

Week 7 - Pandas, SQL

Data Handling

What are the ways that we have learned so far to handle data?

- Flat Files
- Lists of lists
- Dictionaries

None of these are particularly conducive to data exploration and quick manipulation

Introducing Data Frames

When we want to manipulate data in a clean and efficient manner, we want to start thinking about data in terms of vectors:

- Each variable can be considered a vector
- Operations on a variable can be applied to all observations uniformly
- We can quickly reduce the number of variables for specific questions

Introducing Data Frames

In Python, the `pandas` library contains the necessary code to begin working with Data Frames. It is dependent on many functions in the `numpy` library.

```
import pandas as pd # Import the library for use
```

Creating a Data Frame

Create an empty Data Frame:

```
data = pd.DataFrame()
```

A Data Frame is a class that accepts the following parameters:

- `data`
- `index` (for referencing individual rows)
- `columns` (so you can name your variables)
- `dtype` (specify the **kind** of data for each column/variable)
- `copy` (whether or not the data should be copied)

Creating a Data Frame

We can also use pandas to easily read many types of files, and import them as Data Frames:

```
# CSV
data = pd.read_csv(your_filename_here.csv)
# or Excel Files
data = pd.read_excel(your_filename_here.xlsx)
# or Stata Data
data = pd.read_stata(your_filename_here.dta)
# or SAS Data
data = pd.read_sas(your_filename_here.sas7bdat)
# or SQL Queries
data = pd.read_sql(your_query_here, your_connection_here)
# and many others!
```

Referencing a Single Column

To access a list of all of the column names in your Data Frame:

```
data.columns
```

To then reference a single column:

```
data['Column_Name']
```

To reference several columns, pass a list of column names:

```
data[['Column1', 'Column2']]
```

Slicing the Data Frame

Two selection (or slicing) tools allow us to quickly subset our data.

```
data.iloc[row_selection, column_selection]
```

With the `.iloc` method, we can provide **integer**-based selections, or choose to select all rows or columns, and only subset on a single dimension.

```
data.iloc[:, 0] # Selects all rows, and first column
```


Slicing the Data Frame

Two selection (or slicing) tools allow us to quickly subset our data.

```
data.loc[row_selection, column_selection]
```

With the `.loc` method (now with no `i`), we can provide **name**-based selections, choose to select all rows or columns, and create subsets based on conditions.

```
data.loc[:, 'ColumnName'] # Selects all rows, one column
```

Slicing the Data Frame

Two selection (or slicing) tools allow us to quickly subset our data.

```
data.loc[row_selection, column_selection]
```

With the `.loc` method (now with no `i`), we can provide **name**-based selections, choose to select all rows or columns, and create subsets based on conditions.

```
data.loc[data['Column1'] == some_value, :]  
# Selects only the observations (rows) where the  
# condition is met
```

Transforming our Data

We can quickly transform the data in a given column using the slicing techniques from above:

```
# Log the values of a variable
data.loc[:, 'Column1'] = np.log(data['Column1'])

# Difference two variables
data['newColumn'] = data['Column1'] - data['Column2']
# Because the variable doesn't exist yet, we don't use
# the .loc syntax here
```

Transforming our Data

We can choose an index from among our columns, instead of the arbitrary ascending numbers assigned by default:

```
data.set_index('transaction_id')
```

Or, we can establish a multi-level index by passing a list of columns:

```
data.set_index(['year', 'month', 'day'])
```

Remember! Indices should be unique values!

Transforming our Data

Processing Datetimes is also easy with built-in Pandas functionality:

```
data['myDate'] = pd.to_datetime(data['stringDateColumn'],  
    format = '%Y%m%d', # Need to indicate the correct  
    errors = 'ignore') #    format for your data!
```

We can also parse the data into separate columns afterward:

```
data['week'] = data['myDate'].dt.week  
data['day'] = data['myDate'].dt.day
```

Date Processing

A full list of the ways you can process dates is available at

<https://pandas.pydata.org/pandas-docs/stable/api.html#datetimelike-properties>.

Cleaning Data

There are many operations that are not reasonable to perform with missing data. Any numeric transformation will fail to provide useful output where missing values exist.

```
# Resolve missing values in ALL columns at once
data.fillna(0, inplace = True)
# fills ALL missing values, overwrites original data

# Resolve missing values in single column
data['Column'].fillna(method='pad') # fill values forward
# We can use method 'backfill' to use the NEXT value,
# and fill backwards
```

Generating Summary Statistics

Using the `describe` function to create summary tables easily, and can even export them to csv for use in reports.

```
data.describe()
```

If we want the table presented similar to academic journal formats, we can add a few arguments:

```
data.describe().T[['count', 'mean', 'std', 'min', 'max']]  
# We need to transpose the data using .T before  
# we can select the descriptive stats we want to keep  
# Add a .to_csv('myfile.csv') to that line to save
```


Using SQL with Python

In order to handle data on a large scale, we will frequently rely on SQL databases. In this class, we will practice with SQLite.

Here is a link to the documentation:

<https://docs.python.org/3/library/sqlite3.html>

Using SQL with Python

The first thing we need to do is to establish a connection to our database:

```
import sqlite3
engine = sqlite3.connect('nebraska.db')
```

We are using `mysql` via the `mysqlconnector` module. Next, we provide our `username:password`, which in this case is "viewer," with no password, so we do not enter text after the colon.

Retrieve SQL Data with Pandas

Our next step is to write a `SELECT` statement using SQL, and then to pass it to Pandas for retrieval.

```
select = """SELECT hhincome, occ2010, ind1990 FROM  
            ACS WHERE year=2016"""  
  
data = pd.read_sql(select, engine)
```

How do we write SQL Queries?

Let's take a look at some slides about writing SQL query code:

<https://goo.gl/Lq2yC5>

PandaSQL and Data Cleaning

We can actually use SQL to clean our data within Pandas by making use of the `pandasql` library.

Get started by using the following code:

```
from pandasql import sqldf
pysqldf = lambda q: sqldf(q, globals())
```

If it isn't installed, you can install the library by running

```
pip install pandasql
```

PandaSQL and Data Cleaning

```
edited_data = pysqldf(select_statement_here)
```

Using SQLite syntax, we can then clean any dataset using the same tools that we would to extract data from a database!

We can aggregate, create new columns, group, and join across datasets, just like we would with SQL.