While you are waiting

Choose the correct spelling of this frequently misspelled word

vinyette

vinyet

vignette

ENGR 101 – Lecture 12

Program Design

Anagrams

Download these files from the Google Drive:

`words.txt`

`loadWords.m`

Laura Alford, James Juett, Rick Niciejewski 10/18/17

Announcements

- Lecture 13, Monday, 23-OCT-2017, will be Exam Review.
  - a) you will be given a couple of sample exam questions to work through during lecture: on your own, with your neighbours, with your laptop/MATLAB
  - b) solutions will be discussed in the second half
  - c) we will add a MATLAB sample exam to our Google Drive

- Lab_6 involves a group activity exercise. Please attend your lab.

A Word-Guessing Game

- Today we'll implement a word-guessing game in MATLAB.
- The game uses a set of the 1000 most common words.
  - These are stored in `words.txt`.
  - We've provided a function to load these into a cell array of strings.

- The game picks a word at random and scrambles it. Then the user is asked to guess what the original word was.

- RUN **wordGame_solution** with MATLAB

Example: Scrambling a Word

- Let's start by identifying a specific feature we think we'll need for the program and focus on that…
  - This is called bottom-up design.

- We need a way to randomly mix up the characters in a word to create the anagram given to the player.
  - The random permutation function `randperm()` will be useful.
  - Let's try it out interactively in MATLAB…

Example: Scrambling a Word

- Let's experiment with the word **'hello'** and **randperm** !!!

```
>> word = 'hello';
>> word([5 1 4 2 3])                ; scramble word
ans = ohlel
>> length(word)
ans = 5
>> randperm(length(word))           ; scramble 5 integers
ans =
     3     1     5     2     4
>> word(randperm(length(word)))     ; rebuild word
ans = heoll
>> word(randperm(length(word)))
ans = olehl
>>
```

Example: Scrambling a Word

- After figuring out what we want to do, we create a function that serves as an abstraction for our approach.
- We can then use the function to solve this part of the problem without having to worry about the details!

```
function [ scrambledWord ] = scramble( word )
% scramble Puts the characters of a word in random order

    scrambledWord = word(randperm(length(word)));

end
```

Example: Scrambling a Word

- After figuring out what we want to do, we create a function that serves as an abstraction for our approach.
- We can then use the function to solve this part of the problem without having to worry about the details!

```
function [ scrambledWord ] = scramble( word )
% scramble Puts the characters of a word in random order

    scrambledWord = word(randperm(length(word)));

end

>> disp(scramble('hello'))
lolhe
>>
```

Bottom-Up Design

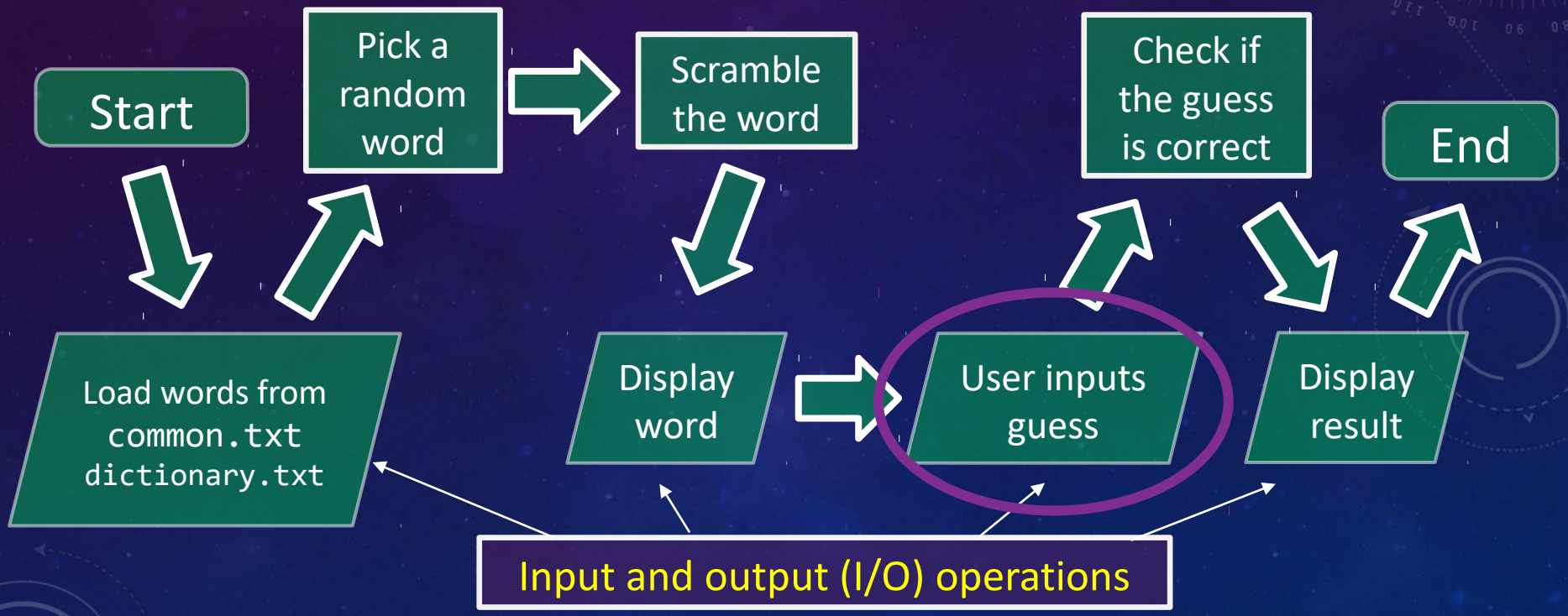
- In bottom-up design, we first start with specific features we think might be useful and implement them.
- **In MATLAB, this often involves trying out different approaches interactively to see how they work.**
  - **Often times we still use built-in functions, so we're not starting literally at the "bottom" of everything.**
- Once we've settled on an approach, we package up the individual pieces of functionality into abstractions.
  - In most programming languages, this amounts to writing functions.
  - A key part of creating an abstraction is deciding on the interface.

