CptS 111 — Lab #12

Practice with Plotting (and a Little More on Dictionaries)

Learning Objectives:

- Practice using dictionary creation and methods
- Use `matplotlib.pyplot` to create plots
- Use the `numpy` module

Prerequisites:

- Familiarity with dictionaries
- Exposure to plotting using `matplotlib.pyplot`
- Exposure to `numpy`

In our last lab for the semester, we'll wrap up Ch. 9 with a little work with dictionaries. Then we'll practice some plotting. I had hoped to have you create a publication-quality figure, but it turns out to be beyond the scope of what this lab should include. However, for those of you who want to learn how to create different kinds of plots, how to use colormaps, or how to have more control over the way your figure looks in general, please check out the tutorials available at https://matplotlib.org/tutorials/index.html. I'm sure there are other articles and tutorials available, but these seem like good starting points. Please let me know if you find some good ones to share with future students!

Task 1: Working with dictionaries

You used a dictionary to create a database of your favorite movies or video games in PA #5, so you have some idea of how useful dictionaries can be. In the first lab task, we're just going to practice some easy commands to gain more familiarity with dictionaries. For this task and the next one, you'll need to use an IDLE Shell window. As usual, please read the comments in the listing below, and type the commands as given.

```
>>> # There are a variety of ways to create a dictionary.  In this
>>> # task we'll use three of them.  Let's start by creating an empty
>>> # dictionary which we'll then populate with key-value pairs.
>>>
>>> # Dictionary creation: Method 1
>>> fruits = {}
>>> fruits['apples'] = 1.29
>>> fruits['cherries'] = 2.89
>>> fruits['nectarines'] = 1.19
>>> fruits['grapes'] = 1.79
>>> fruits['bananas'] = 0.69
>>> fruits
>>>
>>> # fruits is now a dictionary with five types of fruit as keys
>>> # paired with their values which are their prices per pound.
```

```
>>> # Let's consider a second method for creating a dictionary which
>>> # simply adds key-value pairs directly.
>>>
>>> # Dictionary creation: Method 2
>>> trips = {
        2021: ['San Diego', 'New Orleans', 'Caribbean'],
        2022: ['Moab', 'Alaska', 'Tenerife'],
        2023: ['Washington, DC', 'Greek Isles']
        }
>>> trips
>>>
>>> # The next two methods use the dict() function to create a
>>> # dictionary.  One uses keyword arguments to create key-value
>>> # pairs, and the other uses a list of tuples where the first
>>> # element in the tuple is the key and the second element is its
>>> # value.
>>>
>>> # Dictionary creation: Method 3
>>> cell_nums = dict(John='432-1394', Miya='432-7364', Henry='432-8125')
>>> cell_nums
>>>
>>> # Note that for the keys in cell_nums, we didn't have to use quotes.
>>> # This is because keys are always immutable, so they must be
>>> # strings or tuples.  Next let's consider a few operations we can
>>> # perform with dictionaries.
>>> fruits['apples']
>>>
>>> # USE EACH OF THE THREE METHODS ABOVE TO CREATE YOUR OWN DICTIONARIES
>>> # BEFORE CONTINUING!
>>>
>>> # We can use a key with a dictionary to find its value, but what
>>> # happens if a key isn't in the dictionary?
>>> fruits['kiwis']
>>>
>>> # This isn't very satisfying, but we'll consider a method soon
>>> # that's more satisfying.  Let's move on.  Suppose we want to
>>> # modify a value in our dictionary or add another key-value pair?
>>> fruits['apples'] = 3.29
>>> fruits['clementines'] = 1.99
>>> fruits
>>>
>>> # First we modified the value of apples and then added another
>>> # key-value pair to our fruits dictionary.  Next, let's delete a
>>> # key-value pair from a dictionary.
>>> del fruits['apples']
>>> fruits
>>>
>>> # That was easy.  Next let's use the 'in' command which can be
>>> # used with dictionaries in a manner similar to how we used it
```

```
>>> # with strings and lists.  In fact, we did this in PA #5!
>>> if 'John' in cell_nums:
        print(f''To phone John, call {cell_nums['John']}.'')
>>>
>>> # Note that we test to see whether a key is in a dictionary.  In
>>> # the example above, the key is 'John' and the dictionary is
>>> # cell_nums.  Also note the use of quotes.
>>>
>>> # In our final practice with dictionaries, we'll consider some
>>> # dictionary methods.  If we want to remove all the key-value
>>> # pairs from a dictionary, we use the .clear() method.  Be careful
>>> # with this method because you don't want to inadvertently remove
>>> # the contents of a dictionary that you really wanted or needed!
>>> fruits.clear()
>>> fruits
>>>
>>> # Yikes!  That was too easy.  Now earlier we found the value in
>>> # a dictionary for a given key, but if the key didn't exist, we
>>> # ended up with an error.  The next method is much better because
>>> # it allows the use of a default value if a key doesn't exist.
>>> fruits.get('kiwis', 'No such key')
>>>
>>> # Well, given that we blew away the contents of 'fruits', we knew
>>> # we wouldn't find any key for it, but notice that we didn't get
>>> # an error message.  Instead, the message printed was the default
>>> # value.  This method is a much better approach to finding the
>>> # value for a key.
>>>
>>> # We can use the .update() method to update a dictionary using
>>> # key-value pairs from another dictionary.  If a key exists in
>>> # both dictionaries, the value in the original dictionary will be
>>> # replaced by the second dictionary.  Key-value pairs only in the
>>> # second dictionary will be added to the original dictionary.
>>> # Let's see how this works.
>>> cell_nums2 = {'John': '432-1234', 'Sam': '432-6543'}
>>> cell_nums
>>> cell_nums2
>>> cell_nums.update(cell_nums2)
>>> cell_nums
>>>
>>> # In the example above, we created a new dictionary with an
>>> # updated value for the key John and a new key-value pair.  Then
>>> # we updated the cell_nums dictionary so that it now contains
>>> # the new value for John and an additional key-value pair!
>>>
>>> # Earlier we used the command 'del' to delete a key-value pair,
>>> # but as with the operation used to return a value for a given
>>> # key, we'll get an error message if the key doesn't exist (try
>>> # it!).  Instead, it's better to use the .pop() method which again
```

```
>>> # allows us to print a default message if the key doesn't exist.
>>> cell_nums.pop('Ali', 'No such key')
```

