DATA 690 Homework 2 (65 points - Due on Sunday, February 19, 2023 by 11:59 pm ET)

The output of this assignment for submission should be in PDF format **AND** .py or .ipynb. The name of the file should be as follows: Lastname_Firstname_Homework2.pdf (example: Thomas_Sunela_Homework2.pdf) **AND** Lastname_Firstname_Homework2.ipynb (example: Thomas_Sunela_Assignment2.ipynb. In short, you are submitting the python notebook as well as the pdf of that notebook. Do **NOT** submit .html file, the system will give you an error.

Incorrect file name will cost you points!

Instructions for converting a Jupyter Python notebook to PDF: Go to the menu and choose, File --> Download As --> html. Open that html file and print it to PDF. Submit the PDF file **NOT** the html file.

If you are using Google Colab, remember to review the PDF before submitting to ensure that all cells and answers are displayed in the PDF.

Things to note:

- Each cell should display an output
- Use both Markdown and code comments in the Jupyter Notebook as needed

Wherever I have displayed an output, the results of your code should match my results. Both our outputs should be displayed. Do not erase my output displayed.

READ the IMPORTANT NOTE below

NumPy Exercises

Now that we've learned about NumPy let's test your knowledge. We'll start off with a few simple tasks and then you'll be asked some more complicated questions.

IMPORTANT NOTE! Make sure you don't run the cells directly above the example output shown, otherwise you will end up writing over the example output!

1. Import NumPy as np

In [1]:
```
import numpy as np
```

The line **`import numpy as np`** is a statement that imports the NumPy library and gives it an alias `np`. So that whenever we want to use numpy built-in functions, we just need to use np instead of numpy.

2. Create an array of 10 zeros (2 points)

In [2]:
```
# YOUR CODE HERE

np.zeros(10)
```

```
Out[2]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Explanation:

I've used **`np.zeros()`** from NumPy, to create a NumPy array of elements filled with zeros. We have to sepecify the number of elements in the array in the parentheses.

```
In [2]: # DON'T WRITE HERE
```

```
Out[2]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

3. Create an array of 10 ones (2 points)

```
In [3]: # YOUR CODE HERE
        np.ones(10)
```

```
Out[3]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Explanation:

I've used **`np.ones()`** from NumPy, to create a NumPy array of elements filled with ones. We have to sepecify the number of elements in the array in the parentheses.

```
In [3]: # DON'T WRITE HERE
```

```
Out[3]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

3. Create an array of all the even integers from 10 to 50 (3 points)

```
In [4]: # YOUR CODE HERE
        np.arange(10,51,2)
```

```
Out[4]: array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
               44, 46, 48, 50])
```

Explanation:

The **`arange()`** function in NumPy creates a NumPy array of evenly spaced values within a given interval. The third argument in **`np.arange()`** specifies the step size between each number in the array. In this case, it's set to 2, which means that the array contains only even numbers between the given starting number 10 and ending number 51.

```
In [6]: # DON'T WRITE HERE
```

```
Out[6]: array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
               44, 46, 48, 50])
```

4. Create a 3x3 matrix with values ranging from 0 to 8 (3 points)

```
In [5]: # YOUR CODE HERE
        np.arange(0,9).reshape(3,3)
```

```
Out[5]: array([[0, 1, 2],
               [3, 4, 5],
               [6, 7, 8]])
```

Explanation:

The **np.arange(0, 9)** function creates a 1D array containing the numbers 0 to 8, and then the **.reshape(3, 3)** method reshapes this 1D array into a 3x3 2D array.

In [7]:
```
# DON'T WRITE HERE
```

Out[7]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

5. Create a 3x3 identity matrix (2 points)

In [6]:
```
# YOUR CODE HERE

np.eye(3)
```

Out[6]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Explanation:

I used the **np.eye(n)** function in NumPy to create a **nxn** identity matrix, where n is the number of rows and columns. An identity matrix is a square matrix with ones on the main diagonal and zeros everywhere else.

In [8]:
```
# DON'T WRITE HERE
```

Out[8]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

6. Use NumPy to generate a random number between 0 and 1 (3 points)

NOTE: Your result's value should be different from the one shown below.

