Lab 6: Handling Quantitative Data: Introduction to Pandas

Pandas is an open-source Python library that provides high-performance data manipulation and analysis tools. We will discuss two important Pandas data structures: **Series** and **DataFrame**.

1 Series

A **Series** is a one-dimensional array-like structure that contains an array of data and an associated array of data labels called **index**. Given a pandas Series **s**, we can access its data through **s.values** which is an ***ndarray*** object. The index of Series **s** can be accessed through **s.index**.

Example 1: Create a Series from a **list**, and access its data. If you don't specify the index, it will use default indexing that begins with integer 0 and grows up to len(array) – 1.

```
import pandas as pd

s1 = pd.Series([15, 9, -20, 36, 72])

print(s1)

print(s1.values)

print(s1.index)

for i in s1.index:
    print(s1[i])

s1.index = ['a', 'd', 'e', 'b', 'c']

print(s1)

print(s1.index)

for i in s1.index:
    print(s1[i])
```

In [3]: print(s1)

```
0   15
1    9
2  -20
3   36
4   72
dtype: int64
```

In [4]: print(s1.values)

```
[ 15   9 -20  36  72]
```

In [5]: print(s1.index)

```
RangeIndex(start=0, stop=5, step=1)
```

When you print a Series, you will see two columns. The first column shows the index, while the second column shows the values of the real data. You can change the index to other data types such as strings. **The purpose of index is to help you find data efficiently**. So it is not surprising that you can access the Series data through indexing.

Example 2: Create a Series from a **dict**, and access its data. In this case, the Series will use the dict keys as the indices.

```
sdata = {'Ohio':35000, 'Texas':71000, 'Oregon':16000, 'Utah':5000}

s2 = pd.Series(sdata)

print(s2)

print(s2.values)

print(s2.index)

for i in s2.index:

   print(s2[i])
```

In [10]: print(s2)

```
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

2 DataFrame

A Series can only store a single column of data. A **DataFrame** represents a tabular (i.e., two-dimensional) data structure that contains an ordered collection of columns (or an ordered collection of rows). You can think DataFrame as a dictionary of Series. Different from Series, DataFrame has both a **row index** and a **column index**.

We can create a DataFrame from a dictionary of equal-length lists:

Example 3: Create a DataFrame from a dictionary, and access its columns

```
data = {'state':['Ohio', 'Ohio', 'Ohio', 'nevada', 'Nevada'],

        'year':[2000, 2003, 2002, 2001, 2002],

        'pop':[1.5, 1.7, 3.6, 2.4, 2.9]}

frame = pd.DataFrame(data)

print(frame)

print(frame.index)

for i in frame.index:

   print(frame.loc[i])

print(frame.columns)
```

In [20]: print(frame)

```
   pop    state  year
0  1.5     Ohio  2000
1  1.7     Ohio  2003
2  3.6     Ohio  2002
3  2.4   nevada  2001
4  2.9  Nevada  2002
```

```
for i in frame.columns:

   print(frame[i])
```

The DataFrame will use the keys of your dictionary as the column indices, and the value (a list) of each key as the data items in each column. To access a column in a DataFrame, we can use dict-like notation **DataFrame[column_index]**.

In order to access a row of a DataFrame, we need to use the notation **DataFrame.loc[row index]**.

3 A Journey of Data Analysis by Pandas

Acknowledgement: the materials of this section mainly come from the book "Python for Data analysis" written by Wes McKinney, who is also the author of Pandas.

In this section, we are going to use a real dataset, the **US Baby Names** from 1880 to 2016, to show how we can perform data analysis on a large set of data files. It includes some advanced features of Pandas, so you are not expected to understand all the technical details. **The motivation of this sub-section is to show you what can be done by Pandas and how easy it will be.**

The data file comes from https://www.ssa.gov/oact/babynames/names.zip. After unzip, you should see more than one hundred .txt files with file names "yobxxxx.txt". You can open any file using a text editor to take a glance on the data set. The below description is quoted form the NationalReadme.pdf in the unzipped folder:

- Each record in the individual annual files has the format "name,sex,number," where name is 2 to 15 characters, sex is M (male) or F (female) and "number" is the number of occurrences of the name.

This kind of data files are known as Comma Separated Values (CSV) files. If you want to analyse these data files using Excel, it will be very tedious and time consuming. But we can easily handle this task by Python.

Example 4: Try the following Python statements to create a DataFrame from a CSV file:

