

lab03

March 8, 2022

Name: Allan Gongora

Section: 0131

1 Lab 3: Tables

Welcome to Lab 3!

This week, we will focus on manipulating tables. Tables are described in [Chapter 6](#) of the text.

Please complete this notebook by filling in the cells provided. Before you begin, execute the following cell to load the provided tests.

```
[1]: pip install gofer-grader
```

```
Requirement already satisfied: gofer-grader in /opt/conda/lib/python3.7/site-  
packages (1.1.0)  
Requirement already satisfied: tornado in /opt/conda/lib/python3.7/site-packages  
(from gofer-grader) (6.1)  
Requirement already satisfied: jinja2 in /opt/conda/lib/python3.7/site-packages  
(from gofer-grader) (3.0.3)  
Requirement already satisfied: pygments in /opt/conda/lib/python3.7/site-  
packages (from gofer-grader) (2.11.2)  
Requirement already satisfied: MarkupSafe>=2.0 in /opt/conda/lib/python3.7/site-  
packages (from jinja2->gofer-grader) (2.0.1)  
Note: you may need to restart the kernel to use updated packages.
```

```
[2]: import numpy as np  
from datascience import *  
  
# These lines load the tests.  
  
from gofer.ok import check
```

Recommended Reading: * [Tables](#)

- 1) For all problems that you must write explanations and sentences for, you **must** provide your answer in the designated space. This can include:
 - A) Sentence responses to questions that ask for an explanation
 - B) Numeric responses to multiple choice questions

- C) Programming code
- 2) Moreover, throughout this lab and all future ones, please be sure to not re-assign variables throughout the notebook! For example, if you use `max_temperature` in your answer to one question, do not reassign it later on. Otherwise, you will fail tests that you thought you were passing previously!

Once you're finished, select "Save and Checkpoint" in the File menu. Your name and course section number should be in the first and last cell of the assignment. Be sure you have run all cells with code and that the output from that is showing. Then click "Print Preview" in the File menu. Print a copy from there in pdf format. (This means you right click and choose print and choose "save as pdf" from your printer options.) You will need to submit the pdf in Canvas by the deadline.

The gopher grader output and/or output from your coding are essential to helping your instructor grade your work correctly and in a timely manner.

Files submitted that are missing the required output will lose some to all points so double check your pdf before submitting.

1.1 1. Introduction

For a collection of things in the world, an array is useful for describing a single attribute of each thing. For example, among the collection of US states, an array could describe the land area of each. Tables extend this idea by describing multiple attributes for each element of a collection.

In most data science applications, we have data about many entities, but we also have several kinds of data about each entity.

For example, in the cell below we have two arrays. The first one contains the world population in each year (estimated by the US Census Bureau), and the second contains the years themselves. These elements are in order, so the year and the world population for that year have the same index in their corresponding arrays.

```
[3]: population_amounts = Table.read_table("world_population.csv").
      ↪column("Population")
years = np.arange(1950, 2015+1)
print("Population column:", population_amounts)
print("Years column:", years)
```

```
Population column: [2557628654 2594939877 2636772306 2682053389 2730228104
2782098943
```

```
2835299673 2891349717 2948137248 3000716593 3043001508 3083966929
3140093217 3209827882 3281201306 3350425793 3420677923 3490333715
3562313822 3637159050 3712697742 3790326948 3866568653 3942096442
4016608813 4089083233 4160185010 4232084578 4304105753 4379013942
4451362735 4534410125 4614566561 4695736743 4774569391 4856462699
4940571232 5027200492 5114557167 5201440110 5288955934 5371585922
5456136278 5538268316 5618682132 5699202985 5779440593 5857972543
5935213248 6012074922 6088571383 6165219247 6242016348 6318590956
6395699509 6473044732 6551263534 6629913759 6709049780 6788214394
6866332358 6944055583 7022349283 7101027895 7178722893 7256490011]
```

```
Years column: [1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962
```

1963

```
1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977
1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991
1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005
2006 2007 2008 2009 2010 2011 2012 2013 2014 2015]
```

Suppose we want to answer this question:

When did world population cross 6 billion?

You could technically answer this question just from staring at the arrays, but it's a bit convoluted, since you would have to count the position where the population first crossed 6 billion, then find the corresponding element in the years array. In cases like these, it might be easier to put the data into a *Table*, a 2-dimensional type of dataset.

The expression below:

- creates an empty table using the expression `Table()`,
- adds two columns by calling `with_columns` with four arguments,
- assigns the result to the name `population`, and finally
- evaluates `population` so that we can see the table.

The strings "Year" and "Population" are column labels that we have chosen. The names `population_amounts` and `years` were assigned above to two arrays of the same length. The function `with_columns` (you can find the documentation [here](#)) takes in alternating strings (to represent column labels) and arrays (representing the data in those columns), which are all separated by commas.

```
[4]: population = Table().with_columns(
    "Population", population_amounts,
    "Year", years
)
population
```

```
[4]: Population | Year
2557628654 | 1950
2594939877 | 1951
2636772306 | 1952
2682053389 | 1953
2730228104 | 1954
2782098943 | 1955
2835299673 | 1956
2891349717 | 1957
2948137248 | 1958
3000716593 | 1959
... (56 rows omitted)
```

Now the data are all together in a single table! It's much easier to parse this data—if you need to know what the population was in 1959, for example, you can tell from a single glance. We'll revisit this table later.

1.2 2. Creating Tables

Question 2.1 In the cell below, we’ve created 2 arrays. Using the steps above, assign `top_10_movies` to a table that has two columns called “Rating” and “Name”, which hold `top_10_movie_ratings` and `top_10_movie_names` respectively.

```
[5]: top_10_movie_ratings = make_array(9.2, 9.2, 9., 8.9, 8.9, 8.9, 8.9, 8.9, 8.9, 8.
      ↪8)
top_10_movie_names = make_array(
    'The Shawshank Redemption (1994)',
    'The Godfather (1972)',
    'The Godfather: Part II (1974)',
    'Pulp Fiction (1994)',
    "Schindler's List (1993)",
    'The Lord of the Rings: The Return of the King (2003)',
    '12 Angry Men (1957)',
    'The Dark Knight (2008)',
    'Il buono, il brutto, il cattivo (1966)',
    'The Lord of the Rings: The Fellowship of the Ring (2001)')

top_10_movies = Table().with_columns("Rating", top_10_movie_ratings, "Name",
      ↪top_10_movie_names)
# We've put this next line here so your table will get printed out when you
# run this cell.
top_10_movies
```

```
[5]: Rating | Name
     9.2    | The Shawshank Redemption (1994)
     9.2    | The Godfather (1972)
      9     | The Godfather: Part II (1974)
     8.9    | Pulp Fiction (1994)
     8.9    | Schindler's List (1993)
     8.9    | The Lord of the Rings: The Return of the King (2003)
     8.9    | 12 Angry Men (1957)
     8.9    | The Dark Knight (2008)
     8.9    | Il buono, il brutto, il cattivo (1966)
     8.8    | The Lord of the Rings: The Fellowship of the Ring (2001)
```

```
[6]: check('tests/q2_1.py')
```

```
[6]: <gofer.ok.OKTestsResult at 0x7fc83b887690>
```

1.2.1 Loading a Table from a File

In most cases, we aren’t going to go through the trouble of typing in all the data manually. Instead, we can use our `Table` functions.

