api_data_wrangling_mini_project-BrianCamp

May 10, 2019

0.1 Submitted by Brian Camp

This exercise will require you to pull some data from the Qunadl API. Qaundl is currently the most widely used aggregator of financial market data.

As a first step, you will need to register a free account on the http://www.quandl.com website.

After you register, you will be provided with a unique API key, that you should store:

```
In [1]: # Store the API key as a string - according to PEP8, constants are always named in all u

        # to load API_KEY from another file
        # later the file 'my_api.py' will only be kept locally and not ported to github

        %run my_api.py # to load API_KEY
```

```
In [2]: print(type(API_KEY))
        print(API_KEY[:9]+'...')
```

```
<class 'str'>
?api_key=...
```

Qaundl has a large number of data sources, but, unfortunately, most of them require a Premium subscription. Still, there are also a good number of free datasets.

For this mini project, we will focus on equities data from the Frankfurt Stock Exhange (FSE), which is available for free. We'll try and analyze the stock prices of a company called Carl Zeiss Meditec, which manufactures tools for eye examinations, as well as medical lasers for laser eye surgery: https://www.zeiss.com/meditec/int/home.html. The company is listed under the stock ticker AFX_X.

You can find the detailed Quandl API instructions here: https://docs.quandl.com/docs/time-series

While there is a dedicated Python package for connecting to the Quandl API, we would prefer that you use the *requests* package, which can be easily downloaded using *pip* or *conda*. You can find the documentation for the package here: http://docs.python-requests.org/en/master/

Finally, apart from the *requests* package, you are encouraged to not use any third party Python packages, such as *pandas*, and instead focus on what's available in the Python Standard Library (the *collections* module might come in handy: https://pymotw.com/3/collections/). Also, since you won't have access to DataFrames, you are encouraged to us Python's native data structures

- preferably dictionaries, though some questions can also be answered using lists. You can read more on these data structures here: https://docs.python.org/3/tutorial/datastructures.html

Keep in mind that the JSON responses you will be getting from the API map almost one-to-one to Python's dictionaries. Unfortunately, they can be very nested, so make sure you read up on indexing dictionaries in the documentation provided above.

0.2 #0. Preliminaries

In [3]:
```
# First, import relevant modules

import requests
from collections import Counter, OrderedDict, namedtuple

from itertools import filterfalse
```

In [4]:
```
# Now, call the Quandl API and pull out a small sample of the data (only one day) to get
# into the JSON structure that will be returned
```

From the Quandl API documentation site at https://docs.quandl.com/docs/in-depth-usage under 'time-series data':

The API involves a GET command which we will handle with the Python `requests` library

`GET https://www.quandl.com/api/v3/datasets/{database_code}/{dataset_code}/data.{return_format`

In `requests` this will be implemented as data_raw = requests.get(url) and the data is then extracted as a json format using the `.json()` method.

i.e. data_json = data_raw.json()

In [5]:
```
quandl_afxx = 'https://www.quandl.com/api/v3/datasets/FSE/AFX_X/data.json'

# desired window of dates
# for investigation of the structure, the window is kept small
startdate_small='?start_date=2019-05-01'
enddate_small='?end_date=2019-05-07'

# constructing the string for calling the Quandl API
#     (recall that `API_KEY` was obtained from an external program
#      only a portion of it is printed out here)
api_url_small = quandl_afxx + startdate_small + enddate_small + '?order=asc'

print(api_url_small + API_KEY[:9] + '...')

# using reqests.get() to get the raw data
afxx_small_raw = requests.get(api_url_small+API_KEY)

# extracting the data in json format:
afxx_small_json = afxx_small_raw.json()
```

```
https://www.quandl.com/api/v3/datasets/FSE/AFX_X/data.json?start_date=2019-05-01?end_date=2019-0
```

```
In [6]: # Inspect the JSON structure of the object you created, and take note of how nested it i
        # as well as the overall structure

In [7]: print(type(afxx_small_json))
        afxx_small_json.keys()
```

```
<class 'dict'>
```

```
Out[7]: dict_keys(['dataset_data'])
```

