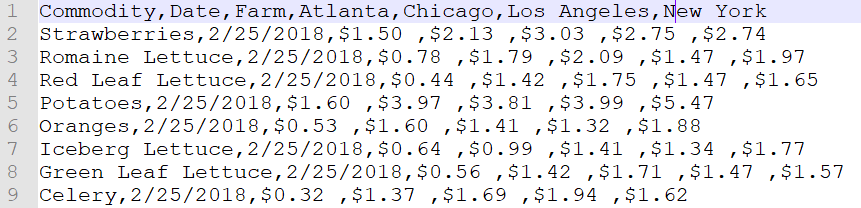
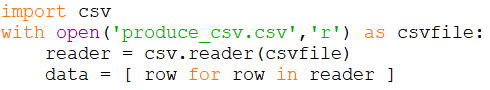
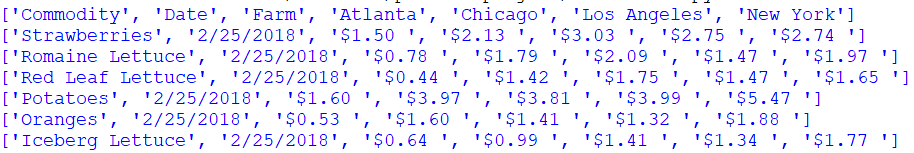
A CSV (comma separated value) file provides structured data in the form of text strings with pre-defined formatting rules. Python has a CSV module that can relieve you of some of the file-processing burden.

Here, I’m introducing a data set that we will use to practice our data handling skills. I’ll be calling it “commodity data”. The data file consists of seven columns. The column headers are in the first row of the data file: *Commodity* followed by *Date* followed by *Farm* followed by the names of four American cities. In each data row, *Commodity* is the name of a fruit or vegetable, followed by the *Date* associated with the row which is followed by five prices in dollars. The data at the top of the produce\_csv.csv file looks something like this …

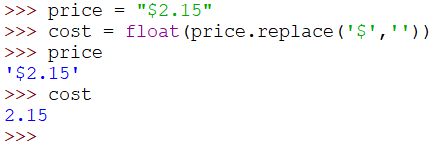


From line 2 of this data, we can see that on 2/25/2018, the price of a standard quantity of strawberries purchased at the farm was $1.50. Whereas, the same item cost $2.13 in Atlanta, $3.03 in Chicago, etc.

By now, you should understand that reading such a file into a data structure is quite simple. The code shown to the right loads data from the file into a list of lists. The first few lines …

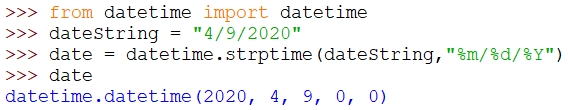


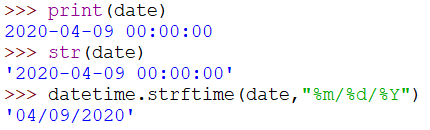
There are some challenging aspects to this data set. For example, some cities consist of two words and other cities are only one word. The dates are not in a standard format: the numbers that make up the month and day may be represented by one character or two. Also, dates represented by strings will not sort correctly if they are not converted to some sort-friendly form. Finally, the prices are strings that include a dollar sign which interferes with directly converting the price strings to float numbers.

For prices (formatted with $) it is necessary to remove the dollar sign character from the string and convert to float. This is easily done using the replace() method and converting to float:   
price = float(price.replace(‘$’,’’)) . This code simply replaces the ‘$’ with a null string (‘’). Another problem with currency is that large amounts often contain commas: $1,247,621.98 . The replace() method could also be used to replace all instances of comma with a null string. But we would have no opportunity to fix this if commas were also used as delimiters in the CSV file. Luckily, our data does not have large numbers like this. But if it did, a tab character would be a better choice than a comma as a delimiter.

