# Assignment #2 - List Comprehensions, Classes, CSV, Tabular Data

This assignment consists of three parts:

1. Highest and Lowest Potentially affected vehicles.
2. `nelta.py`
3. `nelta.py` and Recalls with Potentially affected vehicles > 500,000

👀 Click on this link: https://classroom.github.com/a/eN8sTZ-X (https://classroom.github.com/a/eN8sTZ-X) to accept this assignment in GitHub classroom. This will create your homework respository. Clone your new repository.

## Goals

- create classes
    - implement magic methods
    - create an iterator
- use list comprehensions
- work with csvs
- use a lambda function
- preview working with tabular data in Python

## Part 1 - Highest and Lowest Potentially Affected Vehicles

This part uses a modified dataset from https://www.kaggle.com/datasets/michaelbryantds/automobile-recalls-dataset (https://www.kaggle.com/datasets/michaelbryantds/automobile-recalls-dataset). This data contains Manufacturer name, Recall Type, Component, etc.

The dataset contained in your repository has been modified:

- columns have been removed
- column names have been changed
- some rows have been removed due to non numeric values in numeric columns
- some rows have been removed to consider only subset of the data

In `recall.ipynb`:

1. ❌ do not use external libraries, such as `numpy` or `pandas`
2. 👀 you **SHOULD** use the built-in `csv` module as the dataset we use has embedded commas
    - https://docs.python.org/3/library/csv.html#module-contents
3. ❌ do not use any regular loops ( `while` , `for` )
4. 👀 using list comprehensions, dictionary comprehensions and generators is okay (a `for` within these structures is ok)
5. open the `recalls-truncated.csv` file
6. display the `Manufacturer` and `Potentially Affected` fields for the highest and lowest `Potentially Affected` in the dataset
    - only do this for rows that have `vehicle` value for the `Recall Type` column
    - your discretion on what to do if there are ties
7. print out your results in any format that you prefer (dict, tuple, str, etc.)

Example output:

```
# this example prints out the Manufacturer and Potentially Affected  as two tuples
('Mercedes-Benz USA, LLC', 1) ('Chrysler (FCA US, LLC)', 1224078)
```

# Part 2 - `nelta.py`

## Background Information

Working with tabular data as a two-dimensional `Array` is pretty common, but as soon as you try to do some simple operations like filtering based on column name, things get complicated quickly unless you throw in some additional libraries or tools.

In this homework, we'll being creating (a *maybe over-engineered*) module for reading in csvs and filtering rows based on column values. You'll have to use some Python features in your implementation, like `__iter__`, list comprehensions, etc.

For example, imagine we had a csv containing people's names, the number of fruits they eat weekly, and their favorite color (um... idk?). This data is in a csv called `fruitarians.csv`, and can be visualized as a table (note that row `#` is not in the csv):

```
row #    first       last   weekly_fruits_eaten  fav_color
    0      abe      apple                     0        red
    1      bob     banana                     4     yellow
    2    carol    coconut                   100      white
    3      bob  blueberry                     9       blue
    4      eve     endive                    20      green
    5  frances      fruit                     5          ?
    6      ann      apple                    23      green
```

What if we'd like to run some reports on this? For example:

- read in the csv...
- what do the first three columns look like?
- how many people have the last name, apple?
- what is the first name and number of fruits eaten / week of everyone that eats more than 10 fruits a week
- how many people have a first name less than 4 letters eat less than 10 fruit / week?

