CLPS Matlab Programming Workshop

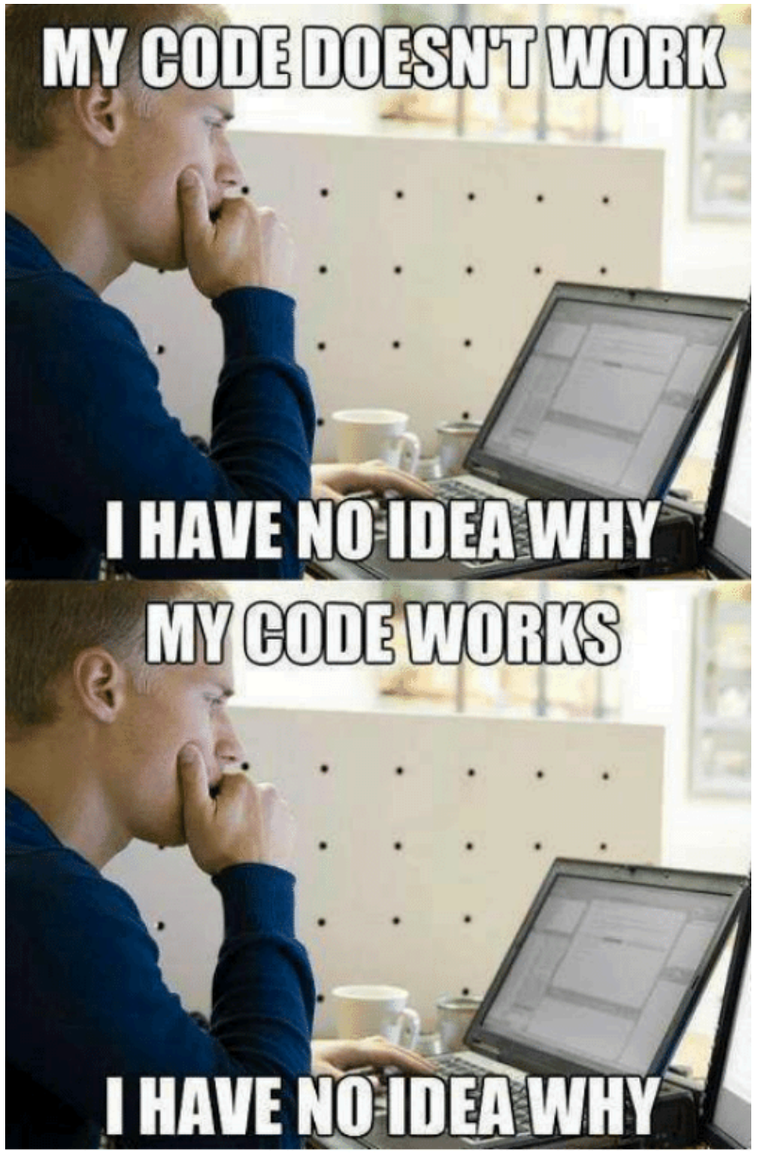
Session 1 of 3

Written by Jae-Young Son

November 28, 2018

Expectation Management

- Programming experiments ≠ scientific computing
  - If you want to learn how to use Matlab to do things like scientific computing and data analysis, this workshop will ***not*** teach you those skills
  - This workshop will only teach you how to program the kinds of computerized experiments you might be expected to create for psychology studies
- Mileage may vary
  - Nobody becomes an expert overnight
  - Learning to code involves lots of practice and aha! moments
  - The more you put in, the more you'll get out



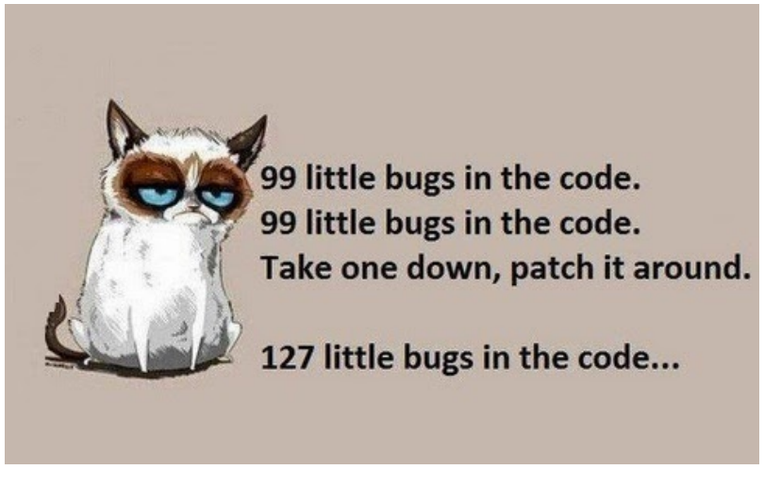
Words of Advice, part 1

- Be patient with yourself!
  - Programming is not a trivial thing to learn: it's like learning a new language
  - Like with all things, getting good at programming takes lots and lots of practice

- I often go into detail about the conceptual underpinnings of why computers work the way they do
  - This is not just for historical interest!
  - Understanding the logic of computers really helps you understand why your code works / doesn't work

Words of Advice, part 2

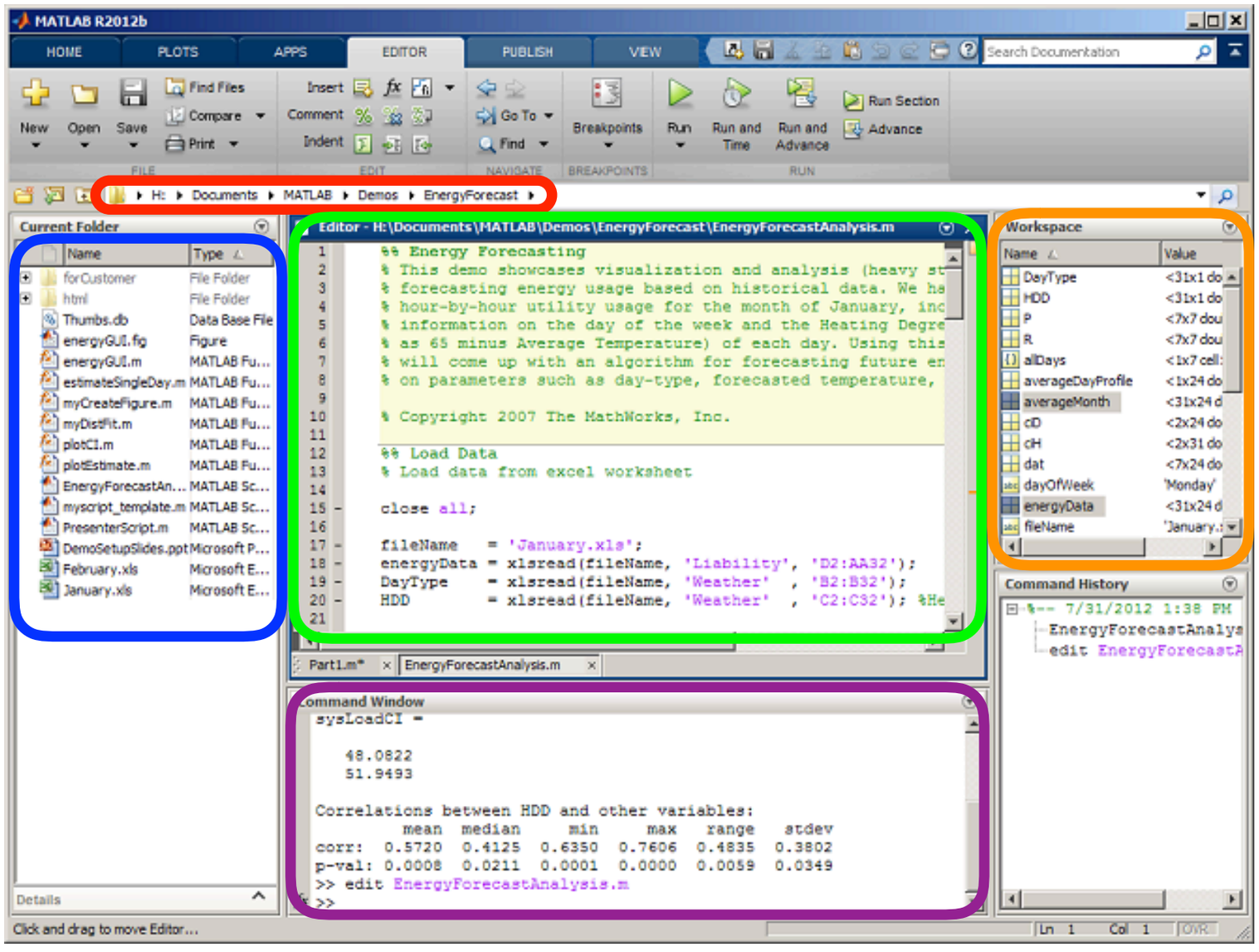
- As a former mentor once told me, "Debugging is not the thing preventing you from programming. Debugging IS programming."
- 90% of your coding failures will involve stupid syntax errors, misspellings/typos, losing count of your parentheses, etc. It is frustrating, but lean into it. This happens to EVERYONE, including friends of mine who work for Google, Facebook, etc
- Pay attention to Matlab's error messages (which are generally pretty informative), and be smart about how you Google solutions to buggy code