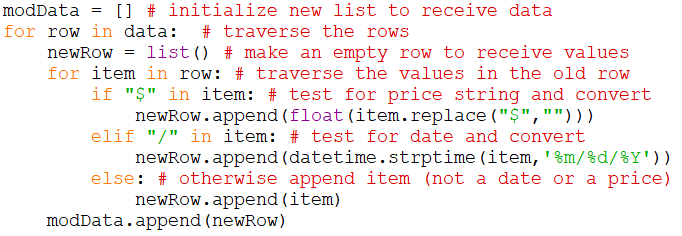
The problem with calendar dates is a bit more complicated. Clearly, there are many different ways that people express calendar dates around the world and we need to be able to handle all of them. Dates (and times) must be encoded in a consistent format and the encoded representations must be sortable. Python includes a module called *datetime* that will turn a date and time into a sortable datetime object. Use this statement to import the datetime module: from datetime import datetime . The datetime *module* provides methods to convert date strings into datetime objects that can be used for sorting and filtering. The dates in our file are formatted with either one or two characters for the month and day. The year contains four characters. We can convert a string containing a date like this:

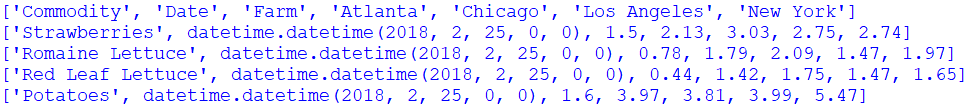
date = datetime.strptime(dateString,"%m/%d/%Y")

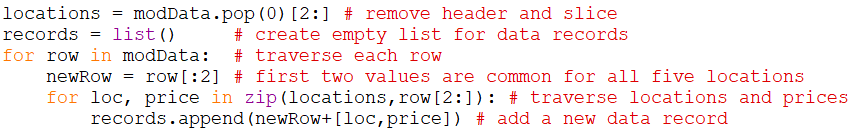
Here, the **date** variable is a datetime object. The method strptime stands for **str**ing **p**arse **time**. The date format is given by the string “**%m/%d/%Y**” . The year format string (“%Y”) is upper-case because it is four digits. If the year were given as only two digits, the format string would have been lower-case (“%y”). Month and day are represented by lower case because, in our data, these numbers may be one or two digits.

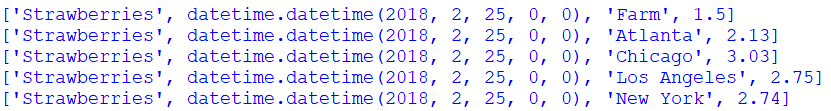
To print a date, it must be converted from a datetime object back to a string. Passing a datetime object to the str() function yields both date and time in standard ISO format (yyyy-mm-dd hh:mm:ss). To obtain the original date format, we pass the datetime object and a format string into the strftime() method.

Why do dates need to be sortable? In general, dates allow us to observe trends over time but only if the date is in a format that is quantifiable. When dates are simply strings, there is no easy way to do arithmetic with them to determine the interval of time between two dates. The datetime module allows us to do arithmetic with dates and times and … importantly … to use them as independent variables along the x-axis of a graph, for example.

Converting the data. Even though we have identified the specific functions and methods we need to convert an item of data, some thought needs to go into how to convert all of the data systematically. Recall that lists are mutable but strings are not. Our data structure is a list that contains lists of strings and strings are immutable. This means we need to either replace it with an appropriate value: floats for price strings and datetime objects for date strings *or* we need to build a new list. *In Python, building a new list is the preferred method* because (1) modifying the same list you are traversing is simply bad form and (2) to do so requires that you index the list element you wish to change which is also not Pythonic. It ought to be clear that we need to traverse the data list to obtain each row and then traverse each row to access each item. This indicates the need for nested for loops. Initialize a new list to receive converted values from the old list before entering the nested loops. The outer loop will create a new empty row before executing the inner loop which will append the new row with values from the original list which have been converted, when necessary, to the desired form. Study the code to the right until it makes sense to you. Now the first five lines of modData look like this …