`Table.read_table` takes one argument, a path to a data file (a string) and returns a table. There are many formats for data files, but CSV (“comma-separated values”) is the most common.

Question 2.2 The file `imdb.csv` contains a table of information about the 250 highest-rated movies on IMDb. Load it as a table called `imdb`.

```
[7]: imdb = Table.read_table("imdb.csv")
      imdb
```

```
[7]: Votes | Rating | Title | Year | Decade
      88355 | 8.4 | M | 1931 | 1930
      132823 | 8.3 | Singin' in the Rain | 1952 | 1950
      74178 | 8.3 | All About Eve | 1950 | 1950
      635139 | 8.6 | Léon | 1994 | 1990
      145514 | 8.2 | The Elephant Man | 1980 | 1980
      425461 | 8.3 | Full Metal Jacket | 1987 | 1980
      441174 | 8.1 | Gone Girl | 2014 | 2010
      850601 | 8.3 | Batman Begins | 2005 | 2000
      37664 | 8.2 | Judgment at Nuremberg | 1961 | 1960
      46987 | 8 | Relatos salvajes | 2014 | 2010
      ... (240 rows omitted)
```

```
[8]: check('tests/q2_2.py')
```

```
[8]: <gofer.ok.OKTestsResult at 0x7fc83b86fa50>
```

Notice the part about “... (240 rows omitted).” This table is big enough that only a few of its rows are displayed, but the others are still there. 10 are shown, so there are 250 movies total.

Where did `imdb.csv` come from? Take a look at this lab’s folder. You should see a file called `imdb.csv`.

Open up the `imdb.csv` file in that folder and look at the format. What do you notice? The `.csv` filename ending says that this file is in the [CSV \(comma-separated value\) format](#).

1.3 3. Using Lists

A *list* is another Python sequence type, similar to an array. It’s different than an array because the values it contains can all have different types. A single list can contain `int` values, `float` values, and strings. Elements in a list can even be other lists! A list is created by giving a name to the list of values enclosed in square brackets and separated by commas. For example, `values_with_different_types = ['data', 8, 8.1]`

Lists can be useful when working with tables because they can describe the contents of one row in a table, which often corresponds to a sequence of values with different types. A list of lists can be used to describe multiple rows.

Each column in a table is a collection of values with the same type (an array). If you create a table column from a list, it will automatically be converted to an array. A row, on the hand, mixes types.

Here’s a table from Chapter 5. (Run the cell below.)

```
[9]: # Run this cell to recreate the table
      flowers = Table().with_columns(
```

```

    'Number of petals', make_array(8, 34, 5),
    'Name', make_array('lotus', 'sunflower', 'rose')
)
flowers

```

```

[9]: Number of petals | Name
      8               | lotus
      34              | sunflower
      5               | rose

```

Question 3.1 Create a list that describes a new fourth row of this table. The details can be whatever you want, but the list must contain two values: the number of petals (an `int` value) and the name of the flower (a string). For example, your flower could be “pondweed”! (A flower with zero petals)

```

[10]: my_flower = [0, "pondweed"]
      my_flower

```

```

[10]: [0, 'pondweed']

```

```

[11]: check('tests/q3_1.py')

```

```

[11]: <gofer.ok.OKTestsResult at 0x7fc83b87f450>

```

Question 3.2 `my_flower` fits right in to the table from Chapter 5. Complete the cell below to create a table of seven flowers that includes your flower as the fourth row followed by `other_flowers`. You can use `with_row` to create a new table with one extra row by passing a list of values and `with_rows` to create a table with multiple extra rows by passing a list of lists of values.

```

[12]: # Use the method .with_row(...) to create a new table that includes my_flower

four_flowers = flowers.with_row(my_flower)

# Use the method .with_rows(...) to create a table that
# includes four_flowers followed by other_flowers

other_flowers = [[10, 'lavender'], [3, 'birds of paradise'], [6, 'tulip']]

seven_flowers = four_flowers.with_rows(other_flowers)
seven_flowers

```

```

[12]: Number of petals | Name
      8               | lotus
      34              | sunflower
      5               | rose
      0               | pondweed
      10              | lavender
      3               | birds of paradise

```

```
[13]: check('tests/q3_2.py')
```

```
[13]: <gofer.ok.OKTestsResult at 0x7fc83b8e7d50>
```

1.4 4. Analyzing Datasets

With just a few table methods, we can answer some interesting questions about the IMDb dataset.

If we want just the ratings of the movies, we can get an array that contains the data in that column:

```
[14]: imdb.column("Rating")
```

```
[14]: array([8.4, 8.3, 8.3, 8.6, 8.2, 8.3, 8.1, 8.3, 8.2, 8. , 8.1, 8.2, 8.3,
            8.3, 8.1, 8.4, 8.5, 8.2, 8.1, 8.4, 8.1, 8.1, 9.2, 8. , 8.2, 8.1,
            8.2, 8.5, 8. , 8.3, 8.1, 8. , 8. , 8.3, 8.1, 8. , 8. , 8.3, 8.4,
            8.1, 8.1, 8.5, 8.5, 8. , 8.3, 8.1, 8. , 8.6, 8.5, 8.3, 8.3, 8. ,
            8.2, 9.2, 8.2, 8.5, 8. , 8.9, 8.4, 8.2, 8.1, 8.3, 8.1, 8.1, 8.1,
            8.3, 8.2, 8.3, 8.7, 8.3, 8.6, 8. , 8.1, 8.2, 8.5, 8.3, 8.9, 8. ,
            8.6, 8.3, 8.1, 8.7, 8.4, 8.1, 8.4, 8. , 8.5, 8.8, 8.2, 8.2, 8.5,
            9. , 8. , 8. , 8.3, 8.4, 8.6, 8.5, 8.7, 8.4, 8.1, 8.1, 8.1, 8.7,
            8.4, 8.9, 8.1, 8.2, 8. , 8.5, 8.5, 8. , 8. , 8.4, 8.1, 8.1, 8. ,
            8. , 8.3, 8.1, 8. , 8.3, 8. , 8. , 8. , 8. , 8. , 8. , 8.7,
            8.3, 8. , 8. , 8.5, 8. , 8.1, 8.1, 8.1, 8.3, 8.2, 8.3, 8.9, 8.2,
            8.2, 8. , 8.3, 8.2, 8.9, 8.5, 8.5, 8.1, 8.1, 8.5, 8.3, 8. , 8.2,
            8.7, 8.3, 8.5, 8.1, 8.3, 8.2, 8.4, 8.1, 8.1, 8.1, 8. , 8.2, 8. ,
            8.6, 8.3, 8.2, 8. , 8.3, 8. , 8.2, 8. , 8.2, 8.8, 8.1, 8. , 8.1,
            8. , 8.2, 8.5, 8.1, 8.4, 8.1, 8.1, 8.7, 8.2, 8. , 8. , 8. , 8.3,
            8.4, 8. , 8.5, 8.1, 8.1, 8.2, 8.2, 8.4, 8.3, 8.6, 8.2, 8. , 8.1,
            8.2, 8.1, 8.3, 8.4, 8.5, 8.6, 8. , 8.3, 8.5, 8.5, 8.3, 8.5, 8.4,
            8. , 8.1, 8.7, 8.9, 8.3, 8.1, 8.1, 8. , 8.2, 8.4, 8.4, 8.1, 8.3,
            8.4, 8.2, 8.5, 8. , 8.2, 8.1, 8.4, 8.1, 8.6, 8.4, 8.1, 8.7, 8.1,
            8.2, 8.1, 8.3])
```

The value of that expression is an array, exactly the same kind of thing you'd get if you typed in `make_array(8.4, 8.3, 8.3, [etc])`.

