Week 3: Scripts, Functions, Basic Input/Output

Objectives:

- Write and run scripts 📝.
- Understand the difference between scripts and functions 🤔.
- Create and call functions with parameters 🛠️.
- Understand control flow and conditional statements 🔄.
- Use `fprintf` and `sprintf` for formatted output 🎯.

Topics Covered:

- Scripts and Functions 📜
- Control Flow and Conditional Statements ⚙️
- Input and Output 📥📤

Scripts and Functions

Scripts:

Definition: A script is a file containing a series of MATLAB commands. Scripts run in the current workspace and can use and modify existing variables.

Creating and Running Scripts: To create a script, write the commands in a new file with a `.m` extension. Run the script by typing its name (without the `.m` extension) in the Command Window.

Example Script:

Create a file named `area_of_circle.m`:

```
radius = input("Please enter the radius: ");
area = pi * radius^2;
disp(['The area of the circle is ', num2str(area)]);
```

💡 **Tip**: Use descriptive names for your scripts to easily remember what they do!

Functions:

Definition: Functions are reusable pieces of code that operate on inputs and produce outputs. Functions have their own workspace, separate from the base workspace.

Components: Name, input arguments, and output arguments.

Creating Functions: Define a function in a new file with the same name as the function and a `.m` extension. Use the `function` keyword to define a function.

Syntax:

```
function output = functionName(input)
    % Function body
    output = ...;
end
```

Example Function:

Create a file named `calculate_area.m`:

```
function area = calculate_area(radius)
    area = pi * radius^2;
end
```

Execution:

```
result = calculate_area(5);
```

⚠️ Gotcha: Remember, functions have their own workspace! Variables created within a function do not affect the base workspace unless explicitly returned.

Input and Output 📥📤

Input:

Static/Hard-coded Input:

Static input refers to assigning values directly within the code using the equal sign.

```
radius = 5;
```

Dynamic Input:

Use the `input()` function to get user input during code execution. This is useful when you need to ask for values interactively.

```
radius = input('Please enter the radius: ');
```

Example:

In this example, we calculate the final height of a falling object based on user input.

```
height0 = input('Please enter the initial height (in meters): ');
time = input('Please enter the time the object was in the air (in seconds): ');
velocity0 = input('Please enter the initial velocity (in m/s): ');
ACC = 9.81;

heightFinal = height0 + velocity0 * time + 0.5 * ACC * time^2;
disp(['The final height is ', num2str(heightFinal), ' meters.']);
```

```
The final height is X meters
```

💡 **Tip**: Always prompt users clearly so they know what type of input is expected.

String Input:

Inputting Strings: To receive string input, use the `'s'` argument.

```
name = input('Please enter your name: ', 's');
disp(['Hello, ', name, '!']);
```

Example:

Here's an example that uses the string input to greet the user with `fprintf`:

```
username = input('Please enter your name: ', 's');
fprintf('You said your name was %s, hello %s!\n', username, username);
```

```
You said your name was John, hello John!
```

Output:

Capturing Output with `diary`:

The `diary` function can capture all Command Window output and save it to a text file. This is useful for long sessions where you want to save output for later review.

```
diary 'output.txt'
disp('Hello, World!');
diary off
```

💡 **Tip:** Use the `diary` function when you want to log output for debugging or record-keeping.

Formatting Output:

disp:

Used for displaying values without much formatting control. It is best for quick checks and simple output needs.

```
someVariable = 25;
disp('The value is: ');
disp(someVariable);
```

```
The value is:
    25
```

fprintf:

`fprintf` provides formatted output, allowing for more control over how numbers and strings are displayed. Always remember to include `\n` for newlines when needed.

```
fprintf('The value is %d\n', someVariable);
```

```
The value is 25
```

sprintf:

`sprintf` works like `fprintf`, but instead of displaying the output, it returns a formatted string. This is useful when you need to store the formatted text for further processing.

```
str = sprintf('The value is %d', someVariable);
disp(str);
```

```
The value is 25
```

sprintf:

`sprintf` works like `fprintf`, but instead of displaying the output, it returns a formatted string. This is useful when you need to store the formatted text for further processing.

```
str = sprintf('The value is %d', someVariable);
disp(str);
```

```
The value is 25
```

💡 **Tip:** Did you catch what happened here, two output calls result in one line?