Top-Down Design

- In top-down design, we start by thinking about the big picture of what our program needs to do.
- We make use of abstraction in order to avoid having to think about all the implementation details.
  - e.g. Using the abstractions from a moment ago, we can treat scrambling a word and checking for anagrams as basic operations.
- It's often helpful to use flowcharts or other diagrams to map out the high-level design of our program.

Flowchart: A Simplified Version of the Game

- For now, let's work with a simplified version of the game that only has the player guess a single word.



Flowchart Components

	Purpose	Example
Terminal	Indicate the start and end of the program.	Start End
Process/Task	Perform a computation or call a function as an abstraction.	Scramble the word; Check if the guess is an anagram
Input/Output	Read or write from/to an external source such as a file or the terminal.	Load from `words.txt`; User inputs guess
…		
…		

Code: A Simplified Version of the Game

```
% A word guessing game. The user is given an anagram of a
% common word and they must guess the correct version.

% Load the words file
words = loadWords('words.txt');

% Pick a single word randomly
% Use content indexing to get the word itself (and not the cell)
word = words{randi(length(words))};

% get the scrambled version
scrambledWord = scramble(word);

% display the scrambled word to the user
disp('Unscramble this word:');
disp(scrambledWord);

% prompt the user for a guess
```

User Input in MATLAB

- Use the `input` function to get input from the user.
- For example:

```
x = input('Please enter a number: ');
```

The result will be stored in x.

The prompt that is displayed to request input from the user.

An extra space to make it look nice.

- Whatever the user enters will be **evaluated** as an expression and returned from `input`. (e.g. if they enter 2 + 3, the value 5 is returned.)
  - If an invalid expression is entered, MATLAB shows the error and lets the user try again.

User Input in MATLAB for Strings

- To get a **string** as input, add the `'s'` parameter to `input`.
- For example:

```
x = input('Please enter a string: ', 's');
```

