# Code Portfolio - Erin Syerson

## Personal Academic integrity statement

I, Erin Syerson, value the opportunity to participate and learn from this institution and thus I commit to submitting my own work and thinking. I will also properly cite sources when I recieve help. I will also have others cite me if I give help. I am aware of the Michigan State University academic honesty policy.

## Imports

When working on some more complex codes, you need to import some modules to work better with your data.

In [ ]:

```
# ensures that the plots made by matplotlib/pyplot show up in the notebo
%matplotlib inline

# imports the pyplot module from matplotlib
import matplotlib.pyplot as plt

# when working with arrays, you need to imput numPy
import numpy as np

# the math module provides functions like sin, cos, tan, etc. This module
import math as mth

# when working with some data, you might need the pandas module
import pandas as pd

# when analyzing relations on graphs
from scipy.optimize import curve_fit

# when analyzing relations on graphs
from scipy.integrate import solve_ivp

# when working with images
from PIL import Image
```

## Create a WHILE loop that will find the index of a desired value

This while loop goes through an array and finds the index value of that item.

values that would have to be changed:

* the values in list_1
* the desired_value

### WHILE loops in general

WHILE loops are used when you don't know exactly how many iterations to run. WHILE loops are good for error checking, finding values, or comparing values

In [ ]:

```
def find_value (list_1, value_desired):     # defining the function and p

    index = 0                               # initializing the index

    while value_desired != list_[index]:    # setting the limit on the wh
        index = index + 1                   # setting a count to the inde

    print('The index associated with', value_desired, 'is', index)     #

list_1 = ["A", "B", "C", "D", "E", "F"]     # initializing the list and t
value_desired = 'C'

find_value(list_1, value_desired)           # calling the function using
```

## Create a FOR loop that prints out cooresponding values from two lists

This function takes two lists, and prints out values from both lists according to the indexes.

values that would have to be changed:

* the values in list_a and list_b

This could also easily be altered to print out values from more than 2 lists.

### FOR loops in general

There are two main ways to make a FOR loop, looping by index and looping by value. The function below utilizes loop by index, where the lists are referenced using their idices. The other way of indicing is looping by value, where the loop uses the value itself instead of in an index. The main difference between the two methods is seen in their applications.

* Looping by index can always be used
* looping by value can ony be used if only one list needs to be referenced.

This also means that the example below can ony be done by looping by index.

In [ ]:
```
def two_lists (list_a, list_b):           # defining the function with th

    for i in range (len(list_a)):         # setting the range to be the l
        print(list_a[i], list_b[i])       # printing out the lists

list_a = ['A', 'B', 'C', 'D', 'E', 'F']   # initializing the lists
list_b = ['1', '2', '3', '4', '5', '5']

two_lists(list_a, list_b)                 # calling the function using th
```

## Create a dictionary

Dictionaries can be used to store large amounts of information in an organized way that is easy to reference. There is an example below of the formatting of a dictionary. dictionaries can store strings, integers, or floats.

Things that would have to be changed:

* The keys in the dictionary
* The information in the dictionary

In [ ]:
```
dictionary = {"key1": 'info1', "key2": 'info2',                    # se
          "key3": 'info3', "key4": 4}

print('Key 1:', dictionary['key1'], 'Key 4:', dictionary['key4'])    # us
```

## General Important Things - Not In A Module

The following is a list of important functions that aren't in a specific module

```
list_name.append()
```

* This can be used to add values to the end of a list

## Working with Images

The following is a list of things to do with 2d images

```
Image.open("filename.jpeg")
```

* Used to load in the image file into a notebook

```
image_array_name[:,:,0], image_array_name[:,:,1], image_array_name[:,:,2]
```

```
    * Used to seperate out the red, green, and blue values from the image ar
    ray
    * image must be in an array form

plt.imshow(red_array_name, cmap = "Reds")

    * Used to plot the red values from the image array
```

## Important Things with SciPy

The following is a list of important functions from the scipy module.

```
result = solve_ivp(function_name, (time_start, time_end), y_initial, t_eval =
time_variable_name)

    * This can be used to help create fit lines for a plot
    * uses differential equations and an initial condition to map out a mode
    l
    * needs a time interval/array to evaluate
    * will give back as many models as you return in your function

parameters, cov = curve_fit(function_name, x, y, p0 = [arg1, arg2, ...])

    * This is used to find the parameters of a model using the function that
    you want it to be based on.
    * The x and y values should be in an array
    * The initial parameters should be in the order expected in the function
    definition
```

In [ ]:

```
# curve_fit formatting
parameters, cov = curve_fit(function_name, x, y, p0 = [arg1, arg2, ...])
param_a = parameters[0]
param_b = parameters[1]
param_c = parameters[2]

y_expected = function_name(x, param_a, param_b, param_c)

plt.scatter(x, y, label = 'actual', color = 'teal')
plt.plot(x, y_expected, label = 'fit', color = 'blue')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

In [ ]:

```
# solve_ivp formatting
result = solve_ivp(function_name, (time_start, time_end), y_initial, t_ev

plt.plot(time_variable_name, result.y[0,:], color = 'blue')
plt.xlabel('time')
plt.ylabel('y')
plt.legend()
```

# Important Things With NumPy

The following is a list of important functions and parts of the numPy module.

