CIS 545: Big Data Analytics

Spring 2018

Homework 1: Data Wrangling

Due February 7, 2018 by 10pm

Big data analytics often requires (1) importing data from multiple sources, possibly *extracting* content from text; then (2) combining data from multiple sources, possibly from multiple different organizations, in heterogeneous formats. For this assignment, our primary goals are to get you comfortable with importing, extracting, saving, and combining data -- largely using Pandas DataFrames as the basic abstraction. You'll read data from files, address issues with missing values, save the data in a database, combine the data, and generate visualizations.

The Task. Most of you likely were on a plane over the break, and chances are that at least one of your flights got delayed. Did you ever wonder how well different airlines do? We'll answer those questions in this assignment! *(Caveat: with apologies to international travelers, most of the data is only available for US cities and routes!)*

Terminology. We'll generally use **field**, **column**, and **attribute** interchangeably to mean a named column in a DataFrame. We'll also generally assume that **table**, **DataFrame**, and **relation** mean the same thing.

Submission of Your Homework. Details are provided in Section 7 below for how you'll do submission to the Web site. We expect you to implement your solution to the Homework in a series of Jupyter files with specified names. In each Jupyter notebook, you'll include **both the code and the output** as produced by Jupyter. For each step we'll ask you to provide a label so we know what Step you are doing, and then we'll ask you to output content in a DataFrame, a visualization, or both. (Directions will appear in a framed box such as this one.)

Please note that components of your submissions will be automatically checked, so please follow directions about field names, etc. very carefully. When you are asked to create a label and a series of cells, please create them consecutively and in the same order!

Step 0: Getting Set Up

We'll be bringing together data from OpenFlights.org with data from the US government Bureau of Transportation at http://www.transtats.bts.gov. To start, open your Terminal or Command Prompt and **cd** to **Jupyter**. Run:

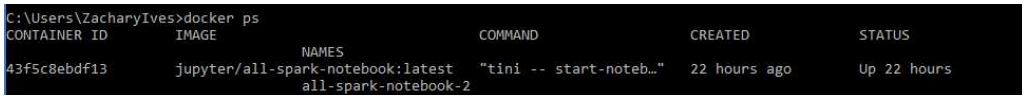
```
git clone https://bitbucket.org/pennbigdataanalytics/hw1.git
```

to get your initial data sets and some "helper" code.

Run:

```
docker ps
```

and look for the CONTAINER ID as in here:



Copy the hex number on the left.

Now run

```
docker cp hw1 {paste container ID}:/home/jovyan/work
```

to copy all of the data into your Docker container.

If this doesn't work for you: *you can launch Kitematic, select the container and click on the link that opens Jupyter. Then from there you can individually upload the files from* ***hw1*** *in your home directory.*

The data files, whose contents are described in the provided notebook Dataset Descriptions, are:

- **airports.dat.txt -** data on airports, in comma-separated values (CSVs) with no header row
- **airlines.dat.txt -** data on airlines, in CSV with no header row
- **routes.dat.txt -** data on flight routes, in CSV with no header row
- **On_Time_On_Time_Performance_2016_10.csv -** data on actual flights, with performance info, with a header row
- **aircraft_incidents.htm -** webpage that lists commercial aircraft incidents by year

Go to your copy of Jupyter running on Docker, by opening your web browser to http://localhost:8888 or wherever Kitematic sends you when you click on the link (as in HW0).

Open the template Jupyter notebook "`Load DataFrames`".

Step 1: Importing the Data

1.1 Import from files

Fill in the Cells that read the input files using Pandas' `read_csv` function. For the first 3 sources you'll need to assign column names to the data, based on the **Dataset Descriptions** and some hints below. For the "on time performance" you can read the file, and have Python use the header names already in the file. We use the variable names **airports_df**, **airlines_df**, **routes_df**, and **flights_df** to refer to the DataFrames below. The "_df" is because Python is dynamically typed, so it helps make code more clear when the type is apparent from the variable name.

1. For your variables and your column names, follow the Python convention that names are in lowercase and underscores are used as spaces.
2. Specifically, when naming columns that are shared between datasets, use the format *datasource_property* (i.e. airport_id, airport_name, airline_id, etc.)