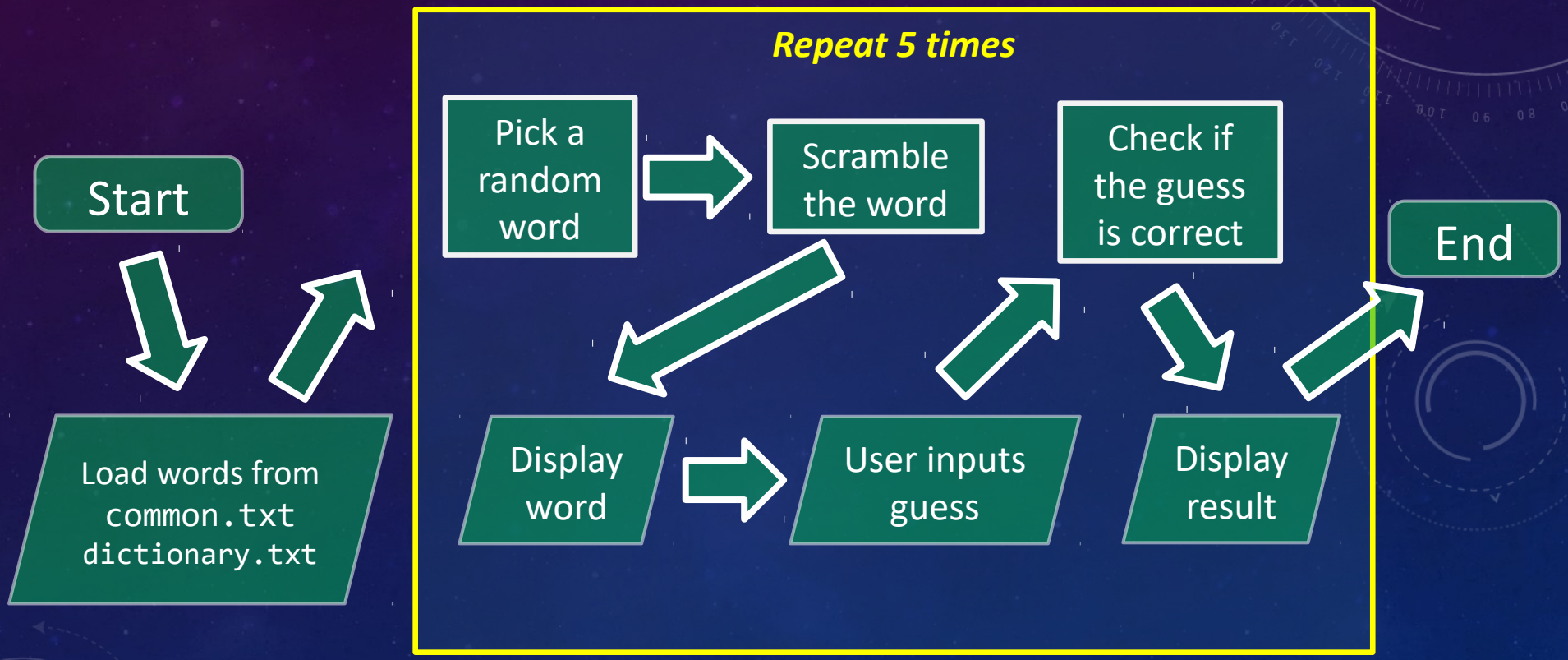
- For string input, whatever the user types is simply returned as a vector of characters (it is not evaluated).
  - This is what we want for our word game program.

Code: A Simplified Version of the Game

```
% A word guessing game. The user is given an anagram of a
% common word and they must guess the correct version.

% Load the words file
words = loadWords('words.txt');

% Pick a single word randomly
% Use content indexing to get the word itself (and not the cell)
word = words{randi(length(words))};

% get the scrambled version
scrambledWord = scramble(word);

% display the scrambled word to the user
disp('Unscramble this word:');
disp(scrambledWord);

% prompt the user for a guess
guess = input('Enter your guess: ', 's');

% check if the guess and the word are the same
disp(isequal(guess, word));
```

Adding Repetition – *something NEW*

- Let's modify our program to give the user 5 words…



Iteration (Loops)

- Most programming languages provide a mechanism to repeat several lines of code over and over.
  - These are called **loops**.
- The process of repetition in programs is called **iteration**.

- We're only going to take a very brief look at loops and iteration in MATLAB. We won't go through all the rules.
  - We'll focus on this much more in C++.

for Loops in MATLAB

- Here's an example:

```
% Print out the numbers 1 through 5

for i = 1:5

  disp(i)

end
```

`i` is called the **index variable**.

We specify a sequence of values for `i`. In this case, the loop repeats 5 times with the values 1, 2, …, 5 used in turn for `i`.

Code that we want to repeat goes inside the for loop.

for Loops in MATLAB

STOP
In MATLAB, you would want to just use `sum(x)` instead!

- Another example:

```
% Compute the sum of numbers in a vector
% Assume there is a vector named x
total = 0;
for i = 1:length(x)

  total = total + x(i);

end
```

We use each value of `i` as an index to get each element from x.