Question 4.1 Find the rating of the highest-rated movie in the dataset.

Hint: Think back to the functions you've learned about for working with arrays of numbers. Ask for help if you can't remember one that's useful for this.

```
[15]: highest_rating = max(imdb.column("Rating"))
      highest_rating
```

```
[15]: 9.2
```

```
[16]: check('tests/q4_1.py')
```

```
[16]: <gofer.ok.OKTestsResult at 0x7fc83b76c2d0>
```

That's not very useful, though. You'd probably want to know the *name* of the movie whose rating you found! To do that, we can sort the entire table by rating, which ensures that the ratings and titles will stay together. Note that calling `sort` creates a copy of the table and leaves the original table unsorted.

```
[17]: imdb.sort("Rating")
```

```
[17]: Votes | Rating | Title | Year | Decade
46987 | 8 | Relatos salvajes | 2014 | 2010
55382 | 8 | Bom yeoreum gaeul gyeoul geurigo bom | 2003 | 2000
32385 | 8 | La battaglia di Algeri | 1966 | 1960
364225 | 8 | Jaws | 1975 | 1970
158867 | 8 | Before Sunrise | 1995 | 1990
56671 | 8 | The Killing | 1956 | 1950
87591 | 8 | Papillon | 1973 | 1970
43090 | 8 | Paris, Texas (1984) | 1984 | 1980
427099 | 8 | X-Men: Days of Future Past | 2014 | 2010
87437 | 8 | Roman Holiday | 1953 | 1950
... (240 rows omitted)
```

Well, that actually doesn't help much, either – we sorted the movies from lowest -> highest ratings. To look at the highest-rated movies, sort in reverse order:

```
[18]: imdb.sort("Rating", descending=True)
```

```
[18]: Votes | Rating | Title | Year | Decade
1027398 | 9.2 | The Godfather | 1972 | 1970
1498733 | 9.2 | The Shawshank Redemption | 1994 | 1990
692753 | 9 | The Godfather: Part II | 1974 | 1970
447875 | 8.9 | Il buono, il brutto, il cattivo (1966) | 1966 | 1960
1473049 | 8.9 | The Dark Knight | 2008 | 2000
384187 | 8.9 | 12 Angry Men | 1957 | 1950
1074146 | 8.9 | The Lord of the Rings: The Return of the King | 2003 | 2000
761224 | 8.9 | Schindler's List | 1993 | 1990
1166532 | 8.9 | Pulp Fiction | 1994 | 1990
1177098 | 8.8 | Fight Club | 1999 | 1990
... (240 rows omitted)
```

(The `descending=True` bit is called an *optional argument*. It has a default value of `False`, so when you explicitly tell the function `descending=True`, then the function will sort in descending order.)

So there are actually 2 highest-rated movies in the dataset: *The Shawshank Redemption* and *The Godfather*.

Some details about `sort`:

1. The first argument to `sort` is the name of a column to sort by.

2. If the column has strings in it, `sort` will sort alphabetically; if the column has numbers, it will sort numerically.
3. The value of `imdb.sort("Rating")` is a *copy of `imdb`*; the `imdb` table doesn't get modified. For example, if we called `imdb.sort("Rating")`, then running `imdb` by itself would still return the unsorted table.
4. Rows always stick together when a table is sorted. It wouldn't make sense to sort just one column and leave the other columns alone. For example, in this case, if we sorted just the "Rating" column, the movies would all end up with the wrong ratings.

Question 4.2 Create a version of `imdb` that's sorted chronologically, with the earliest movies first. Call it `imdb_by_year`.

```
[19]: imdb_by_year = imdb.sort("Year")
      imdb_by_year
```

```
[19]: Votes | Rating | Title | Year | Decade
      55784 | 8.3 | The Kid | 1921 | 1920
      58506 | 8.2 | The Gold Rush | 1925 | 1920
      46332 | 8.2 | The General | 1926 | 1920
      98794 | 8.3 | Metropolis | 1927 | 1920
      88355 | 8.4 | M | 1931 | 1930
      92375 | 8.5 | City Lights | 1931 | 1930
      56842 | 8.1 | It Happened One Night | 1934 | 1930
      121668 | 8.5 | Modern Times | 1936 | 1930
      69510 | 8.2 | Mr. Smith Goes to Washington | 1939 | 1930
      259235 | 8.1 | The Wizard of Oz | 1939 | 1930
      ... (240 rows omitted)
```

```
[20]: check('tests/q4_2.py')
```

```
[20]: <gofer.ok.OKTestsResult at 0x7fc83b77a7d0>
```

Question 4.3 What's the title of the earliest movie in the dataset? You could just look this up from the output of the previous cell. Instead, write Python code to find out.

Hint: Starting with `imdb_by_year`, extract the Title column to get an array, then use `item` to get its first item.

```
[21]: earliest_movie_title = imdb_by_year.row(0)[2]
      earliest_movie_title
```

```
[21]: 'The Kid'
```

```
[22]: check('tests/q4_3.py')
```

```
[22]: <gofer.ok.OKTestsResult at 0x7fc83b784650>
```

1.5 5. Finding Pieces of a Dataset

Suppose you're interested in movies from the 1940s. Sorting the table by year doesn't help you, because the 1940s are in the middle of the dataset.

Instead, we use the table method `where`.

```
[23]: forties = imdb.where('Decade', are.equal_to(1940))
      forties
```

```
[23]: Votes | Rating | Title | Year | Decade
      55793 | 8.1 | The Grapes of Wrath | 1940 | 1940
      86715 | 8.3 | Double Indemnity | 1944 | 1940
      101754 | 8.1 | The Maltese Falcon | 1941 | 1940
      71003 | 8.3 | The Treasure of the Sierra Madre | 1948 | 1940
      35983 | 8.1 | The Best Years of Our Lives | 1946 | 1940
      81887 | 8.3 | Ladri di biciclette | 1948 | 1940
      66622 | 8 | Notorious | 1946 | 1940
      350551 | 8.5 | Casablanca | 1942 | 1940
      59578 | 8 | The Big Sleep | 1946 | 1940
      78216 | 8.2 | Rebecca | 1940 | 1940
      ... (4 rows omitted)
```

Ignore the syntax for the moment. Instead, try to read that line like this:

Assign the name **forties** to a table whose rows are the rows in the **imdb** table **where** the **'Decade's are equal to 1940**.

Question 5.1 Compute the average rating of movies from the 1940s.

