**MSCF Financial Computing I**

**Mini 1, 2022**

**Homework 1**

***Due At 11:59 pm Sunday, Sept. 4, 2022***

***You will lose 10 points per hour after that time***

1. **(20 points) Regular Expressions**

The **expenses.txt** file provides good examples for regular expression matching.

1. Create a Python script file named **expense\_regex.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 for displaying lines from **records** that match a trivial regular expression:

**import re**

**# 1a**

**pat = r'D'**

**for line in records:**

**if re.search(pat, line) != None:**

**print(line)**

Confirm that the output is lines from **records** that contain a **D** somewhere.

1. Comment out the pattern for part 2.a, and add a new pattern for part 2.b, like this:

**# 1a**

**# pat = r'D'**

**# 1b**

**pat = r'\''**

Confirm that the output is lines from **records** that contain a single quote (**'**) character

somewhere.

1. Comment out the pattern for part 2.b, and add a new pattern for part 2.c that will display lines from records that contain a double quote (**"**) character somewhere. Run the module to test.
2. Continue in this manner of commenting out the previous part’s pattern and adding a new pattern. Add a pattern that will display lines that begin with **7**; run the module to test.
3. Add a pattern that will display lines that end with an **r** *or* a **t**; test.
4. Add a pattern that will display lines that contain a literal period (**.**) character; test.
5. Display lines that contain an **r** followed later by a **g**. (The **r** and the **g** do not need to be consecutive characters.)
6. Display lines that contain two *consecutive* uppercase letters (for example, **AA**, **DF**, **LM**, **YW**, …).
7. Display lines that contain a comma (**,**) character.
8. Display lines that contain *three or more* comma characters (not necessarily consecutive).
9. Display lines that *do not* contain any **v**, **w**, **x**, **y**, or **z** characters.
10. Display lines that contain money amounts between 10.00 and 99.99. ***Hint:*** What must the first character be? What must the second character be? What must the third character be? And so forth. The pattern **'[10.00-99.99]'** *will not work*, because one pair of square brackets only matches a single character, not a range of multiple characters. For example, **'[aaaaaaa]'** matches one **a**, not a sequence of seven **a** characters. The pattern **'[abcabcabcaaabbbaaaccccccc]'** matches the same thing as the pattern **'[abc]'** or the pattern **'[acb]'** or **'[bac]'** and so forth: that is, *either* an **a** *or* a **b** *or* a **c**. The regular expression pattern **'[AA-ZZ]'** matches *either* an **A**, *or* a character in the range from **A** through **Z**, *or* a **Z**; that is, the pattern is equivalent to **'[A-Z]'**.
11. Display lines that contain *exactly three* commas.
12. Display lines that contain a **(** character (that is, an *open parenthesis* character).
13. Display lines that describe meals costing at least 100.00.
14. Display lines that have an expense category that is *exactly four characters wide* (your pattern should work even if more lines are added to the file, with new categories that have not yet been defined).
15. Display lines for expenses that occurred in March.
16. Display lines that contain an **a**, followed by a **b**, followed by a **c** (perhaps with other characters between the **a** and the **b** and the **c**).
17. Display lines that contain some sequence of two characters, followed later by that same sequence of two characters, followed later by that same sequence of two characters again. That is, each matched line should contain at least one sequence of two characters at least three times.
18. Display lines in which the expense description contains a both a lowercase **a** and a decimal digit character between **0** and **9**, *in either order*. That is, the **a** might appear *before* the digit, or the **a** might appear *after* the digit.
19. Display lines that contain no uppercase letters.
20. Display lines that contain a **d** character, possibly followed by one *optional* character, followed by an **i** character. (Matches would include words like **diver**, **doily**, **drip**, **diplomat**, etc.)

1. **(40 points) Pandas Series and DataFrame and Matplotlib**
2. The **b\_soup\_1.py** file contains the code from the Week 1 Part 1 lecture notes, showing how to start with the HTML for a web site and process that HTML into a **list** of table data value strings (**str**) using the **BeautifulSoup** module.

First, modify **b\_soup\_1.py** so that the program’s only output is the final sequence of table cell value **list**s: no **bsyc\_temp.txt** file, no intermediate results being displayed, etc.

