Lesson 4 – Python for Data Science

**Questons for Mentor:**

* DataCamp – lesson 4.2 – Grouping Data – ‘Detecting outliers with Z-scores’ – output should give z-scores of fertility and life but seems to only actually do it for fertility (values for life seems to just be the life expectancy). Is my code wrong? It was correct when I submitted.

**Data Types for Data Science:**

* Data types
  + Container sequences
    - Holds other types of data
    - Used for aggregation, sorting, and more
    - Lists, sets, tuple
* Tuples
  + Like lists, hold data, index
  + Immutable (can’t change the data)
  + Create tuples by pairing elements
  + Can be unpacked
  + If creating a variable and the argument is trailed with a comma, it creates a tuple
* Enumerate function
  + Used in loops to return the position and the data in that position
  + Returns index along with the value
* Sets
  + Unique data elements in an unordered fashion
  + Mutable
  + Created from lists (syntax: set(list) )
  + Deletes duplicates
  + .add() adds single element
  + .update() merges in another set or list
  + .discard() safely removes an element from the set by value
  + .pop() removes and returns an arbitrary element from the set
  + .union() set method returns a set of all the names (or)
  + .intersection() method identifies overlapping data (and)
  + .difference() method identifies data present in the set on which the method was used that is not in the arguments
* Dictionaries
  + Hold data in key/value pairs
  + Nestable
  + Iterable
  + Created by dict() or {}
  + .get() allows you to safely access the key without error or exception handling
    - If a key isn’t in the dictionary, returns ‘None’
  + Del instruction deletes key/value and will throw error
  + .pop() will safely remove a key/value from dictionary
  + In operator is more efficient and clearer than .get()
    - ‘11234’ in art\_galleries
    - False
  + Adding a key:value pair in a dictionary while iterating is below:
    - Dict[key] = value
* Collections module
  + Counter accepts a list and counts all values in that list
  + .mostcommon() counts most common value in list
* Dictionary handling
  + defaultdict passes it as a default type that every key will have even if it doesn’t currently exist
  + can be used to append a list or add to a counter variable (adding value by one for every eatery with a phone number)
  + OrderedDict orders dictionaries by key
* Namedtuples
  + Tuple where each column has a name
  + Each one has same properties
  + Eatery = namedtuple('Eatery' , ['name' , 'location' , 'park\_id' , ...: 'type\_name'])
* DateTime
  + From datetime
  + Datetime.strptime() converts a string to a datetime format

**Python Data Science Toolbox:**

* Docstrings
  + Describe what a function does
  + Serve as documentation for your function
  + Placed in the immediate line after the function header in between triple quotes (“”” “””)
* Scope in functions
  + Part of program where an object or name may be accessible
  + Global scope – defined in the main body of a scrips
  + Local scope – defined within a function
  + Built in scope – names in the pre-defined built ins module
  + LEGB rule (Local scope, Enclosing functions, Global, Built-In)
* Nested functions
  + Can next a function that allows you to do repeatable tasks within a function
* Default arguments
  + Create a default argument by adding name, ‘=’, then default argument
  + Syntax: def power(number, pow=1) to raise to the first power
* Flexible arguments
  + Use if we don’t know how many arguments there will be
  + \*args takes multiple arguments and turns all arguments into a tuple
  + \*\*kwargs takes multiple keyword arguments
* Lambda functions
  + More efficient way of writing simple functions
  + Can write functions in quick (and sometime dirty) way
  + Syntax: variable = lambda argument: function body
* Error handling
  + Can provide useful error messages with try, except and also with raise

