Lab 3: Python for Scientific Computing

Due Friday September 23, 2016 @ 5pm PT on bCourses

Part 1: Python Basics

Exercise 1: Anaconda with Python 2.7 Installation

Deliverables:

- None

Directions:

1. Anaconda is a Python distribution with a number of non-standard libraries often used for scientific computing. In other words, by installing Anaconda, you are installing the Python 2.7 language, a user interface called Spyder, and about 200 additional Python libraries/packages for science, math, engineering, and data analysis. Download Anaconda (http://continuum.io/downloads) for **Python 2.7** and your operating system.

Exercise 2: In-class Examples

Deliverables (2 pts):

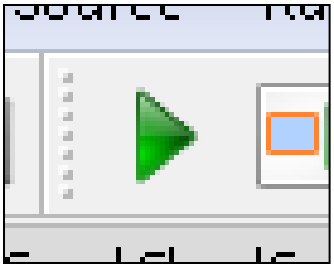
- Submit the Python (.py) files HelloWorld.py, HelloName.py, Math.py, VariableTypes.py, ListDictStr.py, and TryExcept.py, completed according to the following directions. Additionally, in your lab report, copy & paste your Python code into a code box.
- In your lab report, respond to all Questions.

Directions:

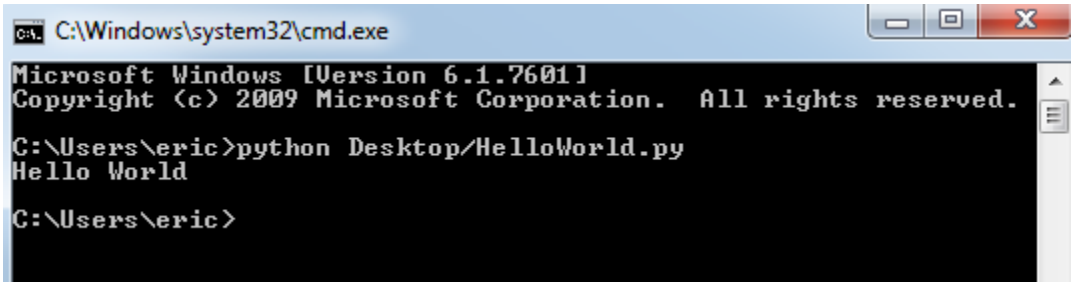
1. Hello World
    a. Open Spyder, the Python Graphical User Interface.
    b. Create a new Python file called HelloWorld.py.
    c. Enter the following code into HelloWorld.py and run the program.

```
def main():
    message = "Hello World"
    print message

main()
```



d. Run the Python program from the terminal or command line:
   i. Open cmd (Windows) or Terminal (Mac/Linux)
   ii. Type "python" followed by the location of the .py file. For example:



2. Hello Name
   a. Create a new Python file called HelloName.py.
   b. Enter the following code into HelloName.py and run the program.

```
def main( name ):
  message = "Hello " + name
  print message

main( "Class" )
```

3. Math
   a. Create a new Python file called Math.py.
   b. Enter the following code into Math.py and run the program.

```
def add( intOne, intTwo ):
  total = intOne + intTwo
  return total

def main():
  summation = add( 5, 4 )
  message = "Sum is "
  message += str( summation )
  print message

main()
```

c. Add the function divide and expand the main function.

```
def divide( intOne, intTwo ):
    total = intOne/intTwo
    return total

def main():
    summation = add( 5, 4 )
    message = "Sum is "
    message += str( summation )
    print message

    quotient = divide( 5, 4 )
    message = "Quotient is "
    message += str( quotient )
    print message
```

d. **Question**:
   i. What is the value of the variable "quotient"?
   ii. What is the data type (int, float, long, etc.) of the variable "quotient"?

4. Variable Types
   a. In many languages (such as Arduino/C), it is necessary to define a variable's type and it is illegal to later change that type. Python, by contrast, tries to determine a variable's data type automatically. In some cases, Python will even change the type for you (e.g. converting an int to a long to avoid overflow).
   b. Open the file VariableTypes_skeleton.py and save it as VariableTypes.py
   c. Edit VariableTypes.py so that it matches the code below and run the program.

```
def main():
    myString = "Hello "
    myInt = 186
    myFloat = 3.14159

    mySum = myInt + myFloat
    print mySum, type(mySum)

    mySum = int( mySum )
    print mySum, type(mySum)

    mySum = myString + str( myInt )
    print mySum, type(mySum)

main()
```

a. **Question**:
    i. In this program, what are the data types of the variable "mySum"?