```
import numpy as np

import matplotlib.pyplot as plt # used for visualisation

columns = ['name', 'sex', 'births'] # used as column indices

names1880 = pd.read_csv('names/yob1880.txt', names =columns)

print(names1880)
```

First, we import two other Python libraries, NumPy and Matplotlib, that will be used later. Then we create a list of strings to represent the column names of our data set. Since the data files are already in the format of CSV, we can use **DataFrame.read_csv()** method to create a DataFrame by loading a CSV file. At the same time, we specify the column indices. Finally we print out the DataFrame. Usually it only shows the beginning and end parts of the DataFrame.

We can perform some simple **data analysis** on this DataFrame. E.g., what is the total number of Female births and Male births in Year 1880, respectively? This question can be answered by a typical data analysis technique called **GroupBy**. A GroupBy process usually involves one or more of the following steps:

- **Splitting**: split the data into groups based on some criteria
- **Applying**: apply a function to each group independently
- **Combining**: combine the results into a single data structure

In the names1880 DataFrame, we have males and females. What we need to do is to split the data sets according to the **sex** column, and we will have two groups: one group of all boys and another group of all girls. For each group, we want to find its total number of births. Once you get this idea, you can implement it using the **DataFrame.groupby()** method:

```
grouped = names1880.groupby('sex')

grouped.births.sum()  # please also compare with names1880.births.sum()

grouped.births.max() # please also compare with names1880.births.max()

grouped.births.min() # please also compare with names1880.births.min()

grouped.births.count() # please also compare with names1880.births.count()
```

You will see the following output:

```
In [33]: grouped.births.sum()
Out[33]:
sex
F       90993
M    110491
Name: births, dtype: int64
```

Analysing the data of a single year may not be that challenging. The real power of Python is to easily handle many data files easily. E.g., how can we aggregate all the data during 1880-2017? Let's try the following codes:

Example 5: Data aggregation: load data from multiple CSV files:

```
years = range(1880, 2018)
pieces = []
for year in years:
   path = 'names/yob%d.txt' % year  # find the path and file name of a year
   frame = pd.read_csv(path, names=columns) # columns has been defined in Example 4
   frame['year'] = year # introduce a new column 'year'
   pieces.append(frame) # add frame into a list
names = pd.concat(pieces, ignore_index=True) # concatenate multiple frames into a single one
print(names)
names.to_csv('names/yob.txt') # store the frame to yob.txt
```

In the statement "path = 'names/yob%d.txt' % year", we use **%d** inside the string to indicate a placeholder, which will be replaced by the value **year** after the percent sign **%**.

In the original data files, there is no explicit information about the year of birth. But when we aggregate all the data, we need to include the year information. This is done by the statement **frame['year'] = year** which includes a new column (with index 'year') into the DataFrame. Every time we load a CSV file, we store the data into a DataFrame and append it to a list **pieces**. At the end, we concatenate all the DataFrames in **pieces** as a single large DataFrame **names** with more than 1.8 million rows, which cannot be handled by my Excel! You can save the aggregated data in a new file, say **yob.txt**, through method **DataFrame.to_csv()**.

Given this huge amount of data, we are able to do more data analysis. The first question is to find out the total number of male births and female births in every year. This time we show a new technique called **pivot table**. A pivot table is usually created by summarizing data from a given source table. For example:

Example 6: Pivot table

