Notice:

This review homework is completely optional. The problems are intended to help you review some of the topics that will be on the final. Each problem is worth 20 points and the ABCs are worth 10 points each, so you can earn up to 140 points of extra credit on this assignment! It is due on Tuesday, April 24th at 8:00pm, with a grace period until 11:59. Since this is the last day of classes, there will not be a resubmission.

Thanks for a great semester!

Coding!
~Homework Team

Function Name: `functiony`

Inputs:

1. (*char*) A string containing a function header

Outputs:

1. *(logical*) A logical indicating whether or not the function header is valid

Background:

You're studying for finals and have finished all the practice tests, but GT Wi-Fi goes down before you can check your answers! To check to see whether all your function headers are valid, you decide to write a function in MATLAB. Afterall, MATLAB will always be there for you when GT inevitably fails you again.

Function Description:

Write a function that takes in a string containing a function header and outputs a true or a false indicating whether the header is valid.

Example:

```
>> [out] = functiony('function [out] = myFunction(in)')
out = true
```

Hints:

- Using several conditionals will be easier than one long conditional. Think of a way to keep track of if one rule fails through a variable or a vector of variables.
- Create a helper function to check for valid variable names.
- The functions `strtrim()` and `iskeyword()` will be very useful.

Function Name: `edgy`

Inputs:

1. (*char*) Filename of an image with file extension

File Outputs:

1. An edgy image

Background:

In an attempt to become less conventional and more *edg*y after the downfall of GT Wi-Fi, you decide to brush up on your numerical methods and learn about edge detection in images.

Function Description:

Write a function to perform basic edge detection on an image, according to the following steps:

1. Convert the image to grayscale.
2. Using the numerical pixel values in the gray image, calculate the numerical derivative of the pixel values across both the horizontal and vertical directions and take the absolute value of each array.
    a. Since the resulting x-direction and y-direction derivative arrays will have different dimensions, remove the last row/column to make both difference arrays the same size.
    b. Additionally, crop the original image to this new size.
3. Calculate the averages and standard deviations of the x-direction and y-direction difference arrays and calculate a minimum threshold difference that will be considered an edge according to the following formula:

$$\Delta x_{\text{threshold}} = \text{mean}\left(\frac{\Delta(\text{brightness})}{\Delta x}\right) + \text{std}\left(\frac{\Delta(\text{brightness})}{\Delta x}\right)$$

$$\Delta y_{\text{threshold}} = \text{mean}\left(\frac{\Delta(\text{brightness})}{\Delta y}\right) + \text{std}\left(\frac{\Delta(\text{brightness})}{\Delta y}\right)$$

   (std is short for standard deviation).
4. Determine which points in the **red layer** of the x-direction and y-direction derivative arrays are greater than or equal to the corresponding thresholds. Points that are greater than or equal to the threshold in either direction should be considered an edge.
5. Change these edge pixels in the original color image to pure red.
6. Write the new edgy image to `edgy_<original filename>.png`.

Continued…

Example:

`omar.png:`
`<200 by 200>`

```
>> edgy('omar.png')
```

`edgy_omar.png:`
`<199 by 199>`

Notes:

- When you find the differences on the grayscale image, **don't convert to double**. Additionally, use the entire gray image, not just a single layer.
- When you take the mean, **do not** convert to double either.

Hints:

- Look at the third input of the `diff()` function.
- The `std()` function takes the standard deviation of a vector (but it only works on type double!).

Function Name: `shapey`

Inputs:

1. (*double*) A 1xN vector containing the lengths of each consecutive line segment of the crop polygon
2. (*double)* A 1xN vector containing the counterclockwise angles in degrees between each consecutive line segment and its previous line segment

Plot Output:

1. The plotted crop polygon

Background:

After the violent death of GT Wi-Fi, you move to the country and become a humble farmer. You have taken an interest in drawing crop circles and you are wondering if MATLAB can do it for you. You want to be able to draw many different crop patterns, not just circles, so you decide to make the images out of straight lines. (You know that you can make something that looks like a curve by taking many small line segments and attaching them end to end at small angles from one another.) You will be able to use this function to draw all different kinds of shapes and patterns!

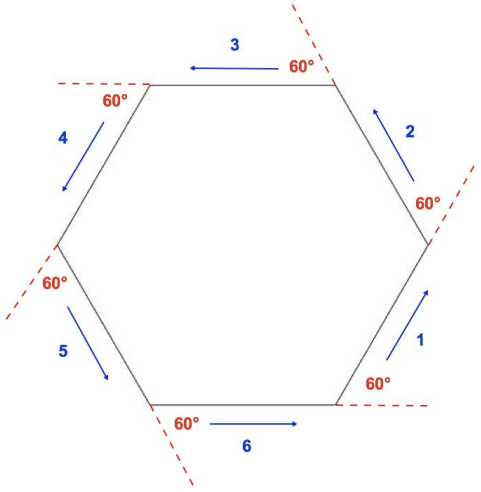
Function Description:

This function will take a vector of line segment lengths and angles and plot a polygon. The polygon will start at (0,0) and each line segment will be drawn from where the last one ends. Your first input will be a vector specifying the lengths of each consecutive line segment in the crop polygon. Your second input will be a vector specifying the *counterclockwise* angles between consecutive line segments.