After you've completed this task, show your work to your TA to get credit. Keep your IDLE session open for the next task.

Task 2: Using `numpy` and `matplotlib.pyplot`

In this task, we'll practice a little with the `numpy` and `matplotlib` modules in an IDLE Shell window, and in Task 3, we'll write a complete program using the `matplotlib` module. The `numpy` module allows us to create n-dimensional arrays, also called matrixes, which are lists or nested lists. What's really powerful is that `numpy` allows us to write simple commands that affect an entire array. We're not actually going to explore the power of `numpy`; we're just going to use the command we need to create some data to plot! The `matplotlib` module is used to create plots in Python, and it's name stands for MATLAB Plotting Library because it replicates the plotting capabilities of MATLAB, a popular program used in engineering, which itself stands for MATrix LABoratory. As you might guess from its name, MATLAB works especially well for matrixes (arrays).

`numpy` and `matplotlib` aren't in the standard library packages that come with Python. Instead you need to download them separately. Unfortunately, with the introduction of Python 3.10, this has become more challenging. If the following doesn't work for you, please ask your TA for help.

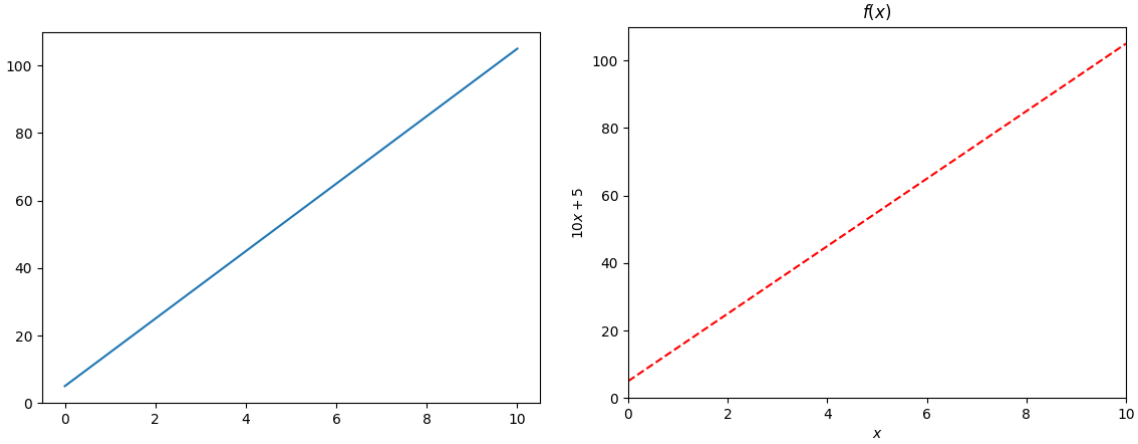
> **Windows:** Search for `cmd` or `PowerShell`. You can also use the Terminal app if you have it. Start your program, and after it has started issue the following command:
>
> - `pip install numpy`
> - `pip install matplotlib`
>
> You may be prompted to reinstall Python first. Download it from **`python.org` and be sure to check the box at the bottom of the window that says "Add Python to PATH"**. After Python has been reinstalled, try the commands again. If none of this works and your TA isn't able to help you, please check out the following `https://www.youtube.com/watch?v=K87M0sMVXZE` which is about 4 minutes long. This is for Python 3.11.x, but it should work for Python 3.10.x as well.
>
> **macOS:** In a Finder window, go to the Applications folder and scroll down to the Utilities folder. In the Utilities folder, launch the Terminal app. At the prompt, issue the following command, one at a time:
>
> - `pip install numpy`
> - `pip install matplotlib`