Once all of the prices have been converted to float and the calendar dates have been converted to datetime objects, our data still needs to be converted from tabular format to be truly useful. If we were to convert the values in this table to data records, each record would have four attributes: commodity, date, location, and price. Every line of input data will generate five data records. Study the code to the right until it makes sense to you. The five data records generated by the first data line will look like this:



After the data records have been created, the data is finally in a format we can use for analysis. Data analysis is a compilation process that may constitute one or more subprocesses: data selection/exclusion (filtering), computation and visualization. The data set is quite large so filtering is absolutely necessary.

Data selection. Each record in my data list is organized in this format: commodity, date, location, price. The values can be accessed using indices 0 through 3, respectively. Knowing this organization is important because the index of the data represents the type of data it is. Python’s built-in **filter()** function can be used to select the data records we want when we provide an appropriate lambda function. Alternatively, we could convert each data record to a dictionary and access each item in a data record with a key string: commodity, date, location, and price.

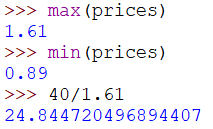
In the source code line below, the lambda function identifies the records for a specific commodity and location. So, the list select will contain the prices recorded for the retail sale of Oranges in Chicago across all available dates. The filter function returns a filter object, so the source code line below converts the filter object to a list. 

Now we can collect the dates and prices for Oranges in Chicago from the select list using simple list comprehensions.



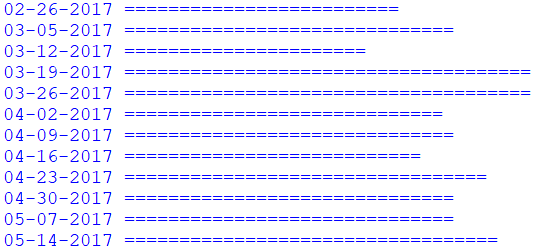
These two lists can be used to visualize how the price of oranges in Chicago changed over time. Remember how you generated a text graph earlier in the semester? We could try to do that here. But we need to reveal some assumptions about our data. Because we have been working with lists and lists are ordered collections, the order of the data in our dates and prices lists will be as they were in the original input data. That data may have been sorted based on date or sorted alphabetically based on commodity or maybe put together at random. We generally don’t have any clue. So, if we want to plot the prices in chronological order, we will need to sort the data based on dates. But we have separate lists for date and price, so we will need to first bring them together and then sort that list based on date. Luckily, this is easy to do with a list comprehension. Here’s how I would merge the two lists and sort the merged list by date …



Now, we can use our textgraph skill (remember that?) to plot the prices over time. But first, I ought to find out the maximum and minimum price because I will need to scale the prices so the maximum fits within 40 columns to get meaningful results on the screen. So I’ll multiply the price by 25 to get a nice graph in a maximum of 40 columns. Study and understand the code below. Here, the datetime object is converted to a date string and printed before a string of equal signs that represents the price on that day.



The first several lines of the textgraph are shown below …



At this point, we have gone through the process of reading a data file, converting the values to meaningful data types, transforming the format from tabular to individual data records, selecting individual data records for analysis, sorting them and generating a crude graph. It is necessary for you to completely understand all of these data manipulation techniques. I strongly suggest that you load this data set into your own program and work with it. Be curious about what questions you might be able to ask of it.

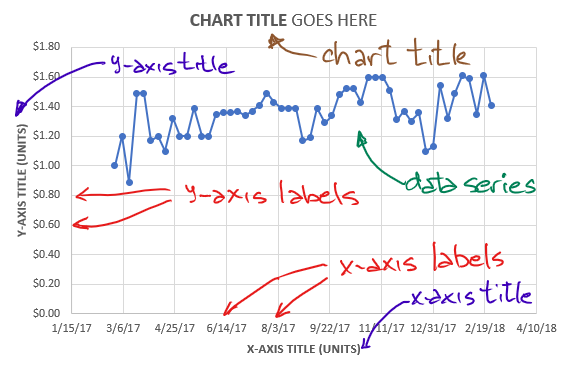
Big time data visualization.

