Lesson 7: Data Collection:

**Questions for Mentor:**

* When I imported the stock exchange data using a dictionary with parameters, it didn’t pull in the data. But when I wrote out the parameters in the url, it pulled in necessary data. Should I be typing parameters into url every time?
  + Figured it out – there wasn’t data for 2018??

**Importing Data to Python:**

* Reading a text file
  + Assign file name to variable
  + Assign open method –
  + Syntax:
    - file = open(filename, mode=’r’)
    - File.close()
  + Best practice to use a with statement so you don’t have to worry about closing
    - Syntax: with open(‘huck\_finn.txt’, ‘r’) as file:
* Flat files
  + Basic text files containing records
    - Table data without distinct table constructs
  + Record = row of fields or attributes
  + Column = feature or attribute
  + Flat files can have headers
  + Importing flat files
    - Use numpy or pandas
* Import flat files using numpy
  + If all values are numerical
  + Numpy arrays are python standard for numerical data
  + Essential for other packages like scikit-learn
  + Data = np.loadtxt(filename, delimiter=’,’, skiprows=1, usecols=[0, 2], dtype=str)
  + genfromtxt()
* Importing flat files using pandas
  + Pd.read\_csv()
* Import other file types
  + Pickled files
    - Used in cases wehre there are datatypes for which it isn’t obvious how to store them
    - Pickled files are serialized
      * Converted to bytestream
  + Excel files
    - Pd.ExcelFile(file)
    - .sheet\_names selects the sheet names
    - Df.parse[‘data’] or df.parse[0] will select only that sheet name
  + SAS/Stata files
    - SAS = statistical analysis system
    - Stata = ‘Statistics’ + ‘Data’
    - SAS files used for advanced analytics
  + HDF5 files
    - Becoming standard for storing large quanitities of numerical data
    - HDF5 can scale to exabytes
    - Syntax: data = h5py.File(filename, ‘r’)
    - Structure of HDF5 file
      * Keys = meta, quality, strain
  + MATLAB
    - ‘matrix laboratory’
    - Industry standard in engineering and science
    - Can use SciPy
      * Scipy.io.loadmat()
      * Scipy.io.savemat()
    - Loads a dictionary
      * Keys = MATLAB objects
      * Values = values tied to those objects
* Introduction to relational databases
  + Tables in databases have records that represent instance of entity type
  + Tables are the same as dataframes
  + Primary key is unique value
  + Tables all linked by primary keys
  + Relational database management systems
    - Postgre SQL, MySQL, SQLite
  + Create database engine in python
    - Can use SQLAlchemy
    - Syntax: variable = create\_engine(‘sqlite:///file name.sqlite’)
    - Table\_names = engine.table\_names()
  + Querying relational databases in Python
    - Query = getting data out from the database
    - Workflow of SQL querying
      * Import packages and functions
      * Create database engine
      * Connect to engine
      * Query database
      * Save results of query to DF
      * Close connection
    - To connect syntax: con = engine.connect()
    - To execute syntax: rs = con.execute(“SELECT \* FROM Orders”)
    - To create dataframe syntax: df = pd.DataFrame(rs.fetchall())
    - To close syntax: con.close()
    - To update column names syntax: df.columns = rs.keys()
    - Can use context manager to do all this ^^ (with xxx as xx: )
    - Can filter results by using SELECT \* FROM Orders WHERE xxx > 6
    - Can sort results by using SELECT \* FROM Orders ORDER BY SupportRepId
    - Can read a SQL query in one line with pd.read\_sql\_query()
    - Advanced querying: exploiting table relationships
      * Will use a join to bring tables together
      * ‘SELECT \* FROM Orders INNER JOIN Customers on Orders.CustomerID = Customers.CustomerID’
* Importing flat files form the web
  + Import and locally save datasets from web
  + Can also import and load into pandas dfs
  + Urllib
    - Urlopen() accepts URLs instead of file names
    - Syntax: urlretrieve(url, file\_name)
  + Going to a website is senting an HTTP request – GET request
  + Urlretrieve() is a GET request
  + HTML – hypertext markup language
  + Requests.get(url)
  + HTML is unstructured and structured data together
  + To turn HTML into useful data, parse exact structured data using BeautifulSoup (python package)
  + JSONs – java scrips object notation
    - with open("a\_movie.json") as json\_file:
      * variable = json.load(json\_file)
  + What is an API?
    - Application programming interfaces
    - Set of protocols and routines for building and interacting with software interfaces
    - APIs are everywhere
    - ?t=hackers
      * Query string
      * T = title
      * Hackers = name of movie
    - APIs are used by developers by using a simple API call in software to implement complex features instead of having to code it by themselves
    - API or library?
      * An API is an interface that defines the way by which an application program may request services from the libraries
    - 5 APIs that every data scientist should know
      * Facebook API
      * Google map API
      * Twitter API
      * IBM Watson API
      * Quandl API
* Python requests – GET
  + When using an API, you can import status\_code
    - This is essentially if the request was successful or an error
  + Convert to JSON to make output an actual dictionary and you can specify which part of the dictionary you want to print (choosing the text in the translator example)

