Exported for Brian O'Connell on Tue, 27 May 2025 19:31:17 GMT

Programming 6 Pre-LAB: MATLAB Data and Structures

This PreLab will differ a little in that there are 2 submittable parts. It's going to start with Part 1, a debugging activity, then get into the concepts, and then Part 2, a script generation activity.

At this point, the expectation is that you have a decent grasp of the following elements of MATLAB and programming concepts. You don't need absolute mastery at this point but enough familiarity that you understand their use and purpose but may still need to utilize some reference resources to fully implement:

- MATLAB basics
  - Access MATLAB
  - Run an M-File
  - Print the M-File and Command window as a pdf
  - General MATLAB features
- Some basics about developing code
  - Pseudo-code/Flowcharts
    - Including the course required syntax
  - Commenting your code
  - Debugging
- Variables
  - Data Types
  - Scalar, array, matrix
- Arithmetic
- Program Inputs & Outputs
- Array Creation
- Array Math
- Plot command
- Plot Features

This lesson will go over:

- Loading external data
- Matrix Manipulation
- Data Manipulation
- Conditionals
- Branching
- Loops

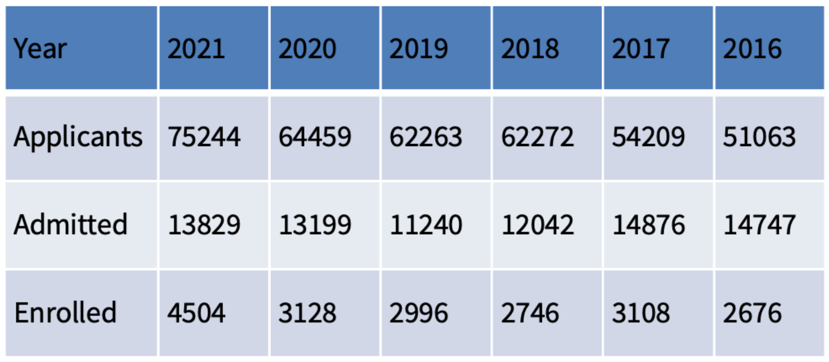
P6L1: Debugging

Start with the following script:

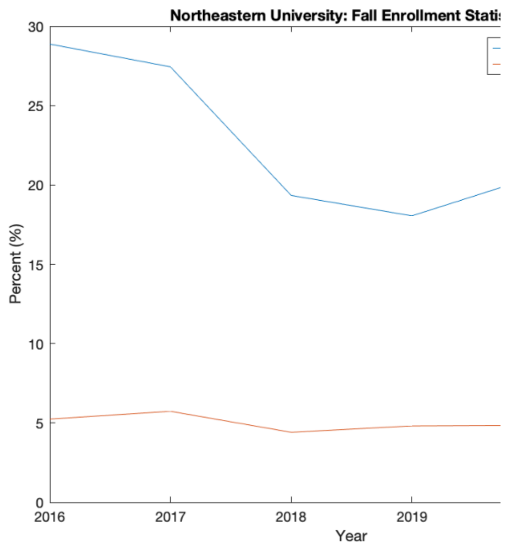
https://raw.githubusercontent.com/boconn7782/CourseCode/main/MATLAB/P6L1_start

Update the title block as required to make this your script.

The program is supposed to take in the Northeastern Enrollment Numbers from 2016 to 2021 and then display the Percent Admitted and the Percent Enrolled per year. The following is the given data table from Northeastern University and the expected figure output.



Year	2021	2020	2019	2018	2017	2016
Applicants	75244	64459	62263	62272	54209	51063
Admitted	13829	13199	11240	12042	14876	14747
Enrolled	4504	3128	2996	2746	3108	2676



You must identify and fix the 5 errors in this script that keep it from running as described. Not all of these are errors that will keep the script from running. Identify the errors and fix them. Include comments describing the errors. Be aware of how long you take but it's a good challenge to see which you can recognize on your own. If you get past that point of productive struggle and into a frustrated non-productive struggle, follow through the rest of the section to see the common errors and some methods for identifying them. I recommend setting a 10-15 minute timer to challenge yourself and then once that goes off, continue along with the per-lab and see how many you identified yourself.

MATLAB Code Analyzer

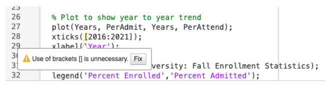
Let's start by using the MATLAB editor's help features. Similar to MS Office Suite and Google Drive and their built in spelling and grammar checks, MATLAB has a built-in Code Analyzer to help. It gives you 3 tiers of warnings:

MATLAB Code Analyzer Show Correct Answer Show Responses

match the warning indicator with it's meaning

Premise			Response	
1	Red Octagon with an Exclamation Point	→	A	File contains warnings or opportunities for improvement as well as syntax errors or other significant issues.
2	Yellow Triangle with an Exclamation Point	→	B	File contains no errors warnings or opportunities for improvement.
3	Green Circle with a Check Mark	→	C	File contains syntax errors or other significant issues.
4	Yellow Triangle with a Check Mark	→	D	File contains warnings or opportunities for improvement but no errors.
			E	Not a warning indicator

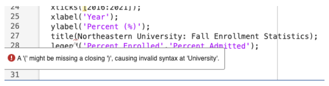
If you look to Line 28, you'll see some yellow highlighting with a red swiggle,



```
26    % Plot to show year to year trend
27    plot(Years, PerAdmit, Years, PerAttend);
28    xticks([2016:2021]);
29    xlabel('Year');
3                                   
3                     versity: Fall Enrollment Statistics);
32    legend('Percent Enrolled','Percent Admitted');
```

⚠ Use of brackets [] is unnecessary. Fix underneath. This indicates just a warning, *not an error.* In this case, the `[ ]` are simply unnecessary. I personally still use them though as they indicate an array.