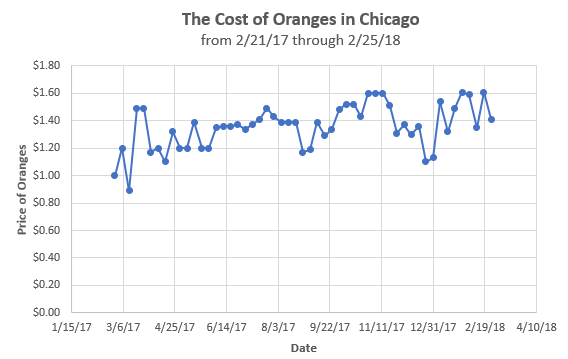
Now it’s time to move up to using some popular packages to visualize this data. Each package conceptualizes the generation of graphics a bit differently. FOR THE REST OF THE SEMESTER WE WILL BE WORKING IN THE ANACONDA ENVIRONMENT USING SPYDER AS THE SOURCE CODE EDITOR. YOU SHOULD ALSO MAKE THIS CHANGE.

Why didn’t we do this from the beginning? In Anaconda, the choices are overwhelming and the opportunities to break the environment are numerous. Using Idle instead, avoided these problems and helped us focus on learning the language.

Why are we doing this now? From this point forward we will be experimenting with various modules. Anaconda gives us access to many useful data visualization modules. If you seriously wish to move forward in the data sciences, Anaconda is a tool designed for data science activities.

We’ll start by using Spyder to edit and run our source code. You can keep several documents open at once in Spyder which is helpful when working with source code and data files.

Visualization basics. *Every graph you generate* should have *all* of the elements shown on the graph to the right.

* An x-axis and a y-axis containing
  + **Axis labels** in an understandable format
  + An **axis title** that also indicates the *physical units* represented on the axis
* A descriptive **chart title**
* One or more **data series**
  + Points on the data series are indicated by **markers**
  + The markers may be connected together with **lines** that are the same **color** as the markers
  + If there is more than one series plotted
    - Each series should have its own distinct marker and color
    - A **legend** should associate a label for each data series with the marker/line shown

Exploring data visualization using matplotlib. The matplotlib package contains several modules. The module that we will use to generate graphs is called pyplot. So we can import matplotlib.pyplot and give it a shorter name to work with (plt): import matplotlib.pyplot as plt .

Now we can use the pyplot module to draw a graph showing the price of oranges in Chicago over time. As you might expect, there are quite a few steps involved to draw even the most basic graph. As we go through those steps, we will often be using default values for things as there are many, many parameters associated with any graph. The first two statements establish the basis for a “figure”. First, we initialize a figure and then we add a set of axes to the figure for our graph.



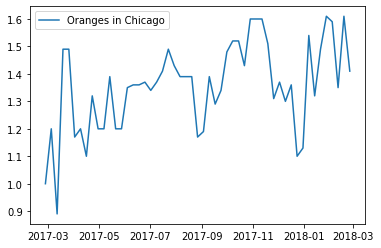
Now we can add our data series to the axis …



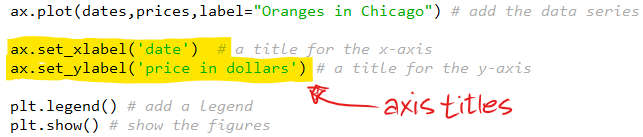
… and to actually see the plot, we need to execute these lines …



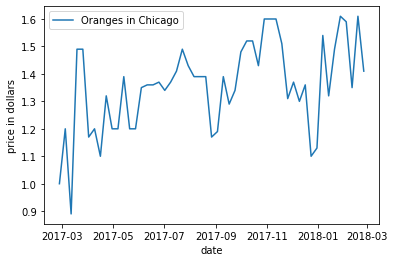
Here’s the result …



It looks like oranges are available year-round in Chicago. But the graph needs some axis labels. We can add axis labels like this …