**Data Organization:**

* Ways to keep organized
  + Implementing version control;
  + Setting up a virtual environment for reproducible results;
  + Separating raw data, intermediate data, and final data;
  + Documenting work, and;
  + Separate modules for custom functions.
* Introduction to Git
  + Git status says which files have been changed since the last save and are in the ‘staging area’
  + Git diff will show you all the changes in your repository
  + Git diff directory will show you changes to the files in some directory
  + Git add Add to staging area syntax: git add ‘filename’
  + git diff -r HEAD allows you to compare state of files with those in staging area
    - HEAD is a shortcut meaning ‘most recent commit’
  + Nano is a text editor we will use
    - Allows you to open a file and make changes in it
      * Ctrl-K: delete a line.
      * Ctrl-U: un-delete a line.
      * Ctrl-O: save the file ('O' stands for 'output').
      * Ctrl-X: exit the editor.
  + Git commit saves the changes in staging area
  + Git commit -m “message” allows you to commit with a message about the changes
  + Git log shows the log of the projects history
  + Git log path shows the history of a specific file or directory where path is where we want to see the history
  + Git commit without message will launch a text editor and will allow you to write a longer message
  + git diff ID1..ID2 shows differences between 2 commits
  + .gitignore lets you stop paying attention to files
  + Git clean -f removes unwanted file that’s not being tracked
  + You should keep most of the settings for git but should always set name and email address
    - Git config --global user.name Joe
    - Git config --global user.email [boardman.joe@gmail.com](mailto:boardman.joe@gmail.com)
  + Git checkout -- filename will undo changes that have not yet been staged
    - Once changes are discarded, they’re gone forever
  + To undo changes that have already been staged, use below syntax:
    - git reset HEAD path/to/file
    - git checkout -- path/to/file
  + git reset will unstage everything
  + you can use git checkout with branch name to switch to that branch
  + git checkout -b branch-name lets you create a new branch
  + initially identical to the original, but once you start making changes, it only effects new branch
  + git merge \*source \*destination merges two branches
  + git init \*project name\* creates a repository
    - do not nest repositories
  + git clone URL where URL is the URL for the repo you want to clone
  + git pull lets you pull in from another repo or directory
  + git push pushes your changes out to a repo or directory
    - syntax: git push \*remote name \*branch name
* Data Definitions
  + Df.profile\_report() – performs quick data analysis, similar to .describe() method but more robust
    - For each column there are a number of statistics - if relevant for the column type - are presented in an interactive HTML report
* **Syntax:**
* import numpy as np
* import pandas as pd
* from pandas\_profiling import ProfileReport
* df = pd.DataFrame(
* np.random.rand(100, 5),
* columns=["a", "b", "c", "d", "e"]
* )
* To generate report use below syntax:
  + - profile = ProfileReport(df, title="Pandas Profiling Report")
  + Pandas Profiling Example questions:
    - Rejects constants?
    - First box in jupyter notebook just adding variable to demonstrate what the profile report can do?
    - Below report just better format inline project?
    - Notebook widgets show the same thing just more matter of factly? Less formatting?
    - Data Cleaning
  + Data hardly ever comes clean
  + Not all outliers are bad data, but some are
  + .columns attribute can be important to see unexpected issues with column names (spaces at beginning or end)
  + .value\_counts() counts the unique values of each column in descending order
    - Dropna=False is sometimes used as default
  + Visual EDA
    - Histograms
      * Show number of data points in bins of values
    - Box plots
      * Great when you want to see relationship across multiple data points
    - Scatter plots
      * Show relationships between 2 numeric columns
  + Tidy Data
    1. Columns represent separate values
    2. Rows represent individual observations
    3. Observational units form tables
* Some data better for reporting, some better for analysis
  + Treatment a vs treatment b example
* Can tidy up data by using df.melt() method
  + Value\_vars= parameter allows your ro specify which columns you want to melt (i.e. treatment a and treatment b)
  + Melting turns columns into rows
* Can perform the reverse of melt with pivot()
  + Turns unique values into columns
  + Can turn data into more reporting friendly shape
* Pivot\_table() can be used if we have duplicate values