In [7]:
```
# YOUR CODE HERE

# Defining the value that's not to be matched
value = 0.65248055

# Loop until a different number is generated
while True:
    random_number = np.random.rand()  # Generate a random number between 0 and 1
    if random_number != value:        # Check if the new number is different from value
        break                         # If a different number is found, exit the loop

# Create a NumPy array with a single element
random_array = np.array([random_number])

# Setting the precision for printing arrays
np.set_printoptions(precision=8)

# Print the result
random_array
```

Out[7]:
```
array([0.63637561])
```

Explanation:

- First assigned the random number **0.65248055** to the variable **value** because that number should not be matched.
- Then created a while loop, to generate a random number that is different than **value**

- Used **`np.random.rand()`** from NumPy to generate a random float between 0 and 1.
- Then used **`np.array()`** to print the output in the desired way.
- **`np.set_printoptions()`** was used to set the precision to limit the number of decimals, which is 8 in **`value`**

In [9]:
```
# DON'T WRITE HERE
```

Out[9]:
```
array([0.65248055])
```

7. Create an array of 20 linearly spaced points between 0 and 1: (3 points)

In [8]:
```
# YOUR CODE HERE
np.linspace(0,1,20)
```

Out[8]:
```
array([0.        , 0.05263158, 0.10526316, 0.15789474, 0.21052632,
       0.26315789, 0.31578947, 0.36842105, 0.42105263, 0.47368421,
       0.52631579, 0.57894737, 0.63157895, 0.68421053, 0.73684211,
       0.78947368, 0.84210526, 0.89473684, 0.94736842, 1.        ])
```

Explanation:

To solve this, I've used **`linspace()`** function in NumPy to create a NumPy array of evenly spaced numbers within a given interval, and the third argument is used to specify the number of elements in the array instead of the step size between them.

In [12]:
```
# DON'T WRITE HERE
```

Out[12]:
```
array([0.        , 0.05263158, 0.10526316, 0.15789474, 0.21052632,
       0.26315789, 0.31578947, 0.36842105, 0.42105263, 0.47368421,
       0.52631579, 0.57894737, 0.63157895, 0.68421053, 0.73684211,
       0.78947368, 0.84210526, 0.89473684, 0.94736842, 1.        ])
```

Numpy Indexing and Selection

Now you will be given a starting matrix (be sure to run the cell below!), and be asked to replicate the resulting matrix outputs:

In [9]:
```
# RUN THIS CELL - THIS IS YOUR STARTING MATRIX
mat = np.arange(1,26).reshape(5,5)
mat
```

Out[9]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25]])
```

8. Write code that reproduces the output shown below. (3 points)

Be careful not to run the cell immediately above the output, otherwise you won't be able to see the output any more.

In [10]:
```
# YOUR CODE HERE
mat[2:, 1:5]
```

Out[10]:
```
array([[12, 13, 14, 15],
       [17, 18, 19, 20],
       [22, 23, 24, 25]])
```

Explanation:

- Used slicing to get the desired output.
- The second line of this code **`mat[2:, 1:5]`** extracts a subarray from mat. It selects the last three rows (having index 2, 3, and 4), and the columns second to last(having index 1, 2, 3, and 4).
- In simple terms, this creates a subarray that is a 3x4 matrix containing the elements in **the bottom two rows and the last four columns of mat.**

In [14]:
```
# DON'T WRITE HERE
```
Out[14]:
```
array([[12, 13, 14, 15],
       [17, 18, 19, 20],
       [22, 23, 24, 25]])
```

9. Write code that reproduces the output shown below. (2 points)

In [11]:
```
# YOUR CODE HERE
mat[0:3,1:2]
```
Out[11]:
```
array([[ 2],
       [ 7],
       [12]])
```

Explanation:

Similar to the previous question, I've used slicing here as well. To explain the line of code in simple terms, we are extracting **the elements in the first three rows and the second column** in the matrix **mat.**

In [16]:
```
# DON'T WRITE HERE
```
Out[16]:
```
array([[ 2],
       [ 7],
       [12]])
```

10. Write code that reproduces the output shown below. (2 points)

In [12]:
```
# YOUR CODE HERE
mat[3:, :]
```
Out[12]:
```
array([[16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25]])
```

Explanation:

Similar to the previous two questions, I've used slicing here as well. To explain the line of code in simple terms, we are extracting **the elements in the first two rows and all the columns** in the matrix **mat.**

In [18]:
```
# DON'T WRITE HERE
```
Out[18]:
```
array([[16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25]])
```

11. Get the sum of all the columns in the following format (3 points)

In [13]:
```
# YOUR CODE HERE
mat.sum(axis=0)
```
Out[13]:
```
array([55, 60, 65, 70, 75])
```

Explanation:

The `sum()` method in NumPy is used to calculate the sum of all the elements in an array or along a specified axis.The `axis=0` argument specifies that the sum should be calculated along the 0th axis, which corresponds to the columns.

In [21]:
```
# DON'T WRITE HERE
```

Out[21]:
```
array([55, 60, 65, 70, 75])
```

12. We worked a lot with random data with numpy, but is there a way we can insure that we always get the same random numbers? If so, write the code. (2 points)

In [14]:
```
# YOUR CODE HERE

