Dominic Pancella

Food for Thought
How do you clean a dataset using the pandas library?
What are the different kinds of joins, and when would you use them?
How do you filter only the specific subset of rows that you want to analyze?
Does pandas allow you to pipeline commands?
What sorts of summary statistics are available in pandas?

Introduction
I'm going to be looking at two datasets containing basic statistics about the countries of the world. They are contained in an Excel file (.xlsx), so the first thing I'm going to do is pull them into Python as pandas dataframe objects. That way I can use functions from the pandas library to clean them, merge them, add a couple columns, and maybe perform some surface level analysis and observation on them using filters and descriptive statistics.

Step 1: Load the Data
The Excel file I am using happens to be in the same folder as the Jupyter Notebook I'm writing and testing my code in. Therefore I don't have to put anything but the file name as the path parameter (called *filename* in the xlrd module). I can't use the *read_excel* function from pandas directly because it only supports the .xls extension, which was replaced by .xlsx in Excel 2007. A nice workaround is to use the xlrd module in tandem with pandas.

```
import pandas as pd
import xlrd

xlsx = xlrd.open_workbook("CSC 357 Week 1 Lesson.xlsx", on_demand=True)
with pd.ExcelFile(xlsx) as wb:
   basic_info = pd.read_excel(wb, "basic_info")
   birth_death_rates = pd.read_excel(wb, "birth_death_rates")
```

Step 2: Browse the Data
Once I have the data loaded into two pandas dataframes, I want to see what each one looks like, and I want to begin the process of cleaning them. I can use the *head*, *info*, and *describe* functions to figure out what my data looks like.

```
basic_info.head(5)
birth_death_rates.head(5)

basic_info.info()
birth_death_rates.info()

basic_info.describe()
birth_death_rates.describe()
```

Step 3: Clean the Data
The *basic_info* dataset appears to have no missing or incorrect values in any column. However, the *birth_death_rates* dataset has three fewer values in both the *Birth Rate* and *Death Rate* columns than in the *Country* column. If my assumption is correct that there are three countries with missing data, I'd like to get rid of those rows outright so they don't affect my analysis.

```
birth_death_rates = birth_death_rates.dropna()
birth_death_rates.info()
```

Step 4: Merge the Data

As it turns out, my assumption was correct: there were indeed three countries with missing values. Fortunately, the data was relatively clean to begin with—no weird outliers or inconsistent formatting—so I can move on to merging the two datasets together. Because I had to remove some rows from one dataset, I want to perform an inner join that will keep only rows that are included in both datasets. (Note that I could have merged them first, then cleaned the single merged dataset; the order does not matter here.)

```
combined_data = pd.merge(basic_info, birth_death_rates, how="inner", on="Country")
combined_data.head(5)
combined_data.info()
```

Step 5: Rename Columns

Once the data has been merged into a single dataframe, I notice that some column names have spaces in them, so I want to rename them accordingly.

```
combined_data = combined_data.rename(columns = {"GDP (millions)":"GDP_millions"})
combined_data = combined_data.rename(columns = {"Birth Rate":"Birth_rate"})
combined_data = combined_data.rename(columns = {"Death Rate":"Death_rate"})
```

Step 6: Add Columns

My combined dataset has five columns with numerical values: Population, Area, GDP (millions), Birth Rate, and Death Rate. Using common sense, as well as the fact that I was the one who compiled these datasets, leads me to believe the units of Area are square miles, the GDP is represented as millions of US dollars, and the birth and death rates for each country are expressed per 1,000 people per year. I would like to add columns for population density and GDP per capita.

```
combined_data["Density"] = combined_data.Population / combined_data.Area
combined_data["GDP_per_capita"] = combined_data.GDP_millions /
combined_data.Population * 1000000
```

Step 7: Reorder Columns

Suppose now that I want to rearrange the columns of my dataframe in a specific order. Mostly, I'm thinking of moving the columns that I added—which pandas will automatically place at the right end of a dataframe—closer visually to the values from which they were calculated.

```
combined_data = combined_data[["Country", "Capital", "Region", "Population", "Area",
"Density", "GDP_millions", "GDP_per_capita", "Birth_rate", "Death_rate"]]
combined_data.head(5)
```

Checkpoint

At this point we have gathered our data, read it into Python as pandas DataFrame objects, cleaned it, and transformed it into a single dataset that we can use for analysis. There are several places we can go from here, perhaps the most fun of which would be to muse about a potential machine learning scenario where we use this data to try and predict the birth and death rates of an unknown country based on region, population density, and GDP per capita.

If we are to proceed with such a scenario—after all, this is a machine learning class—we are not yet done altering our dataset. We need to change the region to a numerical variable and split our data into a training set and a test set, if such a thing is even possible.

Step 8: Encode Important Categorical Variables as Numeric
To avoid messing up the nice, clean dataset from above, I will create a new dataframe that we can make more changes to while still referring back to the original data. The first change I want to make is to encode the Region variable as numeric rather than categorical.

```
ml_data = combined_data
labels, uniques = pd.factorize(ml_data.Region)
ml_data.Region = labels
ml_data.head(5)
```

Step 9: Remove Extra Columns
If we are going to create a machine learning model using only certain columns from our dataset, we need to remove the columns we won't use. In this case, I only want to keep Region, Density, GDP_per_capita, Birth_rate, and Death_rate. Since there are two things we are trying to predict, I will actually create two dataframes, one containing Birth_rate and one containing Death_rate.

```
ml_birth_data = ml_data[["Region", "Density", "GDP_per_capita", "Birth_rate"]]
ml_death_data = ml_data[["Region", "Density", "GDP_per_capita", "Death_rate"]]
```

Step 10: Split Data into Training and Test Sets
The next thing we need to do with those two new datasets is split them into training and test sets using the train_test_split method from sklearn. From there we will be able to apply machine learning training algorithms to them and develop a model to predict the birth and death rates of an unknown country. Using the same value for random_state should ensure that we get the same groups of countries in our training and test datasets, which is important so that we don't skew our results.

```
import numpy as np
from sklearn.model_selection import train_test_split

birth_train, birth_test = train_test_split(ml_birth_data, test_size=0.2,
random_state=42)
death_train, death_test = train_test_split(ml_death_data, test_size=0.2,
random_state=42)
```

Conclusion
The pandas library is an incredibly useful tool for data wrangling in all forms. It works with a great variety of file formats including CSV, JSON, HTML, and XLS to store data into a user-friendly format. Its built-in functions allow users to do pretty much anything from adding a calculated column to filling in missing values to converting back and forth between rows and columns in order to get their data in the shape they want.

Resources
Pandas cheat sheet: https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf
User guide to pandas: https://pandas.pydata.org/pandas-docs/stable/user_guide/index.html
Collection of pandas tutorials: https://pandas.pydata.org/pandas-docs/stable/getting_started/tutorials.html