Loading the entirety of the flights_df DataFrame will generate a warning about mixed types. We don't actually care about all of the columns here. When calling **read_csv**, use the option **use_cols** to specify a *list of columns to import*. Only include the following fields (these were named in the CSV file header, thus don't follow Python naming conventions):
['Year','Month','DayofMonth','AirlineID','Carrier','FlightNum','Origin','Dest','ArrDelayMinutes','Cancelled']

Data Check for Step 1. Let's now create some Cells to get a quick look at what we have loaded (and so we can check your work!):

1. Find the cell that says "Step 1.1"
2. Run the Cell that simply outputs the **airports_df** DataFrame.
3. Update the next Cell to output the **airlines_df** DataFrame.
4. Update the next Cell that simply outputs the **routes_df** DataFrame.
5. Update the next Cell that simply outputs the **flights_df** DataFrame (after extraneous columns were discarded as described above).

Nulls. You should see for **airlines** a variety of entries that say "NaN" (not a number), which represents "unknown" or null information. In fact, if you look closely there are also other values representing "unknown" such as "\N" and even "-". We'll have to regularize all of this later!

Schemas. OK, you've loaded the DataFrames. You can get the *schemas* -- the names and types of the columns of the DataFrames by the **dtypes** property. Use **airlines_df.dtypes** and **routes_df.dtypes** to take a look. Now compare the *types* of **routes_df.airline_id** and **airlines_df.airline_id**. You should see that

compare the types of **routes_df.airline_id** and **airlines_df.airline_id**. You should see that one is "int64" and the other is "object."

Why is this? Python automatically infers the types based on what it reads from the CSV files. Unfortunately, things like "NaN" are actually *floating-point* (fractional) numbers and "\N" is a string. If a column has multiple kinds of values, Python will consider it to be an "object." Unfortunately, this will interfere with how we combine tables later, so we need to "clean" the data.

1.2 Import from a webpage

We are going to scrape data from the **aircraft_incidents.htm** webpage using the "beautifulsoup4" package. More details can be found in the documentation: https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

Run the code snippet that stores the web content in **soup**. Once the content is extracted, we need to get rid of the HTML tags and select the data we are going to be looking at. The data we need are the year and the description of the incidents under that year. It can be seen that the HTML tag (almost always) for the year is `<h3>` and for the incidents it is `<li>.` Using beautifulsoup4's `find_all(list of tags)` function, only select the content under those HTML tags. Store the data in a string variable called **selected_data**, where each year or incident description is separated by '\n'. Finally, check if all the tags have been removed. Output the message 'No Tag Found!', if successful.

Data Check for Step 1. Let's now create some Cells to get a quick look at what we have loaded (and so we can check your work!):

1. Find the cell that says "Step 1.2"
2. Run the Cell that scrapes the webpage **aircraft_incidents.htm**.
3. Update next cell to select only Year and Incident descriptions.
4. Update the next Cell to check for tags and output the message, if no tags.
5. Update the next Cell to output the **selected_data**.
6. Update the next Cell to write **selected_data** as a text file named **incidents_raw.txt**. We will use this file later.

Step 2: Simple Data Cleaning

Part 1

We are going to clean the **airlines_df, airports_df,** and **routes_df** DataFrames. First, in a new Cell, let's define a "helper" function.

```
# Replace NaNs with blanks if the column is a string, so we don't get
# objects in the column
def fillna_col(series):
    if series.dtype is pd.np.dtype(object):
        return series.fillna('')
    else:
        return series
```

Additionally, define a second function called **nullify** that takes a single parameter. Given the parameter with value "\N" it returns **NaN**, otherwise it returns the value of the parameter.

Regularizing and removing nulls. Next, we'll need to use two functions to apply **nullify** and **fillna_col** to our DataFrames.

- The DataFrame **applymap** function can be used to apply a function to *every ce*

this to apply **nullify** to all of the elements in each of our DataFrames (airports, airlines, etc.) and get rid of all of the "\N"s.

- The DataFrame **apply** function can be used to apply a function to *every column* of a DataFrame. Let's use that to call **fillna_col** on the DataFrames -- replacing the NaNs with blank strings if the column is otherwise an object.
- Also, let's get rid of rows in **routes_df** that have null airline, source, or destination IDs. Recall that **dropna** can be used here.

