Function Name: `piCrust`

Inputs:

1. (*cell*) Your pie (a cell array), with kind words from Mary Berry hidden in the layers of the pastry (nested cell arrays)

Outputs:

1. (*char*) A sentence from Mary Berry, extracted from the layers of your pie pastry
2. (*double*) Maximum number of layers in your pie pastry

Topics: (*unnesting*), (*string manipulation*)

Background:

After the signature challenge, Paul Hollywood kneaded your spirits harder than an enriched dough and now you're feeling pretty down. Lucky for you, hidden inside the lamination of your pie pastry from the technical are warm, comforting words of wisdom from Mary Berry herself. If you can find these words and make them into a sentence, you stand a chance at turning your weekend around and pulling out a great bake during the showstopper challenge. Elizabeth may be the Queen of England, but Mary Berry is the Queen of Your Heart!

Function Description:

You are given a pie (cell array) with Mary Berry's words of encouragement tucked into the layers of the pastry (nested cell arrays). For the first output, unnest the words and form them into a sentence. You should put spaces in between the words when you form the output sentence. The second output should be the largest number of layers (deepest nested cell array) in your pastry. On your marks, get set, BAKE!

Example:

```
pie = {'you', {{{{'can'}}}}, {{'definitely see'}}, {{{'the'}}},
                                        {{{{{{'layers here'}}}}}}}
[maryBerrySays, maxLayers] = piCrust(pie)
maryBerrySays -> 'you can definitely see the layers here'
maxLayers -> 7
```

Notes:

- The input is guaranteed to only have 1 row.
- The contents of the unnested cells will always be type *char*.
- There should not be an extra space at the beginning or end of the output sentence.

Hints:

- `iscell()`, `ischar()`, or `class()` may be useful in this problem.

Function Name: `piDetective`

Inputs:

1. (*cell*) 1xM cell array containing various data types
2. (*cell*) 1xM cell array containing various data types

Outputs:

1. (*cell*) 1xN cell array containing only words that start with 'i' or 'p'
2. (*double*) The number of times the word 'pi' appears in the first output

Topics: (*iteration*), (*cell arrays*), (*cell concatenation*)

Background:

You are THE Pi Detective, nobody is anywhere as good as you. At night you lurk the streets with one goal in mind: find all the pi. You snag everything that even somewhat resembles pi so you can go analyze your findings later on back at the lab. You have one advantage that keeps you above the rest though, it's your secret weapon: MATLAB.

Function Description:

Given two 1XM cell arrays, find the words that start with `'p'` (case insensitive) in the first cell array and the words that start with `'i'`(case insensitive) in the second cell array. Your first output should be all of the words from the first cell array that start with `'p'` and all of the words from the second cell array that start with `'i'`. The words must appear in the order in which they appear in the cell array with the words from the first cell array taking precedence over the words from the second cell array, as shown in the example below. Your second output should be the amount of times `{'pi'}` (case insensitive) appears in the first output.

Example:

```
ca1 ={'PiCrust', 'CS1371', 532, true, 'Pi', 'piThonSux', 931}
ca2 = {'Impostor', 213, 'imresize()', 4, 'pi', 'bag', 's'}

[newCa,numPi] = piDetective(ca1, ca2);

newCa → {'PiCrust', 'Impostor', 'imresize()', 'Pi',
'piThonSux'}
numPi → 1
```

Notes:

- Make sure to be case insensitive
- The cell arrays can contain multiple types of data
- The 2 inputted cell arrays will always be the same length
- Only count the occurrences of 'pi' if they are in their own cell in the outputted cell array

Function Name: `piEcesOfPie`

Inputs:

3. (*cell*) MX3 The cell array containing all the data you have on the orders

Outputs:

3. (*cell*) MX3 The cell array with the missing values calculated.

Topics: (*iteration*), (*cell arrays*), (*conditionals*)

Background:

Here at the Archimedes bakery you are preparing for the influx of orders that are sure to come with pi day. In order to handle the increased volume, the bakery is automating some of the orders that are coming in, and have stored that relevant data for each pie in a cell array. Unfortunately, the customers did not properly read the instructions on the online ordering site, and most of the orders are missing some key measurements. Fortunately, you realize a way to calculate each of the missing values from the data you have using MATLAB!