  + Can use aggfunc() method to turn dupes into single lines
* Concatenating data
* Data won’t always come in the same file
* Important to be able to combine
* Pass the concat function a list of dataframes
  + Pd.concat([df1, df2])
* Ignore\_index=True will reset the index upon concatenating
* To concatenate columnwise (add columns to the right), use axis=1 argument
* Finding files with globbing
  + Globbing is pattern matching for file names
  + Wildcards
    - \* means any character
    - ? is any single character
  + For loop and iterate through all file names being read in
  + Can then use the list that was created through for loop to add to concat function
* Can also combine data using merge
  + Merge similar to SQL which allows you to merge tables with a specific key
  + Pd.merge(left=df1, right=df2, left\_on=left\_column, right\_on=right\_column)
  + Can have one-to-one merge or one-to-many/many-to-one
  + Data Types
    - .dtypes checks the data types of a dataframe
    - Category dtype can make the DF smaller in memory
    - To\_numeric will convert a column to numeric
      * Need to use errors=’coerce’ to turn the bad data value into NaN value otherwise it will return an error
    - Most data cleaning is string manipulation
    - Built in suites for string manipulations
      * re library for regular expressions
      * allows us to pattern match with regular expressions
      * \d\* matches an arbitrary number of digits
      * With dollar sign, use \$\d\* which allows us to escape the dollar sign
      * With decimals, use \$\d\*\.\d\* which escapes the dollar sign and the period
      * With money, use \$\d\*\.\d{2} which escapes the dollar sign and the period and specifies only 2 decimals because it is money
      * With money and more than 2 decimals use \$\d\*\.\d{2}
      * Need to compile pattern with pattern = re.compile(format from above)
      * Then can match using pattern.match(string we want to match)
    - Some cleaning steps take multiple steps
    - Can use functions for these cases
      * When applying functions using .apply() method, use axis=1 to do calculation row-wise
    - Duplicates can ruin data
      * Df.drop\_duplicates() will drop dupes
    - Missing data can complicate as well
      * Can leave as is, drop them, or use pandas methods to fill them
      * .dropna() will drop rows with null values
      * .fillna() can fill values
        + Usually filled with a summary statistic like mean or median
    - Assert Statements
      * Allows us to detect early warnings and errors
      * Gives confidence that code is running correctly
* Data Cleaning – Python Data Science toolbox part 2
  + Iter() is iterable
    - Syntax: it = iter(file)
    - Print(next(it)) -> this will go through each line of the file, or iterate through each letter of a string like a for loop does
  + Enumerate() creates an object with all items in original list and the index in a list of tuples
    - Trying to print an enumerate object without converting to a list or unpacking first will result in an error
  + Zip() takes arbitrary number of iterables and returns a list
    - Combines 2 lists and shows as a list of tuples, indexes aligned between two lists
    - Trying to print a zip object without converting to a list or unpacking first will result in an error
    - Using \* operator can unpack an iterable such as a list or tuple into a positional argument. Syntax below:
      * Z1 = zip(mutants, powers)
      * Print(\*z1)
      * This prints the tuples that otherwise wouldn’t have printed without converting to list first
  + Loading data in chunks – can be used with iterators
    - Use read\_csv and argument chunksize
  + List comprehensions
    - Syntax:
      * [*[output expression]* **for** *iterator* *variable* **in** *iterable*]
      * new nums = [num + 1 for num in nums]
    - Collapse for loops for building lists into a single line
    - Can also do nested for loops, just have to do one after the other
    - Need to consider readability
    - Conditionals syntax:
      * [num \*\*2 **for** num **in** range(10) **if** num % 2 == 2]
    - Dictionary comprehensions
      * {num: -num **for** num **in** range(9)}
  + Generator Expressions
    - Like a list comprehension but doesn’t return a list
    - Can still iterate over
    - Lazy evaluation – only uses result as needed
    - Good for when you don’t want to store list in memory (large lists)
    - Can also use conditionals in generator expressions
  + Generator functions
    - Produce generator objects when called
    - Yields a sequence of values (use keyword yield in the formula), doesn’t return a value
  + Str methods
    - Syntax: series.str.lower(), series.str.upper() etc
    - Can use regular expression methods too