`total` is updated on each iteration of the loop.

Code: Playing 5 Rounds

```
% A word guessing game. The user is given an anagram of a
% common word and they must guess the correct version.

% Load the words file
words = loadWords('words.txt');

for i = 1:5
  % Pick a single word randomly
  % Use content indexing to get the word itself (and not the cell)
  word = words{randi(length(words))};

  % get the scrambled version
  scrambledWord = scramble(word);

  % display the scrambled word to the user
  disp('Unscramble this word:');
  disp(scrambledWord);

  % prompt the user for a guess
  guess = input('Enter your guess: ', 's');

  % check if the guess and the word are the same
  disp(isequal(guess, word));
end
```

We've wrapped up all the code to repeat in a for loop.

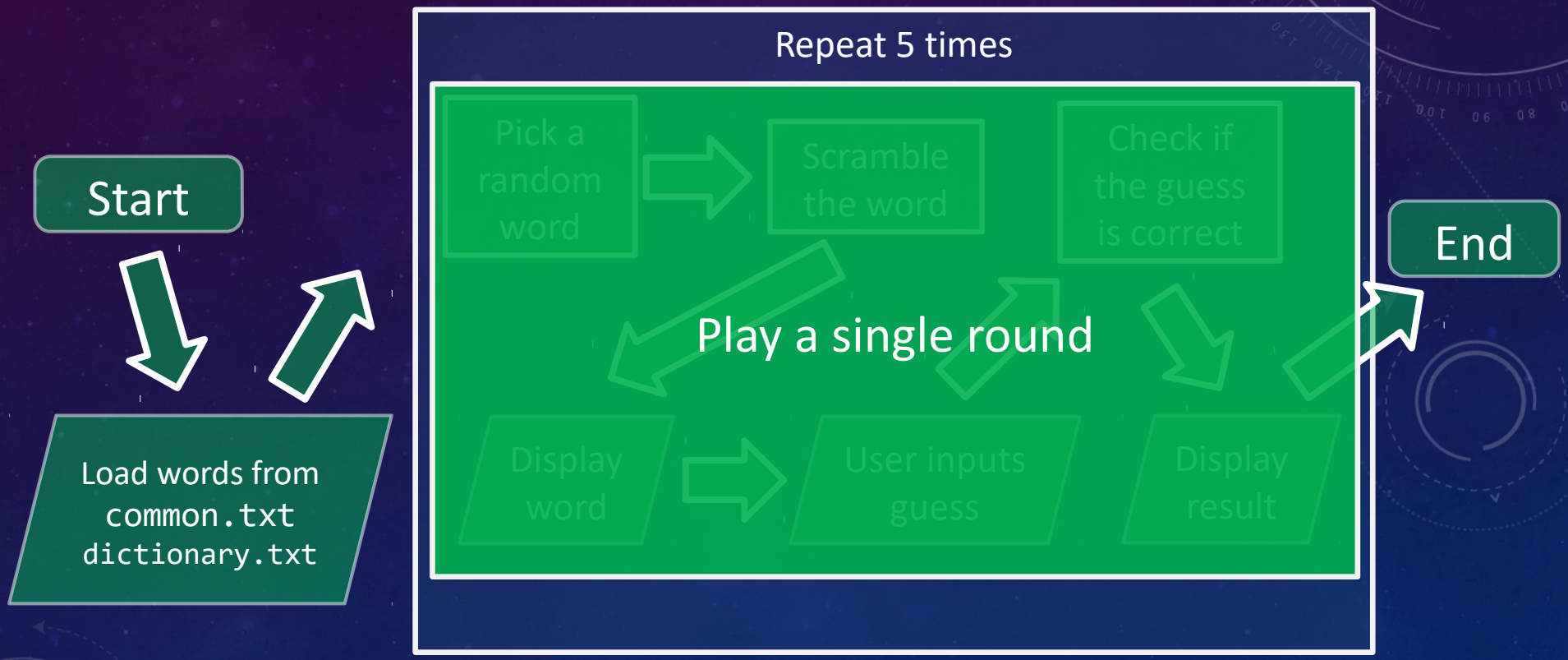
This is a busy script. Comments and useful variable names make it more readable.

Break Time – QUESTIONS ??

We'll start again in 5 minutes.

Abstraction!