If you look a few lines down though, you might (will) find another warning that's more serious. It's not uncommon to forget quotations marks when inputting information, as is the issue here. The interesting element regarding the code analyzer is that is not what the code analyzer will pick up. It will assume that the first word is meant to be a variable and the inclusion of the next if an error of too many or incorrect inputs. So, like the grammar/spell check you're probably familiar with, it's not always right. It's just recognizing where an error would occur and making a best guess a the issue.



```
xticks([2016:2021]);
xlabel('Year');
ylabel('Percent (%)');
title(Northeastern University: Fall Enrollment Statistics);
legend('Percent Enrolled','Percent Admitted');
```

A '(' might be missing a closing ')', causing invalid syntax at 'University'.

Error 1: Fix the error and don't forget to include a comment describing the issue corrected.

- *Recognize the difference between an error and something that is functional but might be unnecessary*
- *Understand what format a function expects for its inputs.*
  - *Most common is forgetting single or double quotation marks to indicate a character array or string*

Run Errors

We've identified and fixed 1 issue. Now let's run the code. You'll see in the command window red text letting you know about errors. This tends to be very descriptive and usually quite useful. Here you likely see:

```
Unrecognized function or variable 'admitted'.
Error in P6L1 (line 19)
PerAdmit = admitted./Applicants*100;
>>
```

It gives the type of error in the first line, the file its in as well as what line it occured at, and then the code from that line. Now keep in mind that it's not always an issue in that line but maybe in the 1 before or the 1 after. That's more often a case in C++ where a missing ; or bracket causes major issues. MATLAB, more often than not, identifies the correct line because it doesn't require the `;` and is better about identifying bracket issues. Here it's a problem with the variable '`admitted`'. Double check how it was created before, remember that capitalization matters in variable names.

Error 2: Fix the error and don't forget to include a comment describing the issue corrected.

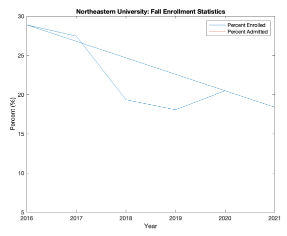
- *Be consistent with variable names*
    - *Capitalization matters*

Recognizing Purpose

Those are the only 2 errors that will keep the program from running. There are still a few that will keep it from running as it supposed to but will still allow the script to finish. Let's understand the purpose through its comments. It should:

- Load total values for Fall Enrollment Statistics
    - Including the # of applicants, # admitted, and the # enrolled from year to year.
- Calculate Percent based on total applicants
    - Based on the next line, we should assume this gives us arrays of year to year data
- Plot to show year to year trend

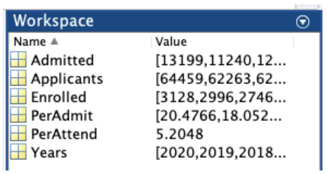
When you run it you should see this image. Note that it only has one line and it doesn't really look like what you would expect for something showing a year to year trend. So let's see what we have:



- Blue line indicating Percent Enrolled based on the legend
- There should be a percent admitted based on the legend but we're not seeing it

Whenever you have an issue that looks like it might involve the data itself, the workspace is very useful.

Based on our legend, I'm guessing there's an issue with the Percent Admitted data. The workspace let's us look at the data we're working with at a glance. We can


Workspace

Name ▲	Value
Admitted	[13199,11240,12...
Applicants	[64459,62263,62...
Enrolled	[3128,2996,2746...
PerAdmit	[20.4766,18.052...
PerAttend	5.2048
Years	[2020,2019,2018...

even double click on the variable name to open it up and get a deeper look. This is sufficient though. There's a clear difference between `PerAdmit` and `PerAttend`. The first looks fine, an array of data points that we assume lines up with year to year data. The `PerAttend` though, which seems to be in the plot, is only giving us a scalar value. This indicates 2 issues, a problem with our calculations and a problem with our labelling.

So let's look at our calculations on Line 19 and 20 first. This may have changed depending on how you're commenting or updates to your title block. You're looking for these lines:

- `% Calculate Percent based on total applicants`
- `PerAdmit = admitted./Applicants*100;`
- `PerAttend = Enrolled/Applicants*100;`

`PerAdmit` is calculated using element by element , `./`, as is expected for taking 2 arrays of data and wanting a corresponding array of calculated data. The `PerAttend` calculation doesn't include the `.` though. With out that, it must be doing matrix division which would result in a scalar value, as we saw in the workspace. Fix the issue so its calculating element by element and run again. You should see 2 lines now.

Error 3: Fix the error and don't forget to include a comment describing the issue corrected.

- *There can be subtle differences between commands, particularly the mathematical operators*
    - *Including the* `.` *or not makes a big difference*

Our initial check of the plot legend seemed to indicate that the issue would be with the Percent Admitted data but that wasn't entirely the case. Let's look at our labelling on line 28. The `legend` command, when manually inputted, is based on order of inputs. Same with the `plot` command. So if we look to both of those, they don't match up so one has to have it's order corrected to match the other, doesn't matter which.

Error 4: Fix the error and don't forget to include a comment describing the issue corrected.