Example:

The following input to `shapey` will result in the hexagon below:

```
>> shapey([5, 5, 5, 5, 5, 5], [60, 60, 60, 60, 60, 60]);
```



Continued...

The function will create the black hexagon. The blue and red lines and numbers are for reference. Each line segment has a length of 5. The blue arrows show the order in which the lines are plotted. The angles specified by the second input are also shown as the counterclockwise angles between consecutive line segments.

To rotate a set of points counterclockwise around the origin, multiply the coordinates by the rotation matrix as follows:

$$\begin{bmatrix} x_{rotated} \\ y_{rotated} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x_{original} \\ y_{original} \end{bmatrix}$$

This is matrix multiplication, not element-wise multiplication!

You do not have to use rotation matrices to solve this problem, but it is given here if you would like to use it.

Notes:

- Line segments should be drawn in black.
- The first angle is measured from the x-axis.
- Use `axis equal` and `axis off` for your plot output.
- Use `sind()` and `cosd()` for sin and cos with degree inputs.
- You could theoretically draw anything you want with this function; See if you can come up with some cool test cases!
- Notice that an n sided shape with equal side lengths and equal angles would have a second input of n angles that are each given by the equation: angle = 360/n. The larger you make n the closers your shape looks like a circle.

Hints:

- Since the rotation matrix rotates points counterclockwise relative to the positive x axis, and the angle of each line segment is given counterclockwise relative to its previous line segment, try keeping track of each line segment's angle counterclockwise relative to the positive x axis. Then, the rotation matrix can be used to rotate and create any line segment.

Function Name: `chessy`

Inputs:

1. (*char*) An 8x8 char array representing a set up chess board
2. (*char*) A filename corresponding to a text file containing the next moves of the game

Outputs:

1. (*char*) The updated game board
2. (*char*) A description of which pieces were captured by each player

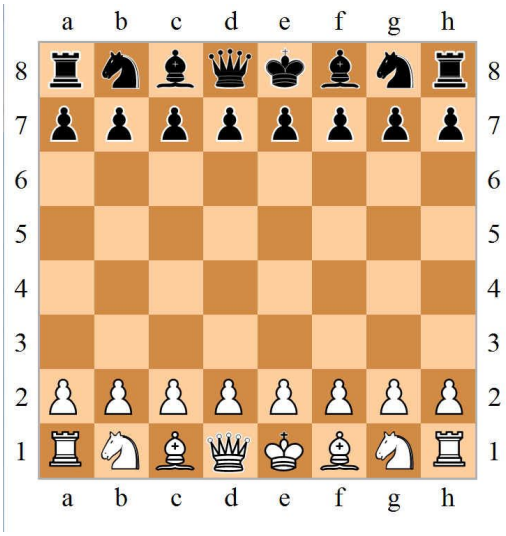
Background:

After GT Wi-Fi went down during your time ticket, you want to run away forever and never return and become a wandering chess grandmaster. You sold your computer to survive, and you can't afford a chessboard, so you decide to play chess with MATLAB iOS instead.

Function Description:

The input array represents a chess board at the start of a game. Each piece is represented by a letter: pawn (P/p), rook (R/r), knight (N/n), bishop (B/b), queen (Q/q), and king (K/k). Empty squares are filled with a space character. Black pieces are lowercase and white pieces are uppercase.

The text file contains instructions, where each line is of the format: `'<a-h><1-8> to <a-h><1-8>'`. For example, the line `'b1 to c3'` would represent moving the white knight from b1 to c3. The board locations can be seen here, alongside the char representation of the starting board:




```
board =

  8×8 char array

    'rnbqkbnr'
    'pppppppp'
    '        '
    '        '
    '        '
    '        '
    'PPPPPPPP'
    'RNBQKBNR'
```

Continued...

Your function should output the current state of the board after all moves in the instruction file have been completed. Your code should also account for any pieces captured during play. For example, a white piece is captured when a black piece is moved to the same space the white piece was on. Return a statement about which pieces were captured by each player, in the following format:

```
'Pieces taken by white: <pieces>. Pieces taken by black: <pieces>.'
```

where the variable portions should be the character representations of the pieces with nothing separating them, in the order they were captured. If a player did not capture any pieces, this variable should have the value `'none :('`.

Notes:

- Each of the lines given in the text file is guaranteed to be a valid move for that piece.

Function Name: `wordy`

Inputs:

1. *(double)* An integer between -999 and 999, inclusive

Outputs:

1. *(char)* A string representation of the integer

Background:

Numbers are great, but sometimes it's necessary to write them out in text. But with GT Wi-Fi down, you can't just WolframAlpha™ it, so you might actually have to think about something. However, this can be a tedious process, which is why you are going to automate it in MATLAB.