5. Lists and Dictionaries
    a. Two of the most common data structures in Python are the list (list) and the dictionary (dict). Lists can contain mixed data types (including lists and dictionaries) and items can be retrieved or reassigned using the item's index in the list (starting at 0). Dictionaries are data structures in which items are indexed by a key. A key can be any string or number and is uniquely associated with an item in the dictionary. Please refer to Python.org for more details: docs.python.org/2/tutorial/datastructures.html
    b. Open the file ListDictStr_skeleton.py and save it as ListDictStr.py.
    c. Read through the program and make sure you understand how lists and dictionaries work. At the bottom of main(), complete 1 through 7 so that the desired value is printed. For each, only a single line of code is necessary.

```
# Complete each of the following
# with one line of code.
print "To Complete:"
# 1. Print the last item in myList
print

# 2. Print the number of items in myDictionary
print

# 3. Print the value 186 from the myDictionary
print

# 4. Print the first character myString (i.e "H")
print

# 5. Print the second character of the first string
# in myStringList (i.e "n")
print

# 6. Print the "first temperature reading" from myData
print

# 7. Using range(), print the list [0,3,6]
print
```

6. Control Flow
    a. Open and run WhileIfElse.py and ForLoops.py. Make sure you understand what the programs are doing. You do not need to modify or submit the code. Please refer to Python.org for more details: docs.python.org/2/tutorial/controlflow.html
    b. **Questions**:
        i. What is the Python equivalent to "else if" in Arduino/C?
        ii. For a while loop or if statement, what is the Python equivalent to "{ }" in Arduino/C?

7. Importing Libraries or "Modules"
    a. Open and run TimeModeule.py. Make sure you understand what the program is doing. You do not need to modify or submit the code.

8. Global Variables
    a. A global variable is a variable that it is visible (and therefore accessible) throughout the program. In Arduino, any variable defined at the top of the program is global and can be used within the setup() and loop() functions without being passed. Please refer to Arduino.cc for more details: arduino.cc/en/Reference/Scope).
    The same is not true for Python. In Python, "variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the

function's body, it's assumed to be a local variable unless explicitly declared as global" (Ref: https://docs.python.org/2/faq/programming.html#what-are-the-rules-for-local-and-global-variables-in-python). If that sounds strange, don't worry, we are going to work through an example.

b. Open Global.py. Make sure you understand what the program is supposed to do. If you run the program, you will get the error message "local variable 'count' referenced before assignment".
c. Uncomment the line "#global myCount" (i.e. remove the hashtag, #). Run the program again and this time everything should work.
d. The Explanation: In the program, we have 3 variables defined at the top of the page, myConstant, myCount, and myList. In the main function, we call the functions appendList() and increaseCount() and then print the variables.
   In the appendList() function, we add myConstant to myList using the append() method. This works because the variables myConstant and myList are implicitly global. As long as we do not try to reassign the variables (e.g. myList = [] or myConstant = 1) within the function, we are able to access myConstant and modify myList.
   In the increaseCount() function, we reassign the variable myCount to myCount+ myConstant. Because of the assignment, the variable is assumed to be local within the increaseCount() function. This, of course, is a problem since the variable myCount is not defined anywhere in the function. By declaring myCount as global, the problem is solved. However, as a general rule, you should avoid explicitly global variables and instead pass variables into a function.
e. You do not need to submit the code.