Hint: The function `np.average` computes the average of an array of numbers.

```
[24]: average_rating_in_forties = np.average(imdb.where('Decade', 1940).
      ↪column("Rating"))
      average_rating_in_forties
```

```
[24]: 8.257142857142856
```

```
[25]: check('tests/q5_1.py')
```

```
[25]: <gofer.ok.OKTestsResult at 0x7fc83b97d9d0>
```

Now let's dive into the details a bit more. **where** takes 2 arguments:

1. The name of a column. **where** finds rows where that column's values meet some criterion.
2. Something that describes the criterion that the column needs to meet, called a predicate.

To create our predicate, we called the function `are.equal_to` with the value we wanted, 1940. We'll see other predicates soon.

where returns a table that's a copy of the original table, but with only the rows that meet the given predicate.

Question 5.2 Create a table called `ninety_nine` containing the movies that came out in the year 1999. Use `where`.

```
[26]: ninety_nine = imdb.where("Year", 1999)
      ninety_nine
```

```
[26]: Votes    | Rating | Title           | Year | Decade
      1177098 | 8.8    | Fight Club      | 1999 | 1990
      735056 | 8.4    | American Beauty | 1999 | 1990
      630994 | 8.1    | The Sixth Sense | 1999 | 1990
      1073043 | 8.7    | The Matrix      | 1999 | 1990
      672878 | 8.5    | The Green Mile  | 1999 | 1990
```

```
[27]: check('tests/q5_2.py')
```

```
[27]: <gofer.ok.OKTestsResult at 0x7fc83b784c50>
```

So far we've only been finding where a column is *exactly* equal to a certain value. However, there are many other predicates. Here are a few:

Predicate	Example	Result
<code>are.equal_to</code>	<code>are.equal_to(50)</code>	Find rows with values equal to 50
<code>are.not_equal_to</code>	<code>are.not_equal_to(50)</code>	Find rows with values not equal to 50
<code>are.above</code>	<code>are.above(50)</code>	Find rows with values above (and not equal to) 50
<code>are.above_or_equal_to</code>	<code>are.above_or_equal_to(50)</code>	Find rows with values above 50 or equal to 50
<code>are.below</code>	<code>are.below(50)</code>	Find rows with values below 50
<code>are.between</code>	<code>are.between(2, 10)</code>	Find rows with values above or equal to 2 and below 10

The textbook section on selecting rows has more examples.

Question 5.3 Using `where` and one of the predicates from the table above, find all the movies with a rating higher than 8.5. Put their data in a table called `really_highly_rated`.

```
[28]: really_highly_rated = imdb.where("Rating", are.above(8.5))
      really_highly_rated
```

```
[28]: Votes    | Rating | Title           | Year | Decade
      635139 | 8.6    | Léon            | 1994 | 1990
      1027398 | 9.2    | The Godfather   | 1972 | 1970
      767224 | 8.6    | The Silence of the Lambs | 1991 | 1990
      1498733 | 9.2    | The Shawshank Redemption | 1994 | 1990
```

447875	8.9	Il buono, il brutto, il cattivo (1966)	1966	1960
967389	8.7	The Lord of the Rings: The Two Towers	2002	2000
689541	8.6	Interstellar	2014	2010
1473049	8.9	The Dark Knight	2008	2000
192206	8.6	C'era una volta il West	1968	1960
1271949	8.7	Inception	2010	2010

... (19 rows omitted)

```
[29]: check('tests/q5_3.py')
```

```
[29]: <gofer.ok.OKTestsResult at 0x7fc83b78c510>
```

Question 5.4 Find the average rating for movies released in the 20th century and the average rating for movies released in the 21st century for the movies in `imdb`.

Hint: Think of the steps you need to do (take the average, find the ratings, find movies released in 20th/21st centuries), and try to put them in an order that makes sense.

Note: Be careful of the common incorrect interpretation of a century. It is NOT 1/1/1900 through 12/31/1999. The correct mathematical definition of a century is: - The 20th century is January 1, 1901 through December 31, 2000. - The 21st century is January 1, 2001 through December 31, 2100.

```
[30]: average_20th_century_rating = np.average(imdb.where("Year", are.
        ↳ between_or_equal_to(1901, 2000)).column("Rating"))
average_21st_century_rating = np.average(imdb.where("Year", are.
        ↳ between_or_equal_to(2001, 2100)).column("Rating"))
print("Average 20th century rating:", average_20th_century_rating)
print("Average 21st century rating:", average_21st_century_rating)
```

```
Average 20th century rating: 8.280113636363636
Average 21st century rating: 8.23108108108108
```

```
[31]: check('tests/q5_4.py')
```

```
[31]: <gofer.ok.OKTestsResult at 0x7fc83b791d90>
```

The property `num_rows` tells you how many rows are in a table. (A “property” is just a method that doesn’t need to be called by adding parentheses.)

```
[32]: num_movies_in_dataset = imdb.num_rows
num_movies_in_dataset
```

```
[32]: 250
```

Question 5.5 Use `num_rows` (and arithmetic) to find the *proportion* of movies in the dataset that were released in the 20th century, and the proportion from the 21st century.

Hint: The *proportion* of movies released in the 20th century is the *number* of movies released in the 20th century, divided by the *total number* of movies.

Note: Be careful of the common incorrect interpretation of a century. It is NOT 1/1/1900 through 12/31/1999. The correct mathematical definition of a century is: - The 20th century is January 1, 1901 through December 31, 2000. - The 21st century is January 1, 2001 through December 31, 2100.

```
[33]: proportion_in_20th_century = imdb.where("Year", are.between_or_equal_to(1901,
    ↪2000)).num_rows / num_movies_in_dataset
    proportion_in_21st_century = imdb.where("Year", are.between_or_equal_to(2001,
    ↪2100)).num_rows / num_movies_in_dataset
    print("Proportion in 20th century:", proportion_in_20th_century)
    print("Proportion in 21st century:", proportion_in_21st_century)
```

Proportion in 20th century: 0.704

Proportion in 21st century: 0.296

```
[34]: check('tests/q5_5.py')
```

```
[34]: <gofer.ok.OKTestsResult at 0x7fc83b87b410>
```

Question 5.6 Here's a challenge: Find the number of movies that came out in *even* years.

Hint: The operator % computes the remainder when dividing by a number. So 5 % 2 is 1 and 6 % 2 is 0. A number is even if the remainder is 0 when you divide by 2.

Hint 2: % can be used on arrays, operating elementwise like + or *. So `make_array(5, 6, 7) % 2` is `array([1, 0, 1])`.

Hint 3: Create a column called "Year Remainder" that's the remainder when each movie's release year is divided by 2. Make a copy of `imdb` that includes that column. Then, use `where` to find rows where that new column is equal to 0. Then use `num_rows` to count the number of such rows.

```
[35]: imdb = imdb.with_column("Even", imdb.column("Year") % 2)
    num_even_year_movies = imdb.where("Even", 0).num_rows
    num_even_year_movies
```

```
[35]: 127
```

```
[36]: check('tests/q5_6.py')
```

```
[36]: <gofer.ok.OKTestsResult at 0x7fc83b78c090>
```

Question 5.7 Check out the `population` table from the introduction to this lab. Compute the year when the world population first went above 6 billion.

```
[37]: year_population_crossed_6_billion = population.where("Population", are.
    ↪above_or_equal_to(6000000000)).sort("Population", descending=False).
    ↪row(0)[-1]
    year_population_crossed_6_billion
```

```
[37]: 1999
```

```
[38]: check('tests/q5_7.py')
```

```
[38]: <gofer.ok.OKTestsResult at 0x7fc83b77ad50>
```

1.6 6. Miscellanea

There are a few more table methods you'll need to fill out your toolbox. The first 3 have to do with manipulating the columns in a table.