- Let's create a function that abstracts playing one round.



Abstraction for Playing one Round – playRound.m

- Before implementing a `playRound` function, we need to decide on the interface (i.e. parameters and what to return).
- This is based on our top-down design and what would be useful.

2. The rest of the program needs to know if they won.

1. We need to pass the list of words to the function.

```
function [ wonRound ] = playRound( words )
% playRound Plays one round of the word-guessing game
%           using the provided words and returns a
%           logical representing whether the user won.


end
```

The comment describes the abstraction.

Implementing playRound

```
function [ wonRound ] = playRound( words )
% playRound Plays one round of the word-guessing game
%           using the provided words and returns a
%           logical representing whether the user won.

  % Pick a single word randomly
  % Use content indexing to get the word itself (and not the cell)
  word = words{randi(length(words))};

  % get the scrambled version
  scrambledWord = scramble(word);

  % display the scrambled word to the user
  disp('Unscramble this word:');
  disp(scrambledWord);

  % prompt the user for a guess
  guess = input('Enter your guess: ', 's');

  % check if the guess and the word are the same
  wonRound = isequal(guess, word);
end
```

Instead of displaying it, we return the result instead.

Using the playRound Abstraction

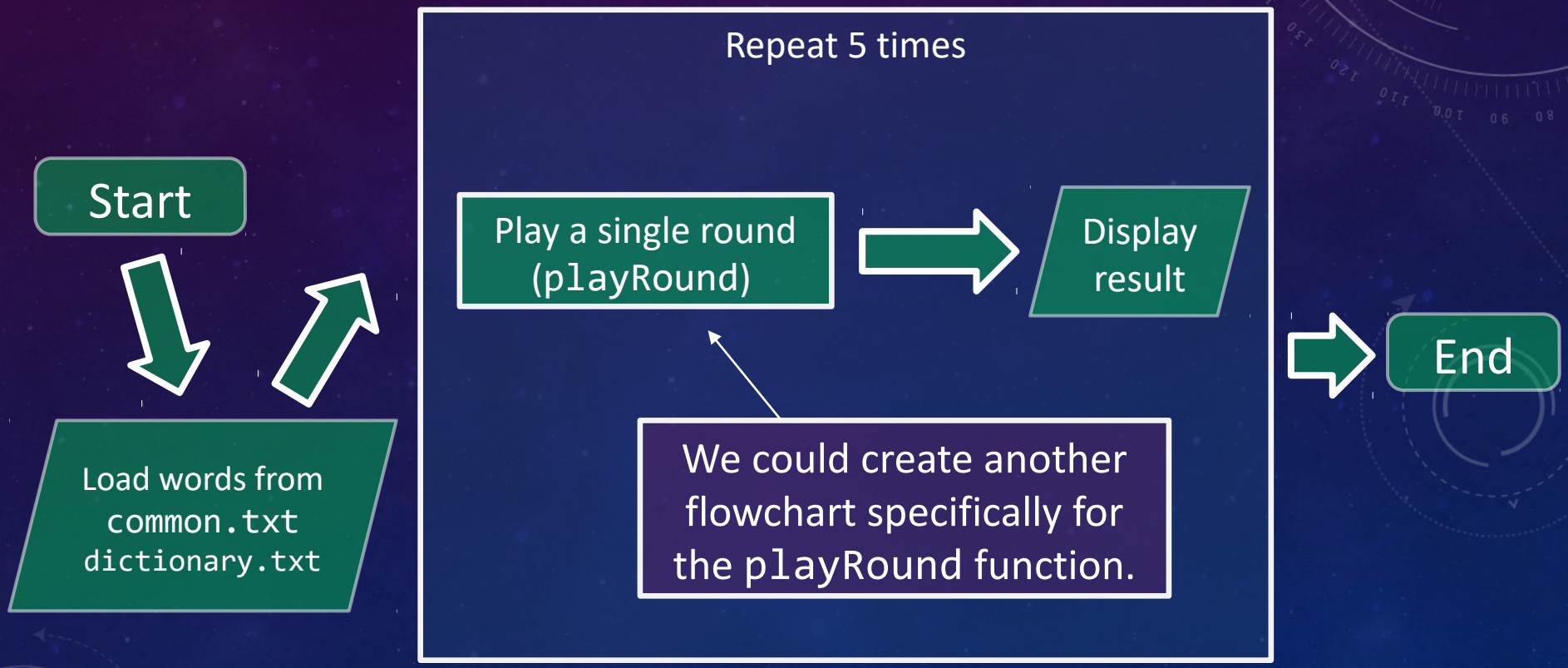
- Now that we've written playRound, we can rewrite the main program to be much more concise and easier to understand!

```
% A word guessing game. The user is given an anagram of a
% common word and they must guess the correct version.

% Load the words file
words = loadWords('words.txt');

for i = 1:5
  % Play a single round and store the result
  wonRound = playRound(words);

  % display whether they won
  disp(['Result of round ', num2str(i)]);
  disp(wonRound);
end
```

