Week 3 Laboratory Activity

Advanced Aggregation

In this week's activity, we look at

1. More sophisticated group-by operations, and
2. Data visualisation.

Download the file `titanic.csv` from Moodle and place it in the same folder as your Jupyter Notebook. Read the file into a DataFrame called `titanic` using the pandas library and print out the first 5 rows.

```
import pandas as pd
titanic = pd.read_csv('titanic.csv')
titanic.head()
titanic.sample(5)
```

1. Data Aggregation

1.1 Multiple aggregation operations

Last week we learnt how to perform aggregation operations to compute the mean or sum on individual columns. We also learnt how to group the data according to certain attributes prior to performing the aggregation.

Oftentimes we'd like to compute multiple aggregation operations at the same time. Here is an example where we compute statistics on multiple columns at once. (Note that you could also use the method to compute different aggregation functions on the same column of data.)

We specify the set of aggregation operations we wish to perform by detailing:

- the columns the aggregation should be applied to (in this case the two columns are 'who' and 'age'),
- the aggregation operations to apply in each case ('count' and 'mean'), and
- the name of the resulting new columns (we'll call them 'passengers' and 'average age')

```
fun = {'who':'count','age':'mean'}
```

Note that the variable `fun` can be anything you want to call it. So now we've defined an aggregation operation that both counts the number of passengers and computes their average age. Let's apply that operation to the rows in the titanic table, grouping them by passenger 'class'. To apply the operation we use the '`agg()`' function:

```
groupbyClass = titanic.groupby('class').agg(fun)
groupbyClass
```

Have a look at the output, which has now been grouped by passenger class. Which class had the most passengers and which one had the oldest passengers on average?

Do note that the columns seem to have 2 levels and also we have not renamed the columns. Let’s try to do this in one single operation, using the renaming() function and to flatten the columns, we reset the index.

```
groupbyClass = titanic.groupby('class').agg(fun)
groupbyClass.rename(
    columns={"who":"passengers", "age":"average age"},
    inplace = True
)
groupbyClass = groupbyClass.reset_index()
groupbyClass
```

***Practice 1a**: Modify the aggregation operation 'fun' above so that it also finds the age of the oldest and youngest passengers in each class. Note that all aggregate operations being applied to the same column need to be placed within the same set of curly braces '{}' and separated by commas ','. So fill in the [MISSING] parts of the function and also attempt to rename the columns:*

```
fun2 = {'who':'count','age':{'mean', [MISSING], [MISSING]}}
groupbyClass2 = titanic.groupby('class').agg(fun2)
# It’s incomplete, add in the renaming here (or later)
groupbyClass2
```

***Practice 1b**: So was the oldest passenger traveling in 'first', 'second' or 'third' class?*

In order to turn the output of the groupby operation into a DataFrame that can be further manipulated, we need to "flatten it" using the '`reset_index()`' and '`droplevel()`' commands. Have a look at the outputs of the following commands one after the other (by printing out the table each time) to see what they produce.

```
groupbyClass2 = groupbyClass2.reset_index()
# turn 'class' groups into column values
groupbyClass2.columns = groupbyClass2.columns.droplevel(0)
# drop the top level in the column hierarchy
groupbyClass2
```

Flattening caused us to lose the column name for the 'class' attribute. We can rename the column as follows:

```
groupbyClass2.rename(columns = {'':'class'},inplace = True)
# rename the first column to be 'class'
groupbyClass2
```

1.2 Custom aggregation operations

There are many inbuilt functions in Python that can be used to aggregate data over columns. For example the '`nunique`' function will count the number of unique values in a list.

Sometimes the function we need isn't available, however, because what we are after is too specific. For example, if we have a list of values, we might wish to count only those elements in the list with value above a certain threshold. Using the '`for`' syntax in Python we can write an expression to count the elements as follows:

```
my_list = (80,20,64,19,56,12,88)
sum(e>50 for e in my_list)
```

The expression is checking for each element 'e' in 'my_list' whether the value is greater than 50 or not, the sum() function is then counting the number of times the greater-than expression returns `TRUE` (i.e. the value 1).

Now that we have a piece of code that can count the number of values that fit a condition, we'd like to use it in an aggregation operation over a column of a DataFrame. We can do that using an anonymous function (called a lambda function) in Python. The syntax to create an anonymous function is to write '`lambda x:`' followed by the function itself, where '`x`' is the name of the variable that appears in the function:

```
fun3 = {'age':{'nunique',lambda x: sum(e>50 for e in x)}}
```

So we have defined a new aggregation operation '`fun3`', which will create two new columns, a column that counts the number of distinct values in the 'age' column using the function '`nunique`' and another column that counts the number of values in the 'age' column that are greater than 50.