9. Try/Except
   a. The Python language makes it easy to simply try to run some code and to catch any errors or exceptions that occur. This is done using "try" and "except" statements. Please refer to Python.org for more details: docs.python.org/2/tutorial/errors.html
   b. Open the file TryExcept_skeleton.py and save it as TryExcept.py. Run the program and make sure you understand what is happening (e.g. what would happen if "myList[10]" is placed outside the "try" statement?).
   c. Currently, the program does not provide any explanation of what went wrong. To fix this, we will print some of the error information. At the top of the program, import the "sys" library (i.e. add "import sys"). Next, edit the except statement to print the error info.

```
except:
    print "Oops, something went wrong"
    print "Unexpected error:", sys.exc_info()[0]
```

   d. If there is a particular error that we expect to encounter, we can add an except statement for handling those cases. In this program, we will catch all index errors.

```
except IndexError:
    print "Index Error"
    print "Print the last item in list"
    print myList[ -1 ]
except:
    print "Oops, something went wrong"
    print "Unexpected error:", sys.exc_info()[0]
```

Exercise 3: Convert a string to an integer

Deliverables (1.5 pts):

- Submit a file named stringToInt.py containing the program described below. Additionally, in your lab report, copy & paste your Python code into a code box.

Directions:

1. Open the stringToInt_skeleton.py file, save it as stringToInt.py, and complete the strToInt() function. The function should accept an integer string (such as "186" or "2014") and return an integer.
    a. Python already has a function called int( ) that performs this task, but we are going to create a function of our own. In other words, the point is not to use int( ).
    b. Hints:
        i. You may find "for loops" and the len( ) function to be helpful.
        ii. Remember that the string "186" is like the list ["1", "8", "6"]. You can use if statements to determine what character is being stored in each index of the list.
2. Add another function called strToFloat() that takes a float string (such as "3.14159") and converts it to a float.
    a. Again, Python already has a float() function that performs this task, but we are going to create a function of our own. In other words, the point is not to use float( ).
    b. Hints:
        i. Use an "if statement" to locate the "." character.

Part 2: Python for Scientific Computing

Exercise 4: Numerical Python

Deliverables (1.5 pts):

- Submit a file named NumpyBasics.py containing the program described below. Additionally, in your lab report, copy & paste your Python code into a code box.
- In your lab report, respond to all Questions.

Directions:

1. NumPy is an open-source computing package for Python which allows numerical and scientific computing to be naturally integrated into a Python program. While there are many similarities between NumPy and MatLab, there are also many differences (see: wiki.scipy.org/NumPy_for_Matlab_Users). This exercise will serve as an introduction to NumPy and highlight many of the distinguishing characteristics. For a more extension tutorial, visit http://wiki.scipy.org/Tentative_NumPy_Tutorial, and for the NumPy documentation, visit http://docs.scipy.org/doc/.
2. Open the NumpyBasics _skeleton.py file, save it as NumpyBasics.py.
3. We will begin by converting **myList** into a NumPy array.

```
# Create a numpy array
myArray = np.array(myList)
print "Array", myArray
print "Shape", np.shape(myArray)
print "Size", np.size(myArray)
```

   a. Note that the size of the array is "(4L,)", meaning that **myList** is 1-dimensional array with a length of 4.

4. Next, convert myList into a NumPy matrix.

```
# Create a numpy matrix
myMatrix = np.matrix(myList)
print "Matrix", myMatrix
print "Shape", np.shape(myMatrix)
print "Size", np.size(myMatrix)
```

   a. Note that the size of the matrix is "(1L, 4L,)", meaning that **myMatrix** is 2-dimensional 1x4 matrix.
   b. **Very Important Note**: While the behavior of the NumPy matrix type is very similar to that of MatLab matrices, it is recommended that you always use the NumPy array type (Also, in this and future exercises, we will only use the array type). A few key

array/matrix differences include (see wiki.scipy.org/NumPy_for_Matlab_Users for a full description):

i. Arrays may be N-dimensional whereas matrices are always 2-dimensional.
ii. 1-dimensional arrays may be treated as either a row or column vector whereas 1-dimensional matrices are always converted to Nx1 or 1xN 2-dimensional matrices.
iii. For arrays, all operators (*, /, +, **, etc.) are elementwise, but for matrices, the operators vary between elementwise and matrix operators.
iv. Array objects are the default NumPy type and thus the most likely type to be returned by NumPy methods and the many third-party packages that build upon NumPy.