Using the index variable to create a different message for each round.

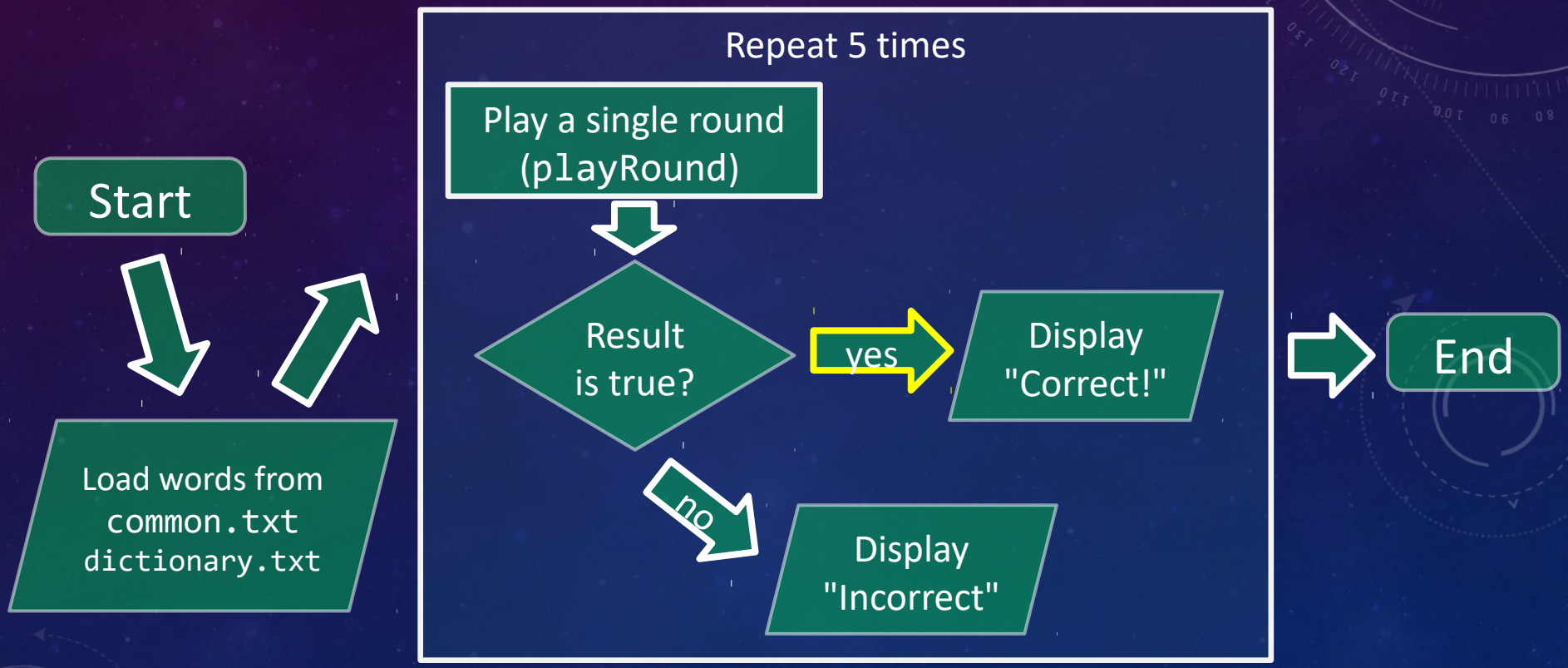
Using the playRound Abstraction

- Now our top-level flowchart is much simpler.



Branching

- When we need our program to make a decision and take one of several paths through the flowchart, we use a branch.