You should now be all set to complete Task 2.

```
>>> # First, let's import the numpy and matplotlib.pyplot modules.
>>> # Note that the matplotlib module has many modules within it.  We
>>> # don't want to import the entire module because it's very large.
```

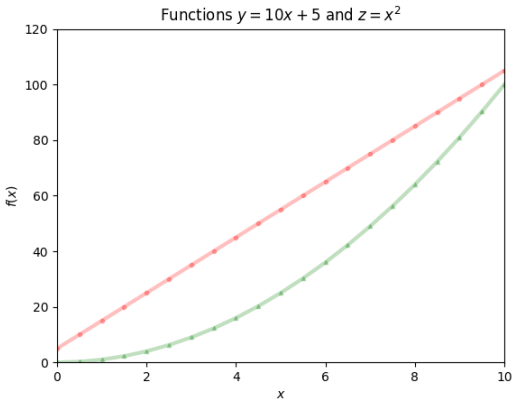
```
>>> # For our purposes, the pyplot module is sufficient.
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> # The aliases assigned to numpy and matplotlib.pyplot are standard
>>> # aliases in the Python world.
>>>
>>> # Next, we'll use numpy to help create some data to plot.
>>> xpoints = np.arange(0, 10.5, 0.5)
>>> xpoints
>>>
>>> # The arange() function in numpy is similar to the range()
>>> # function.  The first argument is the start value, the second is
>>> # the stop value, and the third is the increment value.  As with
>>> # the range function, the sequence of numbers stops before the
>>> # stop value.  However, there are two major differences between
>>> # arange() and range(): range() is restricted to integers, but
>>> # arange() can be used to create a sequence of integers or floats;
>>> # the resulting value is an array.  The reason we wanted to create
>>> # an array of points is because of the ease with which we can use
>>> # them to create other arrays as demonstrated next.
>>> ypoints = 10 * xpoints + 5
>>> ypoints
>>> zpoints = xpoints ** 2
>>> zpoints
>>>
>>> # As we see, we can create an array of points on a line y = mx + b
>>> # simply by multiplying xpoints by 10 and adding 5.  We also
>>> # squared all the values in xpoints to create the array zpoints.
>>>
>>> # Next we'll create a very simple plot using matplotlib.pyplot
>>> # with xpoints and ypoints, each of which is a separate list,
>>> # i.e.,. a 1-D array.
>>> plt.plot(xpoints, ypoints)
>>> plt.show()
>>>
>>> # We always have to use the plt.show() command as our last
>>> # statement for the plot to actually be drawn.  Your figure should
>>> # look like the one on the left below.  Note that the default color
>>> # is blue.
>>>
>>> # Next, let's change the axes, change the format of our line, and
>>> # add labels and a title to our plot.
>>> plt.plot(xpoints, ypoints, 'r--')
>>> plt.axis([0, 10, 0, 110])
>>> plt.title('$f(x)$')
>>> plt.xlabel('$x$')
>>> plt.ylabel('$10x + 5$')
>>> plt.show()
```