The json created is of type `dict`. But it only has one key: `dataset_data`. We dig inside by looking at the value that goes with the `dataset_data` key.

```
In [8]: afxx_small_dict = afxx_small_json['dataset_data']

        print(type(afxx_small_dict))

        print(afxx_small_dict.keys())

        #afxx_small_dict
```

```
<class 'dict'>
dict_keys(['limit', 'transform', 'column_index', 'column_names', 'start_date', 'end_date', 'freq
```

Inside is another dict structure. This one has several keys as indicated above. Here is a closer look at the keys and values printed out below.

```
In [9]: for key, values in afxx_small_dict.items():
            print(key,': ',values)
```

```
limit :  None
transform :  None
column_index :  None
column_names :  ['Date', 'Open', 'High', 'Low', 'Close', 'Change', 'Traded Volume', 'Turnover',
start_date :  2019-05-01
end_date :  2019-05-09
frequency :  daily
data :  [['2019-05-09', None, 87.15, 85.25, 85.75, None, 156083.0, 13415858.0, None, None, None]
collapse :  None
order :  None
```

It seems that the relevant keys are `column_names` and `data`. The `column_names` are the names of the entries in the `data` column.

Let's look a bit more closely at these two keys and their values.

```
In [10]: print(len(afxx_small_dict['column_names']))
         print(afxx_small_dict['column_names'])
         print(len(afxx_small_dict['data'][0]))

         for each in afxx_small_dict['data']:
             print(each)
```

```
11
['Date', 'Open', 'High', 'Low', 'Close', 'Change', 'Traded Volume', 'Turnover', 'Last Price of t
11
['2019-05-09', None, 87.15, 85.25, 85.75, None, 156083.0, 13415858.0, None, None, None]
['2019-05-08', None, 88.95, 85.45, 87.0, None, 252666.0, 22022719.0, None, None, None]
['2019-05-07', None, 90.6, 87.6, 87.8, None, 306539.0, 27240006.0, None, None, None]
['2019-05-06', None, 89.05, 84.9, 89.05, None, 160083.0, 14049181.0, None, None, None]
['2019-05-03', None, 87.5, 86.65, 87.2, None, 138284.0, 12036856.0, None, None, None]
['2019-05-02', None, 87.7, 85.45, 86.55, None, 216631.0, 18702265.0, None, None, None]
```

There are a few other issues that seem to be present. * The dates are given in reverse order even though the original API call asked for ascending order with `?order=asc`. * The end date of 2017-05-07 was not respected. In other words, the API call ignored the `?enddate=2019-05-07` part of the query. * Some of the data in the `Open` field is `None` which will pose problems for some questions later on.

To work with the data further, a `namedtuple` called `MyRow` will be created. This will allow the data on any given day to still be associated with the column name that it is from. Some of the column names have spaces however, so to these will be changed to underscores. The `MyRow` tuple will not include the date as that will be used as an index of sorts for a list that we create later. Keeping the date out of the `namedtuple` will also make it a bit easier to filter data later on (since there are some dates showing up that we don't wont – more below).

```
In [11]: tuple_names = [i.replace(' ','_') for i in afxx_small_dict['column_names'][1:]]
         print(tuple_names)
         MyRow = namedtuple('MyRow', tuple_names)
```

```
['Open', 'High', 'Low', 'Close', 'Change', 'Traded_Volume', 'Turnover', 'Last_Price_of_the_Day',
```

Next we define a function `datalist(data)` that will be used to put our data into a list of tuples. The tuples will be of the form `( date, OrderedDict() )` where the `MyRow()._asdict()` will build Ordered Dictionaries to contain the data for a given date. Finally, the list will be `sorted()` by the `date` entry in the tuple.