- *Order of inputs matters for all functions*

Now when we run it, we should get both lines plotted and laballed correctly. They don't look correct though. Both seem to jump back on one another. This indicates that there's an issue with the data pairings used by the plot command. They take each pairing and plot them in order as XY coordinates for a point then the connect those points in order with lines. In this case, there's a line that goes from the end to the beginning. Let's go back and look at the source data, the manually created arrays for the years, applicants, admitted, and enrolled # of students.

Checking it against the provided data table, it looks someone added the 2021 data to the end instead of the beginning. We can assume that someone had it entered as 2020 to 2016 in decreasing order because that's how Northeastern published its data, with the most recent year first. Whoever updated it for 2021, must have not noticed and added those numbers to the end.

Always double check manual data entry, particularly when the data arrays/tables/matrices are larger.

Error 5: Fix the error and don't forget to include a comment describing the issue corrected.

- *Check for human error.*
    - *Manually inputted data can be a major source of that, but things like order of operations errors can be attributed to that as well.*

The BIGGEST takeaway from this should be you should just run your code a lot. As your developing a script, it doesn't cost you anything to just run it occasionally to see if any errors are occurring or unexpected outputs. Even though you had a full script here to debug, you still ended up running it repeatedly to discover them all. You can do the same for your own scripts but don't have to wait until they're complete.

Concepts and Commands

Now we're going to go over the following new concepts and commands within MATLAB:

- Loading external data
- Matrix Manipulation
- Data Manipulation
- Conditionals
- Branching
- Loops

This may seem like a lot but it's mainly introductions to these elements within MATLAB. Some will be a little familiar as they have corollaries in other topics, some we've covered. There are several example codes you are encouraged to input and enter in the MATLAB command window to observe the results. There are periodic questions on this content that you will need to respond to. They are heavily weighted towards participation. Full credit for the lesson doesn't require 100% correctness but does require 100% participation. The second task will be to create a simple MATLAB script. There is pseudocode and some guidance to help walk you through created a script using some of the following concepts.

File Input and Output

Data is commonly stored in different files or databases to enable repeated access. Here are a few ways to extract data from MATLAB.

Save

The **save** command is used for saving data to the computer. This has a little bit of nuance to be aware of because there are differences in not only what you're saving but it's format.

`save('filename.mat')` will save all workspace files into a MATLAB binary MAT-file. So if you have multiple variables full of information you'd like to save for later use or to transfer elsewhere, this is a way to capture all of them. You can specify which variables are saved by listing them after the filename `save('filename.mat', 'Var1', 'Var2', ...)`. This will only include `Var1`, `Var2` , and any other listed variable from your workspace in the .mat file.

It is more common though to be saving a single variable's content for output to a spreadsheet program or other resource into a more generic form. To save values in a matrix or vector, *y* to a file *`data.txt`* use:

```
y = [1 2 3 4; 5 6 7 8];
save('data.txt', 'y', '-ascii','-tabs');
```

The `'-ascii'` option ensures that the data is saved in ASCII form and the `'-tabs'` tells it to include a tab between elements. This is common for exporting data to be opened in MS Excel or Google Sheets. It can be read by other programs, such as MS Word, Notepad, etc.

We will not use this that often, mainly just for project data if you utilize it, but it's good to know for when you're doing any analysis and want to hold onto that data for future reference. There are also other options for appending data to a file instead of creating a new one or overwriting an old one, `'-apend'`.

fprintf

`fprintf(formatspec,A1,...,An)` will write to the command window. You can instead write to a file by using `fprintf(fileid,formatspec,A1,...,An)` where `fileid` is the name of the file. Typically it's a text file. If you just define it as it's name without a file path, MATLAB will assume it's in the Current Folder. That file will need to have first been opened, using `fileID = fopen(filename,'w')`, to allow for editing. That's what the 'w' flag in that example denotes. You then need to close the file after you're finished with it, `fclose(fileID)`. Try:

- `x = 0:.1:1;`
- `A = [x; exp(x)];`
- `fileID = fopen('exp.txt','w');`
- `fprintf(fileID,'%6s %12s\n','x','exp(x)');`
- `fprintf(fileID,'%6.2f %12.8f\n',A);`
- `fclose(fileID);`

The `type` command will then let you view file contents in the command window, `type('exp.txt')` in that case, or you can manually open the file *'exp.txt'* to see it's contents.

Load

The **load** commands is used for loading data from the computer. This can be used to load a .mat file contents back into the workspace. It can also load other data formats in as a single variable. For example, given an ASCII data file 'data.txt' in the current window, `load('data.txt')` will load it's content into a variable of the same name in the workspace. More often though, we want to specify the variable name ourself so you would more likely use `D = load('data.txt')`. to save that content in the variable `D`. If that text file is tab delimited, arranged with a tab between each element, then it will be loaded in as an array or a matrix if there are multiple lines.

Help   Show Correct Answer   Show Responses

If you needed to save some information from the MATLAB workspace in a specific format, where would you find help in figuring out how to do that?

A Google it

B MATLAB help documentation for 'save'

C Wild guesses

D	MATLAB help forums
E	Knock door to door and ask whoever opens

Matrix Manipulation

First, we're going to want to be able to better deal with Arrays.

