Using SQL with Pandas - Lab

Introduction

In this lab, you will practice using SQL statements and the `.query()` method provided by Pandas to manipulate datasets.

Objectives

You will be able to:

- Compare accessing data in a DataFrame using query methods and conditional logic
- Query DataFrames with SQL using the `pandasql` library

The Dataset

In this lab, we will continue working with the *Titanic Survivors* dataset.

Begin by importing `pandas` as `pd`, `numpy` as `np`, and `matplotlib.pyplot` as `plt`, and set the appropriate alias for each. Additionally, set `%matplotlib inline`.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Next, read in the data from `titanic.csv` and store it as a DataFrame in `df`. Display the `.head()` to ensure that everything loaded correctly.

```
df = pd.read_csv('titanic.csv', index_col=0)
df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Pa
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Pa
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0

Slicing DataFrames Using Conditional Logic

One of the most common ways to query data with pandas is to simply slice the DataFrame so that the object returned contains only the data you're interested in.

In the cell below, slice the DataFrame so that it only contains passengers with 2nd or 3rd class tickets (denoted by the `Pclass` column).

Be sure to preview values first to ensure proper encoding when slicing

- ***Hint***: Remember, your conditional logic must be passed into the slicing operator to return a slice of the DataFrame--otherwise, it will just return a table of boolean values based on the conditional statement!

```
# Preview values first to ensure proper encoding when slicing
df['Pclass'].unique()
```

```
array(['3', '1', '2', '?'], dtype=object)
```

```
no_first_class_df = df[df['Pclass'].isin(['2','3'])]
no_first_class_df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}
```

```
.dataframe thead th {
    text-align: right;
}
```

</style>

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	P
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0
5	6	0	3	Moran, Mr. James	male	NaN	0	0
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1

We can also chain conditional statements together by wrapping them in parenthesis and making use of the `&` and `|` operators ('and' and 'or' operators, respectively).

In the cell below, slice the DataFrame so that it only contains passengers with a `Fare` value between 50 and 100, inclusive.

```
fares_50_to_100_df = df[(df['Fare'] >= 50) & (df['Fare'] <= 100)]
fares_50_to_100_df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}
```

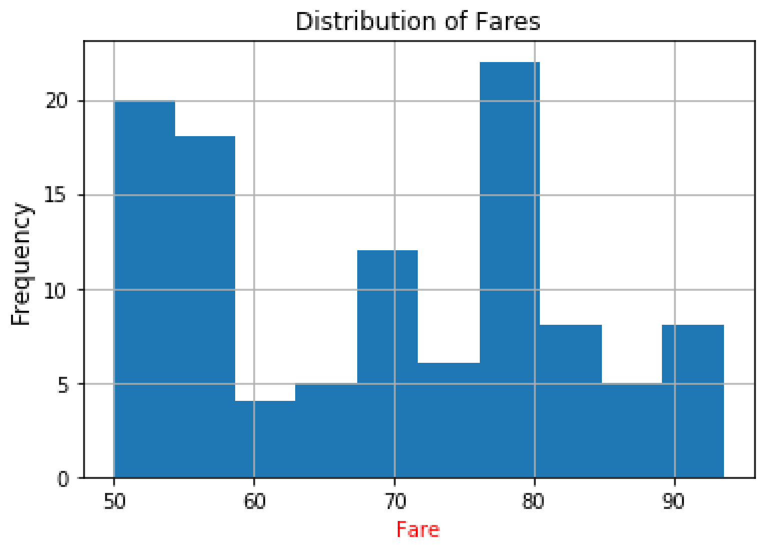
```
.dataframe thead th {
    text-align: right;
}
```

</style>

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0
34	35	0	1	Meyer, Mr. Edgar Joseph	male	28.0	1
35	36	0	1	Holverson, Mr. Alexander Oskar	male	42.0	1

We could go further and then preview the Fare column of this new subsetted DataFrame:

```
fares_50_to_100_df['Fare'].hist()
plt.xlabel('Fare', color='red')
plt.ylabel('Frequency', fontsize=12)
plt.title('Distribution of Fares');
```



Remember that there are two syntactically correct ways to access a column in a DataFrame. For instance, `df['Name']` and `df.Name` return the same thing.

In the cell below, use the dot notation syntax and slice a DataFrame that contains male passengers that survived that also belong to Pclass 2 or 3. Be sure to preview the column names and content of the `Sex` column.