5. We can use a list of arrays (or lists) to create a 2-dimensional array. The matrix transpose can be calculated using the NumPy method, np.transpose(), or the array method, .T.

```
# Create a 2D array
my2DArray = np.array([myArray,myArray,myArray])
print "2D Array"
print my2DArray
print "2D Array Transpose"
print my2DArray.T
```

a. Note that each of the 1D arrays are now rows in the 2D array.

6. Remember that in Python, indexing starts at 0, not 1. We can select a particular row or column as follows

```
# Print column
print "First Column"
print my2DArray[:,0]
# Print row
print "First Row"
print my2DArray[0,:]
```

a. Note that the first row and column are returned as 1D arrays. To return the row and column as a 2D array, you can select all rows/columns from the first and up to but excluding the second.

```
# Print column
print "First Column"
print my2DArray[:,0:1]
# Print row
print "First Row"
print my2DArray[0:1,:]
```

b. Alternatively, you can reshape the 1D arrays. The value -1 tells the NumPy method to infer the unspecified value based on the given array.

```
# Print column
print "First Column"
print np.reshape(my2DArray[:,0],(-1,1))
# Print row
print "First Row"
print np.reshape(my2DArray[0,:],(1,-1))
```

7. **Questions**: Using arrays x, y, A, B, C. and D, as defined below, answer each of the following questions. In your lab report, use the Python `print` output to represent arrays.

```
# Some arrays
x = np.array([0,1,0])
y = np.array([0,0,1])
A = np.arange(0,12,1).reshape(3,4)
B = np.arange(-11,1,1).reshape(3,4)
C = np.arange(1,5,1).reshape(2,2)
D = np.arange(1,25,1).reshape(2,3,4)
```

a. What is $x \times y$, the cross product of **x** and **y**? Hint: np.cross()
b. What is $A^T \cdot B$, the dot product of **A** and **B**? Hint: np.dot()
c. What is $A \circ B$, the elementwise product of **A** and **B**? Hint: * or np.multiply()
d. What is $C \circ I$? Hint: np.eye()
e. What is C^{-1}, multiplicative inverse of **C** (i.e. $CC^{-1} = C^{-1}C = I$)? Hint: np.linalg.inv(). See http://docs.scipy.org/doc/numpy/reference/routines.linalg.html for more on linear algebra operations.
f. What is $x^T \cdot D$? Note: In NumPy, the transpose of a 1D array is itself. Nonetheless, including .T can make your code more readable.
g. What is **D** as a flattened array of shape (24L,)? Hint: ndarray.flatten() where ndarray is the variable D. Alternatively, np.reshape() where the size is input as a tuple, e.g. np.reshape(myArray, (rows,cols))
h. What is the sum of each row in **A**? Hint: ndarray.sum(axis=1)
i. What is the max of each column in **A**? Hint: ndarray.max(axis=0)
j. What is $\sqrt{\frac{1}{12}\sum_{i=1}^{3}\sum_{j=1}^{4}\left(A_{ij}-B_{ij}\right)^2}$, the root mean squared elementwise difference between A and B. Hint: np.sqrt(), np.mean(), np.square(), and – or np.subtract() . Note: this same code can be used to calculate a root mean squared error (RMSE).
k. What is a 3 by 2 array of zeros? Hint: np.zeros() where the size is input as a tuple, e.g. np.zeros((rows,cols))

Exercise 5: Plotting in Python

Deliverables (1 pt):

- Submit a file named MatplotlibBasics.py containing the program described below. Additionally, in your lab report, copy & paste your Python code into a code box.
- In your lab report, respond to all Questions.

Directions:

1. The most popular Python library for 2D and 3D plotting is matplotlib, matplotlib.org. This exercise will serve as an introduction to matplotlib and its basic functionality. Additional examples can be found at matplotlib.org/examples/index.html.
2. Open the MatplotlibBasics _skeleton.py file, save it as MatplotlibBasics.py. At the top of the file, you will see that we are importing the numpy and datetime modules. We are also importing the pyplot module from matplotlib and the DateFormatter, HourLocator, and MinuteLocator from matplotlib.dates. The file also includes two functions for converting between Coordinated Universal Time (UTC) and UNIX time (seconds since 1/1/1970).
3. To begin, we will load some data from a CSV (comma separated values) text file. The file contains time series fridge temperature (˚C), ambient temperature (˚C), and compressor state (on/off) data collected at 1 minute intervals from a 100W mini-fridge. The fridge temperature was measured in the center of the conditioned space and the ambient temperature was measured at 2 inches above the top of the fridge. The file also includes a UNIX timestamp for each data point. We will use the NumPy genfromtxt() method to load the data into an array.

```
# Load some data from a text or csv file.
# Use genfromtxt().
# Set commas as the delimiter and skip the first row of the file.
data = np.genfromtxt("FRIDGE_DATA.CSV",delimiter=",",skiprows=1)
print type(data)
print np.shape(data)
```