Managing Arrays

Now we know we can create arrays and even matrices.

```
1. [0 5 -2 3.14] % Create rows by separating by a space
2. [1,10,5,-3]   % Or by separating by commas
3. [18;1;-4]     % Create columns using semi-colons
4. [7 8 2.3; 7.5 9 0] % Combining them creates 2D arrays (or matrices)
```

We can now manage arrays, constructing and manipulating them after they've been created. There are far too many different commands that will generate arrays or matrices based on input criteria for us to review here. We will go over some of the more common ways but you can review many of the rest in the MATLAB documentation on Matrices and Arrays. This list many of the commands for auto-generation and their details.

For instance, you can create an array of linearly spaced values with the **linspace** function or an array of 0s with the **zeros** function. You can see all the built-in functions in the MATLAB documentation, but I'll go over some common methods for managing matrices here.

Try running many of the below commands, or even creating your own similar ones, in the command window to see the results and how they create and manipulate matrices.

Creating and Calling Matrices

Simply combine the concepts used for arrays:

1. `A = [0 5`
   `-2 3.14]  % Use a newline to separate columns`
2. `B = [1,10; 5,-1]  % or use semi-colon`
3. `C = [18,1; -4] % Keep columns consistent (This command will give you an error due to inconsistent # of columns)`

Calling Matrix Value

Matrix use 2 indices in parentheses as (row, column):

- `B=[2, 1.2; 0, -3]`
- `B(2,1) % Returns 0`
- `B(1,2)+B(2,2) % Can use in equations`
- `B(3,1) % Error since outside of matrix`

Calling Sub-sections of matrix

Colon operator will call entire rows or columns:

- `M=[1,2,3; 4,5,6; 7,8,9]`
- `C=M( : , 2 )   % for all rows, values in column #2`
- `R=M( 2 , : )   % for all columns, values in row #2`

Can use arrays as indices to grab sections or select elements of a matrix:

- `S1=M(2 : 3, 1 : 2)  % rows 2 to 3, col 1 to 2`
- `S2=M( [1,3] , [1,3] )  % for elements (1,1),(1,3),(3,1),(3,3)`

Managing Array Elements Show Correct Answer Show Responses

Given:
```
A=[2,4;6,8];
A(1,2)=5;
```
What is A?
Do not include spaces in your response

Combing elements to create a matrix

Rows, columns, and matrices can be combined into a matrix as long as the final dimensions are consistent:

- `A = [1, 2, 3, 4];`
- `B = [2, 4, 6, 8];`
- `C = [3, 6, 9, 12];`
- `D = [A; B; C]    % Combining rows`
- `E = [1 2 3 4 ];`
- `F = [2 4 6 8];`
- `G = [3 6 9 12];`
- `H = [E, F, G]    % Combining Columns`
- `I = [E, [D; A]] % Complex construction`
- `J = [E, D; A]    % Will fail due to dimensions mismatch`

Redefining Elements

As with arrays, you can redefine matrices by setting elements or subsets of elements equal to (`=`) new values:

- `A = randi(20,3,5)`
- `A(1,1)= 0      % Redefines the element in the first row and column to 0`
- `A(:,[2 3])=[] % Removes that subset of values by setting them equal to an empty matrix`

Matrix operators Show Correct Answer Show Responses

What is the missing line of code?

```
>> X = magic(4);
>> < Missing line of code >
>> disp(Y)
16 2 3 13
```

9 7 6 12
4 14 15 1

For reference, magic(n) returns an n-by-n matrix constructed from the integers 1 through n2 with equal row and column sums.

A	`Y = X([1 3 4], [2 4]);`
B	`Y = X([1 3 4], :);`
C	`Y = X([2 3 4], [1 3 2 4]);`
D	`Y = X(:,[1 3 4]);`

Matrix Functions

There are several functions that allow for manipulating or gaining information about an array or matrix

Transpose

Including an apostrophe(`'`) with the variable denotes a transpose, where the row and column index for each element is interchanged:

- `A = randi(20,3,5)`
- `A' % The ' denotes a transpose, Interchanges the row and column index for each element`

Transpose is one that typically requires visually seeing it to understand. I highly recommend attempting this one, particularly with a matrix that has a different number of rows and columns.

Information

Functions like `size()` and `length()` provide information about the variable:

- `A = randi(20,3,5)`
- `size(A) % Returns the number of rows and columns in a matrix`
- `[R,C]=size(A) % Loads that return into the variable R and C`
- `length(A) % Returns a single value, the largest dimension of matrix. It's used more often for arrays since you know one dimension is equal to 1.`

We can now manage arrays, constructing and manipulating them after they've been created. For more options, the MATLAB documentation on Matrices and Arrays does go over many of the other ways to manipulate and mange these.

Character arrays

This is a good place to re-address character arrays since they can be manipulated the same as other arrays. They can also be used with functions like `upper()` and `lower()` which uppercase or lowercase all the characters in the array. `strcmp()` compares two character arrays or strings to determine if they're the same or not. `end` is a built-in input that refers to the last character in the array and `start` does the same but for the first character.

1. `A='John'`
2. `B='Doe'`
3. `Name=[A,' ',B]`
4. `upper(Name)`
5. `strcmp(lower(Name),'john doe')`
6. `strcmp(lower(Name(end–2:end)),'doe')`

Remember that character arrays use single quotes while strings use double quotes. You can manipulate strings but they cannot be treated simply as 1-D arrays.

String compare Show Correct Answer Show Responses

Given :
`p='BaNaNa';`
What is the result of `strcmp(p(3:6),'Nana')`?

A	True

B	False
C	Logical 0
D	Logical 1

Conditionals

Conditional statements or IFTT statements are founded on comparing if something is true or not and then doing one thing or another based on that. For that first part, defining an expression as true or false, we need to understand relational operators.