Modify the code at the end of the program so that the table cell values are accumulated into a **list** of **list**s, representing the table of rows, something like this:

**daily\_yield\_curves = [**

**[ …** *header list* **… ],**

**[ …** *first data list* **… ],**

**…**

**[ …** *final data list* **… ]**

**]**

The first “inner” **list** should represent the header row:

**['Date', '1 mo', '2 mo', '3 mo', '6 mo', '1 yr', '2 yr',**

**'3 yr', '5 yr', '7 yr', '10 yr', '20 yr', '30 yr']**

Following that should be a **list** for each data row. Be sure to convert each interest rate value from a string to a **float**:

**['01/02/19', 2.40, 2.40, 2.42, 2.51, 2.60, 2.50,**

**2.47, 2.49, 2.56, 2.66, 2.83, 2.97]**

**...**

**['09/13/19', 1.99, 1.98, 1.96, 1.92, 1.88, 1.79,**

**1.76, 1.75, 1.83, 1.90, 2.17, 2.37]**

**...**

Modify **b\_soup\_1.py** again to create a file named **daily\_yield\_curves.txt** containing a neatly formatted table of this information for the year **2021** (instead of 2019).

1. Investigate **matplotlib**’s 3D Surface Plot and Wireframe Plot (**https://matplotlib.org/Matplotlib.pdf**). Produce a 3D Surface Plot of the daily yield curves, with days since 01/02/21 on the X axis, months to maturity on the Y axis (from 1 month to 360 months), and rate on the Z axis. Orient the plot in such a way that this yield curve evolution surface is reasonable to look at. Set axis labels like **‘trading days since 01/02/21’**, **‘months to maturity’**, and **‘rate’** so that the user can tell which axis represents which dimension in the plot. After you have produced a Surface Plot, produce a Wireframe Plot of the same information. (You do *not* need to save screenshots of your plots.)

The Y axis should show *months to maturity*. You will have to “convert” the column labels into the appropriate integer number of months. You can be unclever about this and use a **list** like **[1, 2, 3, 6, 12, 24, 36, 60, 84, 120, 240, 360]**, or you can be more clever and set up a **dict** mapping from column name to number of months, like **cn\_to\_nm = { ‘1 mo’ : 1, ‘2 mo’ : 2, …, ’30 yr’ : 360 }**. It is okay to be unclever.

***Hint:*** You will need to create an **ndarray** of the interest rate values from the **daily\_yield\_curves** list of lists in order to produce plots.

**matplotlib** facilities for creating 3D Surface Plots and Wireframe Plots make use of **numpy** **ndarrays**. Recall that you can convert a **list** of **list**s to a 2-dimensional **ndarray** using **np.array()**. As an example, try:

**X = np.array([ [ 0, .25, .5, .75, 1 ],**

**[ 0, .25, .5, .75, 1 ],**

**[ 0, .25, .5, .75, 1 ] ])**

**Y = np.array([ [ 0, 0, 0, 0, 0 ],**

**[ .5, .5, .5, .5, .5 ],**

**[ 1, 1, 1, 1, 1 ] ])**

**Z = np.array([ [ .4, .2, .1, .1, .2 ],**

**[ .3, .5, .2, .3, .4 ],**

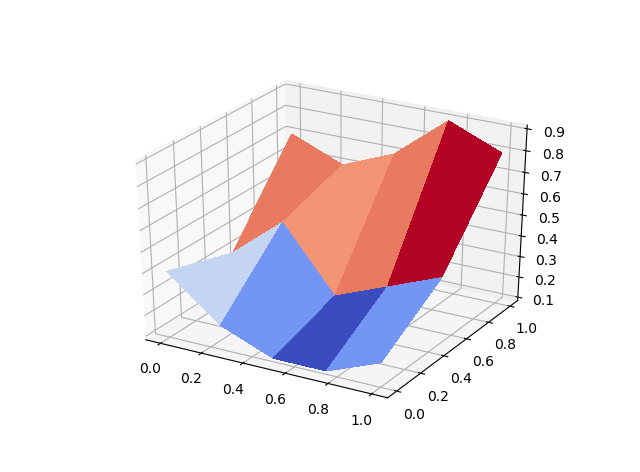
**[ .7, .6, .7, .9, .8 ] ])**

As the last step in creating a plot, you must use the statement

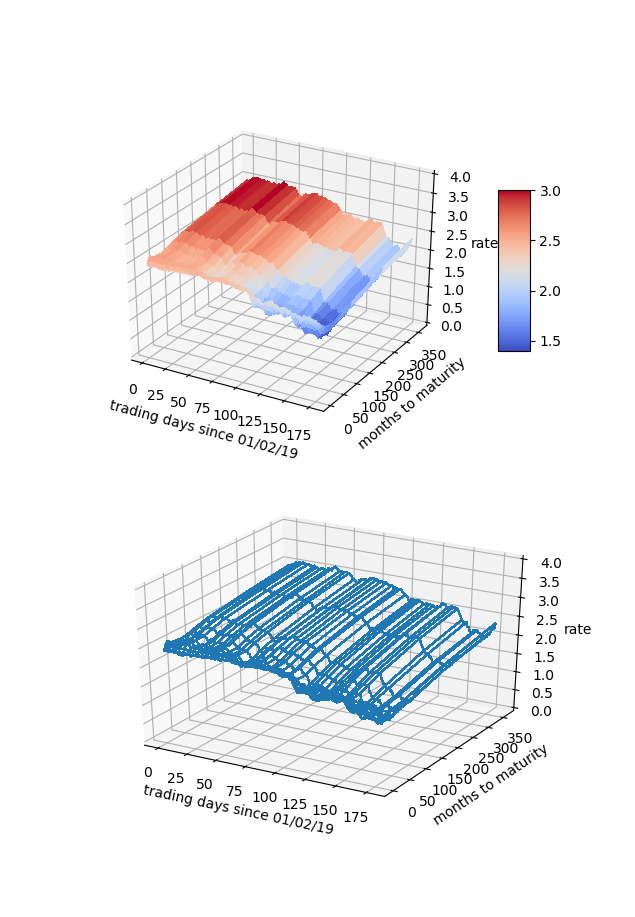
**plt.show()**

to make the plot be drawn on your screen. After the plot has been drawn, click the close button, **X**, in the upper right corner so that your program can continue.

A surface plot of these test **ndarrays**, **X**, **Y**, and **Z**, should look very similar to the screen shot on the next page:



Surface and Wireframe Plots of the yield curve data should look somewhat like these, but with data for the year 2021 instead of for the year 2019. Short-term interest rates in 2021 are pinned at near 0 as a result of the Fed’s response to Covid-19. Long-term rates rose above 2% for a few months, but then fell back below 2%.



1. Our interest rate table is a natural Pandas **DataFrame**, with trading dates as rows and bond maturities as columns. From the **daily\_yield\_curves** **list** of **list**s, create a **DataFrame** named **yield\_curve\_df** with the date strings as the row labels (**‘01/02/2021’**, …, **‘08/27/2021’**, …), the bond maturities as the column labels

(**‘1 mo’**, …, **’30 yr’**), and the corresponding interest rate values as the row/column item values. Use appropriate slices/loops/comprehensions involving **daily\_yield\_curves** to create **yield\_curve\_df**.

**DataFrame** has a **plot()** member function that uses **matplotlib**. You can use **yield\_curve\_df.plot()** to create a plot with rows on the horizontal axis, values on the vertical axis, and with each column represented as a different line. You will still need to use

**plt.show()**

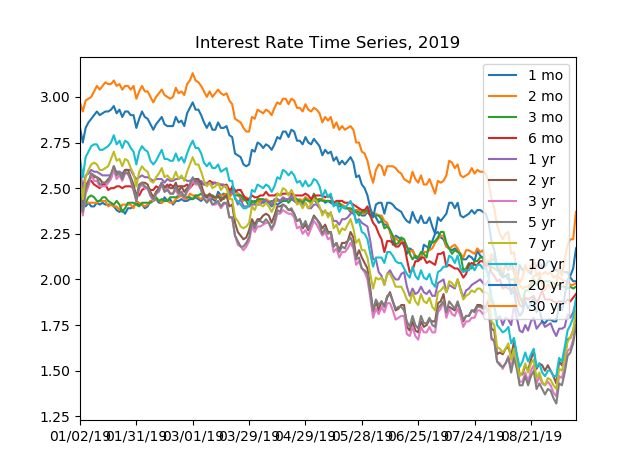
to make the plot be drawn on your screen. Since the rows are trading days, this plot will be of the *time series* of interest rates for each maturity: 1 month, 2 months, 3 months, …, 30 years. You will see that during 2021, interest rates for all maturities have fallen. During March, short term rates fell dramatically when the Fed changed policy due to Covid-19.