For convenience, we will break up the array by columns.

```
# UNIX timestamp (seconds since Jan 1, 1970, GMT)
unix_time = data[:,0]
# Fridge Temperature (degC)
T_fridge = data[:,1]
# Ambient Temperature (degC)
T_ambient = data[:,2]
# Compressor State (On/Off)
m_state = data[:,3]
```

4. Plot the first 100 time steps of the fridge temperature data, just to see what it looks like.

```
# Plot the first 100 minutes
plt.figure()
plt.plot(T_fridge[0:100])

# Show all figures
plt.show()
```

5. To add a legend, label the plot, and call the legend() method.

```
# Plot the first 100 minutes
plt.figure()
plt.plot(T_fridge[0:100],label="Fridge T ($^\circ$C)")
plt.legend()

# Show all figures
plt.show()
```

6. Label the axes with the xlabel() and ylabel() methods.

```
# Plot the first 100 minutes
plt.figure()
plt.plot(T_fridge[0:100],label="Fridge T ($^\circ$C)")
plt.xlabel("Time Steps (Minutes)")
plt.ylabel("Temperature ($^\circ$C)")
plt.legend()

# Show all figures
plt.show()
```

7. Specify the tick positions using xticks() and yticks() and the axis limits using xlim() and ylim().

```
# Plot the first 100 minutes
plt.figure()
plt.plot(T_fridge[0:100],label="Fridge T ($^\circ$C)")
plt.xlabel("Time Steps (Minutes)")
plt.ylabel("Temperature ($^\circ$C)")
plt.xticks(np.arange(0,110,10))
plt.ylim([2,5])
plt.legend()

# Show all figures
plt.show()
```

8. Change the font sizes using the keyworded parameter, fontsize. Also, save the figure as a png.

```
# Plot the first 100 minutes
fs = 18
fig = plt.figure()
```

```
plt.plot(T_fridge[0:100],label="Fridge T ($^\circ$C)")
plt.xlabel("Time Steps (Minutes)", fontsize=fs)
plt.ylabel("Temperature ($^\circ$C)", fontsize=fs)
plt.xticks(np.arange(0,110,10), fontsize=fs)
plt.yticks(fontsize=fs)
plt.ylim([2,5])
plt.legend(fontsize=fs)

# Show all figures
plt.show()
fig.savefig('fridge_temperature.png', format='png')
```

9. Often, when plotting time series data, we would like the x-axis to use timestamps rather than integer time steps. This can be accomplished by adding an hour or minute locator to the x-axis and telling the axis how to display the timestamps.
   We will edit the code to convert the first 100 UNIX timestamps to UTC time and then to Pacific Standard Time (PST). Update plot() and xlim() to include the PST time. Next, we will set the major (labeled) ticks of the x-axis to once an hour and the minor (unlabeled) ticks to every 15 minutes. For each label, we will display only the hour and minute (e.g. 5:30), leaving out the date and seconds. Note that we have also changed figure() to subplots().

```
# Plot the first 100 minutes
utc_datetime = [from_unix_time(x) for x in unix_time[0:100]]
pst_datetime = [t - datetime.timedelta(hours=8) for t in utc_datetime]
fs = 18
fig, axis = plt.subplots()
plt.plot(pst_datetime,T_fridge[0:100],label="Fridge T ($^\circ$C)")
plt.xlabel("Time (PST)", fontsize=fs)
plt.ylabel("Temperature ($^\circ$C)", fontsize=fs)
plt.xticks(fontsize=fs)
plt.yticks(fontsize=fs)
plt.xlim([pst_datetime[0],pst_datetime[-1]])
plt.ylim([2,5])
plt.legend(fontsize=fs)

# Major tick (with label) at once an hour
axis.xaxis.set_major_locator( HourLocator(np.arange(0,25,1)) )
# Minor tick (no label) at every 15 minutes
axis.xaxis.set_minor_locator( MinuteLocator(np.arange(0,60,15)) )
# Format label.
axis.xaxis.set_major_formatter( DateFormatter('%H:%M') )
axis.fmt_xdata = DateFormatter('%m/%d %H:%M:%S')

# Show all figures
plt.show()
fig.savefig('fridge_temperature.png', format='png')
```