We'll create a module to do this... and it'll look something like this:

```
import nelta as nt

# read in the csv
t = nt.read_csv('fruitarians.csv')

# what do the first three columns look like?
t.head(3)

"""
                first               last weekly_fruits_eaten           fav_color
0                 abe              apple                 0.0                 red
1                 bob             banana                 4.0              yellow
2               carol            coconut               100.0               white
"""

# how many people have the last name, apple?
t[t['last'] == 'apple'].shape()[0]

"""
2
"""

# what is the first name and number of fruits eaten / week of everyone that eats more than 10 fruits
a week
t[t['weekly_fruits_eaten'] > 10][['first', 'weekly_fruits_eaten']]

"""
                first weekly_fruits_eaten
2               carol               100.0
4                 eve                20.0
6                 ann                23.0
"""

# how many people have a first name less than 4 letters eat less than 10 fruit / week?
def length_less_than(n):
    def is_length_less_than_n(s):
        return len(s) < n
    return is_length_less_than_n
first_name_three = t[t['first'].map(length_less_than(4))]
first_name_three[first_name_three['weekly_fruits_eaten'] < 10]

"""
                first               last weekly_fruits_eaten           fav_color
0                 abe              apple                 0.0                 red
1                 bob             banana                 4.0              yellow
3                 bob          blueberry                 9.0                blue
"""
```

## Instructions

1. You'll be modifying two classes:
    - `LabeledList` - kind of like a `dict`, but ordered, allows *vectorized* operations, and iterates over values instead of keys
        - ⚠️ This is already **partially implemented**!
        - You'll only need to add missing methods
    - `Table` - a table of data with column labels
2. Open `nelta.py` in a text editor of your choice
3. When writing code for the classes mentioned above...

- **YOU MUST USE AT LEAST 4 LIST COMPREHENSIONS!**
- **YOU CANNOT USE THE FOLLOWING MODULES**: `numpy`, `pandas`
- **YOU SHOULD** use the built-in `csv` module

4. Using your previously implemented `nelta.py` as a module, import it and use it in a notebook to do some very simple data analysis!

---

# `LabeledList`

A `LabeledList` acts like a dictionary... but:

- it's ordered (note that 3.7 and greater has ordered dictionaries by default, though)
- when you loop over it, you get values instead of keys
- if you use a comparison operator with it... and a scalar value (a number, for example), that comparison is done on each value, yielding a new `LabeledList` composed of only `bool` values
- duplicate 'keys' are allowed

Here's how you might use it:

```
ll = nt.LabeledList([1, 2, 3, 4, 5], ['A', 'BB', 'BB', 'CCC', 'D'])
ll['A'] # gives back value at label 'A'
ll['BB'] # gives back new LabeledList composed of labels 'BB' and their values
ll[['A', 'D', 'BB', 'BB']] # gives back new LabeledList composed of the labels specified in list...
along with their values
ll > 2 # gives back a new LabeledList composed of labels in original, along with boolean results of
comparison
```

Implementation:

- 👀 **several properties / methods for this class are already implemented** (see specs below)
- ⚠️ **you'll only need to write the following** (again, see specs below):
  - `__iter__`
  - `__gt__`
  - `__eq__`
  - `__ne__`
  - `__lt__`
  - `map`

---

## Properties:

✅ Already Implemented

- `self.values` - contains the values in this `LabeledList` as a `list`
  - ll = nt.LabeledList([1, 2, 3, 4, 5], ['A', 'BB', 'BB', 'CCC', 'D'])
  - ll.values # [1, 2, 3, 4, 5]
- `self.index` - contains the labels in this `LabeledList` as a `list`
  - ll = nt.LabeledList([1, 2, 3, 4, 5], ['A', 'BB', 'BB', 'CCC', 'D'])
  - ll.index # ['A', 'BB', 'BB', 'CCC', 'D']

---

### `__init__(self, data=None, index=None)`

✅ Already Implemented

Creates a new `LabeledList`.

- `data` - the values stored in `self.values`; represents all of the values in this `LabeledList`
- `index` - the labels associated with each value

`data` and `index` are assumed to be the same length (no error checking is necessary) and the order of the elements in each list determines which values are associated with which labels (if data is `[0, 1]` and values are `['foo', 'bar']`, then the label `0` is associated with the value `'foo'`

You can assume that the labels and values are only `str`, `int`, `float`, and `bool` (no type checking is needed in the constructor). Duplicate labels are allowed.

