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 5 (http://www.inferentialthinking.com/chapters/05/tables.html) 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 (http://www.census.gov/population/international/data/worldpop/table_population.php) 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 [1]:
```
import numpy as np
from datascience import *

# These lines load the tests.
from client.api.assignment import load_assignment
tests = load_assignment('lab03.ok')

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

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 [2]:
```
population = Table().with_columns(
    "Population", population_amounts,
    "Year", years
)
population
```

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 (imdb.com) 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 [2]:
```
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 = ...
top_10_movies = ...
# We've put this next line here so your table will get printed out when you
# run this cell.
top_10_movies
```

In [3]:
```
_ = tests.grade('q2_1')
```

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 [3]:
```
imdb = ...
imdb
```

In [ ]:
```
_ = tests.grade('q2_2')
```

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 (http://edoceo.com/utilitas/csv-file-format).

Question 2.3. Create your own CSV file called `my_data.csv` inside this lab's 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 [7]:
```
# Load your table here.
my_data = ...
my_data
```

In [43]:
```
_ = tests.grade('q2_3')
```

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 [8]:
```
imdb.column("Rating")
```

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 [9]:
```
highest_rating = ...
highest_rating
```

In [ ]:
```
_ = tests.grade('q3_1')
```

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 [10]:
```
imdb.sort("Rating")
```

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

(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 [ ]:
```
imdb_by_year = ...
imdb_by_year
```

In [ ]:
```
_ = tests.grade('q3_2')
```

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 [ ]:
```
earliest_movie_title = ...
earliest_movie_title
```

In [ ]:
```
_ = tests.grade('q3_3')
```

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

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 [ ]:
```
average_rating_in_forties = ...
average_rating_in_forties
```

In [ ]:
```
_ = tests.grade('q4_1')
```

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 [ ]:
```
ninety_nine = ...
ninety_nine
```

In [ ]:
```
_ = tests.grade('q4_2')
```

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 [ ]:
```
lots_of_votes = imdb.where('Votes', are.above(1000000))
lots_of_votes
```

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 [ ]:
```
really_highly_rated = ...
really_highly_rated
```

In [ ]:
```
_ = tests.grade('q4_3')
```

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 [ ]:
```
average_20th_century_rating = ...
average_21st_century_rating = ...
print("Average 20th century rating:", average_20th_century_rating)
print("Average 21st century rating:", average_21st_century_rating)
```

In [41]:
```
_ = tests.grade('q4_4')
```

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 [44]:
```
num_movies_in_dataset = imdb.num_rows
num_movies_in_dataset
```

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 [ ]:
```
proportion_in_20th_century = ...
proportion_in_21st_century = ...
print("Proportion in 20th century:", proportion_in_20th_century)
print("Proportion in 21st century:", proportion_in_21st_century)
```

In [45]:
```
_ = tests.grade('q4_5')
```

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 [ ]:
```
# Our solution used 3 steps that we put on 3 separate lines.
# You can approach this however you like.
num_even_year_movies = ...
num_even_year_movies
```

In [4]:
```
_ = tests.grade('q4_6')
```

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 [ ]:
```
year_population_crossed_6_billion = ...
year_population_crossed_6_billion
```

In [ ]:
```
_ = tests.grade('q4_7')
```

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 [ ]:
```
farmers_markets = ...
farmers_markets
```

In [20]:
```
_ = tests.grade('q5_1')
```

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 [42]:
```
num_farmers_markets_columns = ...
print("The table has", num_farmers_markets_columns, "columns in it!")
```

In [44]:
```
_ = tests.grade('q5_2')
```

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 [34]:
```
farmers_markets_locations = ...
farmers_markets_locations
```

In [ ]:
```
_ = tests.grade('q5_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 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 [40]:
```
average_latitude = np.average(farmers_markets.select('y'))
average_longitude = np.average(farmers_markets.select('x'))
print("The average of US farmers' markets' coordinates is located at (", ave
```

In [25]:
```
_ = tests.grade('q5_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 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 [30]:
```
farmers_markets_without_fmid = ...
farmers_markets_without_fmid
```

In [28]:
```
_ = tests.grade('q5_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 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 [ ]:
```
northern_markets = ...
northern_markets
```

In [37]:
```
_ = tests.grade('q5_6')
```

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

In [ ]:
```
berkeley_markets = ...
berkeley_markets
```

In [39]:
```
_ = tests.grade('q5_7')
```

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

Check that you've passed the tests, and remember to **save** and run the last cell to submit your work.

In [ ]:
```
# For your convenience, you can run this cell to run all the tests at once!
import os
_ = [tests.grade(q[:-3]) for q in os.listdir("tests") if q.startswith('q')]
```

In [7]:
```
# Run this cell to submit your work *after* you have passed all of the test
# It's ok to run this cell multiple times. Only your final submission will b

!TZ=America/Los_Angeles ipython nbconvert --output=".lab03_$(date +%m%d_%H%M
```