Roadmap

- **Session 1: A programmer's essential toolkit**
  - What is information processing?
  - Variables
  - Functions
  - Arrays and matrices
  - Control logic
- **Session 2: Computer graphics**
  - Shapes
  - Animation
  - Placement of objects in space
  - Layering
  - Images
- **Session 3: Putting it all together**
  - Monitoring and recording user input (from keyboards and mice, and getting RTs)
  - Nesting functions
  - Simple economic experiment

Getting oriented



Directory path: tells you what directory you're in

Directory listing: tells you what files are in your current directory

Command window: where you enter commands and/or see the output from the commands you've entered

Editor window: where you can write scripts (collections of commands that are meant to be run in sequence)

Workspace: shows you all of the variables you've created

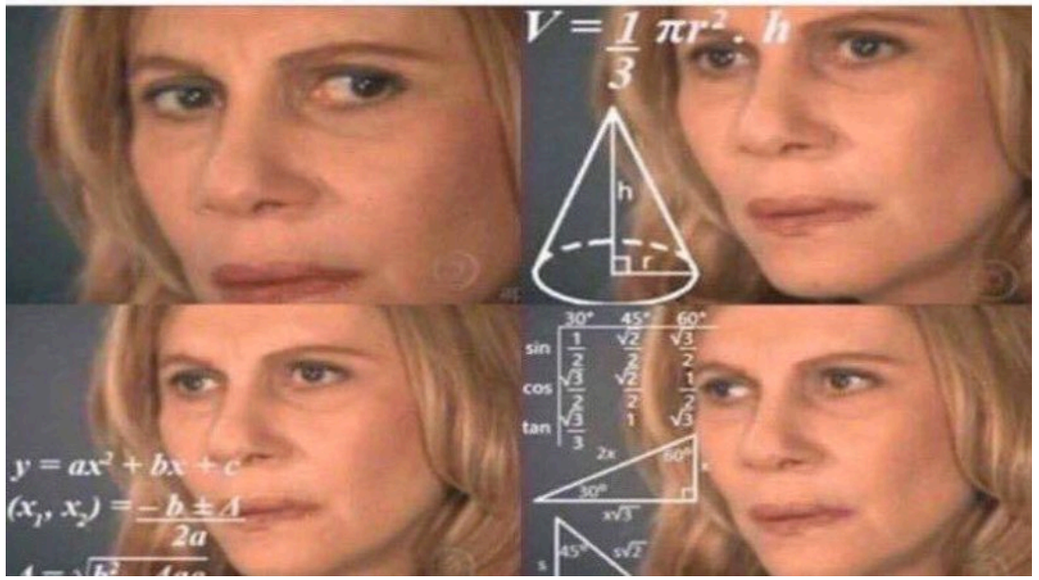
Getting oriented

- You can run any command you want using the command window, but it's often easier to write things in a script
  - Working with the command window is like drawing a picture on an Etch-A-Sketch: you can only run one command at a time, and it must be done manually
  - Working with scripts is like drawing pictures inside a notebook: you can write out your entire program ahead of time and automatically run all of your pre-existing code at once
- To create a new script, press:
  - command+N on Mac
  - control+N on PC

It takes upwards of 4 days to perform a single Google search using this method

1.1 – Information Processing

- What's a computer?
  - A machine that performs computations
- What's a computation?
  - Informally, the process of taking some input information and *doing something with it* so that you get a useful output
- How does your computer implement computations?
  - Your code tells the computer what kinds of information processing to perform

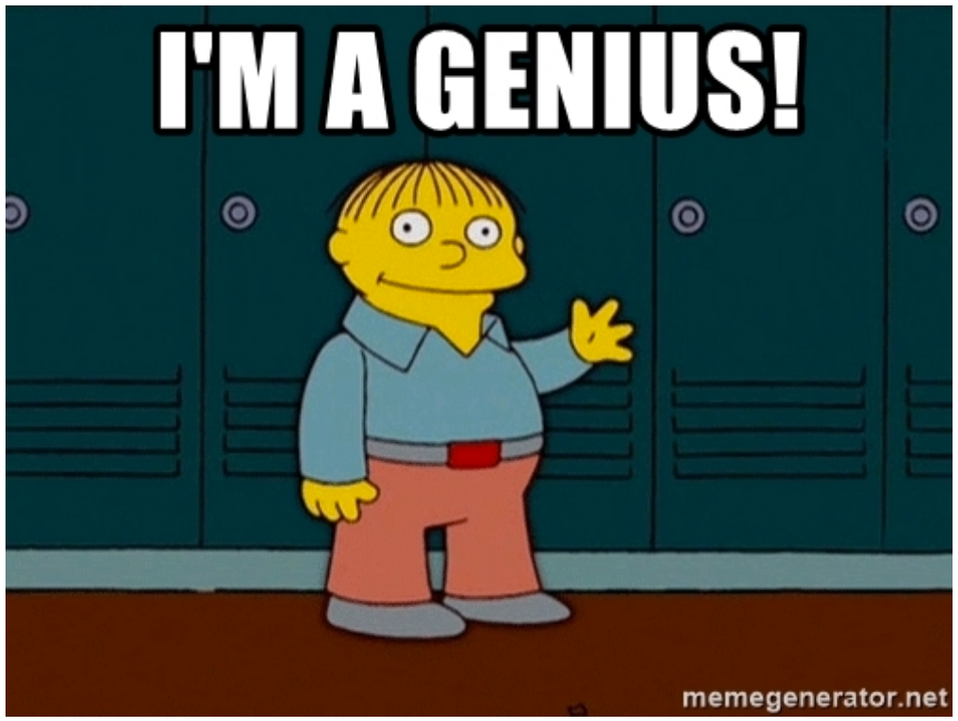


A diagram of what your computer's CPU is doing at any given time

1.1 – Information Processing

- Simple example: type 7+3 into the command window
- Congrats! This is information processing. You asked your computer to:
  - Take two pieces of information (the numbers 7 and 3)
  - Process this information using a computational operation (addition)
  - Return the output of this computation (10)

The abacus is an example of a primitive specific-purpose computer!