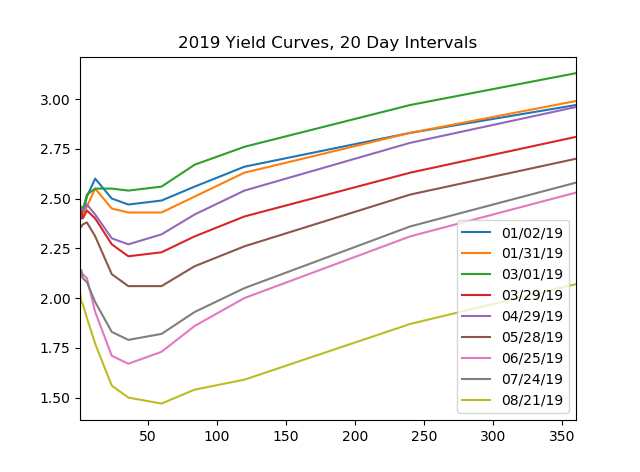
Generally, it is considered more risky to lend for longer periods of time, so a “normal” yield curve slopes up: interest rates are lowest at 1 month, higher at 1 year, higher still at 10 years, and highest at 30 years. This is what we see for most days during 2021.

If we *transpose* **yield\_curve\_df**, so that trading dates become the columns and maturities become the rows, then a **plot()** will show us the daily yield curve for every trading day so far this year. This will be an unreadable mess with over 100 lines.

From **yield\_curve\_df** create a **DataFrame** object named **by\_day\_yield\_curve\_df**, containing the transpose of **yield\_curve\_df** *but* only including a column for every 20th trading day, that is, day 0, day 20, day 40, …, day 240. The column labels should be **‘01/04/21’**, **‘02/02/21’**, **‘03/03/21’**, … if you do this correctly. You will need to modify the row labels from **‘1 mo’**, **‘2 mo’**, and so forth, to the corresponding integer number of months—1, 2, …, 360—in order for the plot’s horizontal axis to make sense.

The by-maturity time series plot and the by-trading-day yield curve plots should look somewhat similar to the examples shown here, but for 2021 data rather than 2019 data:





1. **European Options (40 points)**

In this part of the homework, you will develop code for pricing “plain vanilla” European Call options. By “plain vanilla” we mean that the underlying asset pays no dividend, that the volatility of the underlying asset’s price movements is constant throughout time, that the payoff of the call/put is made at expiration time T and cannot be collected at any prior time, that the risk-free interest rate is constant, and the asset price evolution follows a geometric Brownian motion.

1. In the Python development environment of your choice, open a copy of the **OptionPrices.py** file provided with this homework. **OptionPrices.py** contains the **EuropeanCallOption** class code that we saw in the Week 1 Part 3 Lecture, with the addition of a “pretty printing” function, **binomialTreePretty**, that lets you display small **binomialTree** objects in a clean way. Run the module. You will see that it displays the binomial tree at each step in the **binomialPrice** function’s algorithm. The backward induction code is yours to write—**Your code here**—so the reported option price at this point is $0.0000.
2. Add the backward induction code, the formula for which is shown on slide 16 of the Week 1 Part 3 Lecture notes. Save and test. Here is the output that I get for a 5-step **binomialPrice**:

Binomial Tree Euro call price, 5 time intervals: $6.3598

1. The call value computed with 5 time steps is pretty far off from the BSM price shown on slide 6 of the notes: **$6.12**. As you add time steps to the Binomial Tree (that is, as deltaT becomes smaller), the **binomialPrice** should converge to the BSM price. Uncomment the code for part 1.c that calls **binomialPrice** with time steps of 10, 20, 50, 100, 200, 500, and 1000. Save and test. Do the computed prices appear to converge?
2. In the BSM and Binomial Tree models, given identical volatility, interest rate, and expiration time, the price of an option on a $5 stock with strike price $5 should be exactly 1/10 of the price of an option on a $50 stock with strike price $50. Likewise, the price of an option on a $500 stock with a strike price of $500 should be 10 times the price of $50 stock, $50 strike option. Uncomment the code for part 1.d that tests whether this is the case for the Binomial Tree pricing model. Save and test. What do you observe?

***And Finally***

Remember to put all team members’ names (Andrew IDs) in comments at the tops of your source code files.

Put your **expense\_regex.py**, **b\_soup\_1.py**, and **OptionPrices.py** files into a **zip** archive named **Team\_***N***\_HW1.zip**, where *N* is your team number. *One* team member should upload this zip archive to Canvas for grading before the due date/time.