Note: both `OrderedDict()` and `namedtuple` are from the Python `collections` module.

```
In [12]: def data_list(data):
             # list of tuples
             # first entry is date
             # second entry is namedtuple MyRow that contains the data from that date.
```

```
    l = sorted([(each[0], MyRow(*each[1:])._asdict()) for each in data])
    return l
```

```
In [13]: # for example to create the list of data
         # data_list(afxx_small_dict['data'])
```

These are your tasks for this mini project:

1. Collect data from the Franfurt Stock Exchange, for the ticker AFX_X, for the whole year 2017 (keep in mind that the date format is YYYY-MM-DD).
2. Convert the returned JSON object into a Python dictionary.
3. Calculate what the highest and lowest opening prices were for the stock in this period.
4. What was the largest change in any one day (based on High and Low price)?
5. What was the largest change between any two days (based on Closing Price)?
6. What was the average daily trading volume during this year?
7. (Optional) What was the median trading volume during this year. (Note: you may need to implement your own function for calculating the median.)

0.3 #1. Collect data for 2017

```
In [14]: quandl_afxx = 'https://www.quandl.com/api/v3/datasets/FSE/AFX_X.json'

         startdate='?start_date=2017-01-01'
         enddate='?end_date=2017-12-31'

         api_url = quandl_afxx + startdate + enddate + '?order=asc' + API_KEY
         #print(api_url)
         print(quandl_afxx + startdate + enddate + '?order=asc' + API_KEY[:9] + '...')

         afxx_raw = requests.get(api_url)
```

```
https://www.quandl.com/api/v3/datasets/FSE/AFX_X.json?start_date=2017-01-01?end_date=2017-12-31?
```

0.4 #2. convert to json and then to dict format

The `.json()` method will extract the data in json format.

```
In [15]: afxx_json = afxx_raw.json()
```

As we saw above, we need to extract the dictionary that is paired with `dataset` key.

```
In [16]: afxx_dict = afxx_json['dataset']
         #afxx_dict
```

Finally we use our function `data_list()` which was defined above on the `data` values in the `afxx_dict`.

```
In [17]: afxx = data_list(afxx_dict['data'])
```

```
In [18]: print(type(afxx))
         print(afxx[0])
```

```
<class 'list'>
('2017-01-02', OrderedDict([('Open', 34.99), ('High', 35.94), ('Low', 34.99), ('Close', 35.8), (
```

There are some problems with this data as we noted above. The first problem is that we have too much data. The goal was to have data that occurred between 2017-01-01 and 2017-12-31.

```
In [19]: # many entries are from outside the time period
         # for example, the last entry is outside of the time period of interest
         print(len(afxx))
         print(afxx[-1])
```

```
585
('2019-05-09', OrderedDict([('Open', None), ('High', 87.15), ('Low', 85.25), ('Close', 85.75), (
```

To remove unwanted data we will make use of the `filterfalse()` function from the Python `itertools` module.

```
In [20]: afxx = list(filterfalse(lambda x: x[0]>'2017-12-31', afxx))
```

Now the last piece of data is within the time period and the length of dataset corresponds to roughly fifty weeks of five business days apiece during the year (which is normal for financial markets in a year).

```
In [21]: print(len(afxx))
         print(afxx[-1][0])
         afxx[-1][0]<'2018-01-01'
```

```
255
2017-12-29
```

```
Out[21]: True
```

To answer the remaining questions we need extract several lists of data from our dataset:

```
In [22]: dates = [ x[0] for x in afxx ]
         opening = [ x[1]['Open'] for x in afxx ]
         highs = [ x[1]['High'] for x in afxx ]
         lows = [ x[1]['Low'] for x in afxx ]
         closing = [ x[1]['Close'] for x in afxx ]
         traded_vol = [ x[1]['Traded_Volume'] for x in afxx ]
```

0.5 #3. Highest and Lowest Opening

Now we encounter another problem. Not all of the opening data has a number. Some occur as `None`.