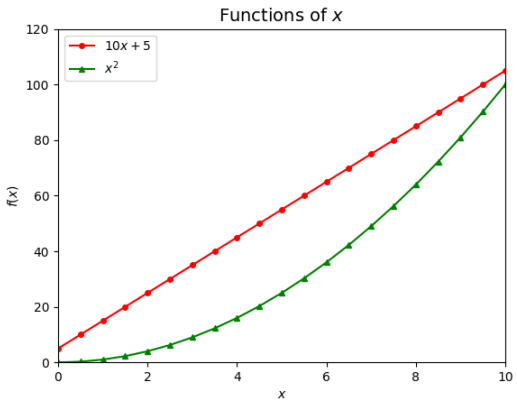
```
>>>
>>> # There are a number of items to point out here:
>>> #    'r--': line color is red (r), line style is dashed (--)
>>> #           (see zyBooks for other colors and line styles)
>>> #    plt.axis([x1, x2, y1, y2]): list used to set x- and y-axis limits
>>> #    plt.title(): use to create title; here the $$'s delimit math
>>> #                 fonts and operations
>>> #    plt.xlabel(): use to create label for x-axis
>>> #    plt.ylabel(): use to create label for y-axis
>>> # Your figure should like the one on the right below.
```




```
>>> # It's very easy to add another line to a plot, and while we're at
>>> # it, let's change the line formatting again, but this time we'll
>>> # use keyword arguments together with a format string.
>>> plt.plot(xpoints, ypoints, 'ro-', linewidth=3, markersize=3, alpha=0.25)
>>> plt.plot(xpoints, zpoints, 'g^-', linewidth=3, markersize=3, alpha=0.25)
>>> plt.axis([0, 10, 0, 120])
>>> plt.title('Functions $y = 10x + 5$ and $z = x^2$')
>>> plt.xlabel('$x$')
>>> plt.ylabel('$f(x)$')
>>> plt.show()
>>>
>>> # So to add another line to the plot, we just add one more
>>> # plt.plot() line but with new values for the second list of
>>> # points.  In addition:
>>> #    'ro-': line color is red and circles (o) mark values of y
>>> #    'g^-': line color is green (g) and triangles (^) mark values of z
>>> #    linewidth: specifies width of line
>>> #    markersize: specifies size of markers (for marks o and ^)
>>> #    alpha: gives degree of line transparency (1 is the default)
>>> # Your figure should look like the one on the left below.
>>> #
>>> # I don't like the look of the plot we just made.  The line width
>>> # and transparency don't look good to me, the markers are too
>>> # small, and if we didn't know which line is straight and which
```