Relational Operators

Relational operators are operators that are used to compare items quantitatively. This is something you are likely to be familiar with once you see them. In mathematics and programming, an operator is a character that represents an action. We are familiar with the basic mathematic operators (+, -, *, /) but you are also likely with relational operators:

Operator	Example	Function	Example	Definition
`<`	`A<B`	`lt( )`	`lt(A,B)`	Less Than
`>`	`A>B`	`gt( )`	`gt(A,B)`	Greater Than
`<=`	`A<=B`	`le( )`	`le(A,B)`	LessThan or Equal to
`>=`	`A>=B`	`ge( )`	`ge(A,B)`	Greater Than or Equal to
`==`	`A==B`	`eq( )`	`eq(A,B)`	Equal to
`~=`	`A~=B`	`ne( )`	`ne(A,B)`	Not equal to

When compared, they return either a **1** if the statement is **True** or a **0** if the statement is **False**. These can be used for comparing scalars and arrays or matrices of the same size. They always work element by element.

For instance, run the following code and see what happens. Try using different relational operators comparing A and B as well:

```
A = [2 4 6; 8 10 12];
B = [5 5 5; 9 9 9];
A < B
A > 7
```

Relational Operator   Show Correct Answer   Show Responses

Given the following:

```
D=[1,2;3,4];
E=[0,2;4,2];
F=D~=E;
```

What does F equal?
Do not use spaces in your response and use commas to separate column elements.

Logical Operators

Along with relational operators, there are logical operators. These are used to return logical values based on the fulfillment of certain conditions, typically the combination of multiple logicals.

Name	Operator	Example	Function	Example	Definition
NOT	`~`	`~A`	`not()`	`not(A)`	True if value is not True
AND	`&`	`A&B`	`and()`	`and(A,B)`	True if A and B are True
OR	`|`	`A|B`	`or()`	`or(A,B)`	True if A or B is True
XOR			`xor()`	`xor(A,B)`	True if A or B is True but False if

					A and B are True

For those familiar with some of the more uncommon Logic gates, MATLAB does not have a built-in NAND, NOR, or XNOR.

There is a precedence involved in using logical operators, like when using basic math operators. `A|B&C` will be evaluated as `A|(B&C)`. This does not often come up but for more information on complex combinations of operators, see MATLABs order documentation on Operator Precedence.

There are other useful logical functions available:

- `all()`
  - Determine if all array elements are nonzero or true
- `any()`
  - Determine if any array elements are nonzero
- `find()`
  - Find indices and values of nonzero elements
- `islogical()`
  - Determine if input is logical array
- `logical()`
  - Convert numeric values to logicals
- `false`
  - Logical 0 (false)
- `true`
  - Logical 1 (true)

I recommend creating a few random arrays and then trying the above commands. If you click on the links, the MATLAB documentation contains some generic examples as well as more complex ones.

Branching

Now that we know how to compare things to retrieve a true or false determination, let's do something with that information. This is where Conditional Statements(aka IFTT) comes into play.

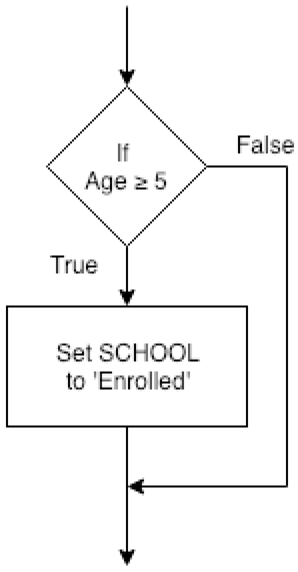
In our pseudocode, we use format IF statements as such:

```
IF expression
  statement;
ENDIF
```

In MATLAB, this becomes:

```
if expression
  statement;
end
```

Let’s walk through an example. Consider this flowchart snippet for an IF Statement:



That can be translated to this in Pseudocode snippet:

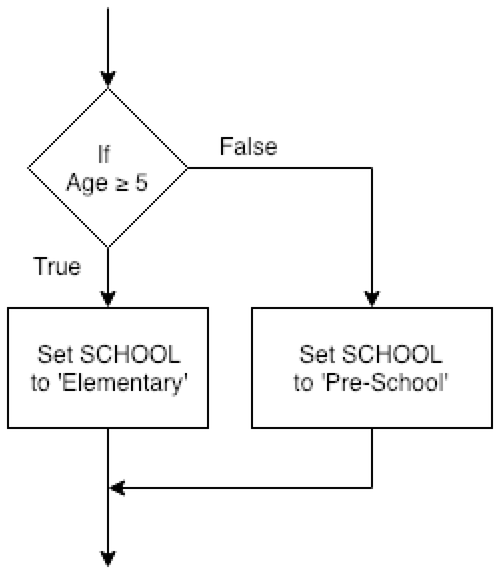
```
IF Age ≥ 5
  Set SCHOOL to 'Enrolled';
ENDIF
```

That then can be converted into MATLAB syntax:

```
if age>=5
  SCHOOL = 'Enrolled';
end
```

They’re pretty similar so it’s a quick conversion. Now let’s keep extrapolating this concept, what about the default case? What about when it doesn’t meet that IF criteria? That’s when we use an ELSE statement to cover that contingency, as shown in the following snippets:

Flowchart:



Pseudocode:

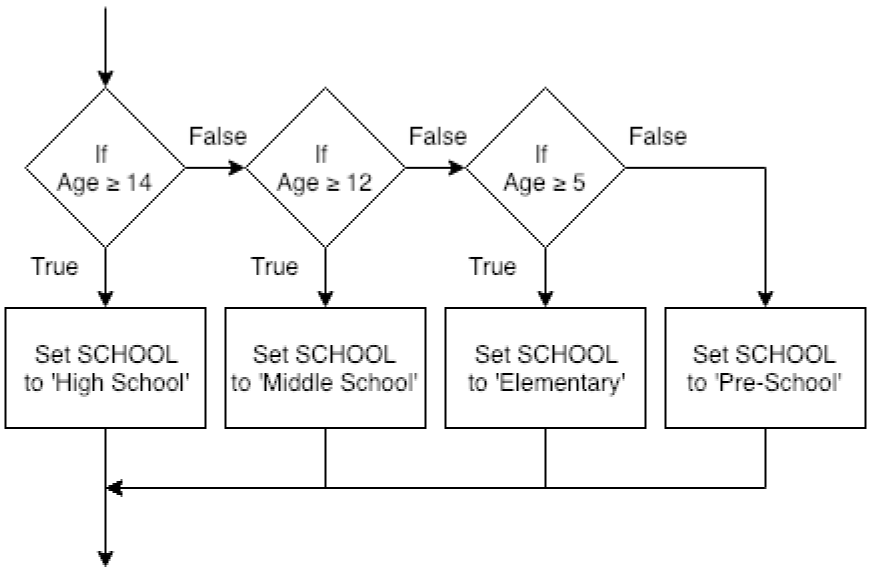
```
IF Age ≥ 5
   Set SCHOOL to 'Elementary';
ELSE
   Set SCHOOL to 'Pre–School';
ENDIF
```

MATLAB:

```
if age>=5
   SCHOOL = 'Elementary';
else
   SCHOOL = 'Pre–School';
end
```

Now we've covered the scenario if the expression is true or if the expression is false and we only care about 2 outcomes. What about when we want to consider more? Like maybe there's a situation where you have multiple criteria to consider in determining letter grades, I mean in determining where a student should be placed:

Flowchart:


If Age ≥ 14
False
If Age ≥ 12
False
If Age ≥ 5
False
True
True
True
Set SCHOOL to 'High School'
Set SCHOOL to 'Middle School'
Set SCHOOL to 'Elementary'
Set SCHOOL to 'Pre-School'

Pseudocode:

```
IF Age ≥ 14
   Set SCHOOL to 'High School';
ELSEIF Age ≥ 12
   Set SCHOOL to 'Middle School';
ELSEIF Age ≥ 5
   Set SCHOOL to 'Elementary';
ELSE
   Set SCHOOL to 'Pre-School';
ENDIF
```

MATLAB:

```
if age>=14
   SCHOOL = 'High School';
elseif age>=12
   SCHOOL = 'Middle School';
elseif age>=5
   SCHOOL = 'Elementary';
else
   SCHOOL = 'Pre-School';
end
```

But wouldn't this change the value of `SCHOOL` for all of them? For instance, if the student is 13 they would be greater than or equal to 12 but also greater than or equal to 5 so wouldn't it get a **False** on the first IF statement, a **True** on the second and `SCHOOL` set to `'Middle School'`, and then a

True for the third IF statement as well, resetting the value of `SCHOOL` to `'Elementary'` erroneously? ***No.*** The first expression found to be true will perform that statement and then skip to the end of the IF statement.

Types of if statements. Show Correct Answer Show Responses

Which implementation of an if statement is valid in MATLAB?
Version A:

```
if X < 3
    disp('Something');
end
```

Version B:

```
Y = X < 3;
if Y
    disp('Something');
end
```

A	A
B	B
C	Both

Loops

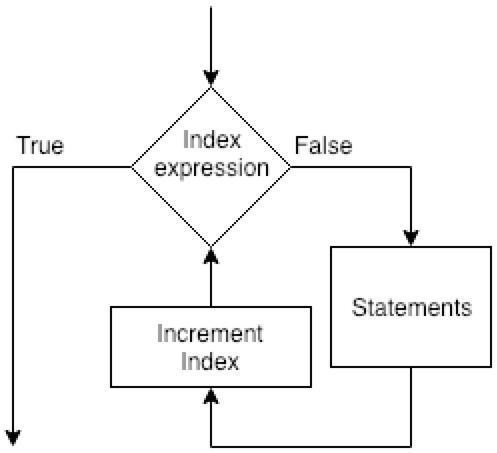
The real power in computational tools like MATLAB are their ability to repeat a task. That where loops get factored into our programs:

For Loops

We have only seen these so far in Pseudocode. We will use these more in class.

For Loops are used to repeat statements a certain number of times. This is particularly useful when you want to do the same things multiple times or use something that also repeats. For instance, if you want to print off elements in a list or make individual manipulations to an array. It uses an index expression meaning something set up to cycle through by incrementing an index automatically until a criteria is met. In Matlab, this is typically cycling through the elements of an array.

We can represent a Flow chart as generically as possible in a Flowchart below:



In our pseudocode, we use the representation:

```
FOR index expression
  statements;
ENDFOR
```

In Matlab, this becomes:

```
for index = values
  statements;
end
```

You say that you will loop through these statements for all elements in the array **values**. The `index` changes each time to be the next element in **values**, starting with the first on the first run of the loop. The loop ends when there are no more elements in the array **values**.