```
# Checking column names for reference
df.columns
```

```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

```
# Checking column values to hardcode query below
df['Sex'].unique()
```

```
array(['male', 'female'], dtype=object)
```

```
poor_male_survivors_df = df[(df['Pclass'].isin(['2', '3'])) & (df['Sex'] == 'male')]
poor_male_survivors_df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1
4	5	0	3	Allen, Mr. William Henry	male	35.0	0
5	6	0	3	Moran, Mr. James	male	NaN	0
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3
12	13	0	3	Saundercock, Mr. William Henry	male	20.0	0

Great! Now that you've reviewed the methods for slicing a DataFrame for querying our data, let's explore a sample use case.

Practical Example: Slicing DataFrames

In this section, you're looking to investigate whether women and children survived more than men, or that rich passengers were more likely to survive than poor passengers. The easiest way to confirm this is to slice the data into DataFrames that contain each subgroup, and then quickly visualize the survival rate of each subgroup with histograms.

In the cell below, create a DataFrame that contains passengers that are female, as well as children (males included) ages 15 and under.

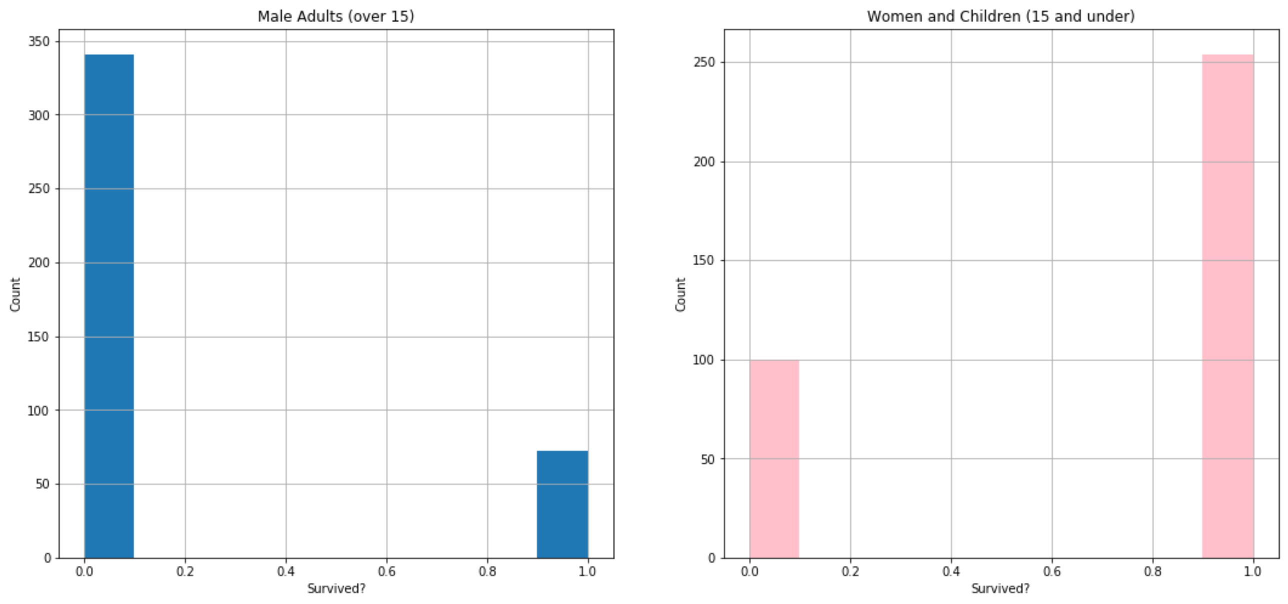
Additionally, create a DataFrame that contains only adult male passengers over the age of 15.