The table `farmers_markets.csv` contains data on farmers' markets in the United States (data collected by the USDA). Each row represents one such market.

Question 6.1 Load the dataset into a table. Call it `farmers_markets`.

```
[39]: farmers_markets = Table.read_table("farmers_markets.csv")
farmers_markets
```

```
[39]: FMID      | MarketName                                     | Website
      | Facebook                                     | Twitter
      | Youtube | OtherMedia
street                                     | city      | County
      | State      | zip      | Season1Date          | Season1Time
      | Season2Date | Season2Time | Season3Date | Season3Time | Season4Date |
Season4Time | x          | y          | Location
      | Credit | WIC | WICcash | SFMNP | SNAP | Organic | Bakedgoods | Cheese |
Crafts | Flowers | Eggs | Seafood | Herbs | Vegetables | Honey | Jams | Maple |
Meat | Nursery | Nuts | Plants | Poultry | Prepared | Soap | Trees | Wine |
Coffee | Beans | Fruits | Grains | Juices | Mushrooms | PetFood | Tofu |
WildHarvested | updateTime
1012063 | Caledonia Farmers Market Association - Danville |
https://sites.google.com/site/caledoniafarmersmarket/ |
https://www.facebook.com/Danville.VT.Farmers.Market/ | nan
      | nan      | nan
      | Danville | Caledonia | Vermont | 05828 | 06/08/2016
to 10/12/2016 | Wed: 9:00 AM-1:00 PM; | nan | nan
      | nan      | nan      | nan      | nan      | -72.1403 | 44.411 |
nan
      | Y      | N      | Y      | Y      | Y      | Y      | Y      | Y      | Y      | N
      | Y      | Y      | Y      | Y      | Y      | Y      | N      | N      | Y      | Y
      | Y      | Y      | Y      | N      | Y      | Y      | Y      | N      | Y      | N
      | Y      | N      | N      | 6/28/2016 12:10:09 PM
1011871 | Stearns Homestead Farmers' Market |
http://Stearnshomestead.com | nan
      | nan      | nan      | nan
      | 6975 Ridge Road | Parma |
Cuyahoga | Ohio | 44130 | 06/25/2016 to 10/01/2016 |
Sat: 9:00 AM-1:00 PM; | nan | nan | nan
      | nan      | nan      | nan      | -81.7286 | 41.3751 | nan
```

Y	Y	N	Y	Y	-	Y	N	N
Y	Y	N	Y	Y	Y	Y	Y	Y
N	N	Y	N	N	N	N	N	N
Y	N	N	N	Y	N	N		

4/9/2016 8:05:17 PM
1011878 | 100 Mile Market
<http://www.pfcmarkets.com>
<https://www.facebook.com/100MileMarket/?fref=ts> | nan
| nan | <https://www.instagram.com/100milemarket/> | 507
Harrison St | Kalamazoo | Kalamazoo
| Michigan | 49007 | 05/04/2016 to 10/12/2016 | Wed: 3:00 PM-7:00 PM;
| nan | nan | nan | nan
| nan | nan | -85.5749 | 42.296 | nan
| Y | Y | N | Y | Y | N | Y | Y | N
| Y | Y | N | Y | Y | Y | Y | Y | Y |
N | N | N | Y | Y | Y | N | Y | N | N
| Y | Y | N | N | N | N | N | N |

4/16/2016 12:37:56 PM
1009364 | 106 S. Main Street Farmers Market
<http://thetownofsixmile.wordpress.com/> | nan
| nan | nan | nan
| 106 S. Main Street | Six Mile | nan
| South Carolina | 29682 | nan | nan
| nan | nan | nan | nan | nan | nan
| -82.8187 | 34.8042 | nan
| Y | N | N | N | N | - | N | N | N
| N | N | N | N | N | N | N | N | N |
N | N | N | N | N | N | N | N | N |
| N | N | N | N | N | N | N | N | 2013
1010691 | 10th Steet Community Farmers Market | nan
| nan | nan
| nan | [http://agrimissouri.com/mo-grown/grodetail.php?type=mo-g ...](http://agrimissouri.com/mo-grown/grodetail.php?type=mo-g...) | 10th
Street and Poplar | Lamar | Barton
| Missouri | 64759 | 04/02/2014 to 11/30/2014 | Wed: 3:00 PM-6:00 PM;Sat: 8:00 AM-1:00 PM;
| nan | nan | nan | nan
| nan | nan | -94.2746 | 37.4956 | nan
| Y | N | N | N | N | - | Y | N | Y
| N | Y | N | Y | Y | Y | Y | N | Y |
N | N | Y | Y | Y | Y | N | N | N | N
| Y | N | N | N | N | N | N | N |

10/28/2014 9:49:46 AM
1002454 | 112st Madison Avenue | nan
| nan | nan
| nan | nan | 112th
Madison Avenue | New York | New York
| New York | 10029 | July to November | Tue:8:00 am - 5:00 pm;Sat:8:00 am - 8:00 pm;
| nan | nan | nan | nan

nan	nan	-73.9493	40.7939	Private business parking lot				
N	N	Y	Y	N	-	Y	N	Y
Y	N	N	Y	Y	Y	Y	N	N
N	Y	N	N	Y	Y	N	N	N
N	N	N	N	N	N	N		

3/1/2012 10:38:22 AM
1011100 | 12 South Farmers Market |
<http://www.12southfarmersmarket.com> | 12_South_Farmers_Market
| @12southfrmsmkt | nan | @12southfrmsmkt
| 3000 Granny White Pike | Nashville |
Davidson | Tennessee | 37204 | 05/05/2015 to 10/27/2015 |
Tue: 3:30 PM-6:30 PM; | nan | nan | nan
| nan | nan | nan | -86.7907 | 36.1184 | nan
| Y | N | N | N | Y | Y | Y | Y | N
| Y | Y | N | Y | Y | Y | Y | Y | Y |
N | N | N | Y | Y | Y | N | N | Y | N
| Y | N | Y | Y | Y | N | N |

5/1/2015 10:40:56 AM
1009845 | 125th Street Fresh Connect Farmers' Market |
<http://www.125thStreetFarmersMarket.com> |
<https://www.facebook.com/125thStreetFarmersMarket> |
<https://twitter.com/FarmMarket125th> | nan | Instagram-->
125thStreetFarmersMarket | 163 West 125th Street and Adam
Clayton Powell, Jr. Blvd. | New York | New York | New York
| 10027 | 06/10/2014 to 11/25/2014 | Tue: 10:00 AM-7:00 PM;
| nan | nan | nan | nan | nan | nan
| -73.9482 | 40.809 | Federal/State government building grounds
| Y | Y | N | Y | Y | Y | Y | Y | Y
| Y | Y | N | Y | Y | Y | Y | Y | Y |
N | Y | N | Y | Y | Y | N | Y | Y | N
| Y | N | Y | N | N | N | N |