# Using seed we can reproduce the same random numbers
np.random.seed(84)
np.random.rand(15)
```

Out[14]:
```
array([0.04604099, 0.37024232, 0.24231827, 0.62848001, 0.99011006,
       0.13058805, 0.17358916, 0.25831406, 0.41839897, 0.13101889,
       0.29045013, 0.49530609, 0.96700431, 0.29738986, 0.65180194])
```

In [15]:
```
np.random.seed(84)
np.random.rand(15)
```

Out[15]:
```
array([0.04604099, 0.37024232, 0.24231827, 0.62848001, 0.99011006,
       0.13058805, 0.17358916, 0.25831406, 0.41839897, 0.13101889,
       0.29045013, 0.49530609, 0.96700431, 0.29738986, 0.65180194])
```

Explanation:

The code `np.random.seed()` sets the random seed for the NumPy random number generator. Setting the seed ensures that the random numbers generated by NumPy will be the same every time the code is run.

As we've seen before the code `np.random.rand(n)` generates an array of `n` random numbers between 0 and 1, sampled from a uniform distribution.

Matplotlib Exercises

NOTE: ALL THE COMMANDS FOR PLOTTING A FIGURE SHOULD ALL GO IN THE SAME CELL. SEPARATING THEM OUT INTO MULTIPLE CELLS MAY CAUSE NOTHING TO SHOW UP.

We will focus on two commons tasks, plotting a known relationship from an equation and plotting raw data points.

Follow the instructions to complete the tasks to recreate the plots using this data:

13. Create data from an equation (10 points)

It is important to be able to directly translate a real equation into a plot. Your first task actually is pure numpy, then we will explore how to plot it out with Matplotlib. The world famous equation from Einstein:

$$E = mc^2$$

Use your knowledge of Numpy to create two arrays: E and m , where **m** is simply 11 evenly spaced values representing 0 grams to 10 grams. E should be the equivalent energy for the mass. You will need to figure out what to provide for **c** for the units m/s, a quick google search will easily give you the answer (we'll use the close approximation in our solutions).

In [16]:
```
# CODE HERE

# Defining the mass (m) values
m = np.linspace(0, 10, 11)
print(m)
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

In [11]:
```
# DON'T WRITE HERE or RUN THE CELL
```

```
The array m should look like this:

[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

In [17]:
```
# CODE HERE

# Defining the speed of light in meters per second (m/s)
c = 299792458

# Calculating energy using the formula E=mc^2
E = m * c**2

# Format the array to have the desired precision and scientific notation
np.set_printoptions(precision=1, suppress=True)

print('Array E :\n\n',E)
```

```
Array E :

 [0.0e+00 9.0e+16 1.8e+17 2.7e+17 3.6e+17 4.5e+17 5.4e+17 6.3e+17 7.2e+17
 8.1e+17 9.0e+17]
```

In [15]:
```
# DON'T WRITE HERE or RUN THE CELL
```

```
The array E should look like this:

 [0.0e+00 9.0e+16 1.8e+17 2.7e+17 3.6e+17 4.5e+17 5.4e+17 6.3e+17 7.2e+17
 8.1e+17 9.0e+17]
```

Explanation:

- Assuming the speed of light `c` is **299792458** meters per second, and
- Created an array of 11 evenly spaced values from 0 to 10 grams for the mass `m` using the `np.linspace()` function.
- Then used the formula `E = mc^2` to calculate the corresponding energy values `E` for each mass value `m` in the array.
- Used `np.set_printoptions(precision=1, suppress=True)` to set the print options to display floating point numbers with a precision of 1 decimal place, and to suppress very small values. So that the array E matches the desired output.

14. Plot E=mc^2 (10 points)

Now that we have the arrays E and m, we can plot this to see the relationship between Energy and Mass.

TASK: Import what you need from Matplotlib to plot out graphs:

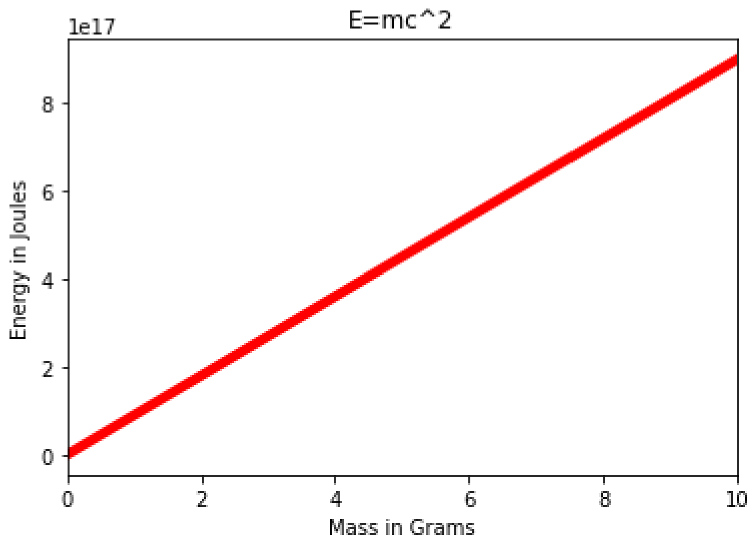
In [18]:

```
import matplotlib.pyplot as plt
#%matplotlib inline
```

TASK: Recreate the plot shown below which maps out E=mc^2 using the arrays we created in the previous task. Note the labels, titles, color, and axis limits. You don't need to match perfectly, but you should attempt to re-create each major component.

In [19]:

```
# CODE HERE
plt.plot(m,E,'red',lw=5)

# Set the limits for the x and y axes
plt.xlim(0, 10)
plt.ylim(np.max(E) * -0.05, np.max(E) * 1.05)

# Labelling x aixs and y axis
plt.xlabel('Mass in Grams')
plt.ylabel('Energy in Joules')
plt.title('E=mc^2') # setting the title of this plot

# To display the plot
plt.show()
```

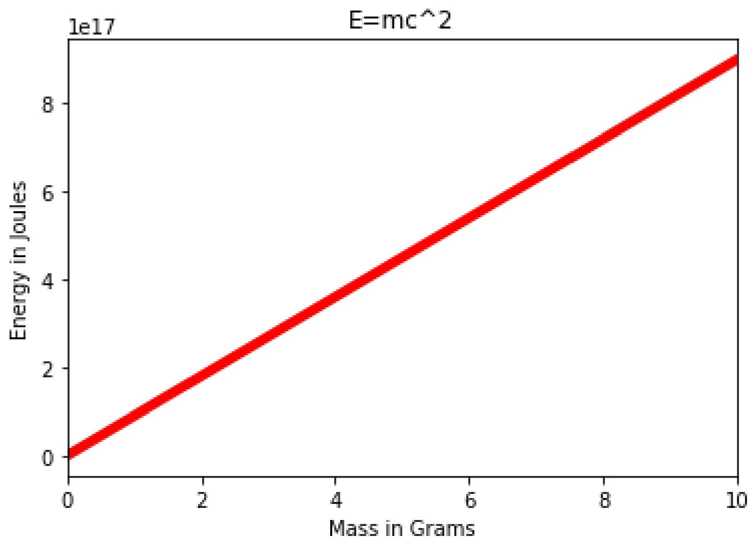


In [23]:

```
# DON'T RUN THE CELL BELOW< THAT WILL ERASE THE PLOT!
```

In [24]:

```

```



In this code, I used:

- `plt.plot()` function to plot m and E
- `plt.xlim()` to set limit of x-axis
- `plt.ylim(np.max(E) * -0.05, np.max(E) * 1.05)` to set the y limits
  - I've set the y-limits to start from a value that is a negative offset from zero to get the desired output.
  - The factor **-0.05** in the plt.ylim function call means that the y-axis will start from a value that is **5%** below the minimum value in the E array.
  - to leave some room at the top right corner of the plot, got the maximun value of `E` and multiplied it with **1.05**
- `plt.xlabel()` and `plt.ylabel()` to label the x-axis and y-axis, respectively, and
- `plt.title()` to give a title to the plot
- `plt.show()` was used to display the plot

---

15. Create plots from data points (15 points)

In finance, the yield curve is a curve showing several yields to maturity or interest rates across different contract lengths (2 month, 2 year, 20 year, etc. ...) for a similar debt contract. The curve shows the relation between the (level of the) interest rate (or cost of borrowing) and the time to maturity, known as the "term", of the debt for a given borrower in a given currency.

The U.S. dollar interest rates paid on U.S. Treasury securities for various maturities are closely watched by many traders, and are commonly plotted on a graph such as the one on the right, which is informally called "the yield curve".

For this exercise, we will give you the data for the yield curves at two separate points in time. Then we will ask you to create some plots from this data.

Part One: Yield Curve Data

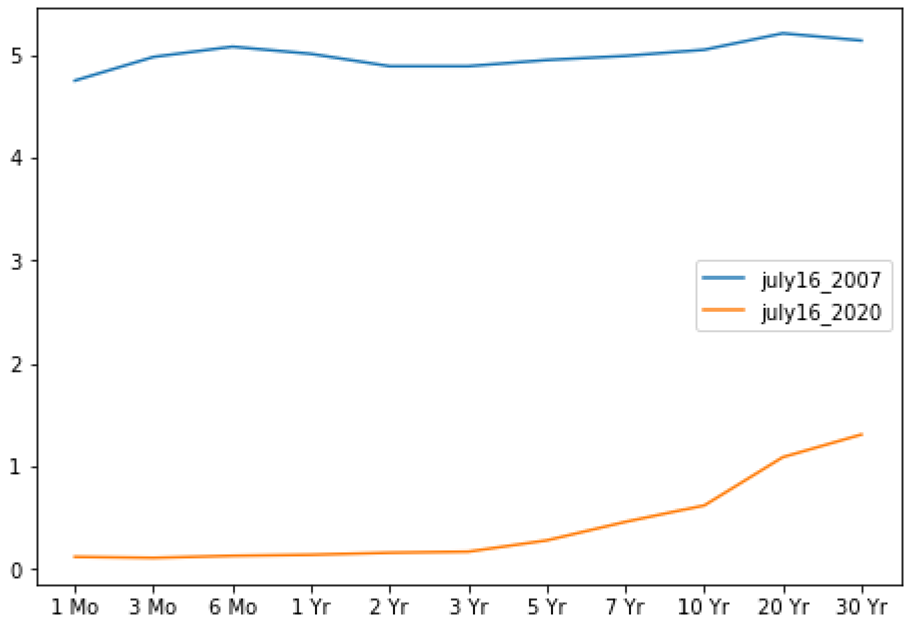
We've obtained some yield curve data for you from the US Treasury Dept.. The data shows the interest paid for a US Treasury bond for a certain contract length. The labels list shows the corresponding contract length per index position.

TASK: Run the cell below to create the lists for plotting.

In [20]:
```python
labels = ['1 Mo','3 Mo','6 Mo','1 Yr','2 Yr','3 Yr','5 Yr','7 Yr','10 Yr','20 Yr','30 Y

july16_2007 =[4.75,4.98,5.08,5.01,4.89,4.89,4.95,4.99,5.05,5.21,5.14]
july16_2020 = [0.12,0.11,0.13,0.14,0.16,0.17,0.28,0.46,0.62,1.09,1.31]
```

TASK: Figure out how to plot both curves on the same Figure. Add a legend to show which curve corresponds to a certain year.

In [22]:
```python
# CODE HERE