Branching

- In code, we use `if` and `if/else` statements to branch.
- For example:

```
% Print a message about the temperature:

temp = input('What temperature is it? ');

if temp > 90
  disp('That is very hot!');

elseif temp > 50
  disp('Go take a walk outside!');

else
  disp('It is pretty cold.');

end
```

When the program runs the whole if structure, <u>only one of several branches will run,</u> depending on the conditions.

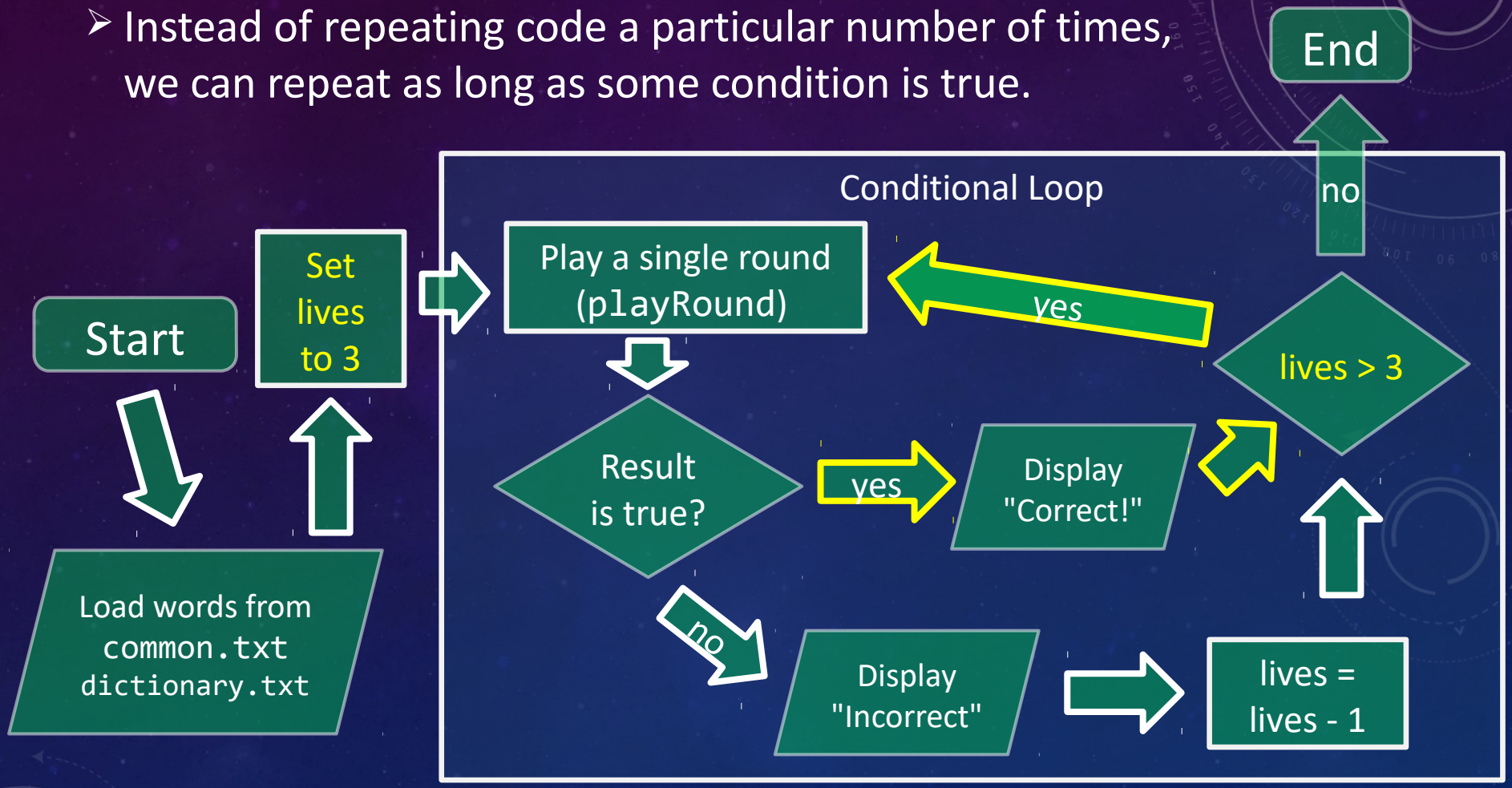
Branching – Modify output based on true/false

```
% A word guessing game. The user is given an anagram of a
% common word and they must guess the correct version.

% Load the words file
words = loadWords('words.txt');

for i = 1:5

  if playRound(words)

    disp(['Round ', num2str(i), ' correct!']);

  else

    disp(['Round ', num2str(i), ' incorrect.']);

  end

end
```

Use the logical result of the playRound as the if condition.