```
women_and_children_df = df[(df['Sex'] == 'female') | (df['Age'] <= 15)]
adult_males_df = df[(df['Sex'] == 'male') & (df['Age'] > 15)]
```

Great! Now, you can use the `matplotlib` functionality built into the DataFrame objects to quickly create visualizations of the `Survived` column for each DataFrame.

In the cell below, create histogram visualizations of the `Survived` column for both DataFrames. Bonus points if you use `plt.title()` to label them correctly and make it easy to tell them apart!

```
fig, axes = plt.subplots(ncols=2, nrows=1, figsize=(18, 8)) # Two figures side by si
ax_lft = axes[0]
adult_males_df['Survived'].hist(ax=ax_lft)
ax_lft.set_title('Male Adults (over 15)')
ax_lft.set_xlabel('Survived?')
ax_lft.set_ylabel('Count')

ax_rght = axes[1]
women_and_children_df['Survived'].hist(ax=ax_rght, color='pink')
ax_rght.set_title('Women and Children (15 and under)')
ax_rght.set_xlabel('Survived?')
ax_rght.set_ylabel('Count');
```



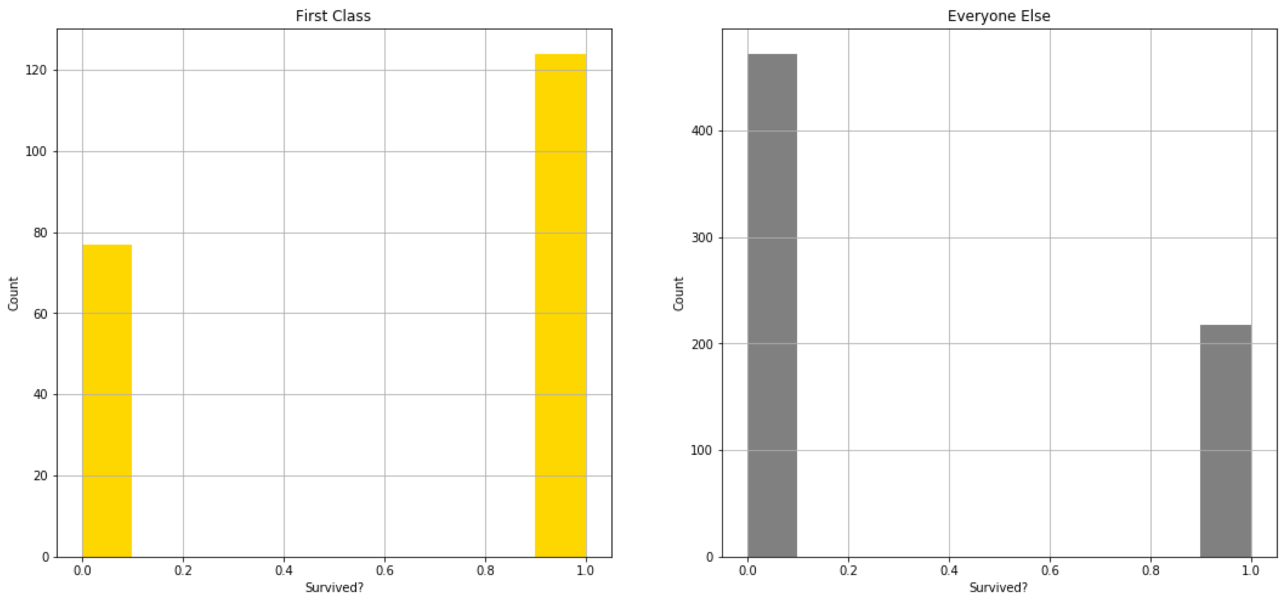
Well that seems like a pretty stark difference -- it seems that there was drastically different behavior between the groups! Now, let's repeat the same process, but separating rich and poor passengers.

In the cell below, create one DataFrame containing First Class passengers (`Pclass == 1`), and another DataFrame containing everyone else.