1.1 – Information Processing

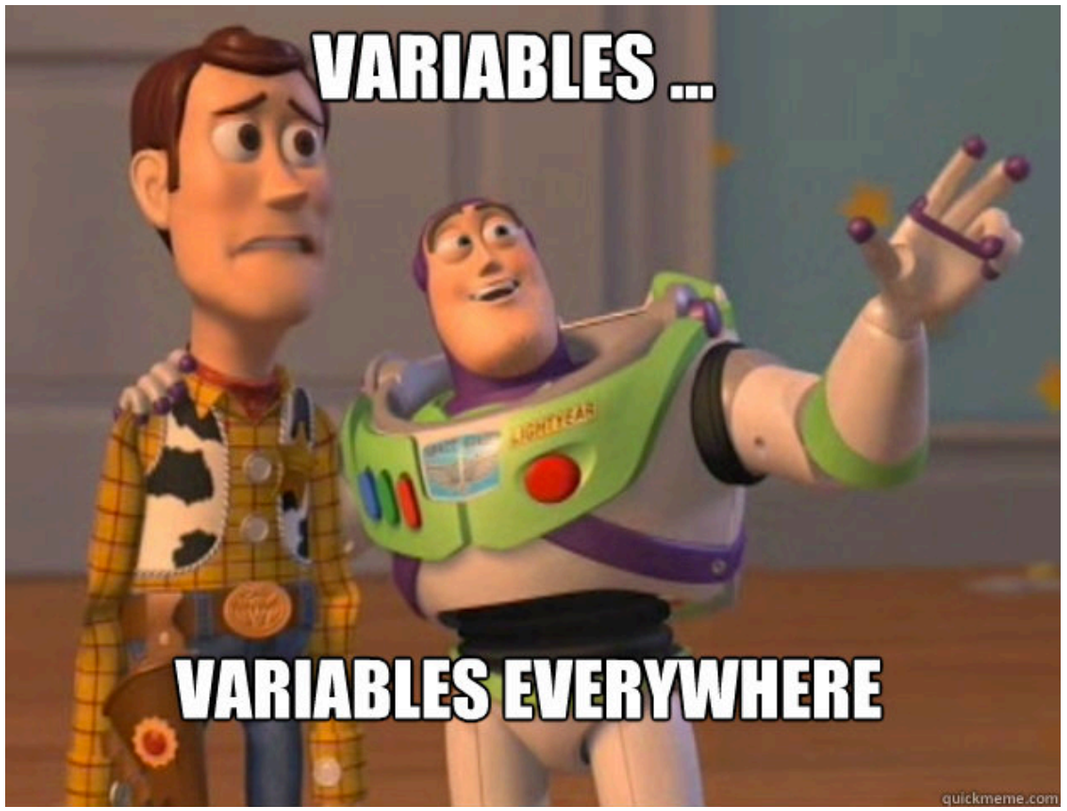
- Try it yourself: what's the effect of adding spaces?
  - `7+10`
  - `7 + 10`
  - `7 +  10`
- Try it yourself: how does Matlab process long operations?
  - `7*6+4/12-8/2`
  - `(7*6)+(4/12)-(8/2)`
  - `7*(6+4)/12-8/2`
  - `7*((6+4)/12-8)/2`

2.1 – Creating Variables

- You'll notice that when you use Matlab as a simple calculator, you get outputs that look something like this:
  - `2*2`
  - `ans = 4`
- Try it yourself: what happens when you type in the following commands?
  - `ans * 2`
  - `ans * ans`
- Matlab automatically saves the answer to user operations in a variable called `ans`

2.1 – Creating Variables

- **What's a variable?**
- Variables in computer programming work in much the same way as variables in algebra
- Variables are like a container that can "store" a single piece of information, which can then be used to perform computations
- For now, let's stick to numbers, which make the logic of variables easy to understand



2.1 – Creating Variables

- Try it yourself!

- `% The percent sign indicates a comment. Anything that comes afterwards is invisible to the program! You can use comments to make notes to yourself`
- `hello = 5;  % A semicolon at the end of a command “suppresses” the output such that the result is not printed in the console window`
- `hello * 5`
- `goodbye = hello;`
- `hello - goodbye`
- `hello = hello - goodbye`
- `hello - goodbye`

2.1 – Creating Variables

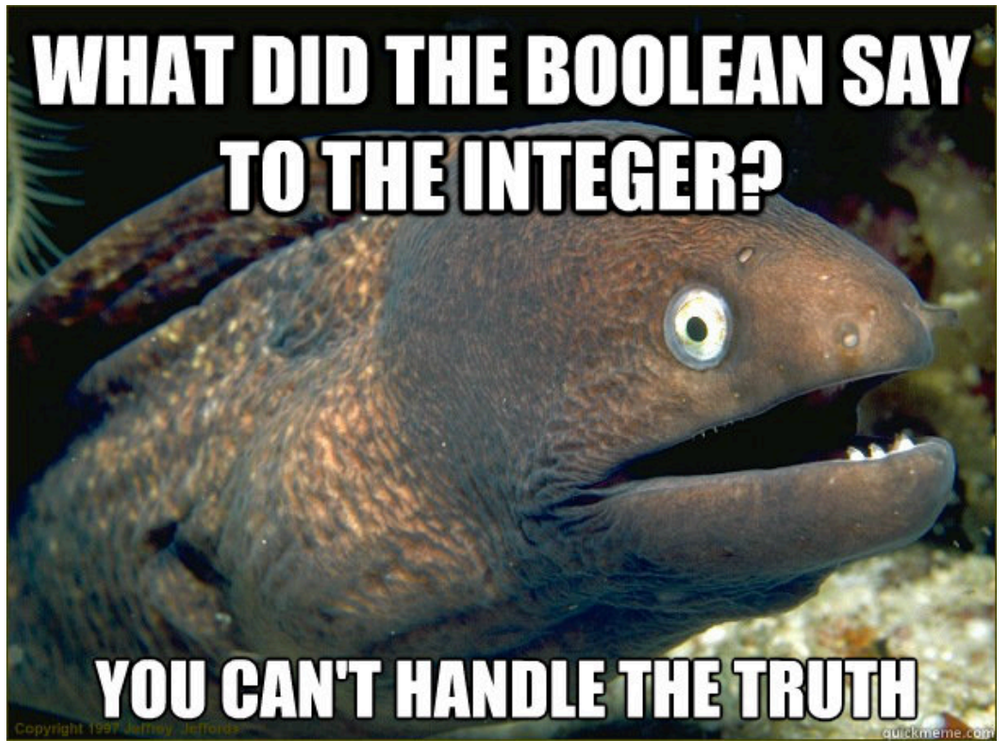
- Here are things that are important to note from the last slide
- You **assigned** the variable `hello` a value by specifying `hello = 5;`
- You could use this variable `hello` in any way you liked *without changing its value* UNTIL you assigned it a new value `hello = hello - goodbye;`
- Once you assigned `hello` a new value, you **overwrote** the original value of 5
- This is because **a variable can only store a single piece of information** (at least until we get to arrays and matrices, but don't worry about that yet!)