**The Power of Pandas, Manipulating DataFrames with pandas:**

* Pandas = library for data analysis
* Calculating summary statistics on a dataframe has many methods
  + Syntax: df.count(), df[‘column’].mean() etc.
* Resampling
  + Downsampling = reducing frequency of sample – going from daily to weekly
  + Upsampling = increasing frequency of sample – going from daily to hourly
  + Typically used with a statistical method like mean() or median()
  + Resample() needs a string to signify frequency (‘D’ = daily)
  + Resample() method is usually followed with statistical method
  + Non-numerical columns are ignored for this
  + Resampling Frequencies – can modify with numbers (i.e. ‘4H’ = every 4 hours)
    - ‘H’ = hour
    - ‘D’ = day
    - ‘B’ = business day
    - ‘W’ = week
    - ‘M’ = Month
    - ‘Q’ = quarter
    - ‘A’ = year
  + .rolling() calculates rolling data
    - Must always use method chaining (mean() etc. )
    - Syntax: .rolling(window=24) for 24 hr window
* Manipulating time series data
  + Str.contains() = which rows have a selected string
    - Returns Boolean response
  + True = 1, False = 0
  + Str.contains(‘ware’).sum() returns how many positive results of rows that have ‘ware’
  + Dt.tz\_localize(‘US/Central’) changes time zone to central time
  + Dt.tz\_convert(‘US/Eastern’)
  + .interpolate(how=’linear’) fills missing data in a linear fashion instead of forward fill or backward fill
* Timeseries visualization
  + Style format string
    - Color(k: black)
    - Marker(.: dot)
    - Line type (-: solid)
  + Subplots=True arguments makes a separate plot with separate Y axes limits for each column
* Indexing DataFrames
  + Syntax: df[‘salt’][‘Jan’] -> DataFrame[‘column label’][‘row label’]
  + Columns can also be treated as an attribute
    - Syntax: df.salt[‘Jan’]
  + An easy way to index is by using accessors .loc and .iloc
    - Syntax: df.loc[‘Jan’, ‘salt’] -> DataFrame.loc[‘row specifier’, ‘column specifier’]
  + To make the result a DataFrame, use a nested list in square brackets.
    - Syntax: df\_new = df[[‘salt’, ‘eggs’]]
  + When slicing with .loc, the right bound is included in the slice, with iloc, it only goes up to the right bound -1 (i.e. [1:4] includes 1, 2 and 3)
  + Lists can be used in place of slicers
    - Instead of putting df.loc[‘Jan’:’May’, ‘eggs’:’spam’], you can put Instead of putting df.loc[‘Jan’:’May’, [‘eggs’:’spam’]] to include just those two columns, not all those in between
  + Df[‘eggs’] yields a series of the column labeled eggs (series = only one dimension of labeled data)
  + Df[[‘eggs’]] yields a DataFrame of the column labeled ‘eggs’
  + df.loc['b' : 'a' : -1] this slices the data frame from point b to point a and the -1 indicates a step size of -1 (reverse order)
* Filtering DataFrames
  + Can create a Boolean series
    - Df.salt > 60
    - Can be combined using the conditional operators (and, or)
  + .isnull() and .notnull() can filter on columns that contain or do not contain a null value
  + .dropna() will drop rows with NaN values
* Transforming DataFrames
  + df.floordiv(12) converts to dozens unit
  + Np.floor\_divide(df, 12) converts to dozens unit
  + Storing a transformation – you can create a new column
    - Syntax: df[‘dozens\_of\_eggs’] = df.eggs.floordiv(12)
    - Creates a new column which houses how many dozens of eggs
  + Df.index.str.upper() makes the index all upper case
* Pandas Data Structures
  + If you want to change the index, you have the change the whole index at the same time because it is immutable. You need to use a list
  + A succinct way is to do a list comprehension – creates a list with a one line formula
    - Syntax: cubes = [i\*\*3 for i in range(10)]
  + You can name your index as well as the columns using the following:
    - Df.index.name = X
    - Df.columns.name = Y
* Hierarchical indexing
  + Syntax: Stocks = stocks.set\_index([‘Symbol’, ‘Date’])
  + Can sort index using sort\_index()
    - Syntax: Stocks = Stocks.sort\_index()
  + Fancy indexing – using a list when indexing using .loc accessor
  + To index into the inner levels of the index, you must use the slice() function
    - Syntax: stocks.loc[(slice(None), slice(‘2016-10-03’, ‘2016-10-04’)), : ]
* Pivoting DataFrames
  + Can be very useful in reshaping DataFrames
    - Syntax: trials.pivot(index=’treatment’, columns=’gender’, values=’response’)
      * Where treatment, gender and response are all existing columns
* Stacking and unstacking DataFrames
  + Unstack will take the chosen multi level index and make it a hierarchical columns (similar to the male/female columns in the pivots exercises)
    - Syntax: trials\_by\_gender = trials.unstack(level=’gender’)
    - Makes thinner DataFrames wider and shorter
  + Stack will take the chosen hierarchical columns and make it a multi level index (opposite of the male/female columns in the pivots exercises)
    - Syntax: trials\_by\_gender.stack(level=’gender’)
    - Makes wide DataFrames thinner and longer
  + Swapping index levels
    - Syntax: swapped = stacked.swaplevel(0, 1)
* Melting DataFrames
  + Goal of melting is to restore a pivoted DataFrame to its original form
  + Syntax: pd.melt(df, )
* Pivot Tables
  + Requires unique index column pairs to identify values in new table
  + Reshapes a dataframe with a pair of summarizing assumptions and their values
  + Syntax: df.pivot\_table(index=’treatment’, columns=’gender’, values=’response’, aggfunc=’count’)
  + Shows average values by default
* Categoricals and Groupby
  + Syntax: sales.groupby(‘weekday’).count()
    - This counts the values by each weekday
  + Can groupby one column and perform calculation on another
    - Syntax: sales.groupby(‘weekday’)[‘bread’].sum()
* Groupby and Aggregation
  + .agg() method allows you to aggregate with multiple aggregator and gives a multi layer index for the columns
  + Syntax: sales.groupby(‘weekday’)[‘bread’].agg([‘max’, ‘sum’])
    - Also allows user defined functions in the .agg() method
* Groupby and Transformation
  + Instead of aggregating, we can also transform our data instead
  + Aggregation reduces and goes from multiple values down to 1
  + Transformation applies a formula across each line
* Groupby and filtering
  + .apply() method applies a formula
  + Titanic.groupby(‘sex’).apply(c\_surv\_by\_sex)
    - This applies the survival rate by sex on the C deck on the titanic
* Dataframe methods
  + .idxmax() returns row with max value
  + .idxmin() returns row with min value