You'll want to update your DataFrames using all of the above functions. *Think carefully about the order in which to apply these.*

Changing column types. After all of this, **routes_df.airline_id** will only have integers, but will still have its existing type of *object*. Later we'll need it to be an integer, so that it can be compared directly with the entries from **airlines_df** and **airports_df**. Let's convert it to integer via:

```
routes_df['airline_id'] = routes_df['airline_id'].astype(int)
```

Repeat the same process for the source and destination airport IDs.

Part 2

We will clean the raw text data we stored in **incidents_raw.txt**. For each incident, we want it in the form:

```
1997 January 9 , Comair Flight 3272, an Embraer EMB 120 Brasília, crashes near Ida, Michigan,
during a snowstorm, killing all 29 on board.
...
```

Points to follow during cleaning:

- Remove '[edit]' from the year
- Only select incidents that have occured in the year >= 1997
- Since we extracted the data using tags <h3> and <li>, it is possible that there was other data extracted too. See the end of file **incidents_raw.txt.** However, if we look at the format of all the incidents, we can see that they all contain: month, the word 'Flight' and the symbol '–'. Use these conditions to filter out unwanted data.
- Store all the cleaned incidents in a list named **clean_incidents**

Now that we have all the aircraft incidents since 1997, we need to convert them into a Pandas DataFrame. Use the empty dataframe **incidents_df**. For each entry in **clean_incidents**, extract the date, airline name and flight number (only the part that comes after 'Flight') then store it as a new row in **incidents_df**. Set the column type of incidents_df['Date'] to datetime.

Data Check for Step 2. Run the Cells under to output **airports_df**, **airlines_df**, **routes_df, incidents_df**.

Step 3: Making Data "Persistent"

Now let's actually save the data in a persistent way, specifically using a relational database. For simplicity we'll use **SQLite** here, but we could alternatively use a DBMS such as MySQL or PostgreSQL on the cloud (or in another Docker container).

Create a Cell that starts with the following two lines, which you'll extend momentarily:

```
import sqlite3
engine = sqlite3.connect('HW1_DB')
```

This establishes a connection to an SQLite database, which will be written to the file **HW**

directory.

Next, in this Cell you should save each of your DataFrames (**airlines_df, airports_df, flights_df, routes_df, incidents_df**) to the database. To do this, call the **to_sql** method (make sure to not save the index column) on the DataFrame. Give it a table name matching the DataFrame name, and set the flag **if_exists='replace'** in case you want to run this multiple times.

Once this is all done, you can save the "**Load DataFrames**" Jupyter notebook for submission.

Step 4: Wrangling and Combining Data

Open the Python 3 notebook called "**Wrangling**". Start off by connecting to the database (as above), `import pandas as pd`, and then load back your DataFrames: **airlines, airports, routes** and **incidents** (not counting **flights**, which will take forever to load) from SQL, using the syntax:

```
dataframe = pd.read_sql('select * from table_name', engine)
```

In the next Cell, run **airports_df.info()**, **airlines_df.info()**, **routes_df.info()** and **incidents_df.info()**.

Observe that this tells you the data types on the various DataFrame elements.

Data Check for Steps 3-4 . Create some Cells to get a quick look at what we have loaded:

1. Find the Cell that says "Step 4.0"
2. Update the Cell to simply output **airports_df**. In another cell, output **airports_df.describe().**
3. Update the Cell to simply output **airlines_df**. In another cell, output **airlines_df.describe().**
4. Update the Cell to simply output **routes_df**. In another cell, output **routes_df.describe().**
5. Update the Cell to simply output **incidents_df**. In another cell, output **incidents_df.describe()**.

Before we use data -- it's essential to understand that data is seldom "pristine" and perfect. Human data entry error, incompleteness, and measurement error (and even flipped bits due to gamma rays!)

4.1 Understanding the Data

We would expect certain characteristics, such as every airport to have flights, and every airport to be a source and a destination. *(Otherwise we'd have a situation where some airports pile up airplanes, and others have no planes!)*

Take a look at our data and see:

- How many airports are not included in a route?
- How many airports serve as both *source and destination*?

We'll revisit these questions momentarily when we have more detail. Meanwhile...

Data Check for Step 4.1. Let's create a structure to report part of what we've discovered.

