CptS 111, Spring 2023
Lect. #25, Apr. 24, 2023
Class Notes

Today's Agenda:

1. Plotting data from files
2. The `numpy` module
3. Figures with subplots

Ch. 10 (cont.)

Plotting (cont.)

1. Extracting Data for Plotting from Files

The toy examples we considered in our last lecture are useful for illustrating how to make plots, but typically we want to extract data from a file or files and then plot these data. **Quite often the approach to doing this is first to look at the format of the data in a file, and when possible, use the `.split()` method to create lists for plotting. This is very effective for `.csv` files because fields are separated by commas.**

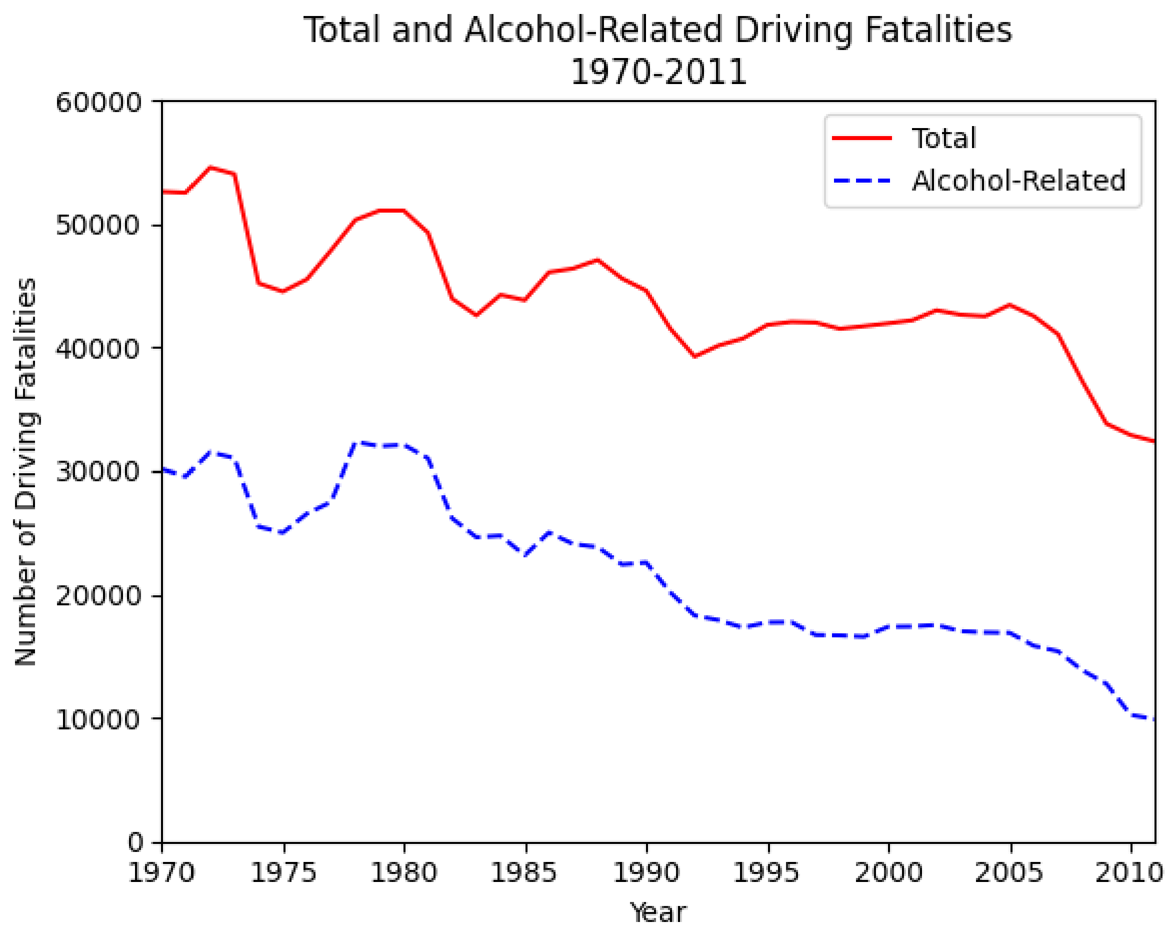
```
In [1]: # Import matplotlib.pyplot, initialize lists, use with to
        # open file, use .split() to create lists from the lines of
        # the file, and append list values as appropriate.

        import matplotlib.pyplot as plt

        yrs = []
        deaths = []
        alcohol_deaths = []
        with (open('dd_stats.csv')) as file_in:
            for line in file_in:
                line_list = line.split(',')      # split on ','
                yrs.append(int(line_list[0]))    # int() first element in list
                deaths.append(int(line_list[1])) # int() second element in list
                alcohol_deaths.append(int(line_list[2])) # and so on
        print(f'years: {yrs}\ntotal deaths: {deaths}\nalcohol-related deaths: {al

        # Create plot with two lines, axes labels, title, and legend
        plt.plot(yrs, deaths, 'r-', label='Total')  # solid red line
        plt.plot(yrs, alcohol_deaths, 'b--', label='Alcohol-Related')  # dashed b
        plt.axis([1970, 2011, 0, 60000])  # [x1, x2, y1, y2]
        plt.xlabel('Year')
        plt.ylabel('Number of Driving Fatalities')
        plt.title('Total and Alcohol-Related Driving Fatalities\n1970-2011')
        plt.legend(loc='upper right')
        plt.show()
```

```
years: [1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 198
0, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 19
92, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2
004, 2005, 2006, 2007, 2008, 2009, 2010, 2011]
total deaths: [52627, 52542, 54589, 54052, 45196, 44525, 45523, 47878,
50331, 51093, 51091, 49301, 43945, 42589, 44257, 43825, 46087, 46390, 4
7087, 45582, 44599, 41508, 39250, 40150, 40716, 41817, 42065, 42013, 41
501, 41717, 41945, 42196, 43005, 42643, 42518, 43443, 42532, 41059, 372
61, 33808, 32885, 32367]
alcohol-related deaths: [30150, 29552, 31500, 31022, 25500, 25004, 2655
0, 27500, 32339, 31980, 32100, 31005, 26173, 24635, 24762, 23167, 2501
7, 24094, 23833, 22424, 22587, 20159, 18290, 17908, 17308, 17732, 1774
9, 16711, 16673, 16572, 17380, 17400, 17524, 17013, 16919, 16885, 1582
9, 15387, 13846, 12744, 10228, 9878]
```