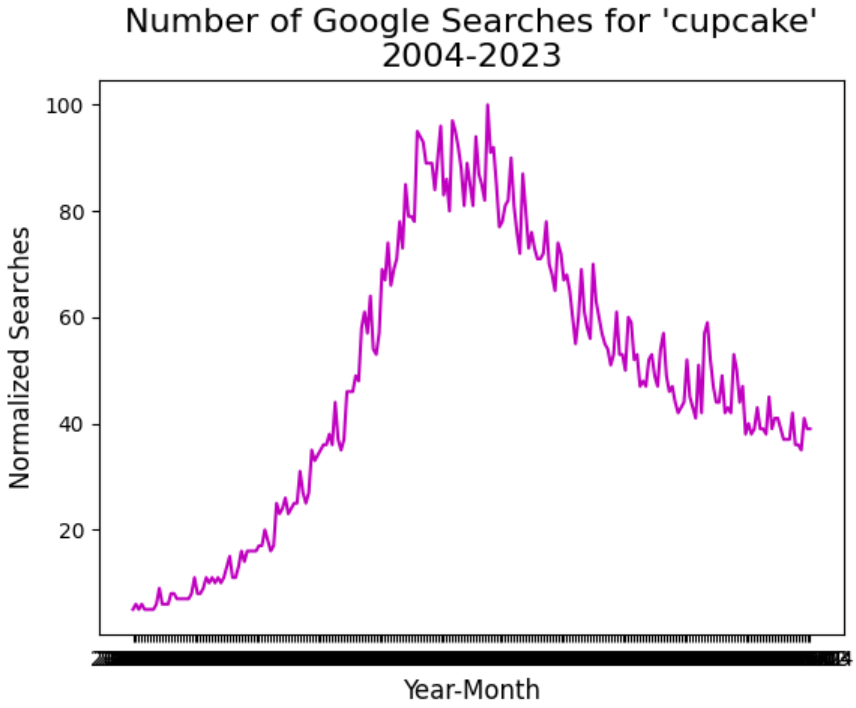
```
>>> # line shows the square of x, there's nothing in the plot to tell
>>> # us.  Let's make a few modifications.
>>> plt.plot(xpoints, ypoints, 'ro-', label='$10x + 5$', markersize=4)
>>> plt.plot(xpoints, zpoints, 'g^-', label='$x^2$', markersize=4)
>>> plt.axis([0, 10, 0, 120])
>>> plt.title('Functions of $x$', fontsize=14)
>>> plt.xlabel('$x$')
>>> plt.ylabel('$f(x)$')
>>> plt.legend(loc='upper left')
>>> plt.show()
>>>
>>> # As we see from the code above, we used the default value for
>>> # linewidth, increased the size of the markers for the marks, and
>>> # added a legend.  Also,
>>> #    fontsize: a fontsize of 14 increased the title size a little
>>> #    label: creates a label for our legend
>>> #    loc='upper left': legend placed in upper left; we can also use
>>> #        'lower left', 'upper center', 'center right', ...
>>> # Your figure should look like the one on the right below.  Also,
>>> # while we didn't do it for this plot, we can change the font sizes
>>> # for xlabel and ylabel using the fontsize keyword.  The markersize
>>> # might be better with a size of 5.
```






```
>>> # In this next exercise, we'll open a .csv file.  Recall that
>>> # values in a .csv file are separated by commas.  Each line in this
>>> # .csv file represents x and y values.  Also, recall that we can
>>> # split a string to create a list.  For this exercise, you need to
>>> # have downloaded the cupcake.csv file.  Be sure it's in the
>>> # correct folder.  The x values are months and years, and the y
>>> # values are the number (normalized) of Google searches for the
>>> # term 'cupcake.'
>>> file_in = open(input('Enter input file name: '))
Enter file name: cupcake.csv
>>> first_line = file_in.readline()  # Remove the header
>>> months = []                      # Initialize lists
>>> cc_searches = []
```

```
>>> for line in file_in:
        line_list = line.split(',')
        months.append(line_list[0].strip())
        cc_searches.append(int(line_list[1].strip()))
>>> months
>>> cc_searches
>>>
>>> # The code above creates the x (months) and y (cc_searches) values
>>> # you're going to plot.  Examine the results for the last two
>>> # commands.  What kind of data structures are they?
>>>
>>> # Next, let's plot the data.
>>> plt.plot(months, cc_searches, 'm-')
>>> plt.title(``Number of Google Searches for 'cupcake'\n2004-2022'', fontsize=16)
>>> plt.xlabel('Year-Month', fontsize=12)
>>> plt.ylabel('Normalized Searches', fontsize=12)
>>> plt.show()
>>>
>>> # Make sure you understand how all the code above works.  Notice
>>> # that we can use a newline character in our string to create two
>>> # lines in the title.  The 'm' stands for magenta, and we've
>>> # increased the fontsize of both our title and labels.  Your
>>> # figure should look like the one below.  Look at your figure.
>>> # What does it tell you about the number of Google searches
>>> # worldwide for the term 'cupcake'?  What is the smeared line
>>> # above the x-axis label?  Values start in Jan. 2004, peak in Feb.
>>> # 2014, and end sometime in Apr. 2023.  From this plot, do you
>>> # think COVID-19 affected the number of searches?
```