Function Description:

Given an integer, convert it to the fully written-out word form. For example, the integers:

`0`, `1`, `2`, `11`, `16`, `20`, `45`, `-100`, and `549`

correspond to

`'zero'`, `'one'`, `'two'`, `'eleven'`, `'sixteen'`, `'twenty'`, `'forty-five'`, `'negative one hundred'`, and `'five hundred and forty-nine'`,

respectively. The full set of rules for how to format numbers is listed below.

Rule Number	Value Range	Rule
(1)	`-999 <= x < 0`	Prepend `'negative '` to the string generated from `abs(x)`
(2)	`x == 0`	`'zero'`
(3)	`1 <= x <= 19`	Spell out the number (`'one'`, `'two'`, ... , `'nineteen'`)
(4)	`20 <= x <= 99`	`'<prefix>-<`**(3)** applied to ones digit`>'` `<prefix>` is `'twenty'`, `'thirty'`, ... , `'ninety'`
(5)	`100 <= x <= 999`	`'<`**(3)** applied to hundreds digit`> hundred [and] <`**(4) OR (3)** applied to remaining digits`>'`

Notes:

- It is pretty easy to make a test script that can test your function against the solution for all possible inputs.
- The word `'and'` should only be present after the `hundred` part of the final phrase if there are more nonzero numbers following that place.

Hints:

- Cell arrays will be very useful for this problem.

Function Name: `waldoy`

Inputs:

1. (*struct*) 1x1 Structure representing a person

Outputs:

1. (*logical*) Whether Waldo can be found
2. (*double*) The degrees of separation from the starting person to Waldo

Function Background:

After the fall of GT Wi-Fi, you can't spend your nights watching Vine compilations anymore, so you dust off your old *Where's Waldo?* books and get cracking. One night, after a Red Bull-fueled hallucinatory fever dream, you finally realize what should have been obvious after all: Waldo is the one responsible for taking down GT Wi-Fi! There isn't time to search for him on every page of every *Where's Waldo* book, you know, so you decide to search for him using MATLAB and finally bring his 31-year reign of terror to an end.

Function Description:

Write a function called `wheresWaldo` that takes in a 1x1 structure representing a single person. The input structure is guaranteed to have at least the following fields:

```
Name: <String representing the name>
Age: <double representing the age>
Friends: <1xN structure array representing all of this person's friends>
```

Note that for each friend, they are guaranteed to have the above 3 fields, but may have any number of extra fields in addition to the 3 above.

Your job is to recursively search through all of the people available by checking all of the first person's friends, then each of their friends, and so on.

The first output of the function will be a single logical indicating if Waldo is a friend of anyone in the group (e.g. if Waldo is present). Unfortunately, Waldo is sneaky and doesn't always put `'Waldo'` in the Name field, so the first output of the function should be true if ANY field of ANY of the people in the group contains the string `'Waldo'`. Note that the spelling and case must appear exactly as shown.

The second output of the function will be a double representing the number of friends between the first person and Waldo. If Waldo does not exist in the group, this output should be 0. For example, if a friend of a friend of the first person is Waldo, then the second output will be 2.

Continued...

Notes:

- There will never be more than 1 Waldo in the group.
- Do not hardcode the field names for the friends: there is no guarantee how many there will be and no guarantee what they will be called.
- You **must use recursion** to solve the problem**.**

Hints:

- Think about the best way to keep track of the results of each friend as you traverse the group.
- A visual example is included below.
- In order to help you test your function, we have provided the function `generatePeople` for you as a .p file. There are no inputs, and only one output – a sample input to `wheresWaldo`. You can run the output structure in the provided solution file and your own code to test the accuracy of your code. These are in supplement to the test cases already provided. Please **do not** call this function in your code!

Example:

The following is a possible input to the function:

`person =`

```
   Name: 'Alexander LoboYoBoy'
    Age: 32
    Job: 'Neither the leader nor the scribe'
Friends: [1x9 struct]
```

This person has 9 total friends. The first of these friends looks like this:

```
>> newFriend1 = person.Friends(1)
newFriend1 =
```

```
    Name: 'Hyder Hasnainy'
     Age: 21
 Friends: []
Students: 20
```

The above person has no friends in the list. Since they are not Waldo, we are done searching for this friend. The 8th friend of our original person looks like this:

```
>> newFriend8 = person.Friends(8)
newFriend8 =
```

```
Name: 'Mr. Zachary Schlesingery'
Age: 'Unknown'
Friends: [1x8 struct]
Students: -2
Hobbies: 'CS-ing'
```

Continued...

The above person has 8 friends, each of which need to be searched if they are (or have any friends who are) Waldo. Now let's look at The Mr. Zachary Schlesingery's 4th friend:

```
>> nestedFriend = person.Friends(8).Friends(4)
nestedFriend =
```

```
Name: 'Omary'
Age: 22
Friends: [1x10 struct]
MATLAB_Lines_Written: 'Waldo'
```

The `'Matlab_Lines_Written'` field has a value of `'Waldo'` which means this person is Waldo in disguise. In this case, the first output of our function would be `true`. The second output of our function would be `2` since we had to go through two layers of friends to reach Waldo.