```
np.pi

   * Retrieves the value of pi from the module

np.array(list_of_data)

   * Turns the list or value into an array, useful for doing math with a la
   rge data set without having to use loops

np.random.seed(seed = 1)

   * Sets the seed for the random function so that when the random function
   is used it will always use the correct random data set

np.random.normal(mean, std, size)

   * creates a random data set defined by the mean, standard deviation, and
   size of the data set

np.polyfit(x, y, deg)

   * This function uses a data set and a specified degree to find coefficie
   nts of the line of best fit
   * This function can be difficult to format when plotting, so there is an
   example below

np.poly1d(array_of_ceofficients)

   * This function uses an array of coefficients to make a polynomial equat
   ion.
   * This function can be difficult to format when plotting, so there is an
   example below

np.zeros([rows,cols], dtype = '') or np.ones([rows,cols], dtype = '')

   * These functions make arrays of all zeros or ones.
   * '[rows,cols]' defines the shape of the array and
   * 'dtype = ''' defines the data type, 'int' for integers and 'float' for
   floats. The default dtype is float
```

`array_name.shape` `array_name.ndim` `array_name.size` and `array_name.shape[dim]`

* These functions give the shape and dimension of an array
* '.shape' gives the shape of the array
* '.ndim' gives the number of dimensions of the array
* '.size' gives the number of elements in the array
* You can access the individual dimensions of an array using '.shape[dim]', which will give the size of the individual dimensions

- indexing an array

* when referencing an array, there are many different ways to list some of the values
* the formatting of this can be difficult to describe, so there is an example below

`np.sum(array_name, axis = 0)`

* This function takes the sum of the rows of the array, if the axis chosen is 1, then it will take the sum of the columns of the array

`np.linspace(first, end, amount)` or `np.arange(first, end, step)`

* These are two ways to create arrays. 'np.linspace' is used to make a list between two values with a set number of elements. 'np.arrange' is used to make a list between two values with a set value between each element.

```
array_1, array_2 = np.loadtxt("Text_file_name", usecols = [0,1], skiprows = 1, unpack=True, delimiter=',')
```

* This function is used to load in textfiles and trun them into arrays. 'usecols' tells the function which columns from the datafile to use. 'skiprows' tells the function how many rows to skip when loading in the textfile. 'unpack = True' is used when there is more than 2 column being imported. 'delimeter' tells the function how the columns of data is separated in the textfile.

In [ ]:
```
# using .polyfit and .poly1d

fit_coef = np.polyfit(x, y, deg)     # finds coefficients for the data se
fit_line = np.poly1d(fit_coef)       # creates the line using the coeffic

plt.scatter(x, y, color = 'orange', label = 'data')        # plots origi
plt.plot(x, fit_line(x), color = 'green', label = 'fit')   # plots line o

plt.legend()               # sets legend and labels for axes and plot ti
plt.xlabel('')
plt.ylabel('')
plt.axis([x_lower, x_upper, y_lower, y_upper])
plt.title('')
plt.plt.suptitle('')
```

In [ ]:
```
# indexing a 1D array

array_name[start : end : step]  # this is the general form for indexing o

array_name[start : end]         # prints all of the values between the st
array_name[start:]              # prints all of the values after the star
array_name[:end]                # prints all of the values leading up to
array_name[:]                   # a copy of the whole array
array_name[start : end : step]  # every "stepth" item from start to end-1
array_name[::step]              # every "stepth" item over the whole arra
```

In [ ]:
```
# indexing a 2D array

array_name[start1:stop1:step1, start2:stop2:step2]   # this is the genera
```

In [ ]:
```
# Loading in a textfile using numPy

array_1, array_2 = np.loadtxt("Text_file_name", usecols = [0,1], skiprows
```

# Important Things With Pandas

The following is a list of important functions and parts of the pandas module.

```
pd.read_csv('file_name.csv', delimeter = ',', skiprows = 1, usecols = [3:])

