**Data Focused Python, Mini 3 2019**

**Homework 2**

***Due 15 minutes before lecture, on Tuesday, Jan. 29***

1. (70 points) **Handling Idiosyncratically Formatted Data**

In Python terms, the purpose of this part of the homework is to gain experience with input and output files, variables, decisions, loops, string processing (including slices and formatting), conversions (string to number or number to string), and the like.

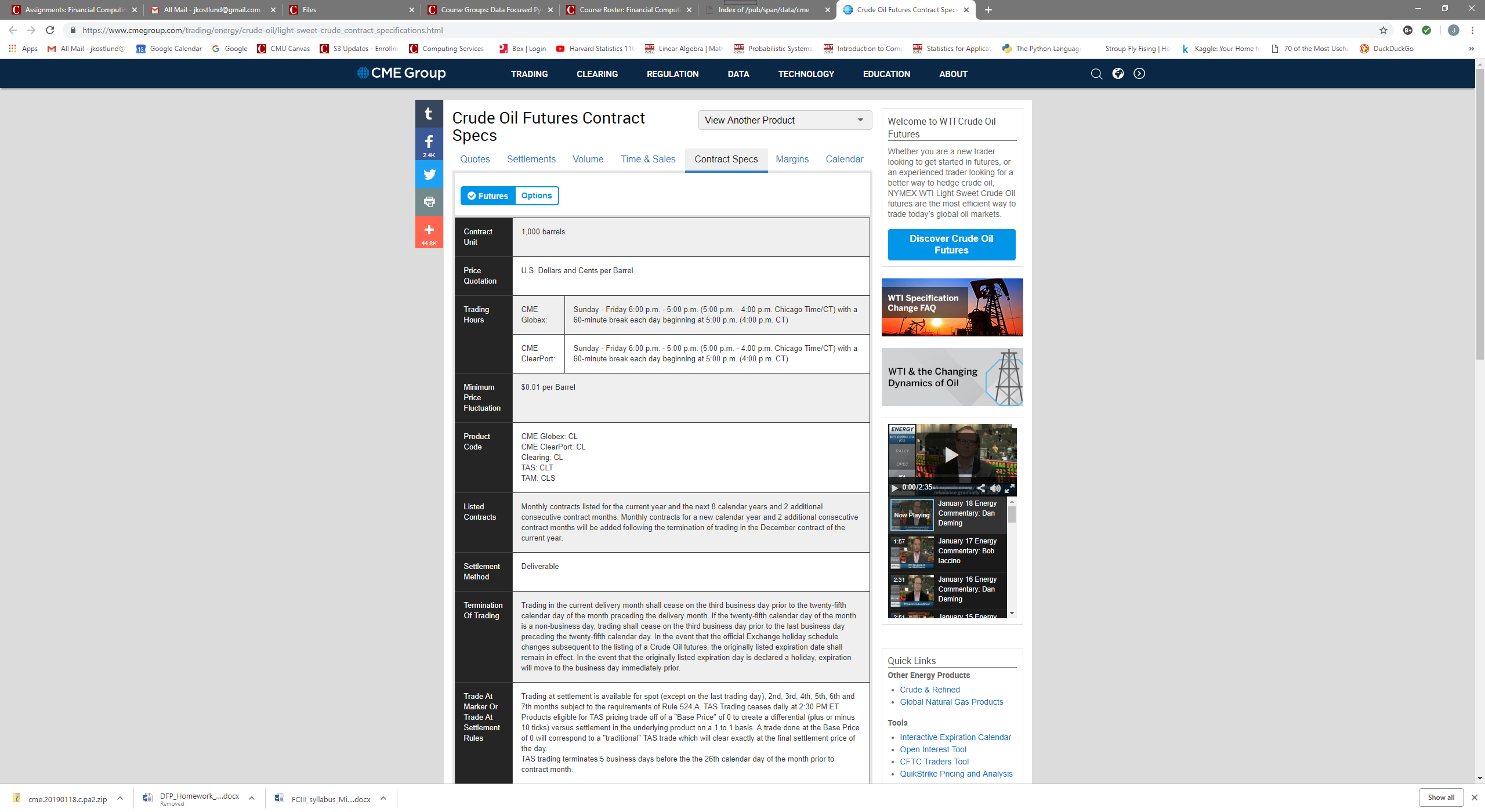
Some data sources are in convenient formats (CSV, JSON, HTML, XML, and the like), and others are mostly unformatted (documents, email messages, system and web logs, and the like). There are also idiosyncratically formatted files, with their own strange formats often made up years in the past, before standards like CSV or JSON were invented. You must be able to handle *all* of these kinds of data sources.