2.2 – Data Types

- Computers are capable of processing different types of information (i.e., data)
  - Not just numbers!
  - Variables can store any of these data types
- However, computers are also literal to an extreme degree
  - A human would have no problem evaluating the statement `fifty-five - 5`
  - However, a computer would immediately freak out because you've just mixed a string (`fifty-five`) with a number (`5`)
- To see why, imagine that a math teacher asked you to solve `fish minus five`. This makes no sense because fish and five are fundamentally different types of data

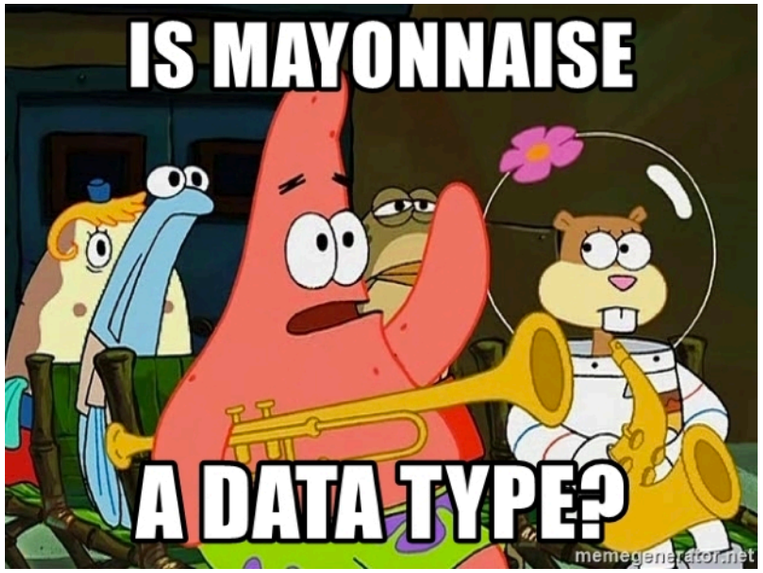
2.2 – Data Types

- Here's a partial list of data types you'll encounter regularly:
  - Floating-point numbers (e.g. 7.14, -3.14159)
  - Integers (e.g. 1, 99, -6)
  - Characters and strings (e.g. "It was a pleasure to burn.")
  - Boolean (e.g. TRUE, FALSE)
  - Structures (we'll cover these later)
  - Cell arrays (we'll cover these later)



2.2 – Data Types

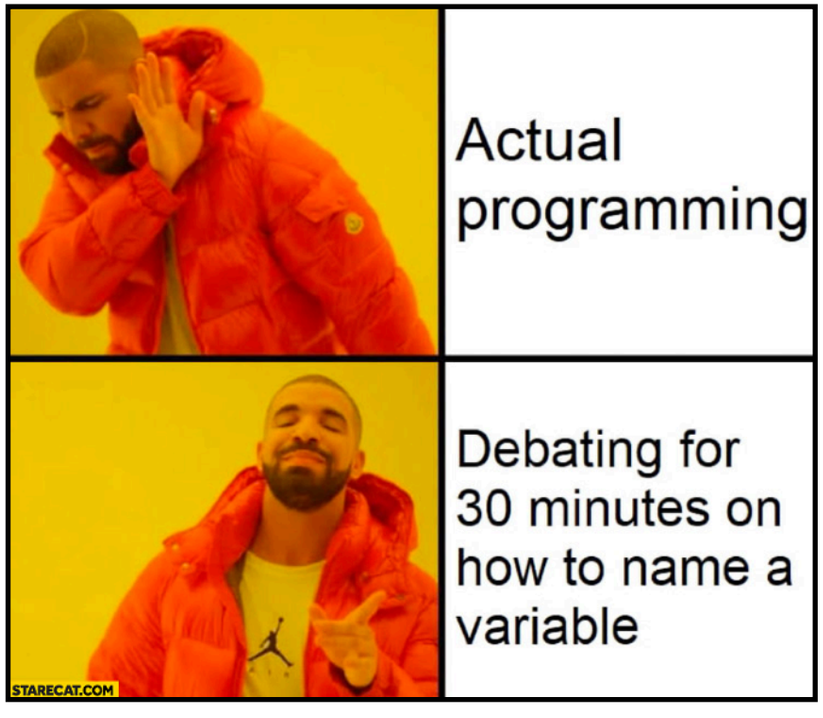
- **Declaring data types**
  - Languages like Java require programmers to declare a data type whenever they create a new variable
  - The variable cannot contain any data that does not match the variable's declared data type
  - This prevents programmers from accidentally telling the computer to evaluate things like fish minus five
- **Matlab does not require you to declare data types**
  - For better or for worse (usually for worse) Matlab's a little more lenient about allowing you to assign data to variables
  - However, it's good practice to keep track of these things as you're working
  - You'll find that you can significantly cut down on debugging time if you keep track of your variable types as you go!

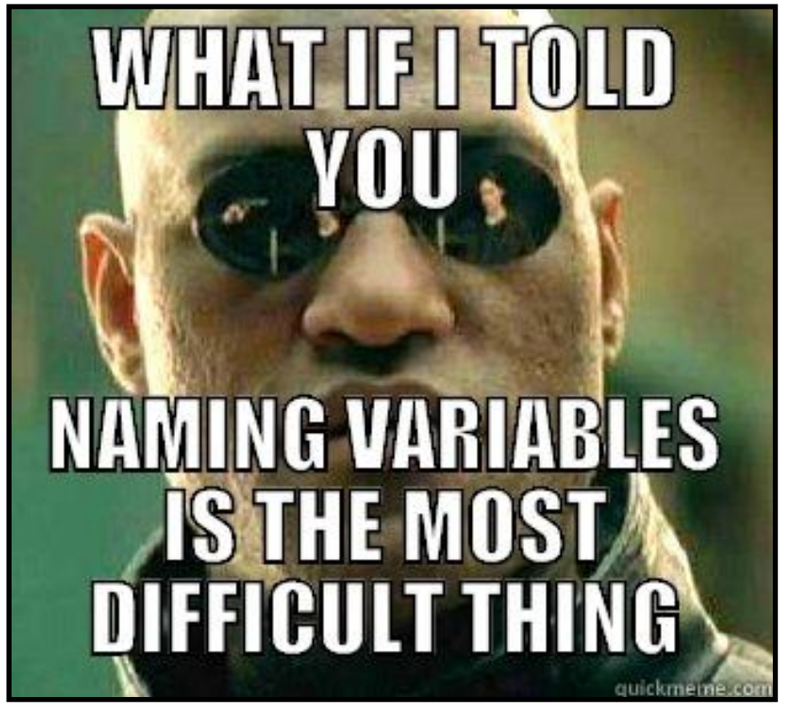


grumble grumble matlab is hippy-dippy new age programming where data types are some post-truth social construct

2.3 – Naming Variables

- When you get good at programming, one of the hardest things you'll be asked to do as a programmer is to name variables
- You think I'm joking, but I'm really not
- Google image search agrees with me:





2.3 – Naming Variables

- Here are two conceptual principles for naming variables:

    1. A variable name should describe the variable's contents clearly enough that somebody else can pick up your code and immediately know why any given variable exists
    2. At the same time, your variable name should be as short as possible

- On paper, these two principles don't sound very complicated
- However, once you start programming more substantial tasks, you'll quickly figure out that it's hard to accomplish even *one* of these objectives

2.3 – Naming Variables

- Permissible variable names:

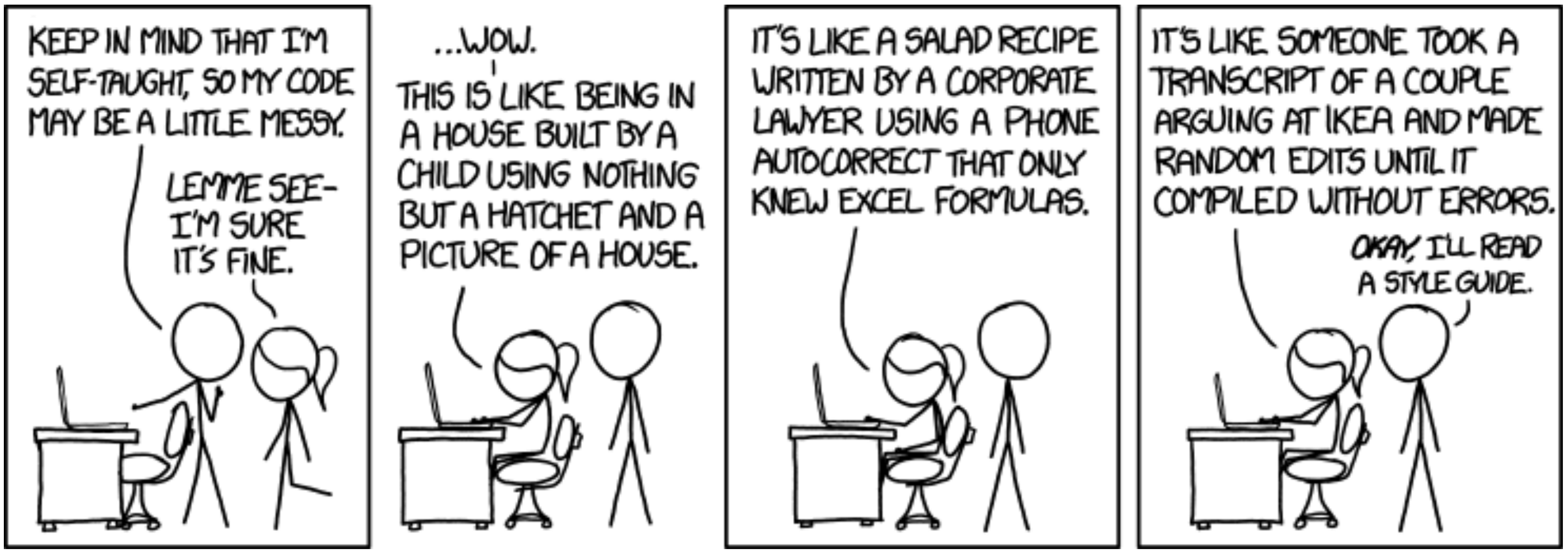
    1. Matlab variable names must begin with a letter. They can contain numbers and (some) special symbols, but they must begin with a letter
    2. Note that Matlab is case sensitive; "hello" and "Hello" are treated as different variables
    3. There can never be spaces in your variable name
    4. Avoid using variable names that are reserved by Matlab
        - For example, you can theoretically define a new variable `ans` containing whatever data you want
        - However, as soon as you perform a new computation, that data inside `ans` is going to be overwritten

2.3 – Naming Variables

- `thisisavariablename` is a garbage variable name because it's nigh-impossible to read
- Camel Case
  - One popular convention is called camel case (because it looks like the bumps on a camel)
  - Using camel case, that variable name would instead be `thisIsAVariableName`
  - Incidentally, you're already used to seeing camel case because of products like the iPhone and PowerPoint
- Underscores
  - Another popular convention is to use underscores as spaces
  - Using this convention, that variable name would instead be `this_is_a_variable_name`
- Which convention you ultimately use is a matter of personal taste, but stick with it!
- There's nothing more maddening than trying to remember whether or not a variable you named 200 lines ago is formatted using camel case or underscores

2.3 – Naming Variables

- …Who cares about variable names?
- Oh, you sweet summer child



2.3 – Naming Variables

- …Who cares about variable names?
- The following code relies on concepts you haven't learned yet, but I bet you could figure out what this snippet does:

```
if playerType == 2
      randomNumber = randi(100);
      if randomNumber <= 36
            payoutToSubject = 0;
      elseif randomNumber > 36 & randomNumber <= 41
            payoutToSubject = endowment/2;
      end
end
```

2.3 – Naming Variables

- …Who cares about variable names?
- Even with slightly condensed variable names, I bet you could figure this out:

```
if playType == 2
      randNum = randi(100);
      if randNum <= 36
            payToSub = 0;
      elseif randNum > 36 & randNum <= 41
            payToSub = endowment/2;
      end
end
```

2.3 – Naming Variables

- …Who cares about variable names?
- Yeah, no. Good luck:

```
if pType == 2
      rNum = randi(100);
      if rNum <= 36
            pay2S = 0;
      elseif randNum > 36 & randNum <= 41
            pay2S = endwmnt/2;
      end
end
```

3.1 – What's a function?

- Suppose that you want to find the absolute value of a number. You could go about writing an algorithm to accomplish this…

```
number = input('Please enter a number: ');
if number > 0
    result = number;
else
    result = number * (-1);
end
disp(result)
```

- …or, alternatively, you could just use the built-in Matlab function.

```
abs(-27)
```

3.1 – What's a function?

- A function is simply a pre-defined chunk of code that performs a useful operation
  - For example, `abs(x)`, `sqrt(x)`, `sin(x)`, and `log(x)` are all functions that perform common mathematical operations
  - Functions are designed to hide the inner workings of the code for your convenience (i.e. you don't want to write an algorithm every time you want to solve a square root, and you don't care how it's solved – you just want to know the answer!)
- In research psychology, most of the coding you will be doing is stitching together pre-existing functions

3.2 – The anatomy of a function

- Functions have two essential components: the **function call**, and the **argument(s)** associated with the function
- Let's look at the simple case of `abs(-27)`
  - Here, `abs` is the function call because it calls the function that computes the absolute value of the argument
  - `-27` is the argument
- The function `abs(x)` only takes one argument, which is a number
- Other functions can take multiple arguments, which can be any arbitrary data type

3.2 – The anatomy of a function

- Try it yourself!

```
abs(99)

abs(-99)


% This returns an error
abs(tree)


% This does NOT return an error. Why?
tree = -200;
abs(tree)
```

3.2 – The anatomy of a function

- Functions can look scary, but they're not so bad once you understand what their arguments are
- For example, take this example PsychToolbox code…

```
% Display 2-sec fixation cross
before advancing to next trial

DrawFormattedText(wPtr, '+',
'center', 'center', white);

Screen(wPtr, 'Flip');

WaitSecs(2);
```