Using Format Specifiers:

`fprintf` and `sprintf` use format specifiers to control the output format. These specifiers determine how data types like integers, floating-point numbers, and strings are displayed.

- `%d` or `%i`: Integer
- `%f`: Floating-point number
- `%s`: String
- `%g`: Compact floating-point format
- `%5.2f`: Floating-point number with 5 characters wide and 2 digits after the decimal point
- `\n`: Newline character

Using `fprintf`:

Here's how `fprintf` works to store formatted text:

```
fprintf('Integer: %d\n', 42);
fprintf('Floating-point: %8.2f\n', 3.14159);
fprintf('String: %s\n', 'Hello, world!');
```

```
Integer: 42
Floating-point:     3.14
String: Hello, world!
```

Another Example:

```
fprintf("Hi");
fprintf(", my name is");
fprintf(", my name is");
fprintf(", my name is");
fprintf(", chka chka Slim Shady\n");
```

⚠️ **Warning:** Don't forget `\n` or you'll end up with a single line

Using `fprintf`:

Here's how `fprintf` works to store formatted text:

```
fprintf('Integer: %d\n', 42);
fprintf('Floating-point: %8.2f\n', 3.14159);
fprintf('String: %s\n', 'Hello, world!');
```

```
Integer: 42
Floating-point:     3.14
String: Hello, world!
```

Another Example:

```
fprintf("Hi");
fprintf(", my name is");
fprintf(", my name is");
fprintf(", my name is");
fprintf(", chka chka Slim Shady\n");
```

```
Hi, my name is, my name is, my name is, chka chka Slim Shady
```

⚠️ **Warning:** Don't forget `\n` or you'll end up with a single line

Using `sprintf`:

Here's how `sprintf` works to store formatted text:

```
str = sprintf('The value is %8.2f', 3.14159);
disp(str);
fprintf("%s", str);
```

```
The value is     3.14
The value is     3.14
```

Another Example:

Here's an example where we use `sprintf` to generate a filename with an incremented value.

```
fileIncrement = 1;
fileName = sprintf("My file (%i)", fileIncrement);
disp(fileName); % Outputs: My file (1)
```

```
My file (1)
```

Comparative/Relational Operators:

How to define a condition

Operators: Used to compare values.

- `==` : Equal to
- `~=` : Not equal to
- `<` : Less than
- `>` : Greater than
- `<=` : Less than or equal to
- `>=` : Greater than or equal to

Example:

```
a = 5;
b = 3;
if a > b
    disp('a is greater than b');
end
```

⚠️ Gotcha: Don’t confuse `=` (assignment) with `==` (equality). Using `=` in a condition can lead to unexpected results!

Logical Operators:

Operators: Used to combine multiple conditions.

- `|` : OR
- `&` : AND
- `~` : NOT
- `||` : Short-circuit OR (stops evaluating if the first condition is true)
- `&&` : Short-circuit AND (stops evaluating if the first condition is false)

Example:

```
x = true;
y = false;
if x & ~y
    disp('x is true and y is false');
end
```

Simple Example: Conditions

Think of real-life checks, e.g., "`(isHungry && hasTime)` => `consumeFood()`." or "`(weather == RAIN || weather == SNOW)` => `bring umbrella`."

Control Flow and Conditional Statements ⚙️

Control flow:

Control statements allow you to execute certain sections of code based on conditions or repeat them a specified number of times. This includes conditional statements and loops.

Conditional Statements:

Determine whether a condition is true or false.

Syntax:

```
if condition
    % Code to execute if condition is true
elseif another_condition
    % Code to execute if another_condition is true
else
    % Code to execute if none of the above are true
end
```

Example:

```
num = input('Enter a number: ');
if num > 0
    disp('The number is positive.');
elseif num < 0
    disp('The number is negative.');
else
    disp('The number is zero.');
end
```

💡 Tip: Use `elseif` to handle multiple conditions without having to nest multiple `if` statements.

🔄 Analogy:

Think of an `if-elseif-else` statement as deciding what to eat based on your mood.

`if` I'm very hungry, eat a meal.

`elseif` I'm a little hungry, grab a snack.

`elseif` I'm thirsty, grab a drink

`else` (I'm not hungry) don't eat.

💡 **Note**: `elseif` and `else` are not always necessary

💡 **Note**: you can have many `elseif`s

Switch Statement:

Definition: A `switch` statement allows you to select one of many code blocks to execute.