4/7/2014 4:32:01 PM
1005586 | 12th & Brandywine Urban Farm Market | nan
| [https://www.facebook.com/pages/12th-Brandywine-Urban-Far ...](https://www.facebook.com/pages/12th-Brandywine-Urban-Far...) | nan
| nan | <https://www.facebook.com/delawareurbanfarmcoalition> | 12th
& Brandywine Streets | Wilmington | New Castle
| Delaware | 19801 | 05/16/2014 to 10/17/2014 | Fri: 8:00 AM-11:00
AM; | nan | nan | nan | nan
| nan | nan | -75.5345 | 39.7421 | On a farm from: a barn, a
greenhouse, a tent, a stand, etc | N | N | N | N | Y | N
| N | N | N | N | N | N | Y | Y |
N | N | N | N | N | N | N | N | N
| N | N | N | N | Y | N | N | N |
| N | N | 4/3/2014 3:43:31 PM
1008071 | 14&U Farmers' Market | nan
| <https://www.facebook.com/14UFarmersMarket> |
<https://twitter.com/14UFarmersMkt> | nan | nan


```
| 1400 U Street NW | Washington |
District of Columbia | District of Columbia | 20009 | 05/03/2014 to 11/22/2014 |
Sat: 9:00 AM-1:00 PM; | nan | nan | nan
| nan | nan | nan | -77.0321 | 38.917 | Other
| Y | Y | Y | Y | Y | Y | Y | Y | N
| Y | Y | N | Y | Y | Y | Y | N | Y |
N | Y | Y | Y | N | N | N | N | N | Y
| Y | Y | Y | N | N | N | N |
4/5/2014 1:49:04 PM
... (8536 rows omitted)
```

```
[40]: check('tests/q6_1.py')
```

```
[40]: <gofer.ok.OKTestsResult at 0x7fc83b77f310>
```

You'll notice that it has a large number of columns in it!

1.6.1 num_columns

Question 6.2 The table property `num_columns` (example call: `tbl.num_columns`) produces the number of columns in a table. Use it to find the number of columns in our farmers' markets dataset.

```
[41]: num_farmers_markets_columns = farmers_markets.num_columns
print("The table has", num_farmers_markets_columns, "columns in it!")
```

The table has 59 columns in it!

```
[42]: check('tests/q6_2.py')
```

```
[42]: <gofer.ok.OKTestsResult at 0x7fc83b784750>
```

Most of the columns are about particular products – whether the market sells tofu, pet food, etc. If we're not interested in that stuff, it just makes the table difficult to read. This comes up more than you might think.

1.6.2 select

In such situations, we can use the table method `select` to pare down the columns of a table. It takes any number of arguments. Each should be the name or index of a column in the table. It returns a new table with only those columns in it.

For example, the value of `imdb.select("Year", "Decade")` is a table with only the years and decades of each movie in `imdb`.

Question 6.3 Use `select` to create a table with only the name, city, state, latitude ('y'), and longitude ('x') of each market. Call that new table `farmers_markets_locations`.

```
[45]: farmers_markets_locations = farmers_markets.select("MarketName", "city", "
↪"State", "y", "x")
farmers_markets_locations
```

```
[45]: MarketName          | city          | State
      | y          | x
      Caledonia Farmers Market Association - Danville | Danville     | Vermont
      | 44.411    | -72.1403
      Stearns Homestead Farmers' Market              | Parma        | Ohio
      | 41.3751  | -81.7286
      100 Mile Market                                | Kalamazoo    | Michigan
      | 42.296   | -85.5749
      106 S. Main Street Farmers Market              | Six Mile     | South Carolina
      | 34.8042  | -82.8187
      10th Steet Community Farmers Market            | Lamar        | Missouri
      | 37.4956  | -94.2746
      112st Madison Avenue                          | New York     | New York
      | 40.7939  | -73.9493
      12 South Farmers Market                        | Nashville    | Tennessee
      | 36.1184  | -86.7907
      125th Street Fresh Connect Farmers' Market     | New York     | New York
      | 40.809   | -73.9482
      12th & Brandywine Urban Farm Market            | Wilmington   | Delaware
      | 39.7421  | -75.5345
      14&U Farmers' Market                          | Washington   | District of
      Columbia | 38.917    | -77.0321
      ... (8536 rows omitted)
```

```
[46]: check('tests/q6_3.py')
```

```
[46]: <gofer.ok.OKTestsResult at 0x7fc83b8d3e10>
```

1.6.3 select is not column!

The method `select` is **definitely not** the same as the method `column`.

`farmers_markets.column('y')` is an *array* of the latitudes of all the markets. `farmers_markets.select('y')` is a *table* that happens to contain only 1 column, the latitudes of all the markets.

Question 6.4 Below, we tried using the function `np.average` to find the average latitude ('y') and average longitude ('x') of the farmers' markets in the table, but we screwed something up. Run the cell to see the (somewhat inscrutable) error message that results from calling `np.average` on a table. Then, fix our code.

```
[48]: average_latitude = np.average(farmers_markets.column('y'))
      average_longitude = np.average(farmers_markets.column('x'))
      print("The average of US farmers' markets' coordinates is located at (",
            ↪average_latitude, ",", average_longitude, ")")
```

```
The average of US farmers' markets' coordinates is located at (
39.18646452349542 , -90.99258081292628 )
```

```
[49]: check('tests/q6_4.py')
```

```
[49]: <gofer.ok.OKTestsResult at 0x7fc88653fb50>
```

1.6.4 drop

`drop` serves the same purpose as `select`, but it takes away the columns you list instead of the ones you don't list, leaving all the rest of the columns.

Question 6.5 Suppose you just didn't want the "FMID" or "updateTime" columns in `farmers_markets`. Create a table that's a copy of `farmers_markets` but doesn't include those columns. Call that table `farmers_markets_without_fmids`.

```
[50]: farmers_markets_without_fmids = farmers_markets.drop("FMID", "updateTime")
farmers_markets_without_fmids
```

```
[50]: MarketName                                | Website
      | Facebook                                | Twitter
      | Youtube | OtherMedia                    |
street                                          | city      | County
      | State              | zip      | Season1Date          | Season1Time
      | Season2Date | Season2Time | Season3Date | Season3Time | Season4Date |
Season4Time | x          | y          | Location
      | Credit | WIC | WICcash | SFMNP | SNAP | Organic | Bakedgoods | Cheese |
Crafts | Flowers | Eggs | Seafood | Herbs | Vegetables | Honey | Jams | Maple |
Meat | Nursery | Nuts | Plants | Poultry | Prepared | Soap | Trees | Wine |
Coffee | Beans | Fruits | Grains | Juices | Mushrooms | PetFood | Tofu |
WildHarvested
      Caledonia Farmers Market Association - Danville |
https://sites.google.com/site/caledoniafarmersmarket/ |
https://www.facebook.com/Danville.VT.Farmers.Market/ | nan
      | nan      | nan
      | Danville | Caledonia          | Vermont          | 05828 | 06/08/2016
to 10/12/2016 | Wed: 9:00 AM-1:00 PM; | nan      | nan
      | nan      | nan      | nan      | nan      | -72.1403 | 44.411 |
nan
      | Y      | N      | Y      | Y      | Y      | Y      | Y      | Y      | N
      | Y      | Y      | Y      | Y      | Y      | Y      | N      | N      | Y      | Y
      | Y      | Y      | Y      | N      | Y      | Y      | Y      | N      | Y      | N
      | Y      | N      | N
      Stearns Homestead Farmers' Market          | http://StearnsHomestead.com
      | nan
      | nan      | nan
      | nan      | nan
Ridge Road                                  | Parma      | Cuyahoga
      | Ohio              | 44130 | 06/25/2016 to 10/01/2016 | Sat: 9:00 AM-1:00
PM; | nan      | nan      | nan      | nan
      | nan      | nan      | -81.7286 | 41.3751 | nan
      | Y      | Y      | N      | Y      | Y      | -      | Y      | N      | N
```