- `DrawFormattedText` is capable of taking 12 arguments, but we can see that in this example, it only takes 5 (this is because some arguments are optional)
- `wPtr` refers to the computer screen to which the formatted text is drawn
- `'+'` is literally the fixation cross itself
- The first `'center'` specifies the x-coordinate at which the text is drawn
- The second `'center'` specifies the y-coordinate at which the text is drawn
- `white` refers to the color of the text

3.3 – HELP

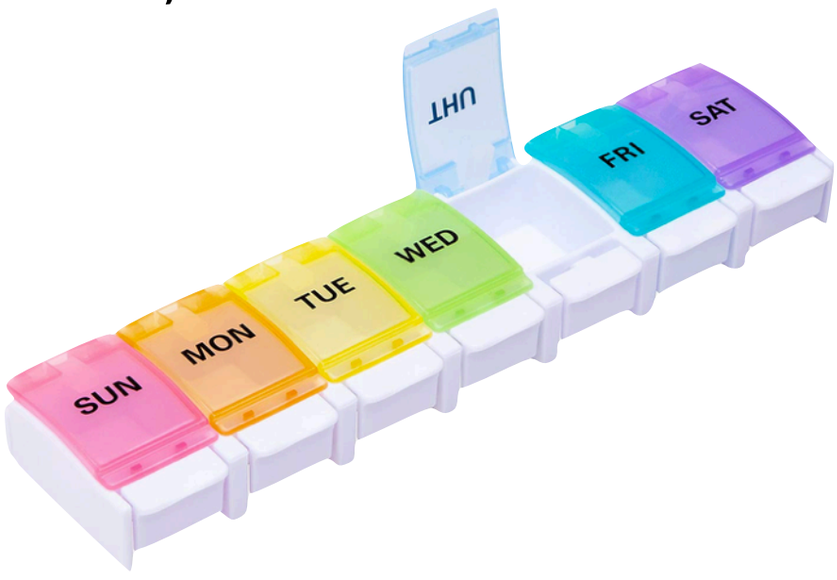
- You will inevitably find yourself in situations where you're asked to decipher the cryptic arguments of mysterious functions you've never used before
- Let's suppose that you had no idea what `WaitSecs` does
- There are three things you should do if you find yourself in this situation, and they are presented below in the order in which you should do them:

1. Use the built-in `help` command – `help WaitSecs`
2. Google – psychtoolbox waitsecs
3. Ask a labmate – last resort

4.1 – Scalars, Arrays, and Matrices

- Suppose that you take pills every day, and that you want to store them inside a container
- Now imagine trying to store your daily pills inside a mint tin that you replenish every day – very inefficient!
- So far, all of the variables we have created have been ***scalar***
- That is, they can only store a single datum

- Now imagine that you store your pills inside a 7-day pill organizer instead – much more efficient!
- This is the definition of an ***array*** – a container for data that would otherwise need to be stored as separate scalars
- Useful for storing a participant's reaction times on trial 1, trial 2, trial 3, etc…



4.1 – Scalars, Arrays, and Matrices

- But what happens if you need to take pills twice a day?
- This is where a ***matrix*** would come in handy! A matrix is just an array with a variable number of rows and columns
- In our pill example, you would want a matrix with 7 columns (one for each day of the week) and 2 rows (morning and night)
- Useful for storing a participant's choice behaviors *and* reaction times on trial 1, trial 2, etc…

4.2 – Numerical matrices

- Creating a matrix
  - Numerical matrices are defined using square brackets
  - You can differentiate elements within the matrix using a space or a comma
- Try it yourself:
  - `[1 2 3 4 5 6]`
  - `[1 2 3; 4 5 6]`
  - `[1; 2; 3; 4; 5; 6]`
  - `[1 2; 3 4; 5 6]`
  - `[1 2 3; 4 5; 6]  % Why does this return an error?`
- Try playing around with these useful matrix operations and see what happens!
  - `test = 1:10  % What values get stored inside this variable?`
  - `test = 5:10  % Recall that a variable can only store a single piece of information at a time. A container counts as a single piece of information!`
  - `test'  % This operation is called transposition`
  - `1:2:10  % How is this different from the first example? Hint: try playing around with the number in the middle`

4.3 – Cell matrices

- Annoyingly, Matlab uses square brackets for two very different operations:
  - The first, as we've seen, is to create a numeric matrix
  - The second is to concatenate (i.e. combine) text
- Try it yourself:
  - `['This is', 'an extremely annoying', 'feature']`
- So how would we go about creating a matrix that contains text instead of numbers?
- The answer is to use something called a ***cell matrix***, which is pretty much every person's least favorite thing about using Matlab

4.3 – Cell matrices

- Why cell matrices are good
  - You can store *anything* you want inside of a cell matrix: numbers, strings, even other matrices!
  - This gives you a lot of flexibility to define matrices of arbitrary complexity, which gives you a lot of power
- Why cell matrices are bad
  - Once you store something inside a cell matrix, it becomes a ***cell data type***.
  - It doesn't matter what it looks like! It's not a number! It's not a text string! It's a cell!
  - Historically speaking, it rarely ends well when you give humans access to flexibility, arbitrary complexity, and power

4.3 – Cell matrices

- Why you have to use cell matrices even though they're bad
  - Because Matlab was designed to work with numbers, and everything else was a poorly thought out afterthought
- More explicitly, this means that the only method of creating and using text matrices is indirectly through the cell data type*

A photograph of the primary developer coming up with cell matrices as a solution for working with text data

** Recent versions of Matlab have implemented string matrices that are specific to text data, but it's a new feature and it can cause problems with PsychToolbox… so we're going to ignore this feature entirely*

4.3 – Cell matrices

- Creating a cell matrix
  - Cell matrices are defined using squiggly brackets
  - Again, you can differentiate elements within the matrix using a space or a comma
- Try it yourself:
  - `{'A' 'pleasure' 'to' 'burn'}`
  - `{'A' 'pleasure'; 'to' 'burn'}`
  - `{'A'; 'pleasure'; 'to'; 'burn'}`
- Again, to reiterate, everything you put into a cell matrix is a cell data type!
  - `thisIsSoDumb = {1 2 3 4};`
  - `thisIsSoDumb(4) - thisIsSoDumb(1)  % Instead of performing the computation 4-1, this command throws an error. Why?`
  - `cell2mat(thisIsSoDumb(4)) - cell2mat(thisIsSoDumb(1))  % cell2mat tells Matlab to convert the cell type to the underlying matrix type`

4.4 – Indexing matrices

- Once you have a matrix, how do you access or edit any given element inside of it?
- To begin, let's define a new variable that contains a cell matrix:
  - `warningMessage = {'The emergency' 'phone number'; 'is' 911};`
- If we only wanted to get the phone number, we could selectively access it by doing this:
  - `warningMessage(2, 2)  % This says to access row 2, column 2, which is the cell that corresponds to 911`
- How would you selectively access the words "phone number?"

4.4 – Indexing matrices

- You can even define new variables by indexing elements out of old matrices!