Next, we'll briefly discuss the use of the `numpy` module as well as how to create figures with multiple plots using `matplotlib`.

2. The numpy Module

`numpy` is a library of math functions, many of which are used with matrixes. In computer science, matrixes are often called `n` -dimensional arrays, where `n` can be as small as 1. One of the benefits of using `numpy` is that operations with arrays can be performed very easily. We won't actually be exploiting the power of `numpy` in this course. However, I want to give you some sense of how useful it can be. Consider the following list of integers.

```
In [2]: # List of integers

        ints = [0, 1, 2, 3, 4, 5]
        ints
```

```
Out[2]: [0, 1, 2, 3, 4, 5]
```

Suppose we want to square each value in this list (without using list comprehension, which is another very useful Python concept beyond the scope of this course; an optional section on list comprehension is available in Ch. 9 in your zyBook). To square the integer values, we can use a `for` -loop.

```
In [3]: # Square elements in list of integers using an iterating for-loop:

        ints_squared = []
        for num in ints:
            ints_squared.append(num ** 2)
        ints_squared

Out[3]: [0, 1, 4, 9, 16, 25]
```

Let's see how this can be done using `numpy`. First we need to import `numpy`.

```
In [4]: # Import numpy as np which is the standard alias for numpy

        import numpy as np
```

Note that `np` is commonly used as the alias for `numpy` by many Python programmers. We can then use the `array()` function in `numpy` to convert the list `ints` to a 1-D array.

```
In [5]: # Use numpy to convert list of integers to array of integers
        # Note that array elements aren't separated by commas

        a_ints = np.array(ints)
        print(f'Notice that our list of integers is printed as {ints}')
        print(f'while our array of integers is printed as {a_ints}.')

        Notice that our list of integers is printed as [0, 1, 2, 3, 4, 5]
        while our array of integers is printed as [0 1 2 3 4 5].
```

What's the difference?

Next, we see how useful `numpy` can be. If we want to find the square of the values of the 1-D array `a_ints`, we don't have to use a `for`-loop. We simply do the following.

```
In [6]: # Squaring integers without use of a for-loop; note that Python
        # knows how to perform math operations with arrays

        a_ints_sq = a_ints ** 2
        print(f'{a_ints_sq} has extra spaces based on the largest value in it.')

        [ 0  1  4  9 16 25] has extra spaces based on the largest value in it.
```

Let's consider two more math operations with arrays.

```
In [8]: # As mentioned above, Python can perform math operations on arrays.

        array_sum = a_ints + a_ints_sq
        array_product = a_ints * a_ints_sq
        print(f'A sum {array_sum} and product {array_product} of two arrays.')

        A sum [ 0  2  6 12 20 30] and product [  0   1   8  27  64 125] of two
        arrays.
```