Function Description:

You will be given a cell array containing numerical values corresponding to the volume `(V)` (column 1), radius `(R)` (column 2), and height `(h)` (column 3) of a cylinder shaped pie. Every row is a different pie. Some values are missing and have been replaced with the corresponding variables. Missing variables in column 1 have been replaced with `'V'`, missing variables in column 2 have been replaced with `'R'`, and missing variables in column 3 have been replaced with `'h'`. With this data given, iterate through the cell array and replace the values of type char with the corresponding numerical values. Calculate the missing values using the following equation and round the value to 2 decimal places:

$$V = \pi R^2 h$$

Example:

```
ca ={'V',3,2;
     10,3,'h'};

[ca_fix]= piEcesOfPie(ca);

ca_fix -> {56.55,3, 2;
            10 ,3,0.35}
```

Notes:

- Each row will have at most one missing variable
- Make sure to round the values that you enter into the cell array to 2 decimal points

Function Name: `pillagePublix`

Inputs:

1. (*cell*) An MX3 cell array containing the Publix inventory
2. *(char)* A string containing your grocery list

Outputs:

1. (*cell*) An MX3 cell array containing the updated Publix inventory
2. *(char)* A string describing the total cost of your groceries

Topics: (*cell array manipulation*), (*iteration*)

Background:

Shiver me timbers! You grew up hearing horror stories about pirate diets, but when you pledged your life to piracy, you didn't expect it to actually be this bad. It's almost pi day, and you're craving something that isn't rotten, stale, or covered in salt. You've finally convinced your fellow crewmates to stop by the grocery store for all the necessary ingredients - keeping things professional, of course. Time to pillage Publix.

Function Description:

You are given a cell array containing the Publix inventory, where the first column holds the item name (type char), the second column contains the quantity of that item that Publix has in stock (type double), and the third column is the unit price of that item in dollars (type double). You are also given a string containing what you need to buy, formatted `'<item>-<quantity of item>'`, with a comma separating each item. There will not be any spaces in the string, and the items may be written in a different case than they appear in the inventory. Also, the string can contain any number of items. Your first output will be the updated Publix inventory. The updated inventory should reflect the new quantity that Publix has in stock after you purchase the items. Your second output will be a string describing the total cost of your Publix run, formatted `'My total is $<total cost>.'` where the total cost always has two decimal places.

Example:

```
inventory = {'Apple', 18, 0.52;
             'Salt',  50, 1.37;
             'Flour', 29, 3.20}
list = 'flour-3,APPLE-15'

[updatedInventory, sent] = pillagePublix(inventory, list)

updatedInventory -> {Apple,  3, 0.52;
```

```
                        Salt,  50, 1.37;
                        Flour, 26, 3.20}
    sent -> 'My total is $17.40.'
```

Notes:

- You can have more than one item on your list
- Every item on your list will exist in the inventory.
- There will always be enough of each item to buy what you need.
- Case does not matter.

Hints:

- Think about how you can format a number with two decimal points in a string.
- Recall which string functions work on cell arrays!

Function Name: `piThonSux`

Inputs:

1. (*cell*) An N x 3 cell array representing a scoresheet containing names, scores, and types of pies

Outputs:

1. (*cell*) An N x 5 cell array of updated and sorted values

Topics: (*cell array manipulation*), (*masking*), (sorting), (if/switch statements), (iteration), (casting), (unnesting)

Background:

It's Georgia Tech's annual Pi Day baking contest, and pie connoisseurs from all across campus have entered this year's competition to bake the best pie around! After many failed attempts at baking your Great Aunt Linda's rhubarb pie, you decide to ditch the baking skills and turn to your coding skills to help sort through all of the entries instead. Plus, you heard you might be able to get free pie this way! Your friend in CS 1301 was going through the entries before you, but made a bunch of silly mistakes and couldn't fix them because, well… python sux! You decide to take over and prove that CS 1371 is the superior Intro Computer Science class and use MATLAB to help you readjust and sort through the entries.

Function Description:

You are given a cell array representing a scoresheet with three columns (in no particular order) of data representing a contestant's name (`'Name'`), the type of pie they baked (`'Type'`), and the score they earned (`'Score'`). Based on the type of pie they baked, determine the category of the pie and calculate the adjusted score based on the algorithm below. The scores may be given as type double, type char, or type cell. In the case that you are given a score in a nested cell array (type cell), you must unnest all of the outer layers to retrieve the double or char value inside. If the score is type char, it will always be a character vector of numbers (like '561'). Convert this string to type double in order to compare with the other scores. Keep in mind, the scores must be in the same format as they were in when they were given to you. Next, append two new columns to the right of the original cell array containing the category of pie (`'Category'`) and adjusted score (`'Adjusted Score'`), respectively, rounding the new scores to the **second decimal place**. Finally, sort the cell array in descending order based on the adjusted scores and return the updated cell array.

Category of Pie	**Types of Pies included**	**Score Adjustment**
`'Fruit'`	`apple, cherry, blueberry`	Increase the score by 0.5 points

'Savory'	chicken pot pie	Increase the score by 20%
'Classic'	key lime, pecan	Decrease the score by 0.3 points (too boring)
'Dessert'	french silk, nutella	Increase the score by 20%
'Other'	All other pies	Reduce the score by 15%

Example:

```
scoreSheet =
     {'Name',              'Score',         'Type';
     'Geoff_Collins',      {{{{4.04}}}},    'PeCaN';
     'Kantwon_Rogers',     7.6,             'french silk';
     'George_Burdell',     {{'8.999'}},     'apple';
     'Mascot_Buzz',        '1.0',           'chicken pot pie'
     'Angel_Cabrera',      '5.468',         'strawberry';}


updatedScores= piThonSux(scoreSheet)

updatedScores →

{'Name'          }    {'Score' }    {'Type'           }    {'Category'}    {'Adjusted Score'}
{'George_Burdell'}    {1×1 cell}    {'apple'          }    {'Fruit'   }    {[        9.5000]}
{'Kantwon_Rogers'}    {[7.6000]}    {'french silk'    }    {'Dessert' }    {[        9.1200]}
{'Angel_Cabrera' }    {'5.468' }    {'strawberry'     }    {'Other'   }    {[        4.6500]}
{'Geoff_Collins' }    {1×1 cell}    {'PeCaN'          }    {'Classic' }    {[        3.7400]}
{'Mascot_Buzz'   }    {'1.0'   }    {'chicken pot pie'}    {'Savory'  }    {[        1.2000]}


*1x1 cell is MATLAB's way of displaying a nested cell; refer to the
original scoreSheet for these nested values as they should be the
same value for both the original and updated scoresheet .
```

Notes:

- The column headers will always be included in the first row and will be given exactly as 'Name', 'Type', and 'Score', but can be in any order
- Entries in the 'Scores' column will always be either of type *double, char*, or *cell*; 'Names' will always be of type *char*; 'Type' will always be of type *char*
- When determining categories, the type of pie is <u>case insensitive.</u> but the category name itself is case sensitive and should be written exactly as stated in the table.
- The order of the original columns should stay the same in the output with the 'Category' and 'Adjusted Score' columns appended to the right of them respectively.

- There will never be a tie between updated scores for sorting purposes

Hints:

- Don't forget about masking!
- Think about how you can check the data type of the scores and properly cast them to the correct type in order to perform mathematical operations on them. Which built-in functions might you use?
- The `cell2mat` function may be useful for sorting

Extra Credit

Function Name: `piEdmontPark`

Inputs:

1. (*cell*) MxN map of the area around piedmont park

Outputs:

1. (*cell*) MxN modified map
2. (*double*) the number of pi's you ate along the way

Topics: (*cell arrays*), (*iteration: pathing*), (*conditionals*), (*data types*)

Background:

You've talked to a therapist, and after many difficult days you've finally overcome your cannoli addiction from last week (Let's be real, that's what it was...). You head to the CULC as the new you, and when you arrive you see a flyer advertising the Annual Pi Festival over at Piedmont Park! You feel your heart rate rise, and your mouth begins to water. Your kneeps are weak, your arms are heavy, you're drooling over the prospect of pi already! Looks like another therapy session might be in store, but first you've got a pi-licious craving to feed!

Function Description:

Your goal is to make it from GT to the Pi Festival at Piedmont Park. Because so many people will be going, they dropped some pi along the way for you to eat! Begin by locating your starting position, indicated by a `{'GT'}`, case sensitive. Then, begin iterating through the cell array. Cells you can move to will contain values of type char (e.g. `{'a'}`) or type cell (e.g. `{{1, 3; 2, 4}}`) and the cells you cannot move onto will contain type double (e.g. `{0}`).

As you iterate through the cell array:

- Navigate through the cells, following the type char values in a single step, horizontal or vertical direction until you reach either of the following:
    - a Teleport Pad (TP)
    - a cell containing the string `'pie'`
- The behavior of TPs is described below in detail.
- Count the number of times you path over a cell containing the string `'pi'`. This is your 2nd output.
- Replace all cells you've pathed over with a `{0}`.
- Once you reach the cell containing `'pie'`, stop iterating. You made it to the Pie Festival! Output the resulting map as your 1st output.

As the name suggests, teleport pads will transport you to a different position on your map cell array. The TPs rely on a previous function you have written: `pythag()`. Each value at the TP will be a 2x2 cell array containing values of type double. These values correspond to two sets of inputs for `pythag()`. The **first column** corresponds to the **first input** of `pythag()` (a), and the **second column** corresponds to the **second input** of `pythag()` (c). The **first row** of

the array, when input into `pythag()`, will output your **change in rows**, and the **second row**, when input into `pythag()`, will output your **change in columns**. Add these changes to the row and column indices of the TP you are currently on (your current position) to obtain the indices of the next path to begin iterating through.

For example, let's say your TP cell was at position `(1,2)` and it contained the array `{1.5, 2.5; 4, 5}`, your change in rows would be `pythag(1.5, 2.5)` → `2` and your change in columns would be `pythag(4, 5)` → `3`. You would then "teleport" to position `(3,5)` on your map and continue pathing. An example of this is shown below.

```
TP = {1.5, 2.5; 4, 5};
map = {'GT'    TP      0       0       0       0;
       0       0       0       0       0       0;
       0       0       0       0       'here'  0;
       0       0       0       0       0       0};
```

*** This does not reflect an actual test case, and is ONLY for showing the teleport functionality*

Ensure to include the `pythag()` function in your submission.

Example:

```
TP = {3,5;4,5}; % TP is of class cell, and defined here for clarity
map = {0      0      0      0      0      0      0      0      0      0      0;
       0      0      0      0      0      0      0      0      0      0      0;
       0      'GT'   'a'    'b'    0      0      0      'f'    'g'    'h'    0;
       0      0      0      TP     0      0      0      'pi'   0      'i'    0;
       0      0      0      0      0      0      0      'pi'   0      'pie'  0;
       0      0      0      0      0      0      0      'd'    0      0      0;
       0      0      0      0      0      0      0      'c'    0      0      0;
       0      0      0      0      0      0      'a'    'b'    0      0      0;
         0       0      0      0      0      0      0      0      0      0
0};

[finalMap, numPi] = piedmontPark(map);

finalMap →
      {0      0      0      0      0      0      0      0      0      0      0;
       0      0      0      0      0      0      0      0      0      0      0;
       0      0      0      0      0      0      0      0      0      0      0;
       0      0      0      0      0      0      0      0      0      0      0;
       0      0      0      0      0      0      0      0      0      'pie'  0;
       0      0      0      0      0      0      0      0      0      0      0;
       0      0      0      0      0      0      0      0      0      0      0;
       0      0      0      0      0      0      0      0      0      0      0;
```

```
        0     0     0     0     0     0     0     0     0     0
0};

% The array above is a 9x11 cell array
numPi →
           2
```

Notes:

- **You MUST submit the pythag function along with your code.**
- There will be a single, unambiguous path with no dead ends leading you to the finish. You will never have to make a decision on which way to go.
- By the nature of the pathing, you will never traverse a previously pathed location.
- You will always have a border on the edge cells of the array to prevent indexing out of bounds.
- You do not move diagonally.
- You should replace **any** tile you have travelled over with a `{0}`, regardless of the class, except the final tile (`{'pie'}`).
- The TPs will always teleport you to a path, and never directly on another TP.
- TP cell values will always result in integers when input into `pythag()` correctly.
- There will not be `{'pi'}`'s and TPs outside of your path.
- The teleporters are listed in the example as TP, but refer to the problem description for what they actually appear as

Hints:

- 👋Helper👋Functions👋
- Consider the order of operations you must make when iterating through the cell array.
- You can call functions that exist in your current folder from other functions!
- Remember that `pythag()` took in the length of one of the triangle's sides as the first input and the triangle's hypotenuse as its second input, and calculated the other side's length.