```
first_class_df = df[df['Pclass'] == '1']
second_third_class_df = df[df['Pclass'] != '1']
```

Now, create histograms of the surivival for each subgroup, just as you did above.

```
fig, axes = plt.subplots(ncols=2, nrows=1, figsize=(18, 8)) # Two figures side by si
ax_lft = axes[0]
first_class_df['Survived'].hist(ax=ax_lft, color='gold')
ax_lft.set_title('First Class')
ax_lft.set_xlabel('Survived?')
ax_lft.set_ylabel('Count')

ax_rght = axes[1]
second_third_class_df['Survived'].hist(ax=ax_rght, color='grey')
ax_rght.set_title('Everyone Else')
ax_rght.set_xlabel('Survived?')
ax_rght.set_ylabel('Count');
```



To the surprise of absolutely no one, it seems like First Class passengers were more likely to survive than not, while 2nd and 3rd class passengers were more likely to die than not. However, don't read too far into these graphs, as these aren't at the same scale, so they aren't fair comparisons.

Slicing is a useful method for quickly getting DataFrames that contain only the examples we're looking for. It's a quick, easy method that feels intuitive in Python, since we can rely on the same conditional logic that we would if we were just writing `if/else` statements.

Using the `.query()` method

Instead of slicing, you can also make use of the DataFrame's built-in `.query()` method. This method reads a bit more cleanly and allows us to pass in our arguments as a string. For more information or example code on how to use this method, see the pandas documentation.

In the cell below, use the `.query()` method to slice a DataFrame that contains only passengers who have a `PassengerId` greater than or equal to 500.

```
query_string = 'PassengerId >= 500'
print(query_string)
high_passenger_number_df = df.query(query_string)
high_passenger_number_df.head()
```

```
PassengerId >= 500
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp
499	500	0	3	Svensson, Mr. Olof	male	24.0	0

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp
500	501	0	3	Calic, Mr. Petar	male	17.0	0
501	502	0	3	Canavan, Miss. Mary	female	21.0	0
502	503	0	3	O'Sullivan, Miss. Bridget Mary	female	NaN	0
503	504	0	3	Laitinen, Miss. Kristina Sofia	female	37.0	0

Just as with slicing, you can pass in queries with multiple conditions. One unique difference between using the `.query()` method and conditional slicing is that you can use `and` or `&` as well as `or` or `|` (for fun, try reading this last sentence out loud), while you are limited to the `&` and `|` symbols to denote and/or operations with conditional slicing.

In the cell below, use the `query()` method to return a DataFrame that contains only female passengers of ages 15 and under.

Hint: Although the entire query is a string, you'll still need to denote that `female` is also a string, within the string. (*String-Ception?*)

```
female_children_df = df.query("Sex == 'female' and Age <= 15")
female_children_df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14.0	1
10	11	1	3	Sandstrom, Miss. Marguerite Rut	female	4.0	1
14	15	0	3	Vestrom, Miss. Hulda Amanda Adolfina	female	14.0	0
22	23	1	3	McGowan, Miss. Anna "Annie"	female	15.0	0
24	25	0	3	Palsson, Miss. Torborg Danira	female	8.0	3

A cousin of the `query()` method, `eval()` allows you to use the same string-filled syntax as querying for creating new columns. For instance:

```
some_df.eval('C = A + B')
```

would return a copy of the `some_df` dataframe, but will now include a column `C` where all values are equal to the sum of the `A` and `B` values for any given row. This method also allows the user to specify if the operation should be done in place or not, providing a quick, easy syntax for simple feature engineering.

In the cell below, use the DataFrame's `eval()` method in place to add a column called `Age_x_Fare`, and set it equal to `Age` multiplied by `Fare`.