**The Power of pandas: Merging DataFrames with pandas:**

* Data import
  + Pd.read\_csv() – over a hundred arguments for very specific importing
  + Can import multiple dataframes with a for loop
* Reindexing DataFrames
  + Using .reindex()
    - Ordered = [‘Jan’, ‘Apr’, ‘Jul’, ‘Oct’]
    - Syntax: w\_mean2 = w\_mean.reindex(ordered)
    - This changes the index of w\_mean to the list named ordered
    - Can also be the index to a different dataframe
      * W\_mean.reindex(w\_max.index)
  + Using .sort\_index()
    - W\_mean2.sort\_index()
  + Reindexing with missing labels creates an entirely new row with NaN value
  + .dropna() will drop rows with NaN values
* Arithmetic with Series and DataFrames
  + .divide() method divides a dataframe by a series that is included inside the .divide() method
  + .pct\_change() method calculates percent change between rows
  + Adding dataframes, will only add where there are like index values
  + .add() does the same thing as + operator but you can do an iferror type functionality with .add()
  + .add(fill\_value=0) makes it so there are no NaN values
* Appending and Concatenating DataFrames
  + Append adds to the end of a series or dataframe
  + Syntax: S1.append(s2)
    - This stacks series 2 below series 1
  + Works with both series and dataframes
  + Keeps old index – can repeat index numbers. To remedy, use .reset\_index(drop=True)
  + Concatenate can stack rows and columns (vertically and horizontally)
    - Syntax: pd.concat([s1, s2, s3])
  + With pd.concat(), there is an argument that you can use – ignore\_index=True
  + When appending dataframes with different column names and/or different index names, it will append the columns to the right and add the index as well as removing the name of the index
    - It will show NaN values for the columns where there is no data
  + Concatenating with argument axis=0 achieves the same result as appending in this case. Stacks vertically below
    - When using axis=1, or axis=’columns’, it stacks horizontally on the right, so if there are index labels that are the same, their values will show on the same line as opposed to the index label being repeated with axis=0
  + Multi-index concatenation
    - Use argument keys=[arg1, arg2] to set the outer level multi-index
      * Can be used with rows or columns, but to do it with columns, must specify axis=’columns’
    - The order of the list of keys must match the order of the input dataframes
  + Pd.concat() can also accept dictionaries – uses keys as column or row labels (depending on axis=0 or axis=1) and uses values as the values
* Outer and Inner joins
  + Joining tables means meaningfully combining rows of multiple tables
  + Outer join
    - union of index sets (all labels, no repeats)
    - Has NaN values for missing fields
  + Inner Join
    - Only has intersection of index sets (ones that are common to both)
  + Inner and outer joins are called as arguments in pd.concat() – syntax: join=’inner’ or ‘outer’
* Merging DataFrames
  + Extends pd.concat() with the ability to align rows using multiple columns
  + Merges multiple dataframs and where there is a like row in both columns, it merges them together. Drops the values where there isn’t a like row
  + Syntax: pd.merge(population, cities)
  + Use argument on= to say what column you want to merge on
    - Can use multiple columns (i.e. pd.merge(population, cities, on=[‘NOC’, ‘Country’]) )
  + Can taylor suffixes of columns using suffixes= argument
    - Suffixes=[’\_bronze’, ‘\_gold’]
  + When column names are different (but values are same), use argument left\_on= and right\_on= to specify which columns you want to merge on for the left argument and the right argument
* Joining DataFrames
  + Another way to join DataFrames together
  + Left join – keep all rows of left DF in the merged DF
  + Right join – keeps all rows of right DF in the merged DF
  + Outer join includes all rows with NaN values for places without data
  + .join() – performs left join using index by default
* Ordered Merges
  + Pd.merge\_ordered() – default is ‘outer join’ and accepts on= and suffixes=
  + Pd.merge\_asof() -