1. Find the Cell that says "Step 4.1"
2. In the next Cell, create and output a dictionary with counts of the number of unique values of items for which we have information in a dictionary:

```
{'airports': ___, 'destinations': ___, 'sources': ___}
```

where you fill in the number of **unique Airport IDs**, **destination airport IDs**, and **source/origin airport IDs**. *Note any inconsistencies here?*

4.2 Looking at Joint Data

Now let's combine information from the different sources!

4.2.1 Combining DataFrames via Merge (also known as *Join*)

The first question is which airports are in the data as destinations, but not as sources.

"Query" for Step 4.2.1.
1. Find the Cell that says "Step 4.2.1"
2. Update the next Cell to contain the **names** (not just IATA codes) of airports that you can fly to, but not from. Be sure to eliminate duplicates.

You'll need to do a *merge* (join) with the original **airports_df** DataFrame. Note that **dataframe.merge**() can be called with a second DataFrame, as well as the columns to match (**left_on**, **right_on**).

4.2.2 Top-k Destinations

Next let's consider what destination airports are most-trafficked, by name. Recall that we have the frequencies of arrivals for each airport by its ID. You may also find the **.reset_index()** method useful.

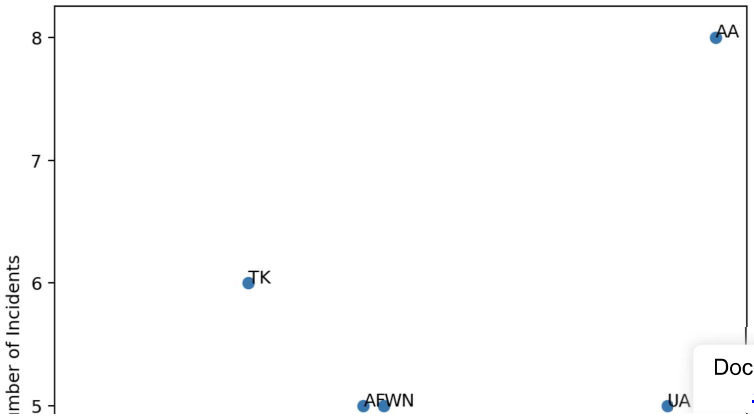
"Query" for Step 4.2.2.
1. Find the Cell that says "Step 4.2.2"
2. Update the next Cell to contain the top-10 most popular airports, in decreasing order of popularity, identified by their **airport names** (not just the IATA codes). Include the counts.

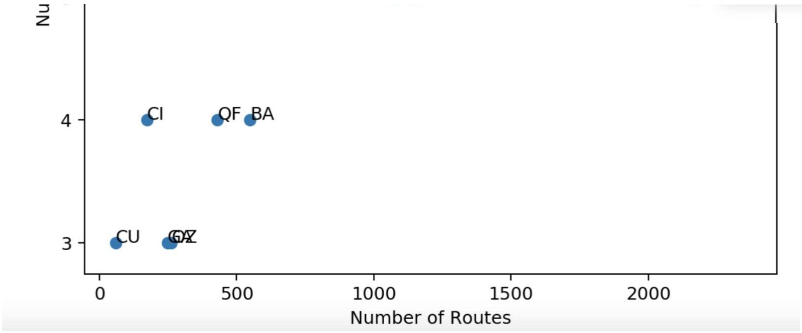
4.3 Visualizing the Data

Let's try to find out how "reliable" an airline is by comparing the number of flight routes it has and the number of incidents it has had in the last 20 years.

"Query" for Step 4.3.
1. Find the Cell that says "Step 4.3"
2. In the next Cell, create a DataFrame that has the number of incidents and the number of routes for each airline IATA.
3. Filter out records that have more than 2 incidents and create a scatter plot of the number of routes vs. the number of incidents. *Notice anything interesting?*

For the scatter plot, use **matplotlib** and it should look like:





Step 5: Integrating Data

Data wrangling typically refers to the process of importing, remapping, and converting data. On the other hand, *data integration* typically refers to linking data across heterogeneous sources. We have an integration problem (in its simplest form) when we want to combine the on-time information (what was originally **flights_df**, and is now in the SQL database) and the route information (**routes_df**).