Syntax:

```
switch expression
    case value1
        % Code to execute if expression == value1
    case value2
        % Code to execute if expression == value2
    otherwise
        % Code to execute if expression does not match any case
end
```

Example:

```
day = input('Enter a day number (1-7): ');
switch day
    case 1
        disp('Sunday');
    case 2
        disp('Monday');
    case 3
        disp('Tuesday');
    case 4
        disp('Wednesday');
    case 5
        disp('Thursday');
    case 6
        disp('Friday');
    case 7
        disp('Saturday');
    otherwise
        disp('Invalid day number');
end
```

💡 Tip: Use `switch` statements for clean, easy-to-read code when dealing with multiple specific cases.

Example 4: Two Ways to Traverse a List with an Index

Approach 1: Use a range for loop

```
my_list = [10, 20, 30, 40, 50];
for i = 1:length(my_list)
    fprintf('Element %i is %i\n', i, my_list(i));
end
```

Explanation:

In this example, `i` takes the values from 1 to the length of `my_list`. Each iteration accesses the `i`-th element in the list, and prints both the index and the element. This approach automatically increments the index after each iteration.

Example 4: Two Ways to Traverse a List with an Index

Approach 2: Manually Incrementing an Index

```
my_list = [10, 20, 30, 40, 50];
i = 1;
for value = my_list
    fprintf('Element %i is %i\n', i, value);
    i = i + 1;  % Manually increment the index
end
```

Explanation:

Here, we have to manually increment the index `i`. On each iteration, the code accesses the i-th element of `my_list`, then increments `i` by 1. This approach offers more flexibility if you want to adjust how or when the index increments (for example, skipping elements or using a dynamic step size).

🔄 Real-World Analogy:

A `for` loop is like counting the number of laps you run around a track. You know you need to run 5 laps, so you repeat the same action 5 times.

`while` Loop:

Definition: Repeats a block of code while a condition is true.

Syntax:

```
while condition
    % Code to execute while condition is true
end
```

Example:

```
i = 1;
while i <= 5
    fprintf('Iteration: %i', i);
    i = i + 1;
end
```

⚠️ **Gotcha**: Be careful with while loops! Ensure the condition will eventually be falsy to avoid infinite loops.

🔄 Real-World Analogy:

A `while` loop is like filling a glass with water. You keep pouring `while (glass ~= isFull)`. You don’t know exactly how much water it will take, but you stop when the condition (full glass) is met.

Break and Continue:

`break`: Exits the loop immediately.

`continue`: Skips the rest of the code in the current iteration and proceeds to the next iteration.

⚠️ **Warning**: using `break` is generally considered a "code smell". If you find yourself needing to use `break` there is generally something you're doing wrong.

Using `continue` is not "great" either but there are more acceptable use cases for `continue`

Example (Break):

At first glance, this will loop 10 times. But break stops after 5.

```
for i = 1:10
    if i == 5
        break;
    end
    disp(i);
end
```

```
1
2
3
4
```

Example (Continue):

At first glance, this will also loop 10 times. But a condition skips even numbers.

```
for i = 1:10
    if mod(i, 2) == 0
        continue;
    end
    disp(i);
end
```

```
1
3
5
7
9
```

⚠️ **Warning:** Try to avoid `break` and `continue` when possible to avoid creating code that's difficult to follow.

🔄 Real-World Analogy:

`break`: Imagine you are searching for a specific book in a library. Once you find it, you stop searching and leave (exit the loop).

`continue`: Imagine you are checking a list of items for defects. If you find a defective item, you skip further inspection of that item and move to the next (skip iteration).