  - `reference = [warningMessage(1,2) warningMessage(2,2)]`

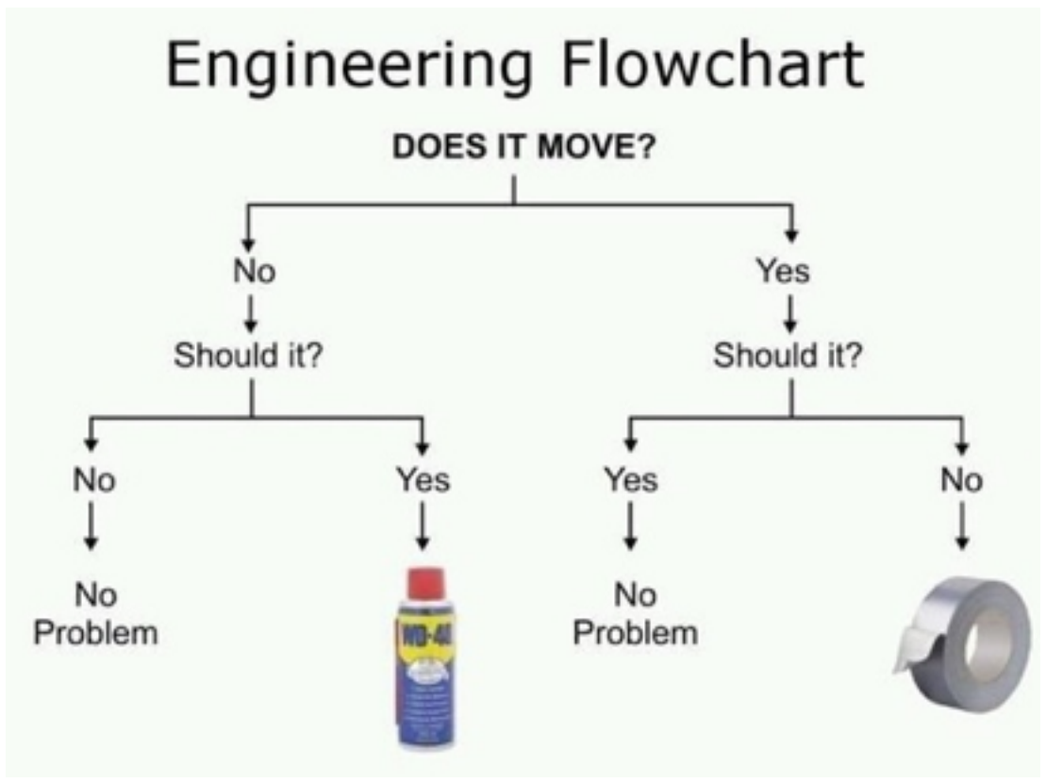
- Let's suppose that you move out of the United States to the United Kingdom, where the emergency phone number is 999. How would you edit the phone number, and nothing else?

  - `warning(2,2) = {999};`

A fairly typical response to dialing 999 in the UK

5.1 – Control Logic

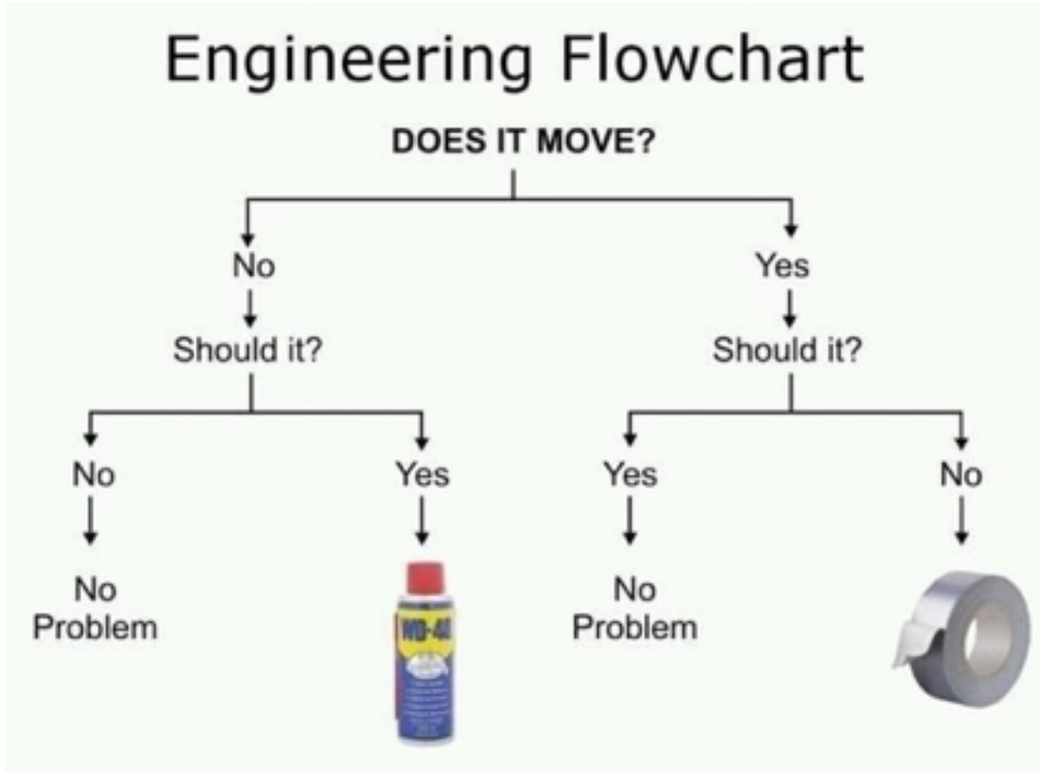
- There are often times when you'll want your program to take one course of action if certain conditions are met, and to take different courses of action if those conditions are not met
- For this reason, logical statements (also known as conditional statements or control logic) can best be visualized as flowcharts



5.1 – Control Logic

- How might we represent this flowchart using fake code?

```
if doesMove(object) == true
    if shouldMove(object) == true
        disp('No problem!')
    else
        disp('Use duct tape!')
    end
else
    if shouldMove(object) == true
        disp('Use WD-40!')
    else
        disp('No problem!')
    end
end
```



5.2 – Logical comparisons

- At the heart of control logic is the ability to make logical comparisons (formalized by the Boolean values true/false)
- Consider this simple program:

```
number = input('Please enter a number: ');
if number > 0
    disp('You have entered a positive number!')
elseif number == 0
    disp('0 is neither positive nor negative!')
else
    disp('You have entered a negative number!')
end
```

- You probably recognize the “greater than” sign from math class. This syntax specifies what comparison is made, and therefore dictates which branch of the flowchart is entered into

5.2 – Logical comparisons

Matlab Syntax	Logical Significance
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
~=	Not equal to
~	Not
&	And
|	Either/Or

5.2 – Logical comparisons

- Once again, strings are treated a little differently than numbers:

```
answer = input('First president of the U.S.? ', 's');
answer = upper(answer);
if strcmp(answer, 'GEORGE WASHINGTON')
    disp('Correct!')
else
    disp('Incorrect!')
end
```

- A few things to unpack:
  - The `input` function now has an optional argument `s` that indicates that the program should expect a string, not a number
  - Computers are very literal, so the program will treat `George Washington` and `GEORGE WASHINGTON` as being two different answers – this is the reason why we redefine `answer` as being uppercase using the function `upper`
  - `strcmp` stands for string compare, and it returns either Boolean TRUE or FALSE

5.3 – Conditionals (if, ifelse, else)

- A logical comparison only returns Boolean TRUE/FALSE
- A conditional statement tells the program how to proceed *conditional upon* a logical comparison being TRUE/FALSE (hence the name)

