Jupyter Notebook Demonstration: Introduction to Statistical Graphics

In the following video demonstration, you will see a walk-through and detailed explanation of the provided Example Jupyter Notebook. This Notebook loads a database of cereals and various properties about the cereals and then generates graphs of these properties using the Matplotlib library.

Here, you will step through the example code and explain each block's function.

Requirements

In order to successfully run the code provided in this example, two additional packages need to be installed:

- ***matplotlib***, a graphing package,
- ***pandas***, a data-processing package.

On OSX, these packages can be installed by entering the following command in the terminal:

pip3 install matplotlib pandas

If you have already started Jupyter Notebook, you must restart the server for the packages to be picked up.

On Windows, these packages can be installed by opening the Anaconda Prompt application that was installed with Juypter Notebook and running the command:

conda install matplotlib pandas

Graphs

The four graphs generated in this example are:

1. Pie Chart
2. Bar Chart
3. Histogram
4. Box-and-Whisker Plot

Cells

In the Notebook provided for this example, the code to generate each of the graphs is in its own cell. This allows you to generate one graph at a time and modify one graph's code without affecting the others. In order to set up the graphing environment, you must run the code in the first cell each time you open the notebook and before attempting to run any graph code.

Part 1: Import Libraries and Establish Database Connection

Code:

```
import sqlite3
import pandas as pd
import matplotlib.pyplot as plt
db_filename = 'cereals.db'
conn = sqlite3.connect(db_filename)
c = conn.cursor()
```

Explanation: The above block of code includes the libraries we will need for the rest of the example. In this example, we need the sqlite3 and pandas libraries and parts of matplotlib library, which will be used for graphing. In addition, we load the database and create a cursor so we can extract the data to be graphed.

Part 2: Graph 1 - Generate data to be graphed

Code:

```
c.execute("SELECT Manufacturer,count(*) FROM cereals GROUP BY
Manufacturer")
counts = c.fetchall()
```

Explanation: This cell generates data to be graphed by asking the database to count the number of records for each manufacturer.

Part 3: Graph 1 - Convert data to be graphed to a DataFrame

Code:

```
manuStats = pd.DataFrame.from_records(counts,
columns=['manufacturer','value'])
```

Explanation: In this line, the data to be graphed is converted to a DataFrame from the records returned by the database. This pseudo-tabular format is used by the matplotlib graphing functions. Using from_records() is an easy way to construct this format from database records.

Part 4: Graph 1 - Plot the Pie Chart

Code:

```
plt.pie(manuStats['value'], labels=manuStats['manufacturer'], shadow=False)
plt.axis('equal')
plt.show()
```

Explanation: In the first of these lines, the graph is created. Specifying the ‘value’ column from the DataFrame created earlier indicates which column of data to graph. Similarly, the ‘label’ column applies a label to each slice of the pie. Finally, the plt.show() function renders the graph in the notebook.

Part 5: Graph 2 - Generate data to be graphed and convert to a DataFrame

Code:

```
c.execute("SELECT Cereal,Sugars FROM cereals")
sugars = c.fetchall()
sugarFrame = pd.DataFrame.from_records(sugars,columns=['Cereal','Sugar'])
```

Explanation: The above cell generates data to be graphed. These lines request pairs of sugar content and cereal name from the database and format the data to be displayed in a matplotlib graph

Part 6: Graph 2 - Plot the Bar Chart

Code:

```
plt.bar(range(len(sugarFrame['Sugar'])),sugarFrame['Sugar'])
plt.xticks([])
plt.show()
```

Explanation: The above lines create and then render a graph in the notebook. In this case, the graph is a bar graph showing values of the 'Sugars' field for each of the named cereals.

Part 7: Graph 3 - Generate data to be graphed and convert to a DataFrame

Code:

```
c.execute("SELECT Sugars FROM cereals")
sugar = c.fetchall()
sugarFrame = pd.DataFrame.from_records(sugar,columns=['Sugar'])
```

Explanation: This cell generated data to be graphed. In this case, the query requests only the 'Sugars' values from each record to the database. The last line formats the data for use with pandas

Part 8: Graph 3 - Plot the graph

Code:

```
plt.hist(sugarFrame['Sugar'], bins=9)
plt.show()
```

Explanation: In the above cell, we generate the actual sugar values Histogram graph. The matplotlib package has a built-in histogram that can perform all of the histogram computations, but here we specify the number of bins to comply with the √N rule. The final line renders the graph in the notebook.

Part 9: Graph 4 - Generate data to be graphed and convert to a DataFrame

Code:

```
c.execute("SELECT Manufacturer,Sugars FROM cereals")
sugarByMan = c.fetchall()
sugarBoxFrame=pd.DataFrame.from_records(sugarByMan,columns=['Manufacturer', 'Sugar'])
```

Explanation: This cell generates the data required to plot the box-and-whisker plot. In this case, the query lists all the sugar contents from the dataset with the addition of manufacturer labeling from each record. The last line reformats the data using pandas

Part 10: Graph 4 - Plot the graph

Code:

```
plt.boxplot(sugarBoxFrame['Sugar'])
plt.show()
```

Explanation: Here, we use the matplotlib built-in 'boxplot' function. In order for the function to provide accurate results, we must indicate the feature of the data used to group the sugars, manufacturer. The function then computes the quantiles. The file line renders the plot in the notebook.