Drive it home ⚾

Use `for` loops when

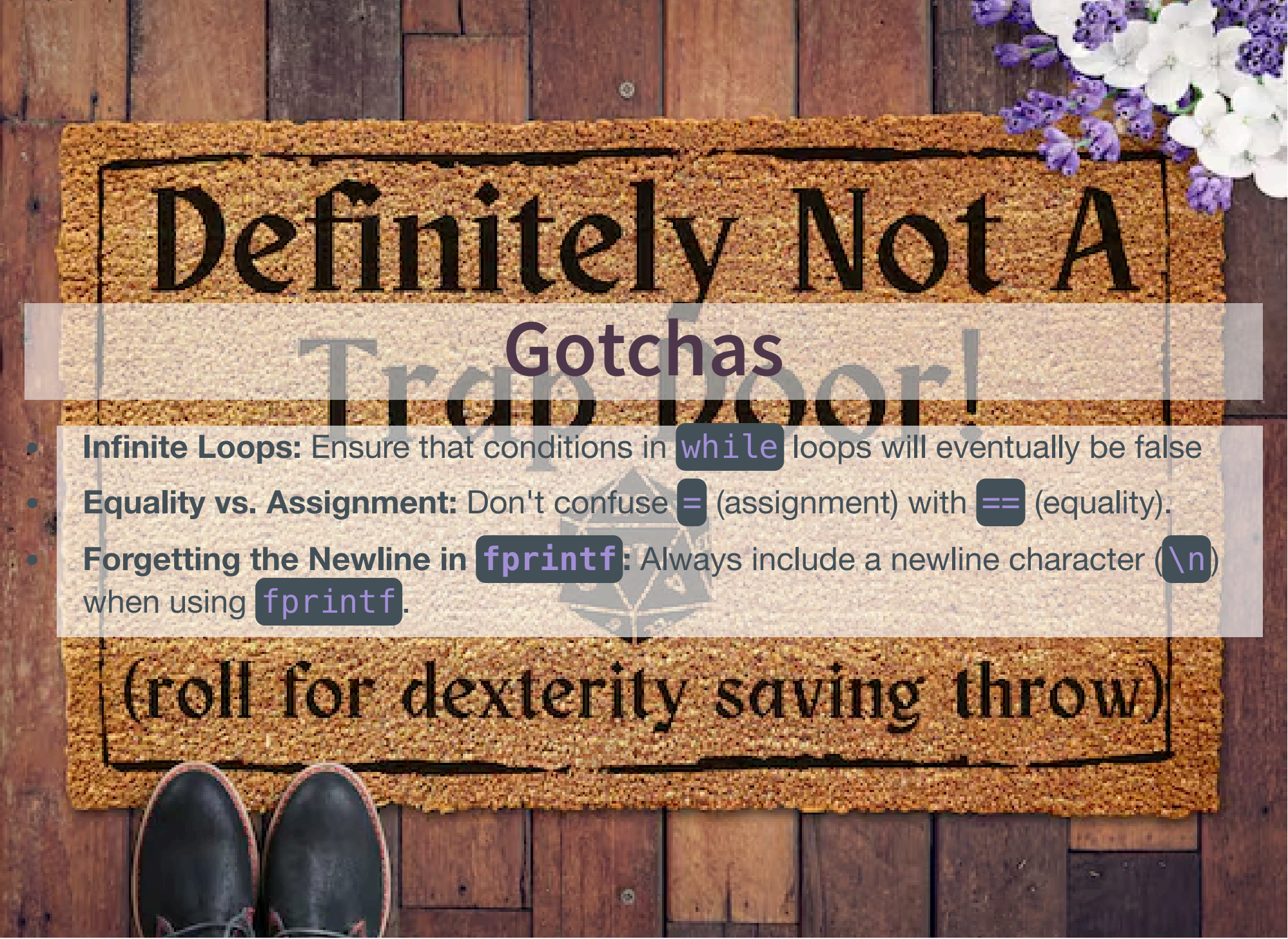
- You know how many times you need to repeat a task.
- You want to traverse (visit) each element in an array.

Use `while` loops when

- You're not sure how many loops to perform
- You want something to repeat until a condition is met

Key Takeaways 🎓

- **Control Flow:**
  - **Conditional statements** (`if-elseif-else`, `switch`) allow decisions based on conditions.
  - **Loops** (`for`, `while`) enable repetitive tasks, whether you know the number of repetitions in advance or not.
- **Input and Output:**
  - Use the `input()` function for dynamic user input.
  - **Formatted output** is achieved through `fprintf` and `sprintf`, with precise control over how data is displayed.
  - Proper use of format specifiers (`%d`, `%f`, `%s`, etc.) allows customization of numeric and string outputs.



Gotchas

- **Infinite Loops:** Ensure that conditions in `while` loops will eventually be false
- **Equality vs. Assignment:** Don't confuse `=` (assignment) with `==` (equality).
- **Forgetting the Newline in `fprintf`:** Always include a newline character (`\n`) when using `fprintf`.

Software Engineering