell A1.
  - For tables, includes variable names (headers) by default.
- **Optional Parameters**:
  - **'Sheet'**: Specifies the worksheet in Excel files.
  - **'Range'**: Specifies the starting cell for writing data.

Example: Exporting Data to Excel

```
% Export matrix data to 'output.xlsx' starting at cell A2
writematrix(data, 'output.xlsx', 'Sheet', 'DataSheet', 'Range', 'A2');

% Export table data to 'output_with_headers.xlsx'
writetable(dataTable, 'output_with_headers.xlsx', 'Sheet', 1, 'Range', 'A1');
```

💡 **Tip**: When exporting, ensure the file extension matches the desired format (e.g., `.csv`, `.xlsx`).

Exporting Data Using Format Specifiers 

Using `fprintf` to Write Formatted Data

```
fileID = fopen('filename.txt', 'w');
fprintf(fileID, formatSpec, variables);
fclose(fileID);
```

- **`formatSpec`**: Format of each variable (e.g., `'%d %s %f\n'`). ⚠️ **Gotcha**: Notice the `fileID` in `fprintf`?

Example: Exporting Data to a Text File

```
% Sample data
IDs = [1; 2; 3];
Names = {'Alice'; 'Bob'; 'Charlie'};
Scores = [85.5; 90.0; 88.0];

% Open the file
fileID = fopen('output.txt', 'w');

% Define format specifier
formatSpec = '%d %s %.1f\n';

% Write data
for i = 1:length(IDs)
    fprintf(fileID, formatSpec, IDs(i), Names{i}, Scores(i));
end

% Close the file
fclose(fileID);
```

💡 **Tip**: Use loops to write each row of data.

Writing CSV Files

```
% Open the file
fileID = fopen('output.csv', 'w');

% Write headers
fprintf(fileID, 'ID,Name,Score\n');

% Define format specifier with commas
formatSpec = '%d,%s,%.1f\n';

% Write data
for i = 1:length(IDs)
    fprintf(fileID, formatSpec, IDs(i), Names{i}, Scores(i));
end

% Close the file
fclose(fileID);
```

💡 **Tip**: Use `writetable` for easier CSV exports when working with tables.

Practical Tips for Data Import/Export

- **Use Import Options for Flexibility**: `detectImportOptions('filename')` and related functions provide control over how data is imported.
- **Check Data After Importing**: Verify the data structure and types to prevent errors in subsequent analysis.
- **Handle Missing Data**: Be aware of how missing data is represented and handled.
- **Consistent Formatting**: Ensure your data files have consistent formatting to simplify import/export processes.
- **Use the Import Wizard**: MATLAB's **Import Wizard** provides a graphical interface to import data and can generate MATLAB code based on your selections.

💡 **Tip**: **Using the Import Wizard**

- **Access**: Go to **Home > Import Data** or double-click the file in the Current Folder browser.
- **Features**:
  - Interactively select data ranges, specify variable types, and handle headers.
  - Preview data before importing.
- **Generate Code**:
  - After setting options, click on **Import Selection** dropdown and select **Generate Function** or **Generate Script**.
  - This helps in learning the syntax and creating reusable code.

Additional Parameters and Tips

Additional Parameters and Tips

Specifying Sheet Names:

- Use the `'Sheet'` parameter to specify the sheet name or index.
- Example:

```
data = readmatrix('data.xlsx', 'Sheet', 'Sheet2');
```

Additional Parameters and Tips

Reading from Text Files:

- Use `readtable` or `readmatrix` for CSV and TXT files.
- Example:

```
data = readtable('data.txt', 'Delimiter', '\t', 'ReadVariableNames', false);
```

Additional Parameters and Tips

Ignoring or Handling Missing Data:

- Use the `'TreatAsMissing'` parameter.
- Example:

```
dataTable = readtable('data.csv', 'TreatAsMissing', 'NA');
```

Additional Parameters and Tips

Handling Large Files:

- **Use `textscan` for Efficiency**: `textscan` can be more efficient for reading large text files with known formats.
- **Chunk Reading**: For extremely large files, consider reading data in chunks.

💡 **Tip**: Document any special parameters or options used during import/export to make your code easier to understand and maintain.