```
df = df.eval('Age_x_Fare = Age*Fare')
df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Pa
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0

Great! Now, let's move on the coolest part of this lab--querying DataFrames with SQL!

Querying DataFrames With SQL

For the final section of the lab, you'll make use of the `pandasql` library. Pandasql is a library designed to make it easy to query DataFrames directly with SQL syntax, which was open-sourced by the company, Yhat, in late 2016. It's very straightforward to use, but you are still encouraged to take a look at the documentation as needed.

If you're using the pre-built virtual environment, you should already have the package ready to import. If not, uncomment and run the cell below to `pip install pandasql` so that it is available to import.

```
# !pip install pandasql
```

That should have installed everything correctly. This library has a few dependencies, which you should already have installed. If you don't, just `pip install` them in your terminal and you'll be good to go!

In the cell below, import `sqldf` from `pandasql` .

```
from pandasql import sqldf
```

Great! Now, it's time to get some practice with this handy library.

`pandasql` allows you to pass in SQL queries in the form of a string to directly query your database. Each time you make a query, you need to pass an additional parameter that gives it access to the other variables in the session/environment. You can use a lambda function to pass `locals()` or `globals()` so that you don't have to type this every time.

In the cell below, create a variable called `pysqldf` and set it equal to a lambda function `q` that returns `sqldf(q, globals())` . If you're unsure of how to do this, see the example in the documentation.

```
pysqldf = lambda q: sqldf(q, globals())
```

Great! That will save you from having to pass `globals()` as an argument every time you query, which can get a bit tedious.

Now write a basic query to get a list of passenger names from `df`, limit 10. If you would prefer to format your query on multiple lines and style it as canonical SQL, that's fine -- remember that multi-line strings in Python are denoted by `"""` -- for example:

```
"""
This is a
Multi-Line String
"""
```

In the cell below, write a SQL query that returns the names of the first 10 passengers.

```
q = """SELECT Name
       FROM df
       LIMIT 10;"""

passenger_names = pysqldf(q)
passenger_names
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	Name
0	Braund, Mr. Owen Harris
1	Cumings, Mrs. John Bradley (Florence Briggs Th...
2	Heikkinen, Miss. Laina
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)
4	Allen, Mr. William Henry
5	Moran, Mr. James
6	McCarthy, Mr. Timothy J

	Name
7	Palsson, Master. Gosta Leonard
8	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
9	Nasser, Mrs. Nicholas (Adele Achem)

Great! Now, for a harder one:

In the cell below, query the DataFrame for names and fares of any male passengers that survived, limit 30.

```
q2 = """SELECT Name, Fare
        FROM df
        WHERE Sex = 'male' AND Survived = 1
        LIMIT 30;"""

sql_surviving_males = pysqldf(q2)
sql_surviving_males
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	Name
0	Braund, Mr. Owen Harris
1	Cumings, Mrs. John Bradley (Florence Briggs Th...
2	Heikkinen, Miss. Laina
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)
4	Allen, Mr. William Henry
5	Moran, Mr. James

	Name
6	McCarthy, Mr. Timothy J
7	Palsson, Master. Gosta Leonard
8	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
9	Nasser, Mrs. Nicholas (Adele Achem)

This library is really powerful! This makes it easy for us to leverage all of your SQL knowledge to quickly query any DataFrame, especially when you only want to select certain columns. This saves from having to slice/query the DataFrame and then slice the columns you want (or drop the ones you don't want).

Although it's outside the scope of this lab, it's also worth noting that both `pandas` and `pandasql` provide built-in functionality for join operations, too!