OTOH, some math operations for arrays require the use of `numpy` .

```
In [11]: # We need to use np.exp() to find the exponential values of the integers

         a_ints_exp = np.exp(-a_ints)
         print(a_ints_exp)
```

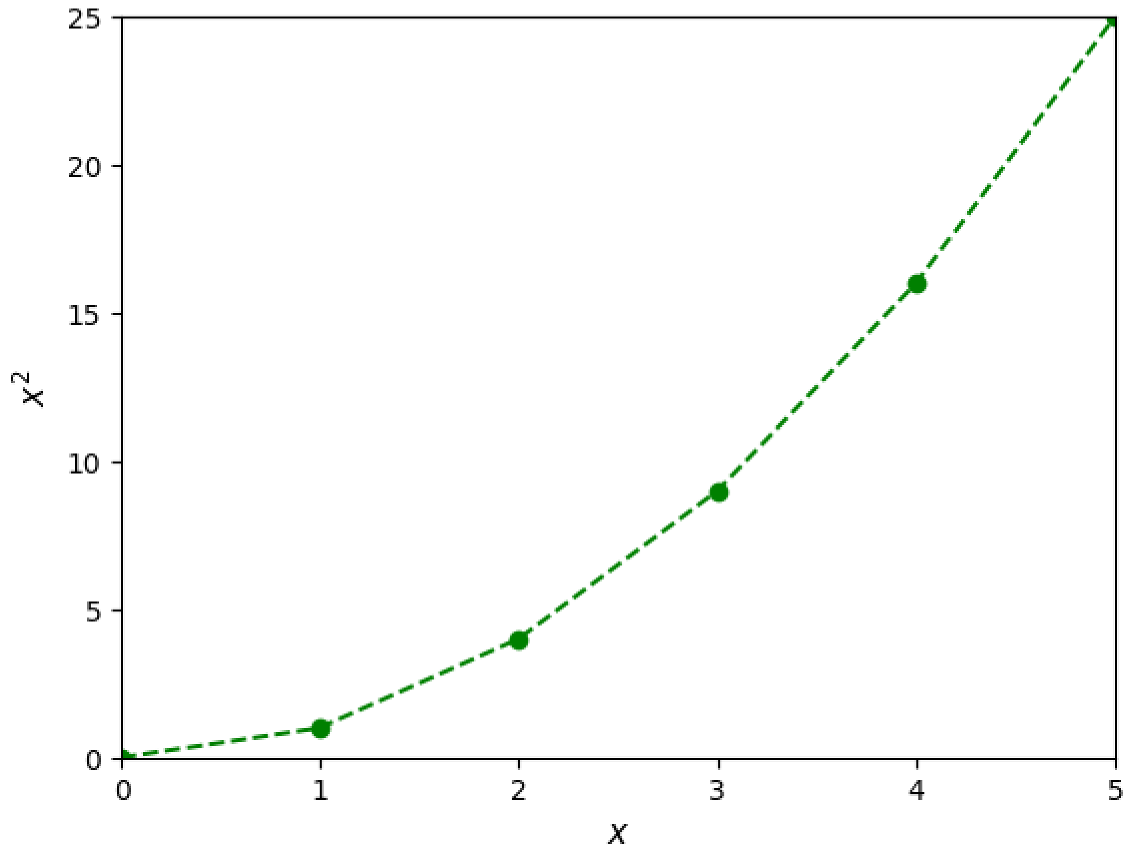
```
[1.         0.36787944 0.13533528 0.04978707 0.01831564 0.00673795]
```

Notice how Python identifies the array.

We've used lists as arguments in the `plot()` function. We can also use 1-D arrays, which are similar to lists, as arguments.

```
In [12]: # Use 1-D arrays as arguments of plot()

         plt.plot(a_ints, a_ints_sq, 'go--') # lists or arrays can be used
         plt.axis([0, 5, 0, 25])
         plt.xlabel('$x$', fontsize=12)  # $$ delimit math mode
         plt.ylabel('$x^2$', fontsize=12)
         plt.show()
```



`numpy` also has a function similar to the `range()` function. The function in `numpy` creates a sequence of values (integers or floats) in an array which can then be used as an argument in the `plot()` function.

In [15]:
```
# The arange() function creates an array of values, floats or integers

x = np.arange(0, 5.5, 0.5) # (start, stop, increment); as with the range(
print(x)                   # function, the stop value isn't included
```

```
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
```

In [16]:
```
# The default values for start and stop are 0 and 1, respectively

y = np.arange(6)
print(y)
```

```
[0 1 2 3 4 5]
```

You'll use the `arange()` function in this week's lab.

3. Figures with Subplots

It's easy to create figures with subplots using `matplotlib` as you'll do in Lab #12. Let's consider an example.