Here's an example of how we can use the index in the loop and how it changes:

```
r = randi(10,1,5);
for i=1:length(r)
    fprintf('Element %i of the array is %i\n',i,r(i))
```

```
end
```

Another common practice in for loops is the use of a counter. If you were going through a jar of spare change counting quarters, you repeat the steps over and over again until all coins are gone but keep a running count specific for the counters , not just the index of coins within that loop.

```
r = randi(10,1,5);
k = 0; % Initialize the counter outside the loop
for i=1:length(r)
    fprintf('Element %i of the array is %i\n',i,r(i));
    if r(i) > 5 % Check counting criteria
        k = k + 1; % Increase counter by 1
    end
end
fprintf('There are %i elements greater than 5.\n',k);
```

Notice that we've nested an if statement in there. We can nest as many if statements as we want as well as other for loops.

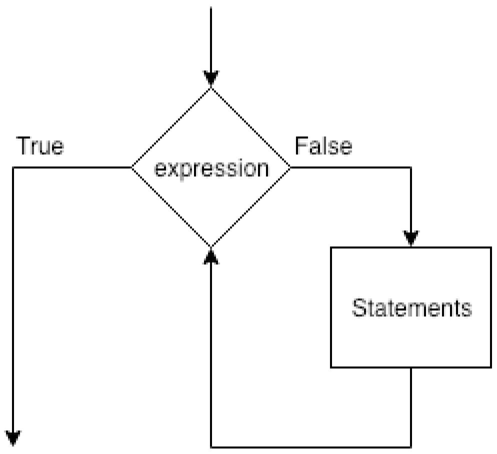
Note that in the examples above, the `1:length(r)` is just creating an array. The for loop will cycle through that array, setting the value of `i` equal to each successive element.

```
X = [5 8 9 10 2 4 18];
for i=X
    fprintf('In this iteration, the value of i is %i\n',i)
end
```

While Loops

We can also use While loops which aren't based on a specified number of iterations but rather iterating until a certain criteria is met. You do need to have some aspect of that criteria have the potential to change with each iteration. Otherwise, the loop will run forever.

Here is a generic representation in a Flowchart:



In our pseudocode, we use the representation:

```
WHILE expression
    statements;
ENDWHILE
```

In Matlab, this becomes:

```
while expression
    statements;
end
```

Let's do an example:

```
r = randi([-5,5],50,1);
i=1; % Indicators for the while criteria
fprintf('Element\tValue\n'); % \t is for tabs
while r(i)<=3&i<length(r)
    fprintf('%2i\t%2i\n',i,r(i));
    i=i+1;
end
fprintf('Loop ends at element %i of %i\n',i,length(r));
```

Do While Show Correct Answer Show Responses

Does MATLAB have a `Do While` loop?

A	Yes
B	No

Now that we've gone over many concepts, let's try and use some:

Pre-Lab Exercise

The following will have you write a script that utilizes some of the above concepts

P6L2 – A Grade Conversion App

Final grades are posted to your transcripts in letter form (A, B, C, etc) but are typically calculated as a weighted average percentage of your work. You will be writing an application that handles that conversion based on the following rules.

Percent Input	Grade Output
90.0% and above	A
80.0% to less than 90.0%	B
70.0% to less than 80.0%	C
60.0% to less than 70.0%	D
less than 60.0%	F

Pseudocode

As always, we start by going over our logic. We can do this in a few ways, for this course there is a requirement that you do so in a loosely formatted pseudocode or flowchart. So let’s start with a a very basic pseudocode that gives us a high altitude view of what we're going to do. We can always update it as we breakdown and understand the required logical steps more. We know we need to do the following:

- Get the percentage grade to convert
- Make that conversion
- Let the user know what the conversion is

Simple as that. let’s write it as loosely formatted pseudocode:

```
1. PROGRAM Grader:
2. Ask user for the final grade as a percentage;
3. Compare that numeric grade to the conversion rules;
4. Return the letter grade to the user;
5. ENDPROGRAM
```

This is a good start but we should get into some details about making the conversion. These are simple rules we’ve been given, if this criterion is met then this is the output. This is textbook **IF THIS THEN THAT**, the *foundation* of modern programming. We introduced this concept in the Algorithmic Thinking and Representation Lab so let’s utilize that here:

```
 1. PROGRAM Grader:
 2. Ask user for the final grade as a percentage;
 3. Compare numeric grade to the conversion rules;
 4. IF percentage is 90.0% or above THEN
 5.     Letter grade is an A;
 6. ELSEIF percentage is 80.0% to 90.0% THEN
 7.     Letter grade is a B;
 8. <Continue ELSEIFs for all grading rules>
 9. ENDIF
10. Return the letter grade to the user;
11. ENDPROGRAM
```

We can get away with not writing out all the logic with something like in line 5 because this isn’t for the computer’s interpretation but for a human reader. They can extrapolate that information, we’ll have to do the extrapolation for the computer when we get to writing the code.

Writing the code

It's now time to start your program. Create an M-File for this part of the assignment. Make sure you remember to:

- Name it properly
- Include a title block at the top
- Include the housekeeping code

You can use this to get started:

https://raw.githubusercontent.com/boconn7782/CourseCode/main/MATLAB/P6L2_Starter.m

We already have a pseudocode to work with so we can of course convert that to our comments. We can even combine it with the first version since we'll be converting the pseudocode to code for the second but we still need a comment explaining what's happening.

```
1. % Provide user information and instruction on the program
2. % Ask user for the final grade as a percentage
3. % Compare that numeric grade to the conversion rules
4. % IF percentage is 90.0% or above THEN
5. %   Letter grade is an A
6. % ELSEIF percentage is 80.0% to 90.0% THEN
7. %   Letter grade is a B
8. % Continue ELSEIFs for all grading rules
9. % Return the letter grade to the user
```

Line 1 is just good user interface practice. You always want to let your user know what to expect from the program and what to do. Programs are limited in their capabilities to respond so you want to make sure the user knows what to input in order to get their desired output.