We can use the `Counter()` function from the Python `collections` module along with its method `.most_common()` to see how many different types of data we have for the opening.

```
In [23]: #open_type = [type(each) for each in opening]
         Counter([type(each) for each in opening]).most_common()
```

```
Out[23]: [(float, 252), (NoneType, 3)]
```

Since we are dealing with stock data, we will replace any missing opening data by using the closing price from the previous day.

Note: The first piece of data does not have this issue. Otherwise we would need to find data from the previous year to fill in the opening price (i.e. the last closing price from 2016).

```
In [24]: print(opening[0])
         print(type(opening[0]))
```

```
34.99
<class 'float'>
```

We define a new list `newopen` to repair the missing values.

```
In [25]: newopen = []
         for i, val in enumerate(opening):
             # if None then use previous closing value
             if val == None:
                 newopen.append(closing[i-1])
             else:
                 newopen.append(val)
```

Now we can more easily find the minimum and maximum opening values.

```
In [26]: lowval = min(newopen)
         highval = max(newopen)
         lowest = {'val':lowval, 'ind':[index for index, value in enumerate(newopen) if value==l
         highest = {'val':highval, 'ind':[index for index, value in enumerate(newopen) if value=


         print('Lowest opening value is %s. Occured on: %s'%(lowest['val'], [dates[i] for i in l
         print('Highest opening value is %s. Occured on: %s'%(highest['val'], [dates[i] for i in
```

```
Lowest opening value is 34.0. Occured on: ['2017-01-24']
Highest opening value is 53.11. Occured on: ['2017-12-14']
```

0.6 #4. Largest range within a given day.

The largest range within a given day will be based upon comparing the high and low prices on any given day.

```
In [27]: daily_changes = [ highs[i] - lows[i] for i in range(len(dates)) ]
         big_change_val = max(daily_changes)
         biggest_change = {'val':big_change_val, 'ind':[index for index, value in enumerate(dail
         biggest_change
         print('Biggest range on any given day (from low to high) was %s.\nOccured on: %s'%(bigg
```

```
Biggest range on any given day (from low to high) was 2.8100000000000023.
Occured on: ['2017-05-11']
```

0.7 #5. Largest change between any two days (based on closing)

There is no missing data from the `closing` data.

```
In [28]: Counter([type(each) for each in closing]).most_common()
```

```
Out[28]: [(float, 255)]
```

```
In [29]: change_consecutive = [ closing[i+1] - closing[i] for i in range(len(closing)-1)]
```

We will need to consider both largest negative (decrease) or largest positive (increase) when finding the biggest change.

```
In [30]: big_inc = max(change_consecutive)
         big_dec = min(change_consecutive)

         print('Biggest increase was %s'%big_inc)
         print('Biggest decrease was %s'%big_dec)

         big_change = max([abs(big_inc), abs(big_dec)])

         print('Biggest change (by absolute value) was %s'%big_change)
```

```
Biggest increase was 1.7199999999999989
Biggest decrease was -2.559999999999995
Biggest change (by absolute value) was 2.559999999999995
```

0.8 #6. Average Daily Trading Volume during 2017

```
In [31]: ave_daily_traded = sum(traded_vol)/len(traded_vol)

         print('The average daily trading volume during 2017 was %s'%ave_daily_traded)
```

```
The average daily trading volume during 2017 was 89124.33725490196
```

0.9 #7. Median Trading Volume during 2017

We create a function `my_median(data)` to calculate the median of a list of **sorted** data.

```
In [32]: def my_median(data):
             n = len(data)
             sortdata = sorted(data)
             # if even number
             if n%2 == 0:
                 med = (sortdata[int(n/2) - 1] + sortdata[int(n/2)])/2
             else:
                 med = sortdata[int((n+1)/2) - 1]
             return med
```

```
In [33]: median_trade_vol = my_median(traded_vol)

         print('Median volume traded was %s'%median_trade_vol)
```

```
Median volume traded was 76286.0
```

```
In [ ]:
```