Commodity futures and option contracts of many kinds are traded on NYMEX, owned by CME Group. Each evening of each trading day, sometime between about 6:00 pm and 8:00 pm Central Time, a SPAN (Standard Portfolio Analysis of Risk) file is posted to <ftp://ftp.cmegroup.com/pub/span/data/cme> containing information about the day’s trading. For a given day, the name of this file is cme.*YYYYMMDD*.c.pa2.zip, where *YYYYMMDD* is the 8-digit year, month, and day of the file. Files for months prior to the current month are moved into the [/pub/span/data/cme/201](ftp://ftp.cmegroup.com/pub/span/data/cme/2016)8 (or 2019) subdirectory.

Download the zipped SPAN file for Friday, Jan. 18, 2018, **cme.20190118.c.pa2.zip**. Unzip, then display this SPAN file. You will see that it is an enormous text file with its own unique format, unfortunately *not* something simple and convenient like CSV or XML or JSON.

The *settlement prices* (in U.S. dollars) contained in the SPAN file are used to *mark to market* each trader’s account, so that gains/losses can be credited/debited each day to reduce the risk of counterparty default (a trader who has to cover modest losses each day is less likely to default than a trader who has to cover huge losses at the end of a year, say). Your job is to extract these settlement prices, as well as contract expiration dates (last trading dates), for one of the globally most heavily traded energy contracts: West Texas Intermediate (WTI) Crude Oil.

To learn about WTI Crude Oil futures contract details, check out: <http://www.cmegroup.com/trading/energy/crude-oil/light-sweet-crude_contract_specifications.html>



Notice that the CME Globex Product Code is **CL**; you will need this for scanning the SPAN file. Using other tabs at the top of this web page, you can see current quotes, recent settlements, volume, etc. If you click the Options button, just to the right of the Futures button near the upper left, you will see information about options contracts based on the underlying futures contracts. There are about two dozen different types of option contracts for this underlying; we are interested in the American Options. When you look at the contract specifications, you will discover that its Product Code is **LO**.

Write a Python program named **hw2.1.py** that reads **cme.20190118.c.pa2** as its input file, and produces **CL\_expirations\_and\_settlements.txt** as its output file. The output should be in exactly this form:

Futures Contract Contract Futures Options Options

Code Month Type Exp Date Code Exp Date

------- -------- -------- -------- ------- --------

CL 2019-03 Fut 2019-02-20

CL 2019-04 Fut 2019-03-20

*… and so forth, through contract month 2021-12 …*

CL 2019-03 Opt LO 2019-02-14

CL 2019-04 Opt LO 2019-03-15

*… and so forth, through contract month 2021-12 …*

Futures Contract Contract Strike Settlement

Code Month Type Price Price

------- -------- -------- ------ ----------

CL 2019-03 Fut 54.04

CL 2019-04 Fut 54.30

*… and so forth, through contract month 2021-12 …*

CL 2019-03 Call 10.50 43.54

CL 2019-03 Put 10.50 0.00

CL 2019-03 Call 11.00 43.04

CL 2019-03 Put 11.00 0.00

*… and so forth, through contract month 2021-12 …*

***Do not try to create a better output format***: it needs to be very easy for us to compare your output to our solution output, and to other students’ outputs. We will take off points if your output format varies too much from what is shown above. Our output format takes into account the order in which records appear in the SPAN file, so you don’t have to remember or accumulate much information as you go. In particular, you do not have to process the contents of **cme.20190118.c.pa2** more than once in order to create the table.

***Do not*** include contract months earlier than 2019-03, or later than 2021-12.

Since there are many, many strike prices for options on futures contracts, the output file is going to be very long, but not nearly as long as the SPAN file itself.

Fortunately, there is documentation online that describes the contents of CME SPAN files. If you Google for “cme span pa2 file format” you will find a page named “Risk Parameter File Layouts for the Positional Formats – SPAN…”. You will want to look at Type “B” Records, Expanded Format, and Type “8” Records, Expanded Format, to learn how to obtain the contract name, type, month, expiration date, strike, and settlement prices that you need.

A few hints:

(a) Notice that the documentation counts character column positions from 1, whereas in your code you will need to count character positions from 0 for **str** slices.

(b) Check the contract specifications to discover the number of decimal places you should display for price (U.S. dollars) of WTI Crude Oil futures and options contracts.

(c) Approach the program in stages: first, make sure you can write a program that simply copies the SPAN file to the output file; next, copy the type B and type 8 records from the SPAN file to the output file; next, copy the type B CL and type 8 CL records; and so forth, making definite steady progress with each revision. As your coding skills improve, you can do two or three or four things in each revision step. Eventually, you will find that you can write dozens of lines of code encompassing many different tasks and goals, and it will work the first time! But maybe not every time.

(d) There are subtypes of the type 8 records: it turns out you can just use the type 8 subtype 1 (or just type 81) records, and ignore the type 82 records. (For WTI Crude Oil, there are no type 83 records.)