We took line 3 from our first pseudocode pass since that'll describe what's happening in lines 4-6, which will likely get converted to regular code. That's a trait of good pseudocode, if done well, thoughtfully prepared, and formatted properly, it can quickly convert to correct code syntax and therefore working code.

Coding Line by Line

We have a scaffold, let's work through it. First a reminder about how these lessons are formatted, the example code-blocks will contain references to code that you should be capable of generating without using the actual code syntax. These will be represented by `< >` brackets. In these cases, you should replace the `< > and everything in them` with the necessary code.

Also, if discussing something in a block of code, line number references will be used. The lesson did so in the previous section. These line numbers will change as we progress and add to the code. In your own code, you may add spaces or use more lines of code than I do, therefore your line numbers might not match up to this text. THAT IS FINE. Just reference the provided code snippets here and compare to your own as you go. The programs generated in class shouldn't be so long that this becomes an issue.

Now back to the code. Let's build off the scaffold. We know how to display information to the command window so we can do that. We also know how to take in information

```
 1. % Provide user information and instruction on the program
 2. <Code to display a welcome message for the user>
 3. <Code to display a description of the program's purpose>
 4. <Code to display an instruction for the user on how to work with it>
 5. % Ask user for the final grade as a percentage
 6. <Input a numeric grade and store it for future use>
 7. % Compare that numeric grade to the conversion rules
 8. % IF percentage is 90.0% or above THEN
 9. %   Letter grade is an A
10. % ELSEIF percentage is 80.0% to 90.0% THEN
11. %   Letter grade is a B
12. % Continue ELSEIFs for all grading rules
13. % Return the letter grade to the user
```

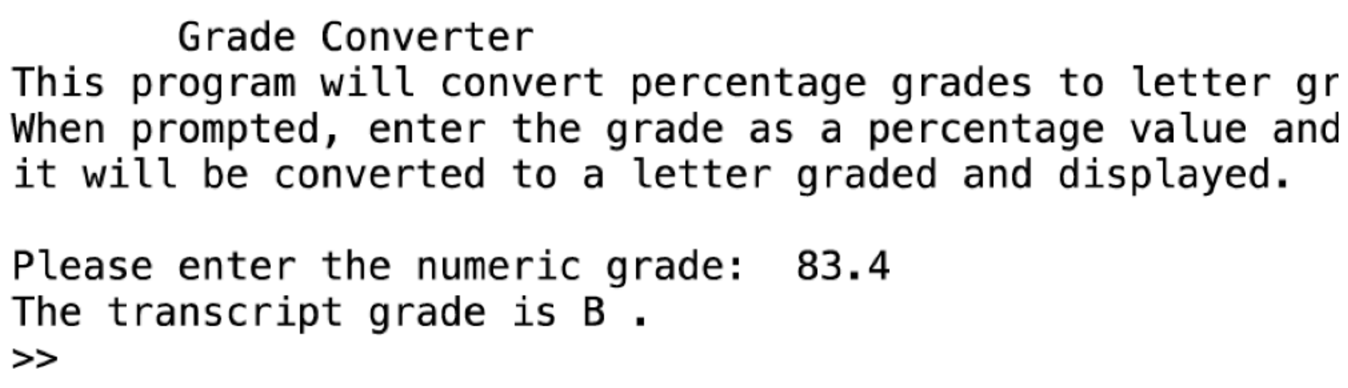
Now we've hit line 9. We have some general knowledge about comparisons and conditional statements, but only from the pseudocode/flowchart standpoint. We need to learn more about this in the context of Matlab so it's time for our first break in PART 1.

Now that we now how to implement IFTT in MATLAB, we can start integrating that into our code.

```
1. % Provide user information and instruction on the program
2. <Code to display a welcome message for the user>
3. <Code to display a description of the program's purpose>
4. <Code to display an instruction for the user on how to work with it>
```

```
% Ask user for the final grade as a percentage
<Input a numeric grade and store it for future use>
% Compare that numeric grade to the conversion rules
if NbrGrade>=90.0
    LtrGrade='A';
elseif NbrGrade>=80 & NbrGrade<90
    LtrGrade='B';
< Continue ELSEIFs for all grading rules >
end
% Return the letter grade to the user
< output the letter grade to the command window. See the example
below. >
```

It should now have an output similar to the following:



```
       Grade Converter
This program will convert percentage grades to letter gr
When prompted, enter the grade as a percentage value and
it will be converted to a letter graded and displayed.

Please enter the numeric grade:  83.4
The transcript grade is B .
>>
```

Testing your code

Make sure to run your code a few times and try it with several different numbers. Always try edge cases as well. What happens when you input 70.0 or 80.0? Is there an issue if you try 90.00001 or 89.99999? What about for some of the other edges like around 90% or 60%?

I'm also hoping you may have recognized the redundancy of an aspect of the provided code. It's fine if you did not but i'm going to point it out so you recognize it in the future.

Look again at:

```
if NbrGrade>=90.0
    LtrGrade='A';
elseif NbrGrade>=80 & NbrGrade<90
```