Y	Y	N	Y	Y	Y	Y	Y	Y	Y		
N	N	Y	N	N	N	N	N	N	N	N	
Y	N	N	N	Y	N	N					
100 Mile Market					http://www.pfcmarkets.com						
https://www.facebook.com/100MileMarket/?fref=ts					nan						
nan					https://www.instagram.com/100milemarket/					507	
Harrison St					Kalamazoo			Kalamazoo			
Michigan					49007			05/04/2016 to 10/12/2016			
PM;					nan			nan			
nan					-85.5749			42.296			
Y	Y	N	Y	Y	N	Y		Y	N		
Y	Y	N	Y	Y		Y	Y	Y	Y		
N	N	N	Y	Y	Y	N	Y	N	N	N	
Y	Y	N	N	N	N	N	N				
106 S. Main Street Farmers Market											
http://thetownofsixmile.wordpress.com/					nan						
nan					nan			nan			
106 S. Main Street								Six Mile			
South Carolina					29682			nan			
nan					nan			nan			
-82.8187					34.8042			nan			
Y	N	N	N	N	-	N		N	N		
N	N	N	N	N		N	N	N	N		
N	N	N	N	N	N	N	N	N	N	N	
N	N	N	N	N	N	N	N				
10th Steet Community Farmers Market					nan						
nan					nan						
nan					http://agrimissouri.com/mo-grown/grodetail.php?type=mo-g ...						
Street and Poplar					Lamar			Barton			
Missouri					64759			04/02/2014 to 11/30/2014			
PM;Sat: 8:00 AM-1:00 PM;					nan			nan			
nan					-94.2746			37.4956			
Y	N	N	N	N	-	Y		N	Y		
N	Y	N	Y	Y		Y	Y	N	Y		
N	N	Y	Y	Y	Y	N	N	N	N	N	
Y	N	N	N	N	N	N	N				
112st Madison Avenue					nan						
nan					nan						
nan					112th						
Madison Avenue					New York			New York			
New York					10029			July to November			
pm;Sat:8:00 am - 8:00 pm;					nan			nan			
nan					-73.9493			40.7939			
Private business parking lot											
N	N	Y	Y	N	-	Y		N	Y		
Y	N	N	Y	Y		Y	Y	N	N		
N	Y	N	N	Y	Y	N	N	N	N	N	
N	N	N	N	N	N	N	N				

12 South Farmers Market |
<http://www.12southfarmersmarket.com> | 12_South_Farmers_Market
 | @12southfrmsmkt | nan | @12southfrmsmkt
 | 3000 Granny White Pike | Nashville |
 Davidson | Tennessee | 37204 | 05/05/2015 to 10/27/2015 |
 Tue: 3:30 PM-6:30 PM; | nan | nan | nan
 | nan | nan | nan | -86.7907 | 36.1184 | nan
 | Y | N | N | N | Y | Y | Y | Y | N
 | Y | Y | N | Y | Y | Y | Y | Y | Y |
 N | N | N | Y | Y | Y | N | N | Y | N
 | Y | N | Y | Y | Y | N | N
 125th Street Fresh Connect Farmers' Market |
<http://www.125thStreetFarmersMarket.com> |
<https://www.facebook.com/125thStreetFarmersMarket> |
<https://twitter.com/FarmMarket125th> | nan | Instagram-->
 125thStreetFarmersMarket | 163 West 125th Street and Adam
 Clayton Powell, Jr. Blvd. | New York | New York | New York
 | 10027 | 06/10/2014 to 11/25/2014 | Tue: 10:00 AM-7:00 PM;
 | nan | nan | nan | nan | nan | nan
 | -73.9482 | 40.809 | Federal/State government building grounds
 | Y | Y | N | Y | Y | Y | Y | Y | Y
 | Y | Y | N | Y | Y | Y | Y | Y | Y |
 N | Y | N | Y | Y | Y | N | Y | Y | N
 | Y | N | Y | N | N | N | N
 12th & Brandywine Urban Farm Market | nan
 | [https://www.facebook.com/pages/12th-Brandywine-Urban-Far ...](https://www.facebook.com/pages/12th-Brandywine-Urban-Far...) | nan
 | nan | <https://www.facebook.com/delawareurbanfarmcoalition> | 12th
 & Brandywine Streets | Wilmington | New Castle
 | Delaware | 19801 | 05/16/2014 to 10/17/2014 | Fri: 8:00 AM-11:00
 AM; | nan | nan | nan | nan
 | nan | nan | -75.5345 | 39.7421 | On a farm from: a barn, a
 greenhouse, a tent, a stand, etc | N | N | N | N | Y | N
 | N | N | N | N | N | N | Y | Y |
 N | N | N | N | N | N | N | N | N | N
 | N | N | N | N | Y | N | N | N | N
 | N | N
 14&U Farmers' Market | nan
 | <https://www.facebook.com/14UFarmersMarket> |
<https://twitter.com/14UFarmersMkt> | nan | nan
 | 1400 U Street NW | Washington |
 District of Columbia | District of Columbia | 20009 | 05/03/2014 to 11/22/2014 |
 Sat: 9:00 AM-1:00 PM; | nan | nan | nan
 | nan | nan | nan | -77.0321 | 38.917 | Other
 | Y | Y | Y | Y | Y | Y | Y | Y | N
 | Y | Y | N | Y | Y | Y | Y | N | Y |
 N | Y | Y | Y | N | N | N | N | N | Y
 | Y | Y | Y | N | N | N | N

... (8536 rows omitted)

```
[51]: check('tests/q6_5.py')
```

```
[51]: <gofer.ok.OKTestsResult at 0x7fc83a95d350>
```

1.6.5 take

Let's find the 5 northernmost farmers' markets in the US. You already know how to sort by latitude ('y'), but we haven't seen how to get the first 5 rows of a table. That's what `take` is for.

The table method `take` takes as its argument an array of numbers. Each number should be the index of a row in the table. It returns a new table with only those rows.

Most often you'll want to use `take` in conjunction with `np.arange` to take the first few rows of a table.