Practical Example: SQL in Pandas

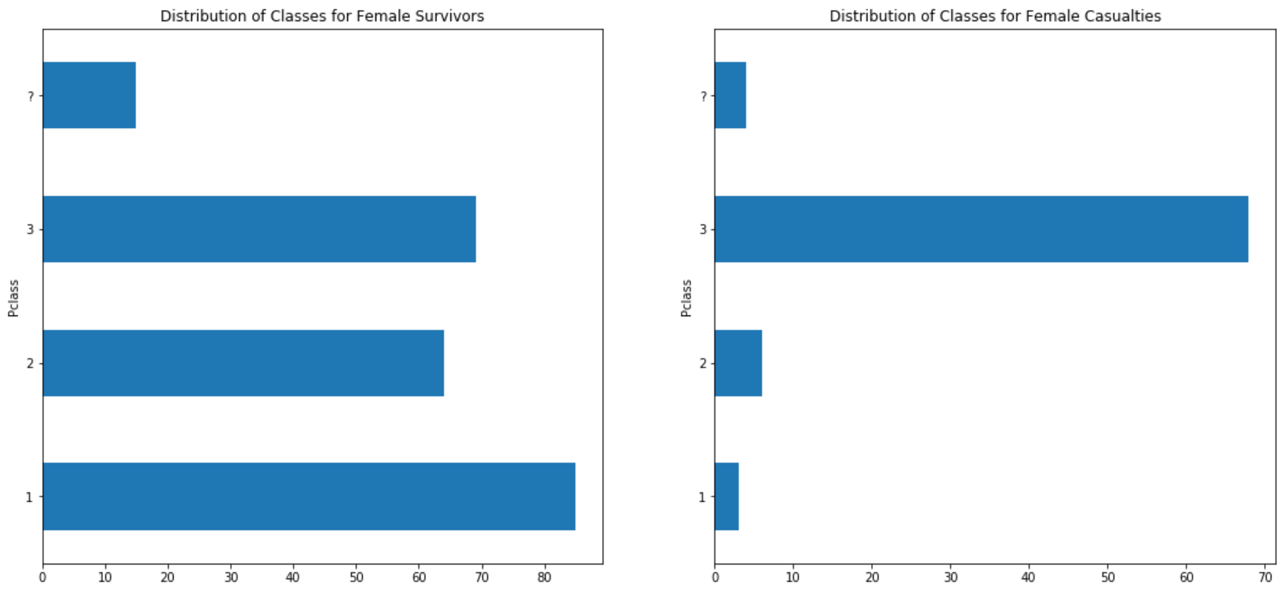
In the cell below, create 2 separate DataFrames using `pandasql`. One should contain the Pclass of all female passengers that survived, and the other should contain the Pclass of all female passengers that died.

Then, create a horizontal bar graph visualizations of the `Pclass` column for each DataFrame to compare the two. Bonus points for taking the time to make the graphs extra readable by adding titles, labeling each axis, and cleaning up the number of ticks on the X-axis!

```
# Write your queries in these variables to keep your code well-formatted and readabl
q3 = """SELECT Pclass, Count(*)
        FROM df
        WHERE Sex = 'female' AND Survived = 1
        GROUP BY Pclass;"""
q4 = """SELECT Pclass, Count(*)
        FROM df
        WHERE Sex = 'female' AND Survived = 0
        GROUP BY Pclass;"""

survived_females_by_pclass_df = pysqldf(q3)
died_females_by_pclass_df = pysqldf(q4)

# Create and label the histograms for each below!
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(18,8))
```

```
survived_females_by_pclass_df.set_index('Pclass')['Count(*)'].plot(kind='barh', ax=a
axes[0].set_title('Distribution of Classes for Female Survivors')

died_females_by_pclass_df.set_index('Pclass')['Count(*)'].plot(kind='barh', ax=axes[
axes[1].set_title('Distribution of Classes for Female Casualties');
```

```
Text(0.5, 1.0, 'Distribution of Classes for Female Casualties')
```



Summary

In this lab, you practiced how to query Pandas DataFrames using SQL.

Releases

No releases published

Packages

No packages published

Contributors 7

Languages

● **Jupyter Notebook** 100.0%