5.1 Relating Routes and Flights: Schema Matching

Now we'll make use of the on-time flight performance table… Recall that we put this into an SQL table and that we'll prefer to avoid bringing that into memory for performance reasons. Let's get a *random sample* of the table into a DataFrame. You can get a sample of *x* rows from *table* by:

```
df = pd.read_sql_query('SELECT * FROM table ORDER BY RANDOM() LIMIT x', con=engine)
```

Replace *x* and *table* to get a 1000-element sample of the flight table; call this **flights_s_df**.

Now let's see if we can actually detect what fields (columns) "match" between **routes_df** and **flights_s_df** (for the sample) DataFrames.

To do this, let's note that we only care about the "categorical" (string-valued) columns. What we want is, e.g., to find fields that have lots of common values (for instance, airline abbreviations or airport codes). To measure this in a principled way, we'll use a measure called the **Jaccard distance** (or Jaccard index or Jaccard measure). The Jaccard distance measures similarity between two sets **A** and **B**, and is merely:

Let's compute the Jaccard distances between (the values of) all pairs of attributes in **routes_df** and **flights_s_df**. This will tell us which attributes include similar values - and thus which might represent the same concept.

5.1.1 Column Matches (Schema Matches)

Create a Cell with an algorithm that iterates over all **routes_df** keys (columns) and all **flights_s_df** keys. In a nested map {***r*** → {***p*** → ***distance***}}, the algorithm should store the **Jaccard distance** between the values in routes column ***r*** to the values in performance column ***p*** **(for distances that are non-zero and only for non-numeric columns)**. Remember, since there may be duplicates, you'll need to convert columns into sets of unique values through **drop_duplicates()**.

Data Check for Step 5.1. Let's look at the output of your Jaccard similarity-based attribute matching algorithm.

1. Find the Cell that says "Step 5.1"
2. In the next Cell, output the nested map output by your computation above.

Your output should look something like:

```
{'airline_iata': {'Carrier': 0.04...},
'code_share': {},
...}
```

This map should help you decide which pairs of columns (one in each DataFrame) are "semantically equivalent." You should be able to find a very clear match between non-numeric codes for airlines in the two DataFrames. The origin and destination codes will also have corresponding pairs (but if you only look at the Jaccard distances, there will be some ambiguity, since both origin and destination cities have values from the same domain; you will need to use your own knowledge of the semantics of the fields to disambiguate which pairs are correct).

Step 5.2 Creating Detailed Flights, Filtered by OpenFlight Routes

Now that you know how **routes_df** and **flights_s_df** data relate, let's see how the routes and flights actually relate. Note that each data source (and thus DataFrame) is in fact incomplete. For this step we'll want to only consider the flight information and route information that "intersects."

5.2.1 Flights by Airline and Destination

Create a *joint* DataFrame **flights_s_df** by merging the two DataFrames on the fields that correspond to flight information. Use **routes_df.merge** as well as **left_on** and **right_on**. Note that **flights_s_df** only represents a *sample* of all flights, since **flights_s_df** only contains a sample of 1000 entries. However, it's also worth noting that OpenFlight only has information about certain routes, and we want to focus on the flights corresponding to routes in OpenFlight.

*Hint: you'll need to pass lists for **left_on** and **right_on** in order to merge on the 3 fields between **routes_df** and **flights_s_df** that you found to "match" in the previous part. These 3 fields should effectively correspond to flights in **flights_s_df**.*

Nonetheless we'll use this sample to figure out how to combine DataFrames. Later, we'll make this scale further by rewriting our computations in SQL.

"Query" for Step 5.2.1.

1. Find the Cell that says "Step 5.2.1"
2. Create a *joint* DataFrame **flights_s_df**.
3. In the next Cell, output a DataFrame that contains, for each **airline IATA code** and **destination city IATA code**, the number of flights that appear in the sample **flights_s_df**.

5.2.2 Delayed Flights by Airline and Destination

Now let's repeat the above, but only for flights that met a frustrating fate.

"Query" for Step 5.2.2.

1. Find the Cell that says "Step 5.2.2"
2. In the next Cell, output a DataFrame, for each **airline IATA code** and **destination city IATA code**, the number of flights that appear in the sample **flights_s_df**, whose arrival was delayed by 30 minutes or more, or which were cancelled (**we'll call this a "bad" flight in 5.2.3**).

