Lab 3: Tables

Welcome to lab 3! This week, we'll learn about *tables*, which let us work with multiple arrays of data about the same things. Tables are covered in chapter 6 of the text.

1. Introduction

Last week we had our first look at *datasets* -- organized collections of many pieces of information. Specifically, we looked at arrays, which hold many pieces of the same kind of data. An array is like a single column in an Excel spreadsheet.

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 an array with the world population in each year (as estimated by the US Census Bureau), and an array of the years themselves (which go from 1950 to 2015). The cell also sets up the lab, so run it now.

In [52]:

```
import numpy as np
from datascience import *

population_amounts = Table.read_table("~/DS_113_S23/Labs/Lab_3/world_populat
years = np.arange(1950, 2015+1)
print("Population:", population_amounts)
print("Years:", years)
```

```
Population: [2557628654 2594939877 2636772306 2682053389 2730228104 2782098
943
 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: [1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 19
63 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?

Just finding the element of `population_amounts` that first goes above 6 billion wouldn't be enough -- we'd have to figure out the year that corresponds to.

Instead, let's put the data in a table.

In [53]:

```
population = Table().with_columns(
    "Population", population_amounts,
    "Year", years
)
population
```

Out[53]:

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)

Before the end of this lab, we'll come back to this table, and you'll have to figure out how to find the answer to our question.

A note for the skeptical

You might protest that it's fairly easy to find the answer to this particular question by just looking through the data and counting.

That's a fair point! Sometimes it's faster to do something without the help of a computer. Questions like this make convenient introductory exercises precisely because it's easy to see how the computer got it.

However, we're building up a toolset that will let us answer questions we couldn't possibly address manually. Learn the toolset, and it will serve you well later.

2. Creating Tables

To make the table `population`, we:

1. Made an empty table by calling the function `Table`.

2. Created a new table that extended the empty table with two columns named "Population" and "Year". We did this by calling the method `with_columns`.

Question 2.1. In the cell below, we've created 2 arrays. `top_10_movie_ratings` contains the IMDb ratings of 8 movies. `top_10_movie_names` contains their names, in the same order. Create an empty table called `empty_table`. Then make a table of the movies by extending that empty table. Call that table `top_10_movies`, and call the columns "Rating" and "Name", respectively.

In [54]:

```
top_10_movie_ratings = make_array(9.2, 9.2, 9., 8.9, 8.9, 8.9, 8.9, 8.9, 8.9
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)')

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

Out[54]:

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)

Loading a table from a file

Usually, you'll want to work with more data than you can comfortably type by hand. Instead, you'll have your data in a file and make a table out of it.

The function `Table.read_table` does this. It takes one argument, a path to a data file (a string). There are many formats for data files, but CSV ("comma-separated values") is the most common. It returns a table.

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`.

In [72]:

```
imdb = Table.read_table("~/DS_113_S23/Labs/Lab_3/imdb.csv")
imdb
```

Out[72]:

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

... (240 rows omitted)

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 in the shared drive. 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.

Question 2.3. Create your own CSV file called `my_data.csv` inside your current folder, then load it into a table called `my_data`.

You can create the file by going to the folder containing this lab and clicking the "New -> Text File" button.

The `my_data` table must have **two columns** and **three rows**. It can have whatever values you want.

In [56]:

```
# Load your table here.
my_data = Table.read_table("~/Labs/my_data.csv")
my_data
```

Out[56]:

Name	Birthday
Yerim	20020323
Eonbi	20020913
Suzy	20021031
Sky	20020228

3. 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:

In [57]:
```
imdb.column("Rating")
```

Out[57]:
```
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. ,  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 3.1. 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.

In [58]:
```
highest_rating = imdb.sort("Rating", descending=True).column("Rating").item(
highest_rating
```

Out[58]:
```
9.2
```

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 table by rating.

In [59]:

```
imdb.sort("Rating")
```

Out[59]:

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

... (240 rows omitted)

Well, that actually doesn't help much, either -- now we know the lowest-rated movies. To look at the highest-rated movies, sort in reverse order:

In [60]:

```
imdb.sort("Rating", descending=True)
```

Out[60]:

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

... (240 rows omitted)

(The `descending=True` bit is called an *optional argument*. If it's confusing, try not to worry about it for now.)

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 *sorted copy of* `imdb`; the `imdb` table doesn't get modified. Since `imdb.sort("Rating")` has a value, you can give a name to that value.
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 3.2. Create a version of `imdb` that's sorted chronologically, with the earliest movies first. Call it `imdb_by_year`.

In [61]:
```
imdb_by_year = imdb.sort("Year")
imdb_by_year
```

Out[61]:

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

... (240 rows omitted)

Question 3.3. 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, then use `item` to get its first item.

In [62]:
```
earliest_movie_title = imdb_by_year.column("Title").item(0)
earliest_movie_title
```

Out[62]:
```
'The Kid (1921)'
```

4. 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`.

In [63]:

```
forties = imdb.where('Decade', are.equal_to(1940))
forties
```

Out[63]:

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

... (4 rows omitted)

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

> Find the rows in the **`imdb`** table **`where`** the **`'Decade'`** s **`are`** **`equal`** **`to`** **`1940`** . Make a table of those rows and name it **`forties`** .

Question 4.1. Compute the average rating of movies from the 1940s.

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

In [64]:

```
average_rating_in_forties = np.average(forties.column("Rating"))
average_rating_in_forties
```

Out[64]:

```
8.2571428571428562
```

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. An object that tells `where` about the criterion those objects should meet. The object is produced by calling the function `are.equal_to` in this case. Technically, this criterion object is called a *predicate*, so that's the word we'll use. You could call it a "criterion" or a "requirement" or whatever you're most comfortable calling it.

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

In [65]:

```python
ninety_nine = imdb.where('Year', are.equal_to(1999))
ninety_nine
```

Out[65]:

Votes	Rating	Title	Year	Decade
1177098	8.8	Fight Club (1999)	1999	1990
735056	8.4	American Beauty (1999)	1999	1990
630994	8.1	The Sixth Sense (1999)	1999	1990
1073043	8.7	The Matrix (1999)	1999	1990
672878	8.5	The Green Mile (1999)	1999	1990

So far we've only done exact matching -- the year is exactly 1999, or the decade is exactly 1940. Often you'll want to do something more flexible. For example, we might want to find all the movies with more than 1 million votes on IMDb. For that, we use a different predicate.

In [66]:

```python
lots_of_votes = imdb.where('Votes', are.above(1000000))
lots_of_votes
```

Out[66]:

Votes	Rating	Title	Year	Decade
1027398	9.2	The Godfather (1972)	1972	1970
1498733	9.2	The Shawshank Redemption (1994)	1994	1990
1473049	8.9	The Dark Knight (2008)	2008	2000
1271949	8.7	Inception (2010)	2010	2010
1177098	8.8	Fight Club (1999)	1999	1990
1073043	8.7	The Matrix (1999)	1999	1990
1074146	8.9	The Lord of the Rings: The Return of the King (2003)	2003	2000
1099087	8.8	The Lord of the Rings: The Fellowship of the Ring (2001)	2001	2000
1166532	8.9	Pulp Fiction (1994)	1994	1990
1078416	8.7	Forrest Gump (1994)	1994	1990

Question 4.3. Find all the movies with a rating higher than 8.5. Put their data in a table called `really_highly_rated`.

In [67]:

```python
really_highly_rated = imdb.where('Rating', are.above(8.5))
really_highly_rated
```

Out[67]:

Votes	Rating	Title	Year	Decade
635139	8.6	Léon (1994)	1994	1990
1027398	9.2	The Godfather (1972)	1972	1970
767224	8.6	The Silence of the Lambs (1991)	1991	1990
1498733	9.2	The Shawshank Redemption (1994)	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)	2002	2000
689541	8.6	Interstellar (2014)	2014	2010
1473049	8.9	The Dark Knight (2008)	2008	2000
192206	8.6	C'era una volta il West (1968)	1968	1960
1271949	8.7	Inception (2010)	2010	2010

... (19 rows omitted)

There are many other predicates. Here are a few:

|Predicate|Example|Result| |-|-|-| | `are.equal_to` | `are.equal_to(50)` |Find rows with values equal to 50| | `are.not_equal_to` | `are.not_equal_to(50)` |Find rows with values not equal to 50| | `are.above` | `are.above(50)` |Find rows with values above (and not equal to) 50|
| `are.above_or_equal_to` | `are.above_or_equal_to(50)` |Find rows with values above 50 or equal to 50| | `are.below` | `are.below(50)` |Find rows with values below 50| | `are.between` | `are.between(2, 10)` |Find rows with values above or equal to 2 and below 10|

The textbook section on selecting rows has more examples.

Question 4.4. Find the average rating for movies released in the 20th century and the average rating for movies released in the 21st century.

Note: Our `imdb` dataset includes 250 of the best-rated movies ever made. There are millions of movies, and most are not represented in this dataset. So whatever you find will be true only among these, not among movies in general.

In [68]:

```
average_20th_century_rating = np.average(imdb.where('Year', are.below(2001))
average_21st_century_rating = np.average(imdb.where('Year', are.above_or_equ
print("Average 20th century rating:", average_20th_century_rating)
print("Average 21st century rating:", average_21st_century_rating)
```

```
Average 20th century rating: 8.28011363636
Average 21st century rating: 8.23108108108
```

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.)

In [69]:

```
num_movies_in_dataset = imdb.num_rows
num_movies_in_dataset
```

Out[69]:

```
250
```

Question 4.5. 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.

In [70]:

```
proportion_in_20th_century = (imdb.where('Year', are.below(2001)).num_rows)/
proportion_in_21st_century = (imdb.where('Year', are.above_or_equal_to(2001)
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
```

Question 4.6. 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.

In [81]:

```
# Our solution used 3 steps that we put on 3 separate lines.
# You can approach this however you like.
imdb_copy = imdb.with_columns("Year Remainder",(imdb.column("Year") % 2))
num_even_year_movies = imdb_copy.where('Year Remainder', are.equal_to(0)).nu
num_even_year_movies
```

Out[81]:

```
127
```

Question 4.7. Check out the `population` table from the introduction to this lab. Compute the year when the world population first went above 6 billion.

In [86]:

```
year_population_crossed_6_billion = population.where('Population', are.above
year_population_crossed_6_billion
```

Out[86]:

```
1999
```

5. 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. (The data are collected by the USDA.) Each row represents one such market.

Question 5.1. Load the dataset into a table. Call it `farmers_markets`.

In [87]:

```
farmers_markets = Table.read_table("~/DS_113_S23/Labs/Lab_3/farmers_markets.
farmers_markets
```

Out[87]:

FMID	MarketName	Website	
1012063	Caledonia Farmers Market Association - Danville	https://sites.google.com/site/caledoniafarmersmarket/	https://www.facebook
1011871	Stearns Homestead Farmers' Market	http://Stearnshomestead.com	
1011878	100 Mile Market	http://www.pfcmarkets.com	https://www.facel
1009364	106 S. Main Street Farmers Market	http://thetownofsixmile.wordpress.com/	
1010691	10th Steet Community Farmers Market	nan	
1002454	112st Madison Avenue	nan	
1011100	12 South Farmers Market	http://www.12southfarmersmarket.com	
1009845	125th Street Fresh Connect Farmers' Market	http://www.125thStreetFarmersMarket.com	https://www.faceboo
1005586	12th & Brandywine Urban Farm Market	nan	https://www.facebo
1008071	14&U Farmers' Market	nan	https://www.f

... (8536 rows omitted)

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

`num_columns`

Question 5.2. 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.

In [88]:

```
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!
```

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.

`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 5.3. 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`.

In [89]:

```
farmers_markets_locations = farmers_markets.select("MarketName", "city","Sta
farmers_markets_locations
```

Out[89]:

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

... (8536 rows omitted)

`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 5.4. 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.

In [91]:
```
average_latitude = np.average(column.column('y'))
average_longitude = np.average(farmers_markets.column('x'))
print("The average of US farmers' markets' coordinates is located at (", ave
```

```
The average of US farmers' markets' coordinates is located at ( 39.18646452
35 , -90.9925808129 )
```

`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 5.5. 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_fmid`.

In [93]:
```
farmers_markets_without_fmid = farmers_markets.drop("FMID", "updateTime")
farmers_markets_without_fmid
```

Out[93]:

MarketName	Website	
Caledonia Farmers Market Association - Danville	https://sites.google.com/site/caledoniafarmersmarket/	https://www.facebook.com/Danvi
Stearns Homestead Farmers' Market	http://Stearnshomestead.com	
100 Mile Market	http://www.pfcmarkets.com	https://www.facebook.com/1
106 S. Main Street Farmers Market	http://thetownofsixmile.wordpress.com/	
10th Steet Community Farmers Market	nan	
112st Madison Avenue	nan	
12 South Farmers Market	http://www.12southfarmersmarket.com	12_S
125th Street Fresh Connect Farmers' Market	http://www.125thStreetFarmersMarket.com	https://www.facebook.com/125t
12th & Brandywine Urban Farm Market	nan	https://www.facebook.com/pa
14&U Farmers' Market	nan	https://www.facebook.co

... (8536 rows omitted)

`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 5.6. 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`.

In [99]:

```
northern_markets = farmers_markets_locations.sort("y",descending=True).take(
northern_markets
```

Out[99]:

MarketName	city	State	x	y
Zona Rosa Farmers' Market	Kansas City	Missouri	-94.5809	64.8628
Zionsville Farmers Market	Zionsville	Indiana	-86.2612	64.8459
Zion Canyon Farmers Market	Springdale	Utah	-113.003	64.8444
Zimmerman Farmers Market	Zimmerman	Minnesota	-93.585	64.5566
Zia Bernalillo Farmers' Market	Bernalillo	New Mexico	-106.547	64.0385

Question 5.7. Make a table of the farmers' markets in Northampton, Massachusetts. (It should include the same columns as `farmers_markets_locations`.)

In [100…

```
northampton_markets = farmers_markets_locations.where("city", are.equal_to('
northampton_markets
```

Out[100]:

MarketName	city	State	x	y
Northampton Gothic Street Farmers Market	Northampton	Massachusetts	-72.6322	41.4436
Northampton Thornes Marketplace Farmers Market	Northampton	Massachusetts	-72.6306	41.4456
Northampton Tuesday Farmers' Market	Northampton	Massachusetts	-72.6302	41.4478
Northampton Winter Farmers' Market	Northampton	Massachusetts	-72.6557	41.4483

Recognize any of them?

6. Congratulations, you're done!

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 *predicate*|	`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|