Question 6.6 Make a table of the 5 northernmost farmers' markets in `farmers_markets_locations`. Call it `northern_markets`. (It should include the same columns as `farmers_markets_locations`.)

```
[55]: northern_markets = farmers_markets_locations.sort("y", descending=True).take(np.
      ↪arange(5))
      northern_markets
```

```
[55]: MarketName      | city      | State | y      | x
      Tanana Valley Farmers Market | Fairbanks | Alaska | 64.8628 | -147.781
      Ester Community Market      | Ester     | Alaska | 64.8459 | -148.01
      Fairbanks Downtown Market    | Fairbanks | Alaska | 64.8444 | -147.72
      Nenana Open Air Market       | Nenana    | Alaska | 64.5566 | -149.096
      Highway's End Farmers' Market | Delta Junction | Alaska | 64.0385 | -145.733
```

```
[56]: check('tests/q6_6.py')
```

```
[56]: <gofer.ok.OKTestsResult at 0x7fc837ce4f10>
```

Question 6.7 Make a table of the farmers' markets in Berkeley, California. (It should include the same columns as `farmers_markets_locations`.)

```
[58]: berkeley_markets = farmers_markets_locations.where("State", "California").
      ↪where("city", "Berkeley")
      berkeley_markets
```

```
[58]: MarketName      | city      | State      | y      | x
      Downtown Berkeley Farmers' Market | Berkeley | California | 37.8697 | -122.273
      North Berkeley Farmers' Market    | Berkeley | California | 37.8802 | -122.269
      South Berkeley Farmers' Market     | Berkeley | California | 37.8478 | -122.272
```

```
[59]: check('tests/q6_7.py')
```

[59]: <gofer.ok.OKTestsResult at 0x7fc837c86990>

1.7 7. Summary

For your reference, here's a table of all the functions and methods we saw in this lab.

Name	Example	Purpose
Table	Table()	Create an empty table, usually to extend with data
Table.read_table	Table.read_table("my_data.csv")	Create a table from a data file
with_columns	tbl = Table().with_columns("N", np.arange(5), "2*N", np.arange(0, 10, 2))	Create a copy of a table with more columns
column	tbl.column("N")	Create an array containing the elements of a column
sort	tbl.sort("N")	Create a copy of a table sorted by the values in a column
where	tbl.where("N", are.above(2))	Create a copy of a table with only the rows that match some <i>predicate</i>
num_rows	tbl.num_rows	Compute the number of rows in a table
num_columns	tbl.num_columns	Compute the number of columns in a table
select	tbl.select("N")	Create a copy of a table with only some of the columns
drop	tbl.drop("2*N")	Create a copy of a table without some of the columns
take	tbl.take(np.arange(0, 6, 2))	Create a copy of the table with only the rows whose indices are in the given array

1.8 8. Submission

Once you're finished, select "Save and Checkpoint" in the File menu. Your name and course section number should be in the first and last cell of the assignment. Be sure you have run all cells with code and that the output from that is showing.

Double check that you have completed all of the free response questions as the auto-grader does NOT check that and YOU are responsible for knowing those questions are there and completing them as part of the grade for this lab. When ready, click "Print Preview" in the File menu. Print a copy from there in pdf format. (This means you right click and choose print and choose "save as pdf" from your printer options.) You will need to submit the pdf in Canvas by the deadline.

The gopher grader output and/or output from your coding are essential to helping your instructor grade your work correctly and in a timely manner.

Files submitted that are missing the required output will lose some to all points so double check your pdf before submitting.

```
[60]: # For your convenience, you can run this cell to run all the tests at once!
import glob
from gofer.ok import grade_notebook
if not globals().get('__GOFER_GRADER__', False):
    display(grade_notebook('lab03.ipynb', sorted(glob.glob('tests/q*.py'))))
```

Population column: [2557628654 2594939877 2636772306 2682053389 2730228104
2782098943

2835299673 2891349717 2948137248 3000716593 3043001508 3083966929
3140093217 3209827882 3281201306 3350425793 3420677923 3490333715
3562313822 3637159050 3712697742 3790326948 3866568653 3942096442
4016608813 4089083233 4160185010 4232084578 4304105753 4379013942
4451362735 4534410125 4614566561 4695736743 4774569391 4856462699
4940571232 5027200492 5114557167 5201440110 5288955934 5371585922
5456136278 5538268316 5618682132 5699202985 5779440593 5857972543
5935213248 6012074922 6088571383 6165219247 6242016348 6318590956
6395699509 6473044732 6551263534 6629913759 6709049780 6788214394
6866332358 6944055583 7022349283 7101027895 7178722893 7256490011]

Years column: [1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962
1963

1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977
1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991
1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005
2006 2007 2008 2009 2010 2011 2012 2013 2014 2015]

Average 20th century rating: 8.280113636363636

Average 21st century rating: 8.23108108108108

Proportion in 20th century: 0.704

Proportion in 21st century: 0.296

The table has 59 columns in it!

The average of US farmers' markets' coordinates is located at (
39.18646452349542 , -90.99258081292628)

['tests/q2_1.py', 'tests/q2_2.py', 'tests/q3_1.py', 'tests/q3_2.py',
'tests/q4_1.py', 'tests/q4_2.py', 'tests/q4_3.py', 'tests/q5_1.py',
'tests/q5_2.py', 'tests/q5_3.py', 'tests/q5_4.py', 'tests/q5_5.py',
'tests/q5_6.py', 'tests/q5_7.py', 'tests/q6_1.py', 'tests/q6_2.py',
'tests/q6_3.py', 'tests/q6_4.py', 'tests/q6_5.py', 'tests/q6_6.py',
'tests/q6_7.py']

Question 1:

<gofer.ok.OKTestsResult at 0x7fc835edefd0>

Question 2:

<gofer.ok.OKTestsResult at 0x7fc835e95e50>

Question 3:

<gofer.ok.OKTestsResult at 0x7fc835e95d90>

Question 4:

<gofer.ok.OKTestsResult at 0x7fc834f27d50>

Question 5:

<gofer.ok.OKTestsResult at 0x7fc835e97910>

Question 6:

<gofer.ok.OKTestsResult at 0x7fc835e95b50>

Question 7:

<gofer.ok.OKTestsResult at 0x7fc835e95a90>

Question 8:

<gofer.ok.OKTestsResult at 0x7fc835e95890>

Question 9:

<gofer.ok.OKTestsResult at 0x7fc835e95590>

Question 10:

<gofer.ok.OKTestsResult at 0x7fc8364b9810>

Question 11:

<gofer.ok.OKTestsResult at 0x7fc83613f190>

Question 12:

<gofer.ok.OKTestsResult at 0x7fc83613f150>

Question 13:

<gofer.ok.OKTestsResult at 0x7fc83613f110>

Question 14:

<gofer.ok.OKTestsResult at 0x7fc836111790>

Question 15:

<gofer.ok.OKTestsResult at 0x7fc835058a90>

Question 16:

<gofer.ok.OKTestsResult at 0x7fc835058d50>

Question 17:

<gofer.ok.OKTestsResult at 0x7fc835058ad0>

Question 18:

<gofer.ok.OKTestsResult at 0x7fc836121f90>

Question 19:

<gofer.ok.OKTestsResult at 0x7fc835058d90>

Question 20:

<gofer.ok.OKTestsResult at 0x7fc835058610>

Question 21:

<gofer.ok.OKTestsResult at 0x7fc8365eaf50>

0.9047619047619048

Name: Allan Gongora

Section: 0131

[]: