Siddhanth Kumar Section 102

Workshop 4

File Input and Output (I/O)

Submit this notebook to bCourses (ipynb and pdf) to receive a grade for this Workshop.

Please complete workshop activities in code cells in this iPython notebook. The activities titled **Practice** are purely for you to explore Python. Some of them may have some code written, and you should try to modify it in different ways to understand how it works. Although no particular output is expected at submission time, it is *highly* recommended that you read and work through the practice activities before or alongside the exercises. However, the activities titled **Exercise** have specific tasks and specific outputs expected. Include comments in your code when necessary. The workshop should be submitted on bCourses under the Assignments tab.

The homework should be submitted on bCourses under the Assignments tab (both the .ipynb and .pdf files). Please label it by your student ID number (SIS ID)

Exercises start here

In this notebook, we're going to explore some ways that we can store data in files, and extract data from files. Let's just get all of the importing out of the way:

In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Practice: Basic Writing and Reading ASCII Files

Think of ASCII files as text files. You can open them using a text editor (like vim or emacs in Unix, or Notepad in Windows) and read the information they contain directly. There are a few ways to produce these files, and to read them once they've been produced. In Python, the simplest way is to use file objects.

Let's give it a try. We create an abstract file object by calling the function `open( filename, access_mode )` and assigning its return value to a variable (usually `f`). The argument `filename` just specifices the name of the file we're interested in, and `access_mode` tells Python what we plan to do with that file:

```
'r': read the file
'w': write to the file (creates a new file, or clears an existing file)
'a': append the file
```

Note that both arguments should be strings. For full syntax and special arguments, see documentation at https://docs.python.org/3/library/functions.html#open

In [2]:
```
f = open( 'welcome.txt', 'w' )
```

A note of caution: as soon as you call `open()`, Python creates a new file with the name you pass to it if you open it in write mode (`'w'`). Python will overwrite existing files if you open a file of the same name in write ('`w`') mode.

Now we can write to the file using `f.write( thing_to_write )`. We can write anything we want, but it must be formatted as a string.

In [3]:
```
topics = ['Data types', 'Loops', 'Functions', 'Arrays', 'Plotting', 'Statistics']
```

In [4]:
```
f.write( 'Welcome to Physics 77, Spring 2022\n' ) # the newline command \n tells Python
f.write( 'Topics we will learn about include:\n' )
for top in topics:
    f.write( top + '\n')
f.close()                                         # don't forget this part!
```

Practice 1: Use the syntax you have just learned to create an ASCII file titled "`sine.txt`" with two columns containing 20 x and 20 y values. The x values should range from 0 to 2π - you can use `np.linspace()` to generate these values (as many as you want). The y values should be $y = sin(x)$ (you can use `np.sin()`) for this. Then, use a `for` loop as above to write a new line for each pair of x and y values. To make sure that each x,y pair is on a new line, remember to add `\n` to the end of each line like above. To separate the values by a tab so that the columns are nicely aligned, you can use the "character" `\t`. So `\n` inserts a new line and `\t` inserts a tab. You may wish to use some kind of string formatting to decimals from running too far. Here is an example with just one data point:

```
x = 0.5 * np.pi
y = np.sin(x)
print("%.5f \t %.5f" % (x,y))
```

Pay close attention to the fact that when you use the `write` function, the argument that you pass to it needs to be a string.

In [23]:
```
# Code for Practice 1

s = open('sine.txt', 'w')

TwoPI = 2 * np.pi

x = np.linspace(0, TwoPI, 20)
def calc(x):
    return np.sin(x)

for val in x:
    s.write('{0:.5f} {1:.5f}\n'.format(val, calc(val)))
    print('{0:.5f} {1:.5f}'.format(val, calc(val)))

s.close()
```

```
0.00000 0.00000
0.33069 0.32470
0.66139 0.61421
0.99208 0.83717
1.32278 0.96940
1.65347 0.99658
1.98416 0.91577
2.31486 0.73572
2.64555 0.47595
2.97625 0.16459
```

```
3.30694 -0.16459
3.63763 -0.47595
3.96833 -0.73572
4.29902 -0.91577
4.62972 -0.99658
4.96041 -0.96940
5.29110 -0.83717
5.62180 -0.61421
5.95249 -0.32470
6.28319 -0.00000
```

Now we will show how to *read* the values from `welcome.txt` back out:

In [28]:
```
f = open( 'welcome.txt', 'r' )
for line in f:
    print(line.strip())
f.close()
```

```
Welcome to Physics 77, Spring 2022
Topics we will learn about include:
Data types
Loops
Functions
Arrays
Plotting
Statistics
```

Practice 2: In the cell immediately above, you see that we print `line.strip()` instead of just printing `line`. Remove the `.strip()` part and see what happens.

Suppose we wanted to skip the first two lines of `welcome.txt` and print only the list of topics `('Data types', 'Loops', 'Functions', 'Arrays', 'Plotting', 'Statistics')`. We can use `readline()` to "read" the first two lines but not store their value, thereby ignoring them.

In [31]:
```
f = open( 'welcome.txt', 'r' )
f.readline()
f.readline() # skip the first two lines
topicList = []
for line in f:
    topicList.append(line.strip())
f.close()
print(topicList)
```

```
['Data types', 'Loops', 'Functions', 'Arrays', 'Plotting', 'Statistics']
```

Python reads in spacing commands from files as well as strings. The `.strip()` just tells Python to ignore those spacing commands. What happens if you remove it from the code above?

Practice 3: Use the syntax you have just learned to read back each line of x and y values from the `sine.txt` file that you just wrote in Practice 1. Don't worry about breaking up the lines into individual values quite yet.

In [3]:
```
# Code for Practice 3

s = open('sine.txt', 'r')

for line in s:
    print(line.strip())
```

```
0.00000 0.00000
0.33069 0.32470
0.66139 0.61421
```

```
0.99208 0.83717
1.32278 0.96940
1.65347 0.99658
1.98416 0.91577
2.31486 0.73572
2.64555 0.47595
2.97625 0.16459
3.30694 -0.16459
3.63763 -0.47595
3.96833 -0.73572
4.29902 -0.91577
4.62972 -0.99658
4.96041 -0.96940
5.29110 -0.83717
5.62180 -0.61421
5.95249 -0.32470
6.28319 -0.00000
```

Practice Reading in Numerical Data as Floats

Numerical data can be somewhat trickier to read in than strings. In the practices above, you read in `sine.txt` but each line was a `string` not a pair of `float` values. Let's read in a file I produced in another program, that contains results from a BaBar experiment, where we searched for a "dark photon" produced in e+e- collisions. The data are presented in two columns:

```
mass    charge
```

Every time we read in a new line, it is going to start out being a `string`. To convert a line like

```
1.57079      1.00000
```

into a pair of values we need to do two things. The first is we need to split that string into two pieces. Fortunately, there is a function to do that for us. Suppose that we read in a `line` and we want to split it. We can do it as follows:

```
line.split()
```

For the line above, calling `.split()` would return the following `list`:

```
['1.57079','1.00000']
```

From there, we need to convert each value in the list into a `float` and store those values somewhere. This can be done using the `float()` function:

```
x_values = []
y_values = []
split_values = ['1.57079','1.00000']
x_values.append(float(split_values[0]))
y_values.append(float(split_values[1]))
```

Now `x_values` is a `list` containing 1 element which is the `float` value `1.57079` and `y_values` is a `list` containing 1 element which is the `float` value `1.00000`.

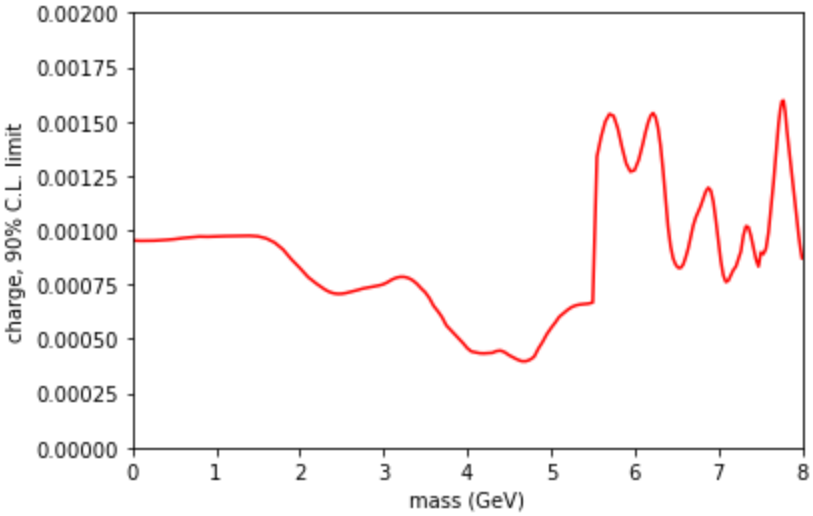
In [5]:
```
# Example using BaBar_2016.dat

f = open('BaBar_2016.dat', 'r')
# read each line, split the data wherever there's a blank space,
# and convert the values to floats

# lists where we will store the values we read in
mass = []
charge = []
for line in f:
    tokens = line.split()
    mass.append(float(tokens[0]))
    charge.append(float(tokens[1]))
f.close()

print(mass)
print(charge)
```

```
[0.001, 0.01, 0.1, 0.18, 0.26, 0.34, 0.42, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3,
1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2, 2.3, 2.35, 2.4, 2.45, 2.5, 2.55, 2.6, 2.65,
2.7, 2.75, 2.8, 2.85, 2.9, 2.95, 3.0, 3.05, 3.1, 3.15, 3.2, 3.25, 3.3, 3.35, 3.4, 3.45,
3.5, 3.55, 3.6, 3.65, 3.7, 3.75, 3.8, 3.85, 3.9, 3.95, 4.0, 4.05, 4.1, 4.15, 4.2, 4.25,
4.3, 4.35, 4.4, 4.45, 4.5, 4.55, 4.6, 4.65, 4.7, 4.75, 4.8, 4.85, 4.9, 4.95, 5.0, 5.05,
5.1, 5.15, 5.2, 5.25, 5.3, 5.35, 5.4, 5.45, 5.5, 5.55, 5.6, 5.65, 5.7, 5.75, 5.8, 5.85,
5.9, 5.95, 6.0, 6.05, 6.1, 6.13, 6.16, 6.19, 6.22, 6.25, 6.28, 6.31, 6.34, 6.37, 6.4, 6.
43, 6.46, 6.49, 6.52, 6.55, 6.58, 6.61, 6.64, 6.67, 6.7, 6.73, 6.76, 6.79, 6.82, 6.85,
6.88, 6.91, 6.94, 6.97, 7.0, 7.03, 7.06, 7.09, 7.12, 7.15, 7.18, 7.21, 7.27, 7.3, 7.33,
7.36, 7.39, 7.42, 7.45, 7.48, 7.51, 7.54, 7.57, 7.6, 7.63, 7.66, 7.69, 7.72, 7.74, 7.76,
7.78, 7.8, 7.82, 7.88, 7.9, 7.92, 7.94, 7.96, 7.98, 8.0]
[0.000952195, 0.000952195, 0.000950897, 0.000950809, 0.000951735, 0.000953462, 0.0009555
98, 0.000958435, 0.000963399, 0.00096678, 0.000970583, 0.000969781, 0.000971594, 0.00097
2387, 0.000973097, 0.000974733, 0.00097422, 0.00097165, 0.000962064, 0.000942872, 0.0009
11077, 0.000865801, 0.000826349, 0.000783788, 0.000753597, 0.000726955, 0.000716662, 0.0
00709861, 0.000706955, 0.000707445, 0.000710584, 0.000715213, 0.000720283, 0.00072533,
0.000731263, 0.000735299, 0.000738478, 0.000741726, 0.000745896, 0.000751421, 0.00076078
8, 0.000772545, 0.000780816, 0.000785241, 0.000784907, 0.000779522, 0.000769029, 0.00075
3078, 0.000732364, 0.000713674, 0.000686495, 0.000650987, 0.000627128, 0.000599517, 0.00
0562417, 0.00054204, 0.000522112, 0.000502032, 0.000480773, 0.000457639, 0.00044199, 0.0
00437854, 0.000434032, 0.000433215, 0.000434069, 0.000435876, 0.000443854, 0.000446076,
0.000438196, 0.000424906, 0.00041502, 0.000404713, 0.000397402, 0.000397045, 0.00040405
2, 0.000417636, 0.00045501, 0.000485403, 0.000520794, 0.000549102, 0.000574152, 0.000601
478, 0.000617846, 0.000633584, 0.000646528, 0.000654344, 0.000658329, 0.000660382, 0.000
662099, 0.000667363, 0.00133831, 0.00142933, 0.00149982, 0.00153409, 0.0015251, 0.001466
65, 0.00138117, 0.00130588, 0.00126924, 0.00127577, 0.00132404, 0.00139962, 0.00144997,
0.00149315, 0.00152501, 0.00153881, 0.00152038, 0.00147044, 0.00138853, 0.00127579, 0.00
114194, 0.00101528, 0.000926469, 0.000870492, 0.000841875, 0.000827211, 0.000826077, 0.0
00841241, 0.000874883, 0.000915342, 0.000968094, 0.00102021, 0.00105777, 0.0010841, 0.00
111115, 0.00114567, 0.00117758, 0.00119518, 0.00118254, 0.00113183, 0.00104808, 0.000951
903, 0.000858968, 0.000792478, 0.000761782, 0.000767772, 0.00079247, 0.0008172, 0.000832
068, 0.000902226, 0.00097536, 0.00101781, 0.00101416, 0.000973784, 0.000920828, 0.000866
269, 0.00083309, 0.0008985, 0.000890239, 0.000915446, 0.000990231, 0.00110334, 0.0012205
2, 0.00135521, 0.00148785, 0.0015524, 0.00159347, 0.00159609, 0.00154314, 0.00145838, 0.
00125745, 0.00118727, 0.00111566, 0.00104421, 0.000977345, 0.000918572, 0.000871319]
```

We got it; let's plot it!

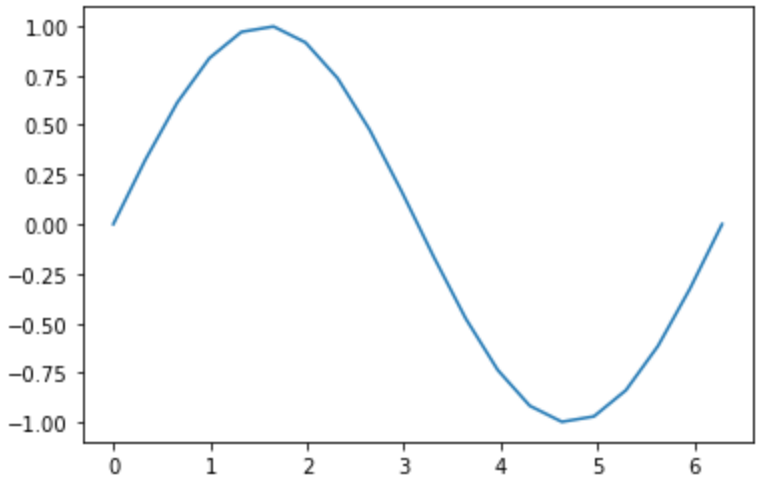
In [6]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(mass, charge, 'r-' )
plt.xlim(0, 8)
```

```
plt.ylim(0, 2e-3)
plt.xlabel('mass (GeV)')
plt.ylabel('charge, 90% C.L. limit')
plt.show()
```



Practice 4: Use the syntax you have just learned to read back each line of x and y values from the sine.txt file that you wrote in Practice 1, and split each line into `float` values and store them. Then, plot your stored x and y values to make sure you have done everything correctly

In [7]:
```
# Code for Practice 4

s = open('sine.txt', 'r')

x = []
sinx = []

for line in s:
    tokens = line.split()
    x.append(float(tokens[0]))
    sinx.append(float(tokens[1]))

plt.plot(x, sinx)
plt.show()
```



Of course, you already know of another way to read in values like this: `numpy.loadtxt()` and `numpy.genfromtxt()`. If you have already been using those, feel free to move on. Otherwise, take a moment to make yourself aware of these functions as they will massively simplify your life.

Fortunately, Python's `numpy` library has functions for converting file information into numpy arrays, which

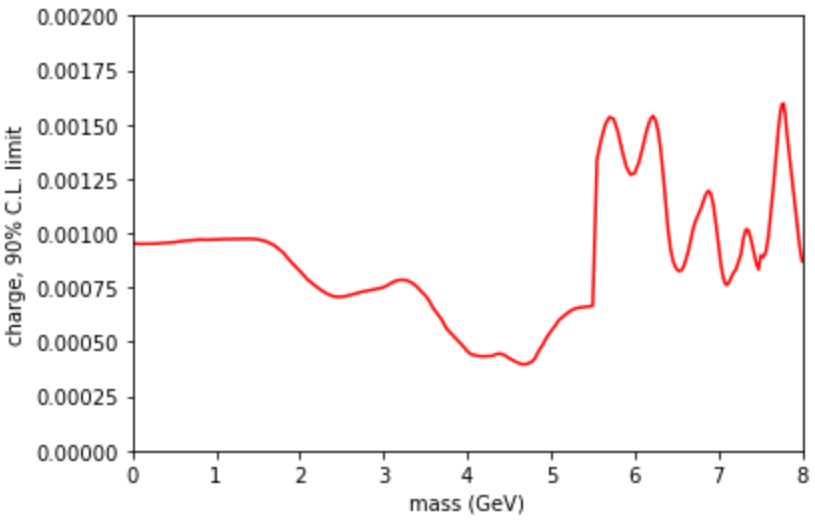
can be easily analyzed and plotted. The above can be accomplished with a lot less code (and a lot less head scratching!)

The `genfromtxt` function takes as it's argument the name of the file you want to load, and any optional arguments you want to add to help with the loading and formatting process. Some of the most useful optional arguments are:

dtype: data type of the resulting array
comments: the character that indicates the start of a comment (e.g. '#')
lines following these characters will be ignored, and not read into the array
delimiter: the character used to separate values. Often, it's whitespace,
but it could also be ',', '|', or others
skip_header: how many lines to skip at the beginning of the file
skip_footer: how many lines to skip at the end of the file
use_cols: which columns to load and which to ignore
unpack: If True (the default is False), the array is transposed (i.e., you can a set of columns, not a set of rows.) You can accomplish the same thing with `genfromtxt( file, opt_args,...).T`

Reload the spectral data and reproduce the plot above using `loadtxt` or `genfromtxt`.
Hint: You may find it helpful to use `numpy.split( array, N )`, which splits `array` into `N` equal-length parts, and returns them as a list.

In [8]:
```
# Same plot as before but now using numpy functions to load the data

import numpy as np
mass, charge = np.loadtxt('BaBar_2016.dat', unpack = True)
plt.plot(mass, charge,'r-')
plt.xlim(0, 8)
plt.ylim(0, 2e-3)
plt.xlabel('mass (GeV)')
plt.ylabel('charge, 90% C.L. limit')
plt.show()
```



Practice: Writing and Reading CSV files

CSV stands for Comma Separated Values. Python's `csv` module allows easy reading and writing of sequences. CSV is especially useful for loading data from spreadsheets and databases.

Let's make a list and write a file!

First, we need to load a new module that you have not used yet in this course: `csv`

In [10]:
```
import csv
```

Next, just as before we need to create an abstract file object that opens the file we want to write to. Then, we create another programming abstraction called a *csv writer*, a special object that is built specifically to write sequences to our csv file. In this example, we have called the abstract file object `f_csv` and we have called the abstract csv writer object `SA_writer`

In [18]:
```
f_csv = open( 'nationData.csv', 'w' )
SA_writer = csv.writer( f_csv,                  # write to this file object
                        delimiter = '|',          # place vertical bar between items
                        quotechar = '',           # Don't place quotes around strings
                        quoting = csv.QUOTE_NONE )# made up of multiple words
```

Make sure that you understand at this point that all we have done is create a writer. It has not written anything to the file yet. So let's try to write the following lists of data:

In [19]:
```
countries = ['Argentina', 'Bolivia', 'Brazil', 'Chile', 'Colombia', 'Ecuador', 'Guyana',
             'Paraguay', 'Peru', 'Suriname', 'Uruguay', 'Venezuela']
capitals = ['Buenos Aires', 'Sucre', 'Brasília', 'Santiago', 'Bogotá', 'Quito', 'Georget
             'Asunción', 'Lima', 'Paramaribo', 'Montevideo', 'Caracas']
population_mils = [ 42.8, 10.1, 203.4, 16.9, 46.4, 15.0, 0.7, 6.5, 29.2, 0.5,\
                    3.3, 27.6]
```

Now let's figure out how to add a line to our CSV file. For a regular ASCII file, we added lines by calling the `write` function. For a CSV file, we use a function called `writerow` which is attributed to our abstract csv writer `SA_writer`:

In [20]:
```
SA_writer.writerow(['Data on South American Nations'])
SA_writer.writerow(['Country', 'Capital', 'Populaton (millions)'])
for i in range(len(countries)):
    print( [countries[i], capitals[i], population_mils[i]] )
    SA_writer.writerow( [countries[i], capitals[i], population_mils[i]] )
f_csv.close()
```

```
['Argentina', 'Buenos Aires', 42.8]
['Bolivia', 'Sucre', 10.1]
['Brazil', 'Brasília', 203.4]
['Chile', 'Santiago', 16.9]
['Colombia', 'Bogotá', 46.4]
['Ecuador', 'Quito', 15.0]
['Guyana', 'Georgetown', 0.7]
['Paraguay', 'Asunción', 6.5]
['Peru', 'Lima', 29.2]
['Suriname', 'Paramaribo', 0.5]
['Uruguay', 'Montevideo', 3.3]
['Venezuela', 'Caracas', 27.6]
```

Now let's see if we can open your file using a SpreadSheet program. If you don't have access to one, find someone who does!

- Download nationData.csv
- Open Microsoft Excel (or equivalent), and select "Import Data."
- Locate nationData.csv in the list of files that pops up.
- Select the "Delimited" Option in the next dialog box, and hit "Next"
- Enter the appropriate delimiter in the next pop-up box, and hit finish.

How did we do?

Practice 5: Use syntax learned above to generate a csv file called `sine.csv` with pairs of x and y values separated by a comma. It should end up looking sort of like

```
0.0,0.0
1.57079632679,1.00000000
```

Notice a few things: we don't need to use any formatting of the numbers. It doesn't matter how many decimal places each value has on each line. Python will just use the comma to figure out where one number ends and another begins

In [28]:
```
import numpy as np

s_csv = open('sine.csv', 'w')
SA_writer = csv.writer(s_csv)

x, y = np.loadtxt('sine.txt', unpack=True)

for i in range(len(x)):
    print([x[i], y[i]])
    SA_writer.writerow([x[i], y[i]])

s_csv.close()
```

```
[0.0, 0.0]
[0.33069, 0.3247]
[0.66139, 0.61421]
[0.99208, 0.83717]
[1.32278, 0.9694]
[1.65347, 0.99658]
[1.98416, 0.91577]
[2.31486, 0.73572]
[2.64555, 0.47595]
[2.97625, 0.16459]
[3.30694, -0.16459]
[3.63763, -0.47595]
[3.96833, -0.73572]
[4.29902, -0.91577]
[4.62972, -0.99658]
[4.96041, -0.9694]
[5.2911, -0.83717]
[5.6218, -0.61421]
[5.95249, -0.3247]
[6.28319, -0.0]
```

We can use a similar process to *read* data from a csv file back into Python. Let's read in a list of the most populous cities from `cities.csv` and store them for analysis.

In [29]:
```
cities = []
cityPops = []
metroPops = []
```

In [30]:
```
f_csv = open( 'cities.csv', 'r')
readCity = csv.reader( f_csv, delimiter = ',' )

# The following line is how we skip a line in a csv. It is the equivalent of readline fr
next(readCity) # skip the header row


for row in readCity:
```

```
    print(row)
f_csv.close()
```

```
['Shanghai', 'China', '24.3', '24.8']
['Lagos', 'Nigeria', '21.3', '21']
['Delhi', 'India', '16.8', '21.8']
['Istanbul', 'Turkey', '14.4', '14.4']
['Tokyo', 'Japan', '13.3', '36.9']
['Mumbai', 'India', '12.5', '20.7']
['Sao Paulo', 'Brazil', '11.9', '20.9']
['Beijing', 'China', '21.5', '21.1']
['Shenzhen', 'China', '10.8', '10.6']
['Seoul', 'South Korea', '10.3', '25.6']
['Jakarta', 'Indonesia', '10', '10.1']
['Guangzhou', 'China', '9.9', '23.9']
['Mexico City', 'Mexico', '8.9', '21.2']
['Lima', 'Peru', '8.7', '9.9']
['London', 'United Kingdom', '8.5', '14']
['New York City', 'United States', '8.5', '20.1']
['Bengaluru', 'India', '8.4', '8.7']
['Bangkok', 'Thailand', '8.3', '8.3']
```

Look at the output of the code above. Every `row` that is read in is a `list` of `strings` by default again. So in order to use the numbers *as numbers* we need to convert them using the `float()` operation. Below, we use this to figure out which city has the largest city population:

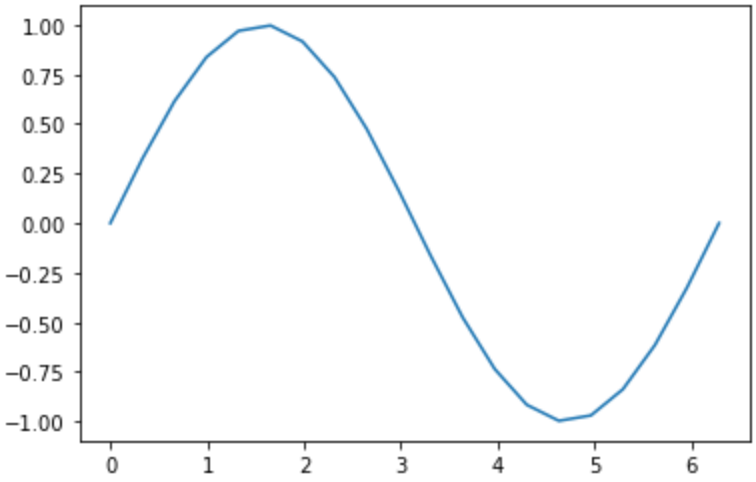
In [31]:
```
f_csv = open( 'cities.csv', 'r')
readCity = csv.reader( f_csv, delimiter = ',' )

largest_city_pop = 0.0
city_w_largest_pop = None

# The following line is how we skip a line in a csv. It is the equivalent of readline fr
next(readCity) # skip the header row


for row in readCity:
    city_country = ', '.join(row[0:2]) # joins the city and country strings using a comm
    cityPop =  float(row[2])
    metroPop = float(row[3])

    # if the population of this city is the largest seen so far, update
    if cityPop > largest_city_pop:
        largest_city_pop = cityPop
        city_w_largest_pop = city_country
f_csv.close()

print("The city with the largest population is: %s with a population of %.1f million peo
```

```
The city with the largest population is: Shanghai, China with a population of 24.3 milli
on people
```

Practice 6: Use the syntax learned above to read in the x and y values from your `sine.csv` file. Plot your data to be sure you did everything correctly.

In [38]:
```
# Code for Practice 6

s_csv = open('sine.csv', 'r')
readSine = csv.reader(s_csv, delimiter=',')

x = []
y = []

for row in readSine:
```

```
    x.append(float(row[0]))
    y.append(float(row[1]))

s_csv.close()

plt.plot(x, y)
plt.show()
```



Exercises

Exercise 1: This exercise is meant to put many of the skills you have practiced thus far together. In this exercise, we are going to use I/O methods to see if we can find some correlations in some fake housing data. You will need the following files which should be in your directory:

```
house_locs_rent.txt
bus_stops.txt
grocery_stores.txt
```

The file `house_locs_rent.txt` is a list of the locations of 500 houses and ther respective rents, and it has 3 columns:

```
x-coordinate y-coordinate rent (in USD)
```

The file `bus_stops.txt` is a list of the locations of bus stops and it has 2 columns:

```
x-coordinate y-coordinate
```

The file `grocery_stores.txt` is a list of the locaitons of grocery stores and it has 2 columns:

```
x-coordinate y-coordinate
```

All 3 files have one-line headers that you will want to ignore when loading the data. The goal of the exercise is to determine how much of the variation in the rent is predicted by variation in the distance between a house and its closest bus stop or by the distance between a house and its closest grocery store.

To determine this, for each of the 500 houses, you will need to first calculate its distance to its nearest bus stop and its distance to its nearest grocery store.

Then, we will use a measure called the Pearson correlation coefficient (which you will use in Homework04) to give an estimate of how much of the variation in the rent is predicted by variation in these distances. The Pearson correlation coefficient is defined as follows:

Correlation Coefficient

Suppose I have N data points each with two variables X_i and Y_i, where $i = 1 \ldots N$. Suppose I want to know how much of the variation in Y is predicted by the variation in X. The correlation coefficient R is a value between -1 and 1 with the following meaning: when $R = 0$, X and Y are independent of each other. When $R > 0$, we say they are positively correlated because if X increases we can expect that Y will increase as well. When $R < 0$ we say they are negatively or oppositely correlated because if X increases we can expect that Y will decrease. R is defined as follows:

$$R = \frac{\mathbb{E}[(X_i - \mu_X)(Y_i - \mu_Y)]}{\sigma_X \sigma_Y} = \frac{1}{N \sigma_X \sigma_Y} \sum_{i=1}^{N} (X_i - \mu_X)(Y_i - \mu_Y)$$

where μ_X is the average of X_i over the dataset, μ_Y is the average of Y_i over the dataset, σ_X is the standard deviation of X_i over the dataset, and σ_Y is the standard deviation of Y_i over the dataset. For calculating those quantities, you may find `np.mean()` and `np.std()` helpful.

However, you must write your *own* correlation coefficient function. It can use `np.mean()` and `np.std()` but it should not call `np.cov` or `np.corrcoef` .

Output: Your code should contain a function to calculate correlation coefficients as well as any other functions that you want to write (for example, a distance function, a minimum distance function...). The output should be the correlation coefficients between the pairs of variables (minimum distances to bus stops, minimum distances to grocery stores, rents) appropriately labeled. Also, create a CSV file titled `distances_rents.csv` and write the values of the minimum distances to the bus_stop and grocery store for each house along with its rent. For example, if the closest bus stop to the first house is 0.5 away and the closest grocery store to the first house is 1.5 away and the rent of the first house is 1250, then the first line of the CSV should read:

```
0.5,1.5,1250
```

Optional: See if you can guess how I generated this fake data. To help sharpen your guess, try transforming the variables before computing the correlation coefficients. If the magnitude of the correlation coefficients goes up, that can be an indicator that you have found the correct form of the function.

In [95]:
```
# Code for Exercise 1 goes here
import numpy as np
import math

housefile = open('house_locs_rent.txt', 'r')
busfile = open('bus_stops.txt', 'r')
groceryfile = open('grocery_stores.txt', 'r')

houseloc = []
houserent = []
```

```
busloc = []

groceryloc = []

housefile.readline()
for line in housefile:
    tokens = line.split('\t')
    houseloc.append([float(tokens[0]), float(tokens[1])])
    houserent.append(float(tokens[2]))

housefile.close()

busfile.readline()
for line in busfile:
    tokens = line.split('\t')
    busloc.append([float(tokens[0]), float(tokens[1])])

busfile.close()

groceryfile.readline()
for line in groceryfile:
    tokens = line.split('\t')
    groceryloc.append([float(tokens[0]), float(tokens[1])])

groceryfile.close()

def distance(p, q):
    return math.dist(p, q)

housebusmin = []

count = 0

housebus = []
housebusmin = []

for house in houseloc:
    for bus in busloc:
        housebus.append(distance(house, bus))
    housebusmin.append(min(housebus))
    housebus = []

housegrocery = []
housegrocerymin = []

for house in houseloc:
    for grocery in groceryloc:
        housegrocery.append(distance(house, grocery))
    housegrocerymin.append(min(housegrocery))
    housegrocery = []


#print(housegrocerymin)


def corrcoef(var1, var2):
    summ = 0
    z = len(var1)
    i = 0
    while i < z:
        inside = (var1[i] - np.mean(var1)) * (var2[i] - np.mean(var2))
        summ += inside
        i += 1
    denominator = z * np.std(var1) * np.std(var2)
    print(summ/denominator)
```

```
corrcoef(houserent, housebusmin)
print(np.corrcoef(houserent, housebusmin))

corrcoef(houserent, housegrocerymin)
print(np.corrcoef(houserent, housegrocerymin))

s_csv = open('distances_rents.csv', 'w')
SA_writer = csv.writer(s_csv)

for i in range(len(houserent)):
    SA_writer.writerow([ housebusmin[i], housegrocerymin[i], houserent[i] ])

s_csv.close()
```

```
-0.23924776103487952
[[ 1.         -0.23924776]
 [-0.23924776  1.        ]]
-0.2255870809795769
[[ 1.         -0.22558708]
 [-0.22558708  1.        ]]
```

OPTIONAL: Practice With HDF5 Files

So far you have encountered a standard ASCII text file and a CSV file. The next file format is called an HDF5 file. HDF5 files are ideally suited for managing large amounts of complex data. Python can read them using the module `h5py.`

In [ ]:
```
import h5py
```

Let's load our first hdf5 file into an abstract file object. We call ours `fh5` in the example below:

In [ ]:
```
fh5 = h5py.File( 'solar.h5py', 'r' )
```

Here is how data is stored in an HDF5 file: hdf5 files are made up of data sets Each data set has a name. The correct Python terminology for this is "key". Let's take a look at what data sets are in `solar.h5py`. You can access the names (keys) of these data sets using the `.keys()` function:

In [ ]:
```
for k in fh5.keys(): # loop through the keys
    print(k)
```

To access one of the 6 data sets above, we need to use its name from above. Here we access the data set called `"names"`:

In [ ]:
```
for nm in fh5["names"]: # make sure to include the quotation marks!
    print(nm)
```

So the dataset called `"names"` contains 8 elements (poor Pluto) which are strings. In this HDF5 file, the other data sets contain `float` values, and can be treated like numpy arrays:

In [ ]:
```
print(fh5["solar_AU"][::2])
print(fh5["surfT_K"][fh5["names"]=='Earth'])
```

Let's make a plot of the solar system that shows each planet's:

- distance from the sun (position on the x-axis)
- orbital period (position on the y-axis
- mass (size of scatter plot marker)

- surface temperature (color of marker)
- density (transparency (or alpha, in matplotlib language))

In [ ]:
```
distAU = fh5["solar_AU"][:]
mass = fh5["mass_earthM"][:]
torb = fh5["TOrbit_yr"][:]
temp = fh5["surfT_K"][:]
rho = fh5["density"][:]
names = fh5["names"][:]
```

In [ ]:
```
def get_size( ms ):
    m = 400.0/(np.max(mass) - np.min(mass))
    return 100.0 + (ms - np.min(mass))*m
def get_alpha( p ):
    m = .9/(np.max(rho)-np.min(rho))
    return .1+(p - np.min(rho))*m
```

In [ ]:
```
alfs = get_alpha(rho)
```

In [ ]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt

norm = mpl.colors.Normalize(vmin=np.min(temp), vmax=np.max(temp))
cmap = plt.cm.cool
m = plt.cm.ScalarMappable(norm=norm, cmap=cmap)

fig, ax = plt.subplots(1)
for i in range(8):
    ax.scatter( distAU[i], torb[i], s = get_size(mass[i]), color = m.to_rgba(temp[i]), a
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_ylim(0.1,300)
ax.set_ylabel( 'orbital period (y)' )
ax.set_xlabel( 'average dist. from sun (AU)' )
ax.set_title( 'Our solar system' )
```

Play around with the data and see what interesting relationships you can find!

If you ever want to write your own HDF5 file, you can open an h5py file object by calling:

```
fh5 = h5py.File('filename.h5py', 'w')
```

Data sets are created with

```
dset = fh5.create_dataset( "dset_name", (shape,))
```

The default data type is float.
The values for the data set are then set with:

```
dset[...] = (  )
```

where the parenthesis contain an array or similar data of the correct shape. After you've added all your data sets, close the file with

```
fh5.close()
```

If you have extra time, try creating your own data set and read it back in to verify that you've done it correctly!

OPTIONAL: Practice With Binary Files

So far, we've been dealing with text files. If you opened these files up with a text editor, you could see what was written in them. Binary files are different. They're written in a form that Python (and other languages) understand how to read, but we can't access them directly. The most common binary file you'll encounter in python is a *.npy* file, which stores numpy arrays. You can create these files using the command `np.save( filename, arr )`. That command will store the array `arr` as a file called filename, which should have the extension .npy. We can then reload the data with the command `np.load(filename)`

In [ ]:
```
x = np.linspace(-1, 1.0, 100)
y = np.sin(10*x)*np.exp(-x) - x
xy = np.hstack((x,y))
```

In [ ]:
```
# save the array
np.save('y_of_x.npy', xy )
```

In [ ]:
```
del x, y, xy # erase these variables from Python's memory
```

Now reload the data and check that you can use it just as before.

In [ ]:
```

```

Bonus challenge! Load the file `mysteryArray.npy`, and figure out the best way to plot it.
Hint: look at the shape of the file, and the dimensions.

In [ ]:
```
data = np.load('mysteryArray.npy')
```

In [ ]:
```

```