Note that if `index` is `None` then the labels should be from 0 to the length of the data - 1:

```
list_with_default_labels = nt.LabeledList(['foo', 'bar', 'baz'])

""""
0 foo
1 bar
2 baz
""""

list.index # [0, 1, 2]
```

However, if `index` is provided...

```
ll = nt.LabeledList([1, 2, 3, 4, 5], ['A', 'BB', 'BB', 'CCC', 'D'])
""""
  A 1
 BB 2
 BB 3
CCC 4
  D 5
""""
```

---

## `__str__(self)` and `__repr__(self)`

✅ Already Implemented

These two methods will give the string representation of a `Labeled List`. `__str__` is used for human readable format (such as when printing) and `__repr__` is used for displaying *what the object actually is* (for example, debugging by just typing the object name in the interactive shell).

For our purposes, these will return the same string (in fact, one can call the other).

The string should be a tabular format where labels are on the left and values on the right. You can space this out any way you like, as long as it's very clear what the labels and columns are.

Using the earlier example, here's a nicely formatted `LabeledList`:

```
ll = nt.LabeledList([1, 2, 3, 4, 5], ['A', 'BB', 'BB', 'CCC', 'D'])
""""
  A 1
 BB 2
 BB 3
CCC 4
  D 5
""""
```

It will be useful to use dynamically padded strings to maintain consistent widths for columns. This can be done with format strings and the format specification mini language (https://docs.python.org/3/library/string.html#format-specification-mini-language). For example:

```
s = 'foo'

# print out 'foo' so that it's padded with spaces and tis total length is 10
print(f'{s:>10})

# results in:
#       foo
```

The `:` signals that a format specifier is coming up. The `>` aligns right. Finally, the `10` is the total width of the new string (spaces will pad the left side).

Of course, you may want the `10` to be variable...

```
vals_max_len = m # imagine that this is the length of the longest label
label = s # imagine that this is a label whose length is shorter than the longest label
# we want to pad this thing ^^^^

# create a format specifier that right justifies and pads
format_spec = f'>{vals_max_len}'

# now add that format specifier to another formatted string by nesting curly braces!?
f'{label:{format_spec}}'
```

Note that if the variable in the format string is a boolean and a format specifier is given, then the boolean will either be a 0 or 1 rather than `True` or `False` which is ok for our purposes (a work-around is to convert the boolean to a string first... then format with `f''`)

👀 **do your best to have dynamic widths, but grading will be generous for this feature**

---

## `__getitem__(self, key_list)`

✅ Already Implemented

`__getitem__` allows our object to be indexed / 'keyed' into as if it were a `dict`. In `LabeledList` the label is the key. Our implementation's key behavior depends on the type of the key:

1. if the key is another `LabeledList` then the key is the `values` property of that labeled list (which is, of course a `list` ... see below for how to handle `list` and a `list` of only `bool` values)
2. if the key is a `list`, then that means that we're retrieving multiple labels and values, so a new `LabeledList` is returned with each label specified and its associated value (if a label occurs more than once, add all occurrences)
3. if the key is specifically a list of `bool` values, then give back a new `LabeledList` where the only label and value pairs given are the ones where the position matches the position of a `True` in the incoming key list (you can assume that the list of `bool` values must be the same length as the `index` of labels... you can error handling if it makes it easier to debug your code, though!)
4. given any single value as the label (such as `str`, `int`)... you will get back: a. the value associated with that exact label if the label occurs only once b. a new `LabeledList` composed of that label repeated, along with its values

Ok. So that's pretty confusing. Here are some examples:

```
ll = nt.LabeledList([1, 2, 3, 4, 5], ['A', 'BB', 'BB', 'CCC', 'D'])

# 1 (values are taken from LabeledList as a list...
# more than one label yields all label and value pairs)
ll[nt.LabeledList(['A', 'BB'])]

"""
 A 1
BB 2
BB 3
"""

# 2 (same as above, but with plain list)
ll[['A', 'BB']]

"""
 A 1
BB 2
BB 3
"""

# 3 (only the last two label value pairs have the same positions as True
ll[[False, False, False, True, True]]

"""
CCC 4
  D 5
"""

# 4a (value is returned as is... just like a dict)
ll['A']

"""
1
"""

# 4b (new LabeledList is returned even though only single value key)
ll['BB'] #

"""
BB 2
BB 3
"""
```