# Creating an empty canvas
fig = plt.figure()
ax = fig.add_axes([0,0,1,1]) # Adding set of axes to figure

# Plotting on that set of axes
ax.plot(labels, july16_2007, label ='july16_2007')
ax.plot(labels, july16_2020, label ='july16_2020')
ax.legend()
plt.show()
```



In this code:

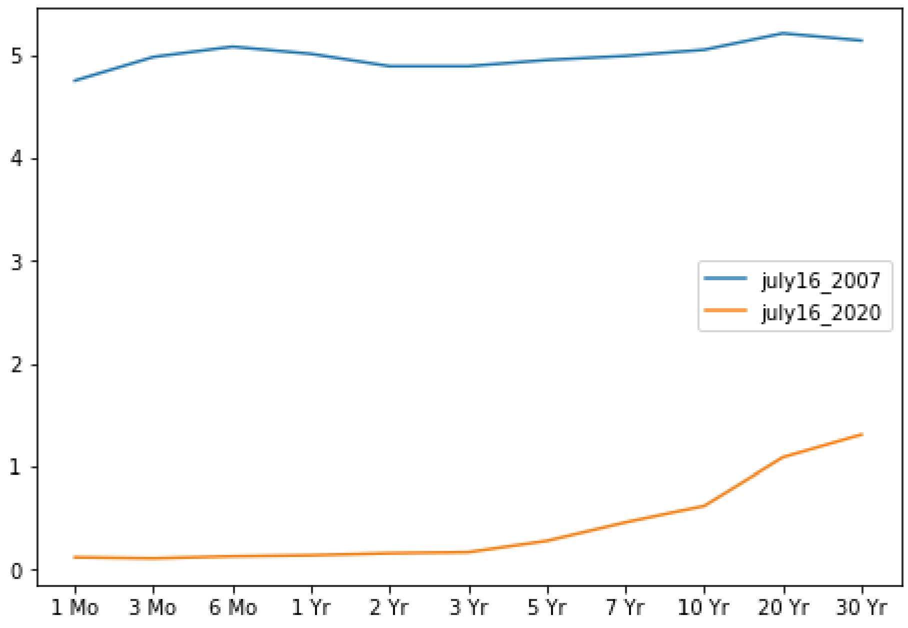
- First I've created a figure(an empty canvas) using **`plt.figure()`** ,

then used:

- **`.add_axes([0,0,1,1])`** function to creates a new set of axes that fills the entire figure.
- In the first **`ax.plot()`** line, plotting the july16_2007 curve on the axes, with the x-axis values provided by labels. The label parameter is set to july16_2007 to identify the curve for the legend.
- The second **`ax.plot()`** line does the same for the july16_2020 curve, with a label of july16_2020.
- Finally, **`ax.legend()`** adds a legend to the plot, with the labels for each curve automatically generated from the label parameters in the `ax.plot` calls.

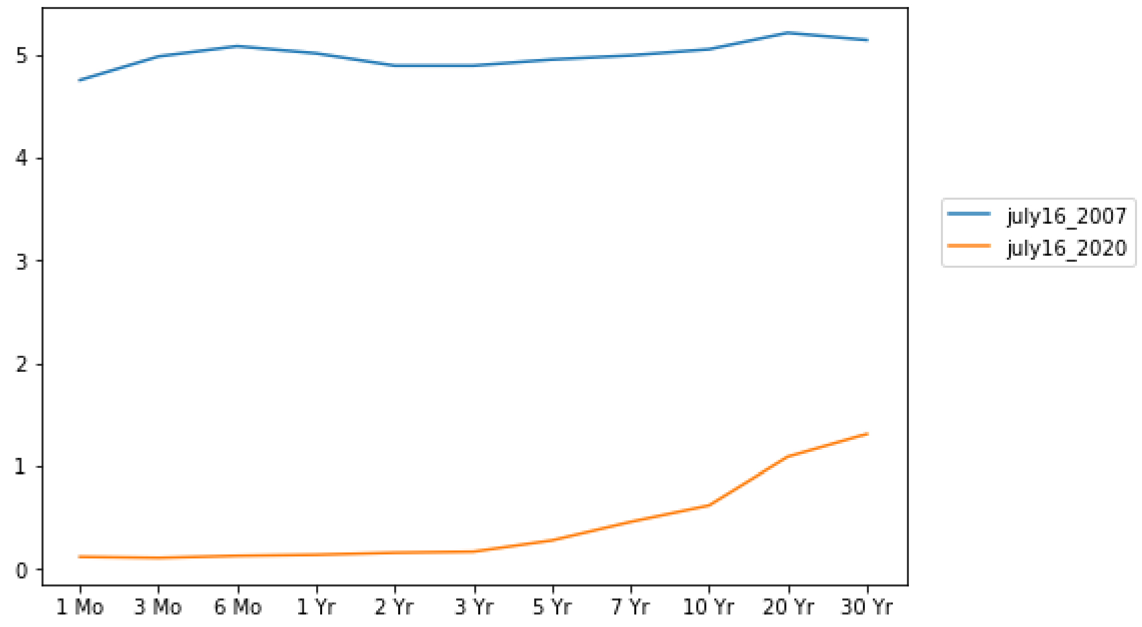