```
df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo", "bar", "bar", "bar", "bar"],
                   "B": ["one", "one", "one", "two", "two", "one", "one", "two", "two"],
                   "C": ["small", "large", "large", "small", "small", "large", "small", "small", "large"],
                   "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
```

```
print(df)
p1 = df.pivot_table(values='D', index=['A'], columns=['C'], aggfunc=sum)
print(p1)
p2 = df.pivot_table(values='D', index=['C'], columns=['A'], aggfunc=sum)
print(p1)
```

In function **df.pivot_table()**: the parameter **values** specifies which column should be aggregated in the pivot table by the function specified by parameter **aggfunc**, the parameter **index** specifies which column(s) is the pivot table index, the parameter **columns** specifies which keys are the columns in the pivot table.

The output looks like:

```
Out[259]:
     A    B      C    D
0  foo  one  small    1
1  foo  one  large    2
2  foo  one  large    2
3  foo  two  small    3
4  foo  two  small    3
5  bar  one  large    4
6  bar  one  small    5
7  bar  two  small    6
8  bar  two  large    7
Out[260]:
C    large  small
A
bar     11     11
foo      4      7
Out[261]:
B      one  two
C
large    8    7
small    6   12
```

Example 7: Pivot table for US Baby Name

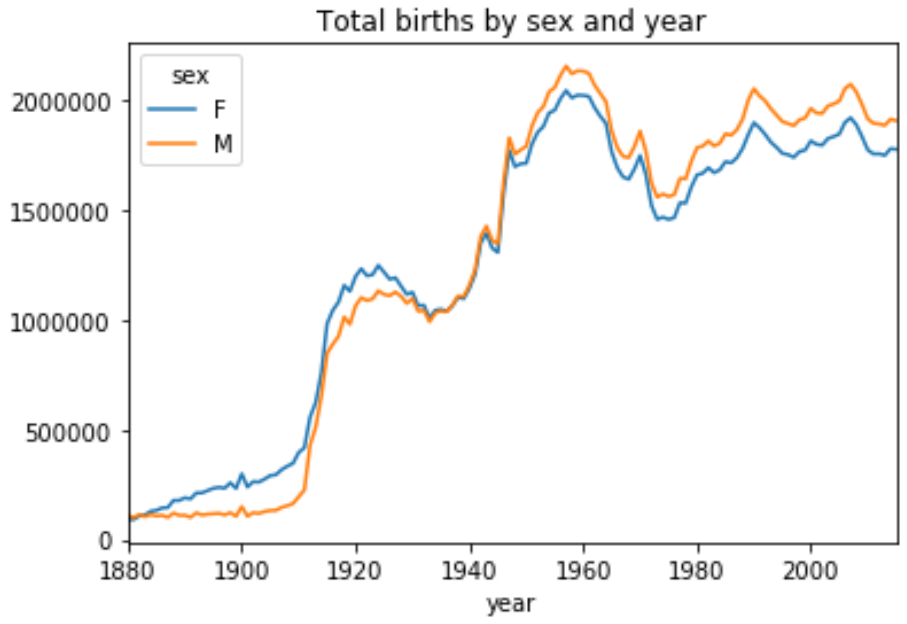
```
import matplotlib.pyplot as plt

print(names.index)

print(names.columns)

total_births = names.pivot_table(values='births', index='year', columns='sex', aggfunc=sum)

print(total_births)

total_births.plot(title='Total births by sex and year')
```

The important statement is:

- total_births = names.pivot_table(values='births', index='year', columns='sex', aggfunc=sum)

The resulted pivot table **total_births** has 136 rows because there are 136 different years, and two columns because there are two types of sexes: F and M. Each data in the pivot table is the summation of the values that match the indices of 'year' and 'sex'. We can visualize the data using the **plot()** method provided by Matplotlib.

You should be able to see the following figure on your screen:



To learn more about Pandas, please try the following series of tutorials:

https://www.tutorialspoint.com/python_pandas/index.htm