Use the built-in function, `isinstance` (https://docs.python.org/3/library/functions.html#isinstance) to check if a value is a particular type:

```
isinstance(key_list, LabeledList)
isinstance(key_list, list)
```

---

## `__iter__(self)`

Implement `__iter__` so that it returns a new object that has a `__next__` method. Since `self.values` is a list, you can return the result of calling the built-in function `iter` with `self.values` as the argument.

```
# using the previous version of ll
for val in ll:
    print(val)
"""
1
2
3
4
5
"""
```

---

## `__eq__(self, scalar)`, `__ne__(self, scalar)`, `__gt__(self, scalar)`, `__lt__(self, scalar)`

These methods all correspond to an associated comparison operator ( `==` , `>` , etc.). They should return a new `LabeledList` of `bool` values corresponding to the operation specified for every value compared to the `scalar` passed in.

- ⚠️ if the value being compared to the `scalar` is `None` , return `False` .
- although the parameter name is `scalar` , you can let this work with any compatible types
- you don't have to deal with any `TypeError` s (just let them occur)
- **if there is an index present, keep that index**
- this might be a good place to get in your four list comprehensions

```
# compares every element in the labeled list to 2 using >
nt.LabeledList([0, 1, 2, 3, 4]) > 2
0 False
1 False
2 False
3  True
4  True
```

```
# compares every element in the labeled list to 1 using ==
ll = nt.LabeledList([1, 2], ['x', 'y'])
ll == 1
x  True
y False
```

```
# note that you can allow these operations to work on different types
# here, we have strings compared with ==
nt.LabeledList(['a', 'b', 'c', 'b', 'b']) == 'b'
0 False
1  True
2 False
3  True
4  True
```

```
# if a value in the labeled list is None, then always return False
nt.LabeledList([None, 0, 2]) > 1
0 False
1 False
2  True
```

---

## `map(self, f)`

Gives back a new `LabeledList` with all of the values transformed to the result of calling `f` on that value.

```
def squared(n):
    return n ** 2
nt.LabeledList([5, 6, 7]).map(squared)
0 25
1 36
2 49
```

---

# Table

A `Table` represents tabular data with row labels ( `index` ) and column names ( `columns` )... along with 2 dimensional grid of data ( `data` ).

It supports operations for filtering by values in a column... as well as selecting specific columns.

## Properties

- `self.values` - contains the values in this `Table` as a `list`
  - `t = Table([[1, 2, 3],[4, 5, 6]],['a', 'b'], ['x', 'y', 'z'])`
  - `.values # [[1, 2, 3],[4, 5, 6]]`
- `self.index` - contains the row labels in this `Table` as a `list`
  - `t = Table([[1, 2, 3],[4, 5, 6]],['a', 'b'], ['x', 'y', 'z'])`
  - `t.index # ['a', 'b']`
- `self.columns` - contains the column names in this `Tabled` as a `list`
  - `t = Table([[1, 2, 3],[4, 5, 6]],['a', 'b'], ['x', 'y', 'z'])`
  - `t.index # ['x', 'y', 'z']`

### `__init__(self, data, index=None, columns=None)`

If either `index` or `columns` are not included, then default to numeric values from 0 up to length of `index` - 1 or `columns` - 1

```
t = Table([['foo', 'bar', 'baz'],['qux', 'quxx', 'corge']])

      0     1     2
0   foo   bar   baz
1   qux  quxx corge
```