Conditional Loops

- Instead of repeating code a particular number of times, we can repeat as long as some condition is true.



while Loops in MATLAB – iteration via conditional

- For example:

```
% Force the user to enter a positive number
n = -1;
while n < 0

  n = input('Please enter a positive number: ');

end
```

This expression is the **loop condition**.

Code inside the **body** of the while loop will run indefinitely as long as the condition remains true.

Conditional Loops

```
% The game now repeats until you miss 3 times and keeps score.

% Load the words file
words = loadWords('words.txt');

lives = 3;
score = 0;
while lives > 0

  if playRound(words)
    disp('Correct!');
    score = score + 1;
  else
    disp('Incorrect.');
    lives = lives - 1;
    disp(['You have ', num2str(lives), ' lives remaining.']);
  end
  disp(['Score: ', num2str(score)]);
end

disp(['Game over. You got ', num2str(score), ' points.']);
```

Initialize `lives` and `score`.

Keep going as long as lives > 0

Update loop variables inside separate branches.

The score is printed on each loop iteration since it's outside the if.

Control Flow

- Branching and iteration are techniques for managing **control flow** in our programs.
  - The line of code that is currently executing is said to have "control".
- In particular, flowcharts are an effective tool for mapping the control flow of our program design.
- Mechanisms like `if`, `for`, and `while` allow us to structure our code to follow the desired control flow.

Flowchart Components

	Purpose	Example
Terminal	Indicate the start and end of the program.	Start End
Process/Task	Perform a computation or call a function as an abstraction.	Scramble the word Check if the guess is an anagram
Input/Output	Read from/write to an external source such as a file or the terminal.	Load from `words.txt` User inputs guess
Branching	Take one of several "branches" through the program depending on a condition.	Result is true? lives > 3
Iteration	Repeat part of the program a certain number of times or while a condition holds.	

Why did we wait so long?

- We've taken a unique approach to introducing programming concepts so far in ENGR 101.
  - In particular, we didn't show control flow mechanisms until now.

- This is because the problems you solve in MATLAB **rarely** require the use of control flow constructs like `if` and `for`.

Prefer Array Operations to Loops:

- Let's say you wanted to plot the function $x^2 + 2x$:
- Using a loop:

```
x = linspace(0, 5, 101);
for i = 1:100
  y(i) = x(i)^2 + 2*x(i);
end
plot(x,y);
```

This is less elegant and runs more slowly than the vectorized version!

- Using array operations: (preferable)

```
x = linspace(0, 5, 101);
plot(x, x.^2 + 2.*x);
```

The array operations work element-by-element, so they do the same thing as the loop.

Prefer Logical Indexing to Loops

- Let's say you wanted to replace all negative elements with 0:
- Using a loop:

```
% assume we have a vector x
for i = 1:length(x)
  if x(i) < 0
    x(i) = 0;
  end
end
```

This is less elegant and runs slower than the vectorized version!

- Using logical indexing: (preferable)

```
% assume we have a vector x
x(x < 0) = 0;
```

Prefer Ranges to Loops

- Let's say you wanted to create a vector of the numbers 1-100:
- Using a loop:

```
% assume we have a vector x
for i = 1:100
  x(i) = i;
end
```

You're using a range to create a range. That's irony or something.

- Using a range: (preferable)

```
x = 1:100;
```

Literally just use the range.

When to use loops in MATLAB?

- Hint: If you ever find yourself using a loop to do anything with elements of an array, you're probably doing it wrong.
  - Use array operations, logical indexing, and ranges instead!
  - Use built-in MATLAB functions. Most are vectorized.

- If you need repetition for something else…
  - Double check to make sure you really need a loop.
  - Are you sure you can't do it with vectorization?
  - Are you really sure?

Summary

- What you have learned so far in ENGR101 !!!
  - How to use MATLAB properly
  - Enough to proceed to next week's exam
  - Enough to use MATLAB in your senior years

- Sample exam problems will be published shortly. Answers will be shared before Monday's lecture

- I will remain for office hours behind Stamps

See you next Monday !!!