Use your cursor to explore the cupcake graph. Find the x value (month and year) for the maximum value which is used to normalize all the other values. Check out maximum and minimum values after 2014 and speculate on what may have caused these to have occurred. This is what data analysts do. Note that you can save your plot as a .png file.

After you've completed this task, show your work to your TA to get credit.

Task 3: A look at the `log_growth()` function

The logistic growth function, $ax(1.0 - x)$, was introduced as a model for population growth for species in a limited habitat. This function behaves very differently for different values of the amplitude a. When a is between 0 and 1, the population will die out regardless of the initial population. When a is between 1 and 2, the population will approach a constant value relatively quickly. When a is between 2 and 3, the population will eventually approach a constant value, but it will fluctuate around this value for some time. When a is between 3 and slightly less than 3.57, the population will permanently oscillate between several values. When a is between 3.57 and 4, the population will vary randomly.

We can demonstrate the behavior of the log growth function for different values of a by giving values of x for various values of a. However, plots of this behavior allow a more effective visualization. In this task, you'll plot the log growth function for four different values of a and for a starting value of $x = 0.5$. You'll need to download the four data files from the course website.

Open an IDLE Editor window and save it as `lab12_t3.py` under `CS111` in a folder called `Lab12`. Don't forget a call to **`plt.show()`**, a call to **`main()`**, your header, and function docstrings. Import **`matplotlib.pyplot`** in the first line of your code, and then code the following two functions.

- **`create_lists()`**: A non-void function with one parameter, a file, that returns two lists. You'll need to initialize the two lists, one for the x values (generations) and one for the y values. Use an iterating `for`-loop to add the values to the lists.
- **`main()`**: In `main()`, you'll need to do the following:
  - Open the four data files you downloaded.
  - Call `create_lists()` four times, once for each file, to obtain lists for all four sets of data.
  - Plot the data using subplots. The first two lines needed are:

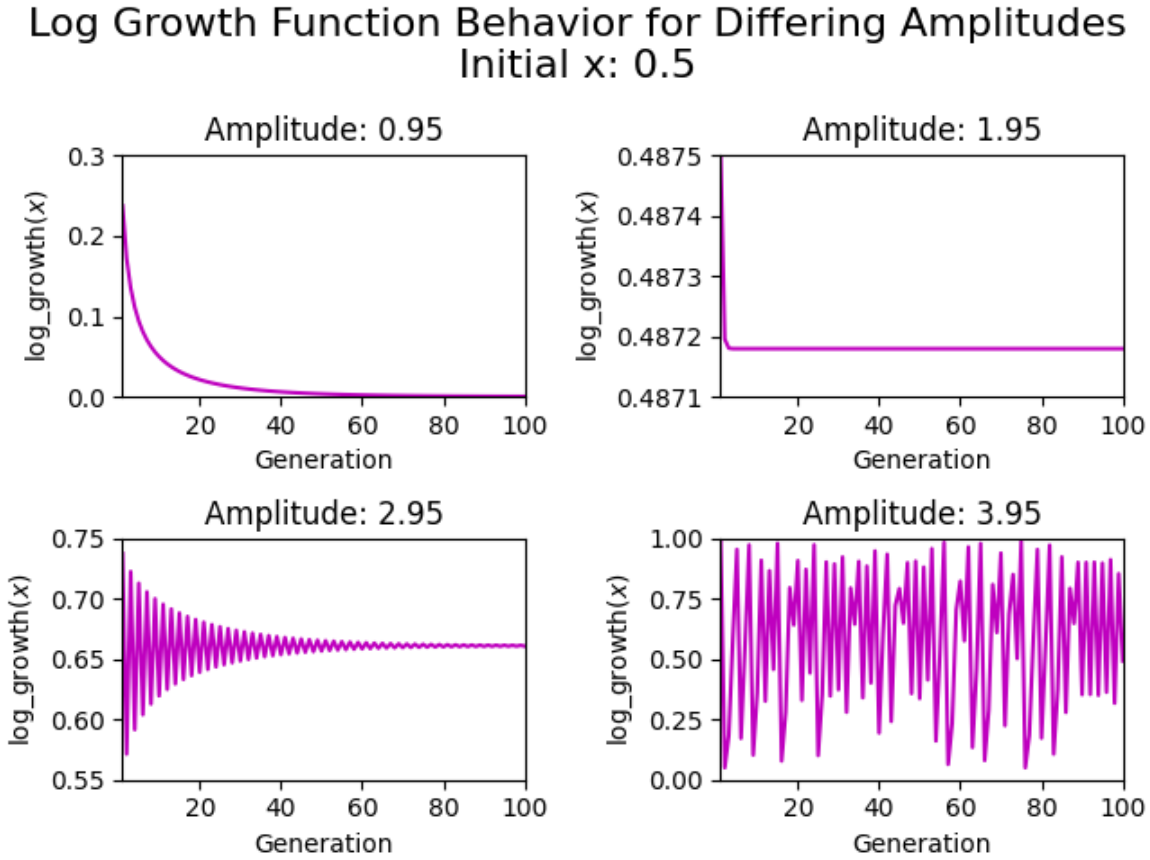
    ```
    fig, subfig = plt.subplots(2, 2, constrained_layout=True)
    fig.suptitle(<title>)
    ```

    Use the title shown in the figure below and a fontsize of 16. The `(2, 2)` in the first line tells `matplotlib` that we want to create a figure with two rows and two columns of subplots. The code for the first subplot is shown below. You'll need to repeat this code three more times, changing the indexes, list names, y limits, and subplot titles appropriately (see figure below).

```
subfig[0,0].plot(x1, y1, 'm-')
subfig[0,0].set_xlim(1, 100)
subfig[0,0].set_ylim(0, 0.3)
subfig[0,0].set_xlabel('Generation')
subfig[0,0].set_ylabel('log_growth($x$)')
subfig[0,0].set_title('Amplitude:  0.95')
```

I've used `x1` and `y1` for my lists; you need to use the names you chose for your lists. Note that the first index in `[m,n]` gives the row `m` and the second index gives the column `n`, i.e., the location of the subplot. Thus, e.g., `[0,0]` is in the upper left and `[1,0]` is in the lower left.

Your figure should look the same as the one below. Note how easy it is to understand the behavior of the log growth function from this figure for different values of the amplitude. As mentioned previously, when the amplitude is between 0 and 1, the population dies out; when it's between 1 and 2, the population quickly converges to a stable value; when it's between 2 and 3, the population oscillates before slowly converging to a stable value; and when it's between 3.57 and 4, populations vary randomly. This random behavior can actually be used in cryptography!



After your program is working correctly, demonstrate it to your TA to get credit.

You're done with labs now! I hope you learned a lot from them and found them challenging but doable and also somewhat fun or interesting! Please be sure to indicate in your evaluations for this class which labs need to be improved and which ones you liked!