Otherwise, adding the `index` and `columns` as arguments will explicitly set the row labels and column names

```
t = Table(d, ['foo', 'bar', 'bazzy', 'qux', 'quxx'], ['a', 'b', 'c', 'd', 'e'])

         a    b    c    d    e
  foo 1000   10  100    1  1.0
  bar  200    2  2.0 2000   20
bazzy    3  300 3000  3.0   30
  qux   40 4000  4.0  400    4
 quxx    7    8    6    3   41
```

### `__str__(self)` and `__repr__(self)`

Again, these two methods will give the string representation of an object. `__str__` is used for a human readable format (for example, used with `print` ), and `__repr__` is used for displaying *what the object actually is* (for example, typing the object name Jupyter). Both of these methods return strings, and for our purposes, these can be the same string.

For a `Table` object, create a string representation in any way such that rows and columns can be clearly distinguished. See the example below for a potential format (it's ultimately up to you how you'd like to format it, though... as long as the grader can read it and determine which rows and columns are aligned).

Please read the notes for the `LabeledList` `__str__` and `__reper__` methods for info on setting a consistent width for cells using string formatting ( `f'{foo:{format_spec}}'` ).

```
t = Table([['foo', 'bar', 'baz'],['qux', 'quxx', 'corge']])

      0     1     2
0   foo   bar   baz
1   qux  quxx corge
```

## `__getitem__(self, col_list)`

`__getitem__` allows our `Table` to be indexed / 'keyed' into as if it were a `dict` . The behavior of indexing or retrieving by key depends on the type of the value used as a key!

Essentially, most keys result in selecting all rows, but specifying which columns to include in a new `Table` (for example `t['a']` selects column a from all rows, and t[['a', 'b']] selects column a and b from all rows. The main exception is a list or `LabeledList` of `bool` values... which specifies which rows to include based on position of the `bool` value and the position of the row (for example, if `t` has 2 rows, then t[[True, False]] will only give back the first row as a new `Table` .

If there's ever only one column returned, give back a `LabeledList` . Otherwise, give back a `Table` .

For exact details, see below:

1. if the key is a `LabeledList` , then the key is the `values` property of that labeled list (which is a `list` )... and a `Table` consisting of **only** the columns contained in the `LabeledList` is returned (note that all rows are returned) ... note that if the `LabeledList` `values` are all `bool` , then follow the procedure for dealing with a list of `bool` values as shown below
2. if the key is a `list` , then that means that we're retrieving multiple columns, so a new `Table` is returned including only the columns specified by the elements in the key list passed in. If a key list has repeated column names, duplicate the column. If a column name matches more than one column, add both columns in the resulting Table.
3. if the key is specifically a list of `bool` values, then give back a new `Table` where the only rows given are the ones where the position matches the position of a `True` in the incoming key list (you can assume that the list of `bool` values must be the same length as the total number of rows in the `Table` object)
4. given any single value as the label (such as `str` , `int` )... you will get back: a. *that* column for all rows as a `LabeledList` if there is only one occurrence of that column name b. a new `Table` composed of that column repeated if there are duplicate column names

Once again, the behavior is complex enough to warrant examples:

```
#####
# Remember... if only one column is given back, return a LabeledList
# ...but if there's more than one column, give back a Table
#####

# 1 (using a LabeledList to select columns)
t = Table(d, ['foo', 'bar', 'bazzy', 'qux', 'quxx'], ['a', 'b', 'c', 'd', 'e'])
t[LabeledList(['a', 'b'])]
"""
         a    b
  foo 1000   10
  bar  200    2
bazzy    3  300
  qux   40 4000
 quxx    7    8
"""

# 2 (the first two columns are selected using a list of columns...
# notice that repeat columns are allowed)
t = Table([[15, 17, 19], [14, 16, 18]], columns=['x', 'y', 'z'])
t[['x', 'x', 'y']]

"""
   x  x  y
0 15 15 17
1 14 14 16
"""

# 3 (select only the first and third rows by using a list of booleans)
t = Table([[1, 2, 3], [4, 5, 6], [7, 8 , 9]], columns=['x', 'y', 'z'])
t[[True, False, True]]

"""
  x y z
0 1 2 3
2 7 8 9
"""

# 4a (using a column name that matches only a single column gives
# back a LabeledList... note no column names, but there are labels!)

t = Table([[1, 2, 3], [4, 5, 6]], columns=['a', 'b', 'a'])
t['b']

"""
0 2
1 5
"""

# 4b (however, if more than one column matches column name, include
# all matched columns in the resulting Table object)
t = Table([[1, 2, 3], [4, 5, 6]], columns=['a', 'b', 'a'])
t['a']

"""
  a a
0 1 3
```

```
1 4 6
"""
```