Key Takeaways 🎓

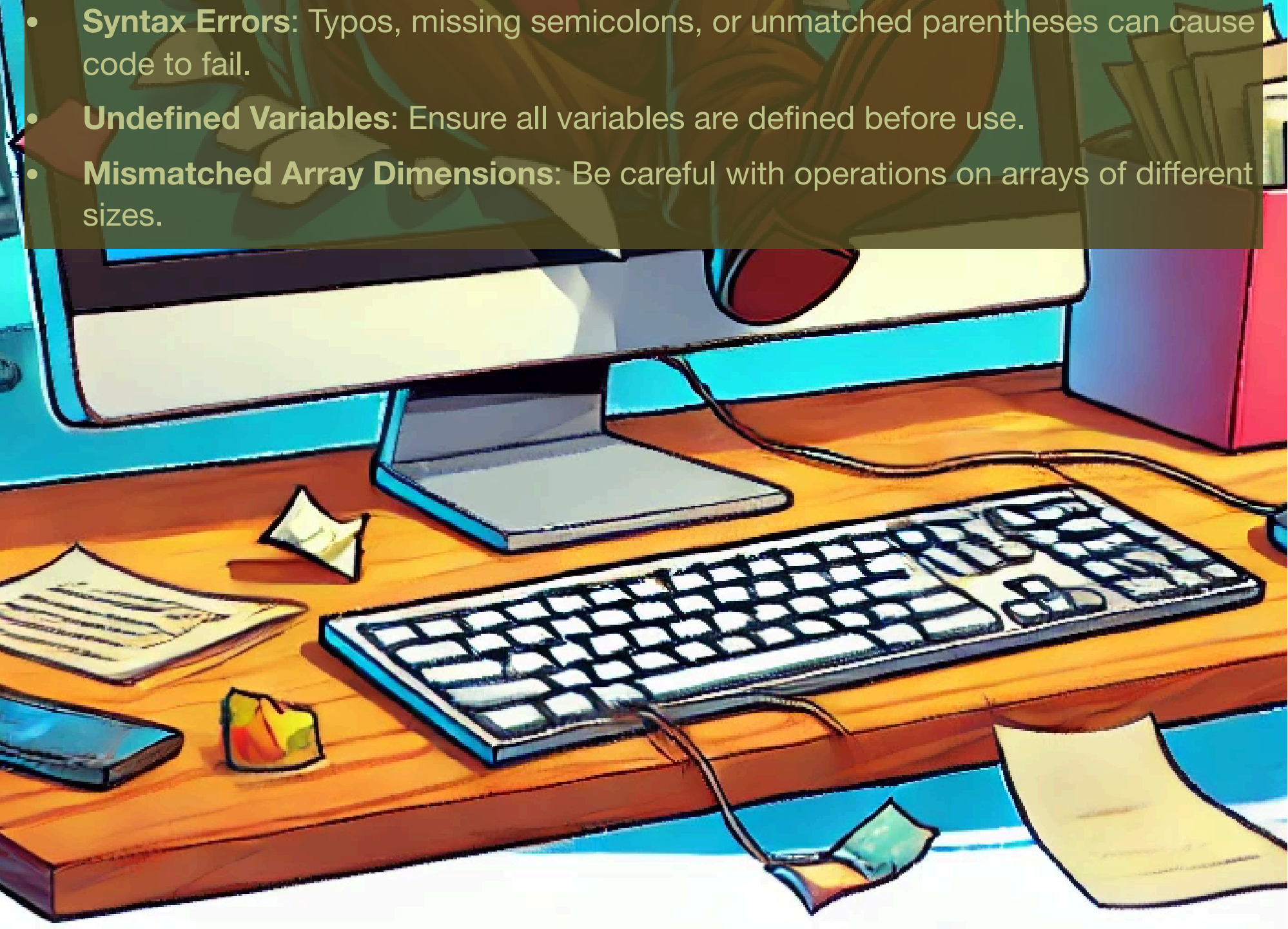
- **Program Design**: Involves identifying problems, developing algorithms using pseudocode or flow charts, implementing solutions, and testing/debugging.
- **Functions**: Modularize code, making it reusable and easier to maintain. Remember to document your functions!
- **Data Import/Export**: MATLAB provides powerful tools for importing and exporting data in various formats. Use the appropriate function for your data type.


GOTCHA!
Gotcha!

Gotchas:

- **Function Naming**: Ensure the function name matches the filename (e.g., `calculate_trajectory.m`).
- **Input/Output Arguments**: Verify the correct number and type of input/output arguments when calling functions.
- **Data Formats**: Ensure the data format matches the expected format when importing/exporting. Inconsistent formats can lead to errors.
- **Units**: Always check if functions expect angles in degrees or radians to avoid calculation errors.
- **Tables**: If a CSV has partial numeric columns and partial text, you might get unexpected data types. Double-check with `summary()` or `varfun()`.
- **Table Variable Names**: If a file's header row has strange characters or duplicates, MATLAB might rename them automatically. Check `Properties.VariableNames` after import.

Common Errors:



- **Syntax Errors**: Typos, missing semicolons, or unmatched parentheses can cause code to fail.
- **Undefined Variables**: Ensure all variables are defined before use.
- **Mismatched Array Dimensions**: Be careful with operations on arrays of different sizes.

Software Engineering

- codecademy.com
- "The Pragmatic Programmer"

Heading 1

Heading 2

Heading 3

Heading 4

Heading 5

```
fprintf('some stuff with a string (%s)', "heyo");
```

Using inline `code` blocks