In [38]:
```
# DONT RUN THE CELL BELOW! IT WILL ERASE THE PLOT!
```

In [36]:
```

```



TASK: The legend in the plot above looks a little strange in the middle of the curves. While it is not blocking anything, it would be nicer if it were *outside* the plot. Figure out how to move the legend outside the main Figure plot.

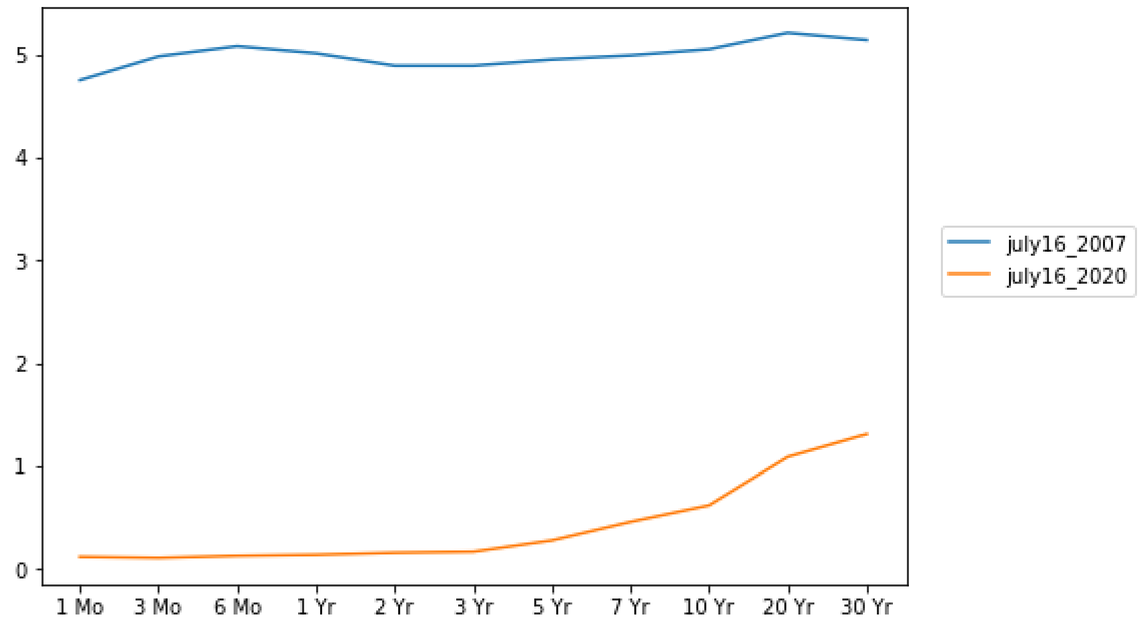
In [34]:
```
# CODE HERE

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

ax.plot(labels, july16_2007, label ='july16_2007')
ax.plot(labels, july16_2020, label ='july16_2020')
ax.legend(loc=(1.04,0.55))
plt.show()
```



To change the location of the legend on the plot used the `loc` parameter. Which specifies the location of the legend as a tuple of (x, y) coordinates. In this case I passed the values **`(1.05,0.55)`** to get the desired results.

In [45]:
```
# DONT RUN THE CELL BELOW! IT WILL ERASE THE PLOT!
```

In [47]:
```
```


july16_2007
july16_2020
1 Mo
3 Mo
6 Mo
1 Yr
2 Yr
3 Yr
5 Yr
7 Yr
10 Yr
20 Yr
30 Yr