(e) There is a brief description of string formatting in McKinney’s book, under 2.3 Python Laguage Basics => Scalar Types => Strings. Many more examples can be found here: https://docs.python.org/3.6/tutorial/inputoutput.html

(f) Remember that collaboration is encouraged: in addition to your homework partner(s), feel free to compare what you are doing with other students, as well. Just make sure you submit your own homework team’s code, after whatever discussions you have with others.

(g) Remember the Discussion board and **jostlund@andrew.cmu.edu**.

1. (30 points) **Lists, Tuples, Sets, Dicts, and Comprehensions**

**expenses.txt** is a small text file describing business expenses. Each line (after the header) gives the money amount, category, date, and description of an expense.

1. Create a Python script file named **hw2.2.py**. In this script, define an empty list named **records**, then read the lines from **expenses.txt** and **append** each line (*excluding* its terminating newline character) to the **records** list. Add this code to display the lines from **records**:

**for line in records:**

**print(line)**

Confirm that the output is not double-spaced; that is, confirm that each line (string) in the **records** list does not include a terminating newline.

1. Close the open **expenses.txt** file, then open **expenses.txt** again. Use ***list*** *comprehension* notation to create and initialize a new list, **records2**, from the lines in the **expenses.txt** file, excluding the terminating newline characters. Confirm that you have done this correctly, by adding this code at the end of the script:

**print("\nrecords == records2:",**

**records == records2, '\n')**

This should display **records == records2: True**.

1. Close the open **expenses.txt** file, and open **expenses.txt** again. Learn about the **str** class’s **split** function. Fields in the **expenses.txt** file are separated with colon characters, **‘:’**, since expense descriptions often contain commas. Use *nested* ***tuple*** *comprehension* notation to create and initialize a new *tuple* *of tuples*, **records3**, in which each “inner” tuple has the form **(***amount***,***category***,***date***,***description***)**, and the “outer” tuple contains one “inner” tuple for each line of input. We use a tuple of tuples because tuples are *immutable*, and we want to protect the input data from accidental change.

Add this code to display the tuple of tuples **records3**:

**for tup in records3:**

**print(tup)**

The output from this loop should look like:

**('Amount', 'Category', 'Date', 'Description')**

**('5.25', 'supply', '20170222', 'box of staples')**

**...**

**('8.98', 'supply', '20170325', 'Flair pens')**

1. A function is a mapping from arguments to values. A sequence or map (**dict**) can also be thought of as a mapping from arguments to values. Creation of sequences/maps from data can simplify function definitions, or even eliminate the need for some of them. A **list** or **tuple** is a mapping from an integer subscript to a value; a **set** is a mapping from a value to **in == True** or **in == False**; and a **dict** is a mapping from a key to a value.

Using ***set*** *comprehension* notation with **records3**, define: **cat\_set**, the set of categories (do not include the string **'Category'**) in the expense records; and, **date\_set**, the set of dates (again, do not include the string **'Date'**) in the expense records. Add this code to display these two sets:

**print('Categories:', cat\_set, '\n')**

**print('Dates: ', date\_set, '\n')**

Since **set**s are unordered, your exact output may differ, but the output should look something like:

**Categories: {'supply', 'meal', 'travel', 'util'}**

**Dates: {'20170222', '20170223', …, '20170325'}**

1. Using ***dict*** *comprehension* notation with **records3**, define a **dict** named **rec\_num\_to\_record** in which each entry’s *key* is the record (line) number, and each entry’s *value* is the tuple representing the data. *Hint*: use a combination of **range()** and **zip**() along with **records3**. In **rec\_num\_to\_record**, store the field names as record number **0**.

Add this code to display **rec\_num\_to\_record**:

**for rn in range(len(rec\_num\_to\_record)):**

**print('{:3d}: {}'.format(rn,**

**rec\_num\_to\_record[rn]))**

The output from this loop should look like:

**0: ('Amount', 'Category', 'Date', 'Description')**

**1: ('5.25', 'supply', '20170222', 'box of staples')**

**...**

**22: ('212.06', 'util', '20170308', 'Duquesne Light')**

Add this code, using the **items()** iterable, to display **rec\_num\_to\_record**:

**for i in rec\_num\_to\_record.items():**

**print('{:3d}: {}'.format(i[0], i[1]))**

Since a **dict** is unordered, the output will be the same as before but perhaps with the lines in a different order … or perhaps not!

Alternatively, using *tuple unpacking* into two loop variables, you can use (for example):

**for k, v in rec\_num\_to\_record.items():**

**print('{:3d}: {}'.format(k, v))**

***When finished, put your hw2.1.py and hw2.2.py source code files into a .zip file, and upload your .zip file to Canvas.***