5.3 – Conditionals (if, ifelse, else)

```
number = input('Please enter a number: ');
if number > 0
    disp('You have entered a positive number!')
elseif number == 0
    disp('0 is neither positive nor negative!')
else
    disp('You have entered a negative number!')
end
```

- Explicitly, what is your computer doing?
- Check if `number` is greater than `0`
- If so, stop checking any subsequent branches and run `disp('You have entered a positive number!')`
- If not, check if `number` is equivalent to `0`
- If so, stop checking any subsequent branches and run `disp('0 is neither positive nor negative!')`
- If all preceding conditional statements have returned Boolean FALSE, run `disp('You have entered a negative number!')`

5.3 – Conditionals (if, ifelse, else)

There are many ways to combine conditionals that will get you the same output

It's up to you as a programmer to figure out which methods are most sensible / easiest to understand

```
% Using EQUAL TO
if number == 0
    disp('This number is 0.')
else
    disp('This number is not 0.')
end

% Using NOT EQUAL TO
if number ~= 0
    disp('This number is not 0.')
else
    disp('This number is 0.')
end
```

5.3 – Conditionals (if, ifelse, else)

There are many ways to combine conditionals that will get you the same output

It's up to you as a programmer to figure out which methods are most sensible / easiest to understand

```
% Using OR
if number > 0 | number < 0
    disp('This number is not 0.')
else
    disp('This number is 0.')
end

% Using AND
if ~number > 0 & ~number < 0
    disp('This number is not 0.')
else
    disp('This number is 0.')
end
```

5.4 – Loops

- Imagine that you're an industrialist in the year 1913
  - Henry Ford has just introduced the idea of an assembly line, and it's reduced the time it takes to build a car from 12 hours to a mere 2.5 hours
  - Being an industrialist, you can't wait to try it out for yourself
  - As you're thinking about how to optimize the efficiency of your factory, you come to the realization that what makes the assembly line so efficient is the fact that each worker is doing one small task repeatedly
- This is essentially how loops work in computer programming

Grad school might not be the most fun thing ever, but at least it beats having to turn the same screw 100,000 times a day

5.5 – for loops

- A ***for loop*** repeats the same chunk of code over and over again *for a certain number of repetitions* (hence the name)
- Let's say that the industrialist needs four tires affixed to the car before it's ready to move on to the next stage of the assembly line
- We can break this objective down into all of its individual components:
- The central insight: there's only one action that actually needs to happen, which is affixing a single tire to the car
- Once you've figured out how to affix a single tire to the car (pick up the tire, place it on the axel, use a bolt to attach it to the axel), all you have to do is repeat that single action 3 more times

5.5 – for loops

- Using fake code, let's see what this might look like…

```
for n = 1:4
   pickUp(tire(n));
   moveTo(axel(n));
   placeOn(axel(n));
   pickUp(bolt(n));
   moveTo(tire(n));
   placeOn(tire(n));
end
```

- The first component of a ***for*** loop is the declaration
- Next, you make a new variable (in this case, ***n***) and set it equal to some sort of numerical range (in this case, 1 through 4)
- The program runs through all the code that is contained between the declaration and the end for the stated number of repetitions
- Then (and only then), the program moves on to the next segment of code
- The first time the loop is run, ***n*** = 1. The next time the loop is run, ***n*** = 2, and so on until ***n*** = 4, after which the loop ceases to run
- We take advantage of ***n*** by specifying *which* tire/axel/bolt to fasten on each iteration

5.5 – for loops

This is implemented in C++, but you can probably parse it using your new knowledge of Matlab!

5.6 – while loops

- How is a ***while*** loop different from a ***for*** loop?
  - There's only one key difference
  - Recall that a ***for*** loop runs the same chunk of code for a certain number of repetitions
  - In contrast, a ***while*** loop runs the same chunk of code while some logical condition is satisfied
- More concretely:
  - Let's imagine again that you're an industrialist putting together an assembly line
  - As much as you'd like to automate production so that you can manufacture cars 24/7, things like labor laws, union rules, and equipment safety protocols prohibit you from simply manufacturing cars all the time
  - So when should your assembly line be in operation? There are a couple of conditions that we might use to determine whether the assembly line should be operating: Is the current time somewhere between 9am and 5pm? Is the factory floor being staffed by the minimum number of workers to ensure safety?

5.6 – while loops

- What might that look like using fake code?

```
while getTime >= 9am & getTime <= 5pm & workerNum >= 50

   runFactory;

end
```

- Critically, runFactory will not run unless all of the conditional statements return Boolean TRUE
- Make sure that you program an exit condition, or else your code will loop infinitely!
  - If you're using simple conditional logic, you can accomplish this by specifying all of your conditions in the declaration (above)
  - If your conditional logic is less straightforward, you can also accomplish this by manually specifying a break condition (next slide)

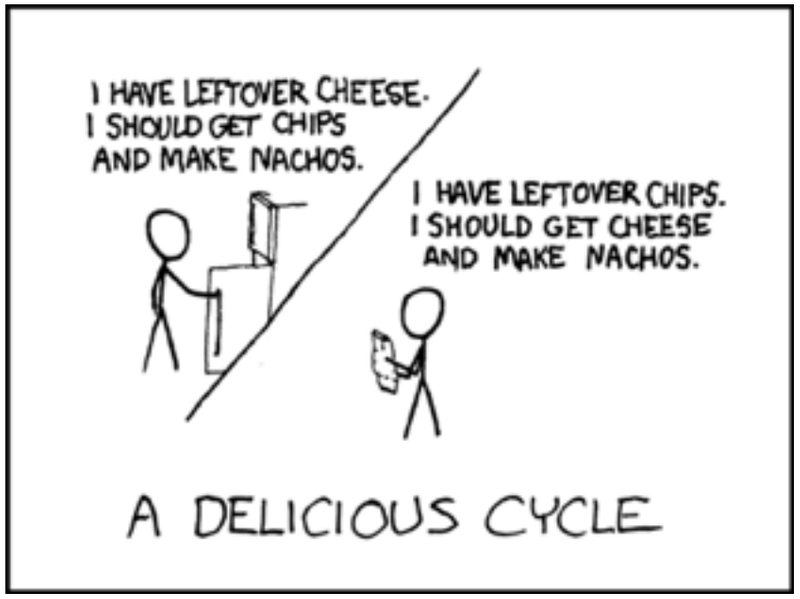
5.6 – while loops

- Here's how you might manually specify a break condition:

```
while 1  % 1 = TRUE, so this is already an infinite loop
   if getTime >= 9am & getTime <= 5pm & workerNum >= 50
      runFactory;
   else
      break
   end
end
```

5.7 – Loop summary

Use ***for*** when…	Use ***while*** when…
You know exactly how many times the loop needs to iterate from the onset	You don't know how many times the loop needs to iterate from the onset
There is only one condition that needs to be met for the loop to break (typically the number of repetitions)	There are multiple conditions that could break the loop (which could include the number of repetitions)



An example of the foolish use of a ***while*** loop. Using a ***for*** loop would have prevented this problem altogether!

5.7 – Loop summary

- This is a profound insight: everything is control logic
  - Logical comparisons return Boolean TRUE/FALSE
  - Conditional statements use logical comparisons to execute different branches of your program
- A ***while*** loop allows you to repeatedly use conditional statements to execute different branches of your program
- A ***for*** loop allows you to implement a ***while*** loop without having to manually maintain a counter variable