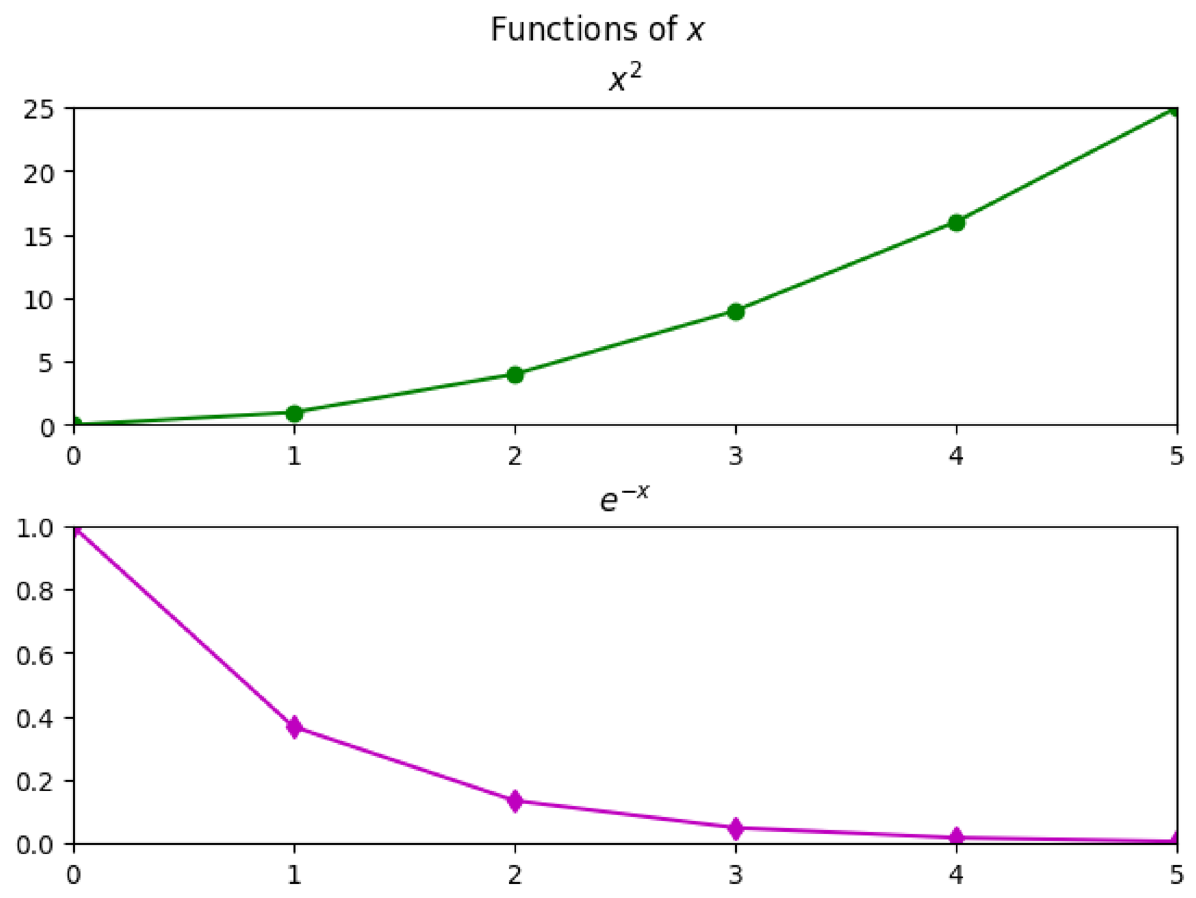
```
In [17]: # Use of subfig to create subplots using matplotlib

         fig, subfig = plt.subplots(2, 1, constrained_layout=True)
         fig.suptitle('Functions of $x$')              # overall title

         subfig[0].plot(a_ints, a_ints_sq, 'go-')      # single index for subfig
         subfig[0].set_xlim(0, 5)                      # x-axis limits
         subfig[0].set_ylim(0, 25)                     # y-axis limits
         subfig[0].set_title('$x^2$')                  # subplot title

         subfig[1].plot(a_ints, a_ints_exp, 'md-')
         subfig[1].set_xlim(0, 5)
         subfig[1].set_ylim(0, 1)
         subfig[1].set_title('$e^{-x}$')

         plt.show()
```



In the code above:

- `fig` : identifies elements of the overall figure
- `subfig` : 2x1 array of subplots, where 2 is the number of rows and 1 is the number of columns; in Lab #12, you'll use a 2x2 array of subplots
- `constrained_layout=True` : prevents subplots from overlapping
- `x` , `y` limits of subplots require different functions and argument types
- titles of subplots require different functions