    * This function will read text files and turn them into a datafile
    * 'delimeter' describes how the columns in the file are seperated, usual
    ly it is a comma, semicolon, or a space
    * 'skiprows' tells the function how many rows to skip while importing th
    e data
    * 'usecols' tells the function which specific colums to use while import
    ing the data
```

.dropna(axis="columns", how="all") or .drop(unwanted_thing, axis = 1)

* These functions are used to drop unwanted values from a datafile, '.dropna()' drops any rows of data that contain a 'nan', while the '.drop()' can be used to drop specified values
* The 'axis="columns", how="all"' tells the .dropna function how to search for and drop the nan values, while the 'axis = 1' tells the .drop function to drop the column the value is in

.head() .describe() and .tail()

* These functions are used to print off different parts of a dataframe. '.head()' prints out the first few lines of the dataframe, '.describe()' prints out stats on the colums of the dataframe, and '.tail()' prints out the last few lines of the dataframe.

pd.unique(dataframe['column_name'])

* This function finds all of the unique values in a specified column of a dataframe and puts them into an array

indexing : dataframe['column_name'].iloc[rows, cols] or
dataframe['column_name].loc['row_name']

* These are two different ways to reference data in a dataframe. '.iloc[]' is used when referencing the data based on its index number, while '.loc[]' is used to reference the data based on its row name.

.columns() and .index()

* These functions are used to label the column, row, or index names

dataframe['column_name'].plot.hist(title = '', bins = )

* this function plots a histogram of the columm of data, the 'bins = ' sets how many bins in the histogram, the higher the number, the greater the number of sections

plt.scatter(x, y, color = '')

* This function plots a scatterplot relating two arrays/dataframe columns

dataframe.sort_values(by = ['column_name'], ascending = True)

* This function sorts a dataframe based on the values in a specified column from lowest to highest, the 'ascending = true' tells the function to order the values least to greatest, this can be reversed by changing 'True' to 'False'

Creating a dataframe

* The pandas module can either be used to importa dataframe or create a dataframe using arrays.
* The formatting of this can be difficult, so there is an example below.
* Things that would have to be changed in the example:
    * column, array, and dataframe names
* If the array already has 2 columns, then the data frame can be made directly, but this example was for seperate arrays. This can also be applied to more than 2 array at a time.

In [ ]:
```
# setup for plotting a histogram

dataframe['column_name'].plot.hist(title = 'title', bins=50, color = 'pu

plt.xlabel('')
plt.title('')
plt.plt.suptitle('')
```

In [ ]:
```
# creating a dataframe

list_of_tuples = list(zip(array_1, array_2))       # combines the arrays

datframe = pd.DataFrame(list_of_tuples, columns = ['array_1_name', 'array
```

## Important Things With Matplotlib

The following is a list of important functions and parts of matplotlib. When plotting, different markers, linestyles, and colors can be found here (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot) or here (https://matplotlib.org/stable/api/markers_api.html).

.subplot(rows, cols, place)

* Used to plot more than one plot in one figure

.plot(x, y, color = 'red', marker = 'o', linestyle = 'dashed', label = 'Line Label')

* This is used to plot, the points will be connected to each other in the order of the array or list that they are in

.scatter(x, y, color = 'red', marker = 'o', label = 'Line Label')

* This function will make a scatter plot of the data points, meaning that they will not be connected to one another

.xlabel('')

* Used to label the x axis

.ylabel('')

* Used to label the y axis

.axis([x_lower, x_upper, y_lower, y_upper])

* Used to set upper and lower limits to the x and y axes

.title('')

* Used to set a title to the plot

.plt.suptitle('')

* Used to set a title to the figure, useful with using .subplot

.legend()

* Uses the labels set in the plot line to put a legend on the plot

In [ ]:

```
# setting up a scatter plot

plt.scatter(x, y, color = 'teal')

plt.xlabel('')
plt.ylabel('')
plt.axis([x_lower, x_upper, y_lower, y_upper])
plt.title('')
plt.plt.suptitle('')
```

## Masking

Masking values is useful when we want to either get rid of unwanted values or focus only on wanted values. Below is the formatting of a few ways to mask parts of a data set.

values that would have to be changed:

* the name of the dataframe
* the wanted value

since masks are defining a new variable and then immeadiatly usinging it as an index, the mask can be done all in one line.

In [ ]:

```
# various ways to mask values

mask_wanted = dataframe['column_name'] == wanted_value       # sets the v
dataframe_wanted = dataframe[mask_wanted]                    # seperates

mask_unwanted = dataframe['column_name'] != unwanted_value   # sets the v
dataframe_unwanted = dataframe[mask_unwanted]                # seperates

mask_wanted_above = dataframe['column_name'] > wanted_value  # sets the v
dataframe_wanted = dataframe[mask_wanted_above]              # seperates

mask_wanted_below = dataframe['column_name'] < wanted_value  # sets the v
dataframe_wanted = dataframe[mask_wanted_below]              # seperates

mask_wanted = dataframe.loc['row_name'] == wanted_value      # sets the v
dataframe_wanted = dataframe[mask_wanted]                    # seperates

mask_unwanted = dataframe.loc['row_name'] != unwanted_value  # sets the v
dataframe_unwanted = dataframe[mask_unwanted]                # seperates

mask_wanted_above = dataframe.loc['row_name'] > wanted_value # sets the v
dataframe_wanted = dataframe[mask_wanted_above]              # seperates

mask_wanted_below = dataframe.loc['row_name'] < wanted_value # sets the v
dataframe_wanted = dataframe[mask_wanted_below]              # seperates


```