5.2.3 Multiply-Delayed Flights by Airline and Destination

"Query" for Step 5.2.3.

1. Find the Cell that says "Step 5.2.3"
2. In the next Cell, output a DataFrame with the number of "bad" flights for each (airline IATA code, destination city IATA code) pair if that pair has more than one "bad" flight.
3. In a third Cell, plot a bar chart, where each bar represents the "bad" flights for each (airline IATA code, destination city IATA code) pair as in Step 2.

5.2.4 Multiply Delayed Flights by Airline

What if we re-examine the above question, but by airline instead of by city?

"Query" for Step 5.2.4.

1. Find the Cell that says "Step 5.2.4"
2. In the next Cell, output a DataFrame that contains, for each airline IATA code, the number of flights that appear in the sample **flights_s_df**, whose arrival was delayed by 30 minutes or more, or which were cancelled. Sort by number of delays/cancellations in decreasing order.

Save the results of the "Wrangling" notebook for submission. If you want to go on to the Advanced version of the assignment, you'll pick up from here. If not, skip to Step 7.

Step 6: "Advanced" Assignment

Create a new Jupyter notebook titled "**Advanced**." Initialize it by loading the **airlines_df**, **airports_df**, and **routes_df** from the SQL database, as before. This time, we want to look at the flights more comprehensively -- and at this point we will have exceeded the point where Python DataFrames are efficient in handling the data. We'll instead make heavier use of the SQL database.

6.1 Pulling Data from the Web

First, we will go beyond the 1 month's data that currently exists, instead expanding to 3 months' flight data.

The fastest way to read a remote CSV file into a DataFrame is to call **read_csv** with the URL. Create a DataFrame **aug_flights_df** in this manner using flight data from August at URL http://big.dataanalytics.education/data/Flights-Aug-2016.csv . Now write the DataFrame to an SQL table **flights**.

Repeat this for …-Sep-2016.csv and …-Oct-2016.csv. Each time, write using `if_exists='append'` so you can keep the existing contents.

An alternate way to get this data (which is slightly slower, but that can be used to access Web data that requires querying Web forms) uses a few additional Python packages that should be imported: **requests** and **StringIO** (located in the io package). We illustrate below how to obtain the flight data from August in this manner.

You can use the **get** call in the **requests** Python package to pull data from the Web. You can use this to retrieve http://big.dataanalytics.education/data/Flights-Aug-2016.csv into a variable **aug_response**. The result (as a large string) will be available as **aug_response.text** (Note: trying to display the whole string in your Jupyter notebook may cause it to hang).

Let's use another tool, **StringIO** (located in the io package), to treat a string as if it is the contents of a file. Call:

```
aug_csv = io.StringIO(aug_response.text)
```

```
aug_csv = io.StringIO(aug_response.text)
```

To "wrap" the CSV text. Now create a DataFrame **aug_flights_df** using `read_csv` over the wrapped file. Project it to only include the fields:

```
'Year','Month','DayofMonth','Carrier','FlightNum','Origin',
'Dest','DepTime','ArrTime','ArrDelayMinutes','Cancelled'
```

Great, now let's see how much data we have in the SQL database.

Use (a modified version of) the Pandas command:

```
df = pd.read_sql(query, db_connection)
```

with the appropriate query to get a `count` on the number of rows in the **flights** table.

Data Check for Step 6.1.

1. Create a Cell in Markdown with contents"## Step 6.1" and hit **Shift-Enter.**
2. Output the **value of the count** in a Cell, as an integer. As a hint, you can use the **ix** function to get a cell at a particular position in a dataframe.

Hint: It should be a bit more than *1.4 million rows*.

Recall that you can call the `info()` or `describe()` functions on the DataFrame to get its fields, types, and information about values.

6.2 Comprehensive Flight Information Using SQL

In this part of the assignment, you'll investigate the correspondence between DataFrame and SQL abstractions. Recall that in 5.2 you used Python's DataFrame **merge** command to merge two dataframes:

```
output_df = left_df.merge(right_df, left_on=['left_col1','left_col2',...],
                                    right_on=['right_col1','right_col2',...])
```

We can do the same thing between left and right SQL tables using:

```
output_df = pd.read_sql("SELECT * FROM left_table JOIN right_table
         ON left_col1=right_col1, left_col2=right_col2", db_connection)
```

If you only want a subset of the attributes (columns) to show up in the DataFrame, replace "*" with a list of columns names.