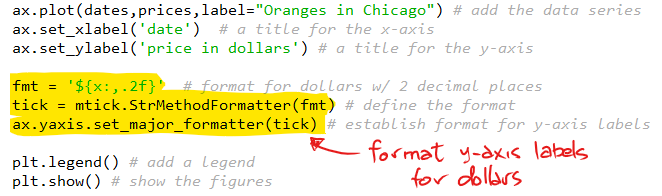
… and get this result …



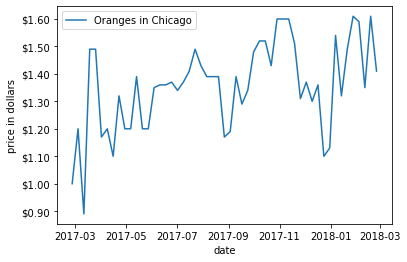
Still, the y-axis should be formatted in currency with a dollar sign. To do this, we need to import another module called the matplotlib ticker.



Now, we can establish a format string for the y-axis values …



… and the result is a plot like this …

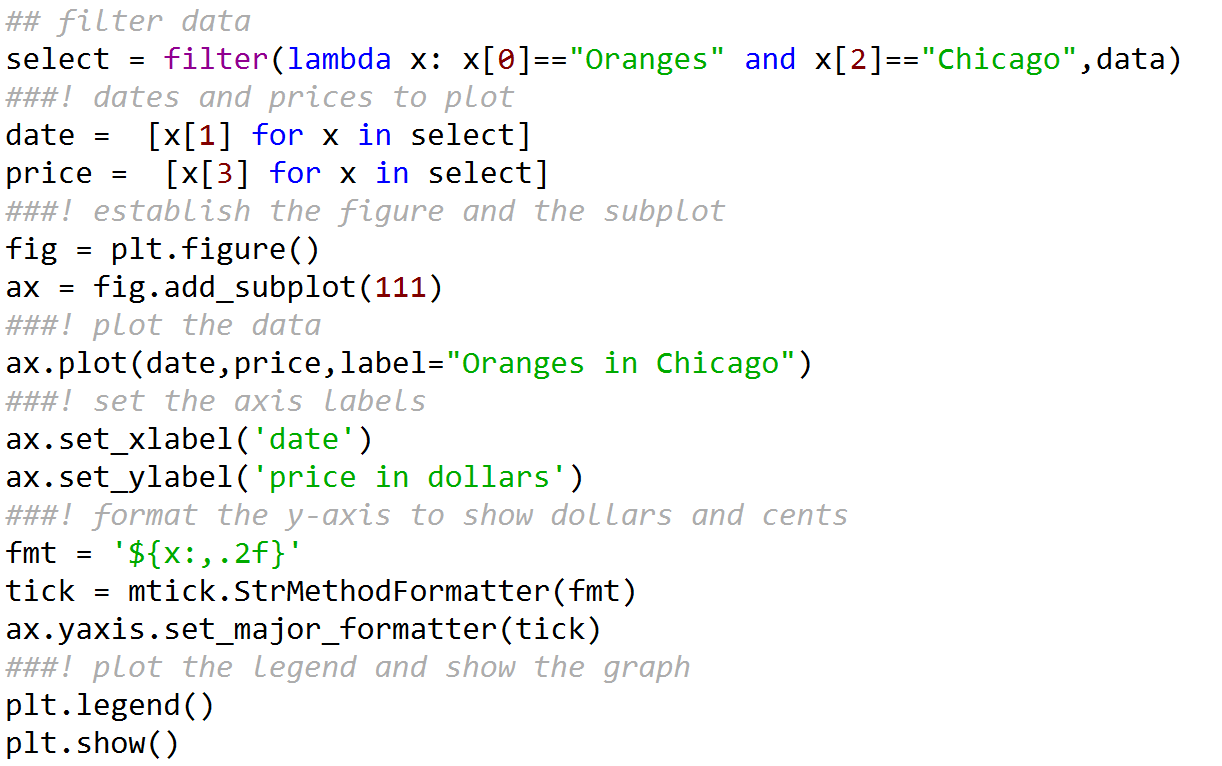


There are still many tweaks that could be made to this graph. Where are the markers? Can I change the color of the line? What about the title for the graph? How can I add another series to compare the price of oranges in two cities? Many other plot types are possible in matplotlib. However, to get the exact result you are looking for may take a bit of searching and fiddling. When fiddling, always read documentation as you proceed.