### `head(self, n)` and `tail(self, n)`

Returns a `Table` showing the first or last `n` row respectively.

```
t = Table([[1, 2], [3, 4], [5, 6], [7, 8]], columns=['x', 'y'])

print(t.head(2))

"""
  x y
0 1 2
1 3 4
"""

print(t.tail(2))

"""
  x y
2 5 6
3 7 8
"""
```

### `shape(self)`

Gives back a tuple containing the number of rows and columns:

```
t = Table([[1, 2], [3, 4], [5, 6], [7, 8]], columns=['x', 'y'])
t.shape()

"""
(4, 2)
"""
```

## `read_csv(fn)`

Finally, in `nelta.py`, write a function that reads in a `csv` file and gives back a `Table` object:

1. assume that the first row is the header (it can be treated as column names)
2. there can be commas embedded in the actual data, so you'll have to use the built in `csv` module
3. **convert numeric values to `floats`** (use `try` / `except` ... and look for `ValueError` specifically)

⚠️ **when creating a `Table` object, use a generated auto-incrementing index** (do not use the first column as the index... as in the example of the `fruitarians.csv` file where "row" is not actually included in the file)

To test your code, you can try to open a csv from your `nelta.py` module. Once you're finished testing, you can comment out the code (or alternatively, you can wrap your test code in `if __name__ == '__main__':` to only run it when the module isn't being imported)

# Part 3 - `nelta.py` and Recalls with Potentially Affected > 500,000

In this part, we will use the same `recalls-truncated.csv` file in the `data` directory. **Write your code in the provided notebook, `recall.ipynb`**

⚠️ When you open your data file

- do your best to make sure you use a relative path (simply `nt.read_csv('name_of_file.ext')` should be sufficient)
- if you are having issues doing this, add a comment before your line that says: `TODO: modify the path`

Once you've read in your file as a `Table` using your `nelta` module, and saving it to a variable:

1. display the number of rows and columns for your `Table` (you can use `shape` to do this)

```
(350, 5)
```

2. show the columns in your `Table`

```
['Date',
 'Manufacturer',
 'Subject',
 'Recall Type',
 'Potentially Affected']
```

3. display the first 4 rows of the dataset

```
         Date                       Manufacturer                                       Subject
Recall Type  Potentially Affected
 0  01/06/2023     Triple E Recreational Vehicles           Battery Disconnect Switch May Short
Vehicle                 341.0
 1  01/05/2023                 Volvo Car USA, LLC                    Steering Wheel May Lock Up
Vehicle                  74.0
 2  12/29/2022  Volkswagen Group of America, Inc.       12-Volt Battery Cable May Short Circuit
Vehicle                1042.0
 3  12/29/2022          Indian Motorcycle Company  Kickstand May Not Retract Properly/FMVSS 123
Vehicle                4653.0
```

4. save the last 4 rows of the data set into a variable called `last_four`
5. show only the label ("index") and Subject column of last_four as a LabeledList (no column name needed)

```
346     Improperly Secured Front Seat Belt Anchor
347     Rearview Camera Image May Not Display/FMVSS 111
348     Secondary Hood Latch Corrosion
349     Tire Sidewall Separation/ FMVSS 139
```