Also, if you want to compute *aggregates*, you can do:

```
output_df = pd.read_sql("SELECT col_g_1, col_g_2, agg1(col_a_1), agg2(col_a_2)
                                    FROM table
                                    GROUP BY col_g_1, col_g_2", db_connection)
```

Now we'll use these basic abstractions to look in more detail at the bigger flight dataset, repeating some of the questions of Step 5.2 above.

For all of the steps in this section, be sure that you are using SQL queries to retrieve the data.

6.2.1 Flights by Airline and Destination

As in 5.2.1, we have two different data sources with flight-related information: the **routes** from Openflight.org and the **flights** from the US government. For this part of the assignment (all of Step 6.2), we are ... **flights** that correspond to routes in Openflight -- i.e., those that match known routes.

"Query" 6.2.1.

1. Create a Cell in Markdown with contents"## Step 6.2.1" and hit **Shift-Enter.**
2. Create a Cell that outputs in a DataFrame, for each **airline IATA code** and **destination city IATA code**, the number of flights from August-October. Make sure that all flights in the DataFrame are along a valid route by **joining** with the routes table (like in 5.2.1).

   Add to the end of the SQL statement (after your GROUP BY) the clause:
   ORDER BY airline_iata, Dest LIMIT 60

   to sort by the airline IATA and destination, and return only the top 60 results.

6.2.2 Delayed Flights by Airline and Destination

To filter individual rows from a table, add a predicate of the form

```
SELECT … FROM …
WHERE condition_on_attributes
 GROUP BY …
```

to filter.

In a Cell, create a new SQL query based off the one in 6.2.1 but with a WHERE constraint conditioning on the arrival delay/cancellation.

"Query" 6.2.2.

1. Create a Cell in Markdown with contents"## Step 6.2.2" and hit **Shift-Enter.**
2. Create a Cell that outputs in a DataFrame, for each **airline IATA code** and **destination city IATA code**, the number of flights (from the data of Section 6.2.1), whose arrival was delayed by 30 minutes or more, or which were cancelled. Add to the end of the SQL statement (after your GROUP BY) the clause:
   ORDER BY airline_iata, Dest LIMIT 60

   to sort by the airline IATA and destination, and return only the top 60 results.

   You'll need to use the DataFrame **set_index()** function to convert the airline IATA code and destination city IATA code into the index of the DataFrame.

6.2.3 Frequently-Delayed Flights by Airline and Destination

To filter results from a grouping, by some property like the count, add a **HAVING** clause.

```
SELECT … FROM …
 GROUP BY …
 HAVING agg-expression
```

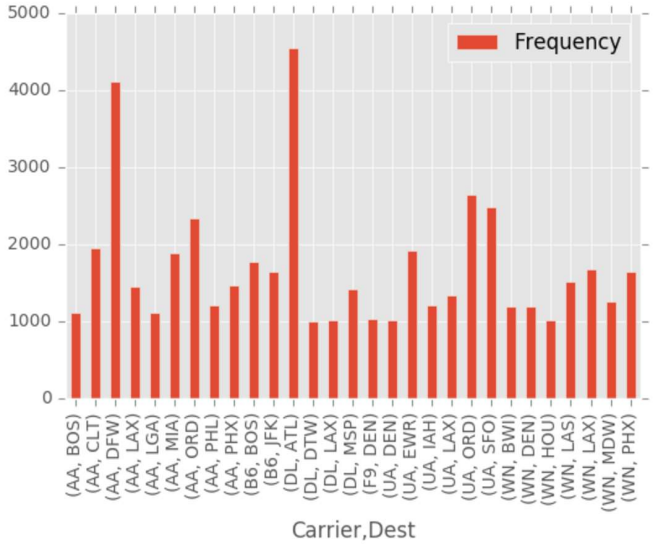
to filter.

In a Cell, create a new SQL query based off the one in 6.2.2 but with the HAVING constraint specifying that the COUNT must be at least 1000 flights delayed by 30 minutes or more, or cancelled.

Data Check and Visualization for Step 6.2.3.