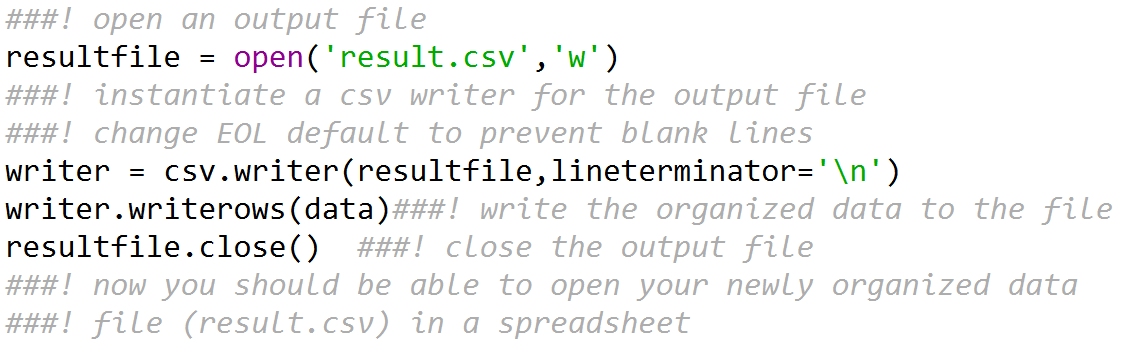
**ASSIGNMENT:**

**This week is a major shift in focus towards the final project which will include a large set of commodity data. It’s important that you (1) have all of the tools in place to replicate what I have demonstrated above and (2) develop an understanding of all of the operations I have described. Follow the instructions above and replicate the results in Anaconda. Upload your source code to Moodle. As always, your code must be fully commented to receive full credit.**

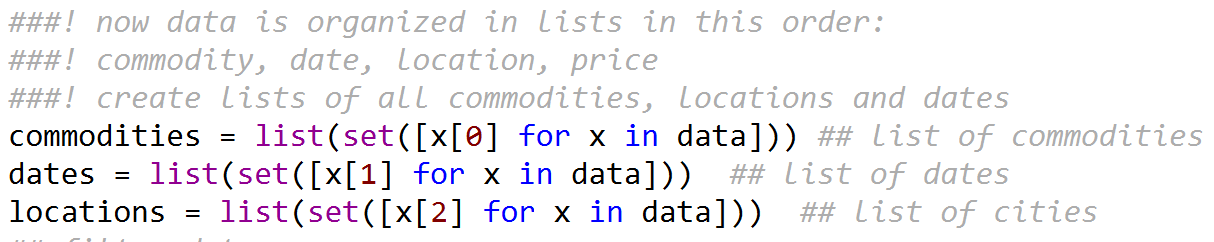
METHOD FOR FILTERING DATA AND PLOTTING A LINE GRAPH AS DESCRIBED



ADDITIONAL CODE TO SAVE YOUR DATA RECORDS TO A NEW CSV FILE



ADDITIONAL CODE TO CREATE LISTS OF COMMODITIES, LOCATIONS AND DATES W/O DUPLICATES



METHOD FOR READING TEXT FROM THE DATA FILE INTO INDIVIDUAL RECORDS (W/CONVERSION)

