**3. More data auditing**

Load the titanic data ([titanic.csv](http://moodle.vle.monash.edu/mod/resource/view.php?id=4348541)) into a pandas DataFrame called ‘titanic’ using Pandas and print out the first 5 rows:

import pandas as pd  
titanic = pd.read\_csv('titanic.csv')  
titanic.head()

**3.1 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:

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

fun = {'who':{'passengers':'count'},'age':{'average age':'mean'}}

So now we've defined a 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?

**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 below:

fun = {'who':{'passengers':'count'},'age':{'average age':'mean', **[MISSING]**, **[MISSING]** }}

**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.

groupbyClass = groupbyClass.reset\_index() *# turn 'class' groups into column values*groupbyClass.columns = groupbyClass.columns.droplevel(0) *# drop the top level in the column hierarchy*

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

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

**3.2 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:

fun = {'age':{'unique age count':'nunique','over 50s count':**lambda** x: sum(e>50 for e in x)}}

So we have defined a new aggregation operation 'fun', which will create two new columns, a 'unique age count' column that counts the number of distinct values in the 'age' column using the function 'nunique' and a 'over 50s count' 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:

groupbyClass = titanic.groupby('class').agg(fun).reset\_index()  *# turn groups into column values*groupbyClass.columns = groupbyClass.columns.droplevel(0) *# drop the top level in column hierarchy*groupbyClass.rename(columns = {'':'class'},inplace = **True**) *# rename the first column*groupbyClass # print out the table

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

**4. Basic plotting**

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

**import** matplotlib.pyplot **as** plt

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

**4.1 Simple plot**

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

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?

**4.2 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?

**4.3 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 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]

​​**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.

**4.4 Barchart**

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.

fun = {'age':{'child count': **[MISSING]**, 'adult count': **[MISSING]** }}

**Practice 6b**: Now follow the steps from **Section 3.2** 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.

Generating a bar chart is a little more involved than generating the previous plots since we need to layout the bars ourself. Before we can generate the chart, we'll create a parameter 'ind' containing the position of the bars:

**import** numpy **as** np  
ind = np.arange(0, 3**\***2, 2)  
width = 0.5      *# the width of the bars*

Note that we used the 'numpy' library just to make it easier to create the index array we needed. What is the value of 'ind'?

​Now that we have the parameters we need we can build the bar chart itself:

fig, ax = plt.subplots()  
rects1 = ax.bar(ind, groupbyClass2['child count'], width, color='g')  
rects2 = ax.bar(ind **+** width, groupbyClass2['adult count'], width, color='c')

The chart looks nice, but needs some labels! We can labels to the y-axis, a title and a legend as follows. (Note that you'll need to copy the following code into the same code input window as the graph above.)

ax.set\_ylabel('Passenger Count')  
ax.set\_title('Child vs Adult Passengers for Different Classes')  
ax.legend((rects1[0], rects2[0]), ('Child', 'Adult'))

Finally, we can fix the text on the x-axis as follows. (Again, copy the code into the previous code window.)

ax.set\_xticks(ind **+** width**+**0.1)  
ax.set\_xticklabels(groupbyClass['class'],rotation='vertical')

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

**5 Basic regression**

Input a simple data frame.

df = pd.DataFrame({'Name' : ['Mike','Aaron','Brad','Steve','George','Mitchell','Shaun','Glenn','Pat','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 8**: 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?

We will use a standard package for generating linear regressions in Python:

**from** scipy.stats **import** linregress

Calling the 'linregress()' function provides us with the slope and intercept of the linear fit through the data:

slope, intercept, r\_value, p\_value, std\_err = linregress(df['Age'],df['Runs'])

With the slope and the intercept we can use the 'for' syntax to compute the line (a list of y-values corresponding to the input x-values)

line = [slope**\***xi **+** intercept **for** xi **in** df['Age']]

We can then plot the 'line':

plt.plot(df['Age'],line,'r-', linewidth=3)

And add the original data points to the same plot:

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