1. Create a Cell in Markdown with contents "## Step 6.2.3" and hit **Shift-Enter.**
2. Create a Cell outputting the results of the SQL query as a DataFrame.
3. Plot a **bar chart** of all (airline, destination city) pairs with these delays.

Your bar chart distribution should end up resembling something like this:



6.3 Shorter Travel Times?

Our last step will be to perform a *shortest path* computation over the flight data. For each fastest direct flight between a pair of cities (C1, C2) on the same day we want to find *shorter pairs of flights* (from initial departure time to final arrival time) that get us to the same destination. Assume that it takes one hour to arrive in an airport on one flight, and make it to another flight before it departs (*hint: take a look at the data in the DepTime and ArrTime fields before you assume that you should add 60 minutes to get an hour!*).

The next flight from the stopover must be on the same day. For simplicity you may ignore flights that wrap to the next day. For performance reasons limit your computation to data for **September 2016**. *[Note that we are quite performance-bottlenecked in this part of the assignment -- later we'll see how to handle large-scale graph traversals more efficiently.]*

First create a "base case" where you take compute, and store in a SQL table, the earliest arrival for each departure time for each pair of cities on the same day. Make sure that all flights in the table are along a valid route by **joining** with the routes table (like in previous parts of this assignment). Put these these into a table called **shortest**. You should use SQL's `min()` aggregate function to find the earliest **ArrTime**. Be sure to drop entries with **null** dates, times, sources, or destinations. In SQL, you can output a field named **f** with literal value **v** using:

```
SELECT v AS f … FROM ...
```

And you can also replace *v* with an expression (such as `(w-3)`) instead of a variable.

As a second step, you will want to create an **index** over the **shortest** table to speed up computation. You'll probably want to do:

```
engine.execute('CREATE index my_index ON shortest(Year,Month,DayofMonth,Origin,Dest)')
```

Next, find a pair of shorter paths that leave at the same time, but arrive earlier to the same destination on the same day. Make sure to ignore the flights that arrive on the following day. You will probably want to make use of (1) variables referring to different iterators (aliases) over the SQL table, and (2) the **exists**() predicate in SQL, as in the use of *s1,s2,s3* in:

```
SELECT v1,v2 …
FROM shortest AS s1 JOIN shortest AS s2 ON s1… = s2…
WHERE  EXISTS
       (SELECT v1 FROM shortest AS s3 WHERE s3… = s1… AND ...)
```

Note: "AS" is optional. The query will work exactly the same if "AS" is omitted, but adding "AS" makes the query more explicit.
Hint: You can get the right query behavior (and keep the computation from going on forever) by essentially limiting the 2 hop flights by seeing if there EXISTS a corresponding flight (same time, route, and departure time) that takes longer in the **shortest** (1 hop) table.

6.3.1 Two-Hop Flights that Turned out to be Shorter

Query for Step 6.3.1.

1. Create a Markdown cell labeled "## Step 6.3.1"
2. In the next Cell, output the results of 2-hop (1-stop) computations that are shorter than direct flights originating at the same time from the same place, in a DataFrame with the fields **Origin,Dest,Hops,DepTime,ArrTime,Year,Month,DayofMonth**. Output the IATA codes for the origin and destination airports.

   Limit your computation to flights starting from LAX and ending in DEN for the first 8 days of the month.

Step 7: Submitting Homework 1

For submission, **we will expect you to submit your Jupyter notebooks with both the code and the output results in each Cell.** We'll be trying to give you rough feedback by auto-validating that the data looks OK. Before submitting, we recommend you go to the **Cell** menu in Jupyter and select **Run All**, then check that every cell has output that makes sense.

Once your Jupyter notebooks are sanity-checked, add the notebook files to **hw1.zip** using the **zip** command at the Terminal, much as you did for HW0. The notebooks should be:

- `Load DataFrames.ipynb`
- `Wrangling.ipynb`
- `Advanced.ipynb` (if applicable)

Recall that you can look at the different options here to determine how to access the files from "outside" Docker.

Next, go to the submission site, and if necessary click on the Google icon and log in using your Google@SEAS or GMail account. At this point the system should know you are in the appropriate course. Select CIS 545 **Homework 1** and upload `hw1.zip` from your Jupyter folder, typically found under `/Users/`*{myid}*.

If you check on the submission site after a few minutes, you should see whether your submission passed validation. You may resubmit as necessary.