Again, we can reformat the DataFrame so it's ready for use:

```
groupbyClass3 = titanic.groupby('class').agg(fun3).reset_index()
# turn groups into column values
groupbyClass3.columns = groupbyClass3.columns.droplevel(0)
# drop the top level in column hierarchy
groupbyClass3.rename(
    columns = {
        '':'class',
        'nunique':'unique age count',
        '<lambda_0>':'over 50s count'},
    inplace = True
)
groupbyClass3
# print out the table
```

***Practice 2**: Interpret the output and discuss your findings with other students.*

2. Data Visualization

In order to plot data in Python, we use the '`matplotlib`' library:

```
import matplotlib.pyplot as plt
```

Reminder: when using Python in a Jupyter Notebook, you need to add also the following 'magic line' to make sure that graphs are shown inline in the notebook.

```
%matplotlib inline
```

2.1 Basic Plots

We'll continue to use titanic data set and let's call the 'plot' routine to have a look at the data:

Note: This part is different from the Tutorials on Moodle, it is just extra.

```
plt.plot(titanic.fare)
plt.show()
```

The figure looks a little complicated, but it is just plotting the fare for each passenger.

***Practice 3:** How many passengers were there in total?*

2.2 Histogram

More informative in this case would be to look at the distribution over fares. We can visualise the distribution by plotting a histogram.

```
titanic.fare.hist(bins = 200)  # try different numbers of bins
plt.xlim(0,300)                # setting limit on x-axis
plt.ylim(0,300)                # setting limit on y-axis
```

Practice 4: *Reduce the x-axis limit to see how much most people paid to go on the titanic. Approximately how many people paid 10 or less?*

2.3 Boxplot

Alternatively, we can use a boxplot (also called a box and whisker diagram) to visualise the same data. A boxplot is a simple visual representation of key features of a univariate sample. It displays five-point summaries and potential outliers in graphical form. To create a boxplot we call:

```
titanic.boxplot(column = 'fare')
plt.ylim(0, 600) # setting limit on y-axis
```

The red (maybe other colours depending on your setup) line across the centre of the box indicates the median value, i.e. half the data lies below the red line and half lies above it. The box itself defines the quartiles -- one quarter of the data lies above the box, and another quarter below it. We can see many high 'fare' values to the top of the graph. One might assume they are outliers, but it probably makes more sense to first investigate the different classes. We can generate boxplots divided by class, as follows:

```
titanic.boxplot(column = 'fare', by = 'class')
plt.ylim(0, 600)
```

If we wanted to, we could filter out the large values in the different classes. For example, to filter out values greater than 160 in first class:

```
filt = ~((titanic['class'] == 'First') & (titanic['fare'] > 160))
titanic = titanic[filt]
titanic.boxplot(column = 'fare', by = 'class')
plt.ylim(0, 600)
```

Practice 5: *Use the same technique to filter out values greater than 50 for the second class and 30 for the third class. Plot the boxplot, and observe the graph. What is the median price for each class? Hint: set a lower y-axis limit to see clearer.*

2.4 Bar Chart

We can compare the fare for different classes and for children/adults using a bar chart.

Practice 6a: *Fill in the missing code to make the aggregation function below count the number of children (age under 18) and adults (age 18 or over) in the different classes*

```
# Tutorial on Moodle uses lambda functions, I am using a normal
# function here in case you feel "challenged" by the lambda function :)
```

```
fun_child_adult = {'age':{[MISSING], [MISSING]}}

groupbyClass2 = titanic.groupby('class').agg(fun_child_adult)
groupbyClass2

# The column names will be meaningless, so you may want to rename them.
groupbyClass2.rename(
    columns = {
        '<lambda_0>':'child count',
        '<lambda_1>':'adult count'
    },
    inplace = True)
groupbyClass2
```

***Practice 6b**: Now follow the steps from Section 2.2 to group the 'titanic' data by class, and apply the above aggregation function to it. Call the resulting DataFrame 'groupbyClass2' and display it:*

```
[MISSING LINE]
[MISSING LINE]
[MISSING LINE]
groupbyClass2
```

We'll now display the aggregated counts as a bar chart.

```
ax=groupbyClass2.plot.bar(figsize=(8,5))
# figsize sets size of plot
ax.set_xticklabels(groupbyClass2['class'],rotation=45)
# use values of column 'class' as the x axis labels. Remove this line
of code to see what will happen if we do not have this line
plt.xlabel('Ticket Class')
# setting a label for x axis
plt.ylabel('Number of Passengers(child or adult)')
# Setting a label for y axis
plt.title('Passengers ticket class based on their adulthood')
# Setting the title of chart
```

***Practice 7**: So which class had the most families do you think?*

2.6 Pie Chart

***Practice 8**: Use the groupbyClass2 below to plot a pie chart, showing the number of children in each passenger class ('child count' column).*

```
plt.pie(groupbyClass2['MISSING'])
plt.show()
```

When should we use a Pie chart?

2.7 Scatter Plot

Input a simple data frame.

```
df = pd.DataFrame({
    'Name' :
['Mike','Aaron','Brad','Steve','George','Mitchell','Shaun','Glenn','Pa
t','Robert','David'],
    'Age' : [39,28,44,25,32,33,31,26,22,25,28],
    'Runs' :[1310,662,1403,828,672,1140,655,1040,557,1030,1140]
})
df
```

Let's have a quick look at the data by plotting it using an x-y scatter plot:

```
plt.scatter(df['Age'], df['Runs'])
plt.show()
```

***Practice 9**: We now have two views of the same data, the table (DataFrame) view and the plot. What information do you gain/lose in these different views? When should we use a scatter plot?*

End of Tutorial 3