6. loop over the `Subject` column as a `LabeledList` and print out each value

```
Improperly Secured Front Seat Belt Anchor
Rearview Camera Image May Not Display/FMVSS 111
Secondary Hood Latch Corrosion
Tire Sidewall Separation/ FMVSS 139
```

7. show the label ("index") and both the `Manufacturer` and `Subject` columns of `last_four` as a `Table` (include column names)

```
         Manufacturer   Subject
346     Rivian Automotive, LLC  Improperly Secured Front Seat Belt Anchor
347     Chrysler (FCA US, LLC)  Rearview Camera Image May Not Display/FMVSS 111
348     General Motors, LLC     Secondary Hood Latch Corrosion
349     PT. Elangperdana Tyre Industry  Tire Sidewall Separation/ FMVSS 139
```

8. using your original, unaltered `Table`, find all rows with `Recall Type` Vehicle and save to a variable called `vehicles`
9. find the number of rows in your `vehicles Table`:

```
313
```

10. using your `vehicles` `Table`, use the column `Potentially Affected` use the greater than operator ( `>` ) to compare with 500,000… save this to a variable called `my_filter`
11. get the type of `my_filter` using the function `type` (it should be a `LabeledList` )
12. index into your `my_filter` with `3` (you should get `False` )
13. use your `my_filter` variable as the index to your `vehicles` `Table` – this translates to showing all recalls in your `vehicles` `Table` that have `Potentially Affected` greater than 500000

```
        Date           Manufacturer                                         Subject  Recall Ty
pe  Potentially Affected
  60  12/08/2022     General Motors, LLC      Running Lights May Not Deactivate/FMVSS 108      V
ehicle             740108.0
  61  12/08/2022  Chrysler (FCA US, LLC)                 Tailgate May Open While Driving      V
ehicle            1224078.0
 110  11/18/2022      Ford Motor Company  Cracked Fuel Injector May Leak and Cause a Fire      V
ehicle             521746.0
 284  09/19/2022             Tesla, Inc.                Power Windows May Pinch/FMVSS 118      V
ehicle            1096762.0
```

14. perform the same calculation as above, but save your resulting `Table` into a variable called `rare_occurence`
15. using your `rare_occurence` table, transform the `Manufacturer` to uppercase by using `map` on the column (that is, index into the `Table` first, and then use `map` )

```
 60    GENERAL MOTORS, LLC
 61 CHRYSLER (FCA US, LLC)
110     FORD MOTOR COMPANY
284            TESLA, INC.
```

16. this is a tricky one! combine some of the functionality from the previous steps to find the recalls that happened in `2023` (hint, you'll have to split the date)

```
      Date                   Manufacturer                              Subject  Recall Type  Po
tentially Affected
 0  01/06/2023  Triple E Recreational Vehicles  Battery Disconnect Switch May Short      Vehicle
341.0
 1  01/05/2023             Volvo Car USA, LLC          Steering Wheel May Lock Up      Vehicle
74.0
```

# Annotations

Add a `README.md` that links to the lines of code (see this link for instructions how) (https://help.github.com/articles/creating-a-permanent-link-to-a-code-snippet/) where you have:

Use this exact markdown format in your `README.md` to add links (including []'s and ()'s):

```
## 4 List Comprehensions

1. [short description 1](https://path.copied/for/permalink/to/code)
2. [short description 2](https://path.copied/for/permalink/to/code)
3. [short description 3](https://path.copied/for/permalink/to/code)
4. [short description 4](https://path.copied/for/permalink/to/code)
```

Additionally, to annotate the use of lambdas in your notebook, add a table of contents at the top of your notebook as a markdown cell that contains links to other parts of your notebook:

```
# Table of Contents

* [first lambda](#first-lambda)
* [second lambda](#second-lambda)
```

And then, before each cell that contains a require lambda, add a header. For example:

```
## first lambda
```

Note that the link to the cell above is the lowercase, "dashed", version... prefixed with a hash.

Download this notebook (../notebooks/internal-links.ipynb) for an example