10. Next, we will consider the case of a single figure with multiple subplots. Specifically, we will plot the first 100 fridge temperature (˚C), ambient temperature (˚C), and compressor state (on/off) data points in 3 subplots. The complete code is given below. Note that previously we plotted

using plt.plot(), which inferred the set of axes to which to plot. When using subplots, we must specify the set of axes using the **axes_array**. As a result, many of the methods change slightly (e.g. ylabel() becomes set_ylabel()). Lastly, we will call a couple methods, tight_layout() and subplots_adjust(), to reduce the unnecessary white space.

```
# Plot the first 100 minutes
utc_datetime = [from_unix_time(x) for x in unix_time[0:100]]
pst_datetime = [t - datetime.timedelta(hours=8) for t in utc_datetime]
fs = 18
fig, axes_array = plt.subplots(3, sharex=True, figsize=(9, 7))

axes_array[0].plot(pst_datetime,T_fridge[0:100],label="Fridge T")
axes_array[0].set_ylabel("Temp ($^\circ$C)", fontsize=fs)
axes_array[0].tick_params(axis='both', which='major', labelsize=fs)
axes_array[0].set_ylim([1.5,5.5])
axes_array[0].set_yticks(np.arange(2,6,1))
axes_array[0].legend(fontsize=fs)

axes_array[1].plot(pst_datetime,T_ambient[0:100],"r", label="Ambient T")
axes_array[1].set_ylabel("Temp ($^\circ$C)", fontsize=fs)
axes_array[1].tick_params(axis='both', which='major', labelsize=fs)
axes_array[1].set_ylim([24.5,27.5])
axes_array[1].set_yticks(np.arange(25,28,1))
axes_array[1].legend(fontsize=fs)

axes_array[2].plot(pst_datetime,m_state[0:100], "g", label="Compressor")
axes_array[2].set_ylabel("State (On/Off)", fontsize=fs)
axes_array[2].tick_params(axis='both', which='major', labelsize=fs)
axes_array[2].set_ylim([-0.5,1.5])
axes_array[2].set_yticks([0,1])
axes_array[2].legend(fontsize=fs)

axes_array[2].set_xlabel("Time (PST)", fontsize=fs)
axes_array[2].set_xlim([pst_datetime[0],pst_datetime[-1]])

# Major tick (with label) at every 30 minutes
axes_array[2].xaxis.set_major_locator( MinuteLocator(np.arange(0,60,30)) )
# Minor tick (no label) at every 15 minutes
axes_array[2].xaxis.set_minor_locator( MinuteLocator(np.arange(0,60,15)) )
# Format label.
axes_array[2].xaxis.set_major_formatter( DateFormatter('%H:%M') )
axes_array[2].fmt_xdata = DateFormatter('%m/%d %H:%M:%S')

# Adjust layout to reduce white space
plt.tight_layout()
plt.subplots_adjust(bottom=0.1,top=0.98, hspace=0.02)

# Show all figures
plt.show()
fig.savefig('fridge_data.png', format='png')
```

11. Questions: To answer each of the following questions, copy and paste the corresponding line of code into your lab report.

a. What line of code generates 3 subplots?
b. What line sets the y-axis limits of the top subplot?
c. What line sets the y-axis ticks for the middle subplot?
d. What line sets the x-axis label as hours and minutes (e.g. 5:00)?
e. What line sets the horizontal white space between each subplot?

Exercise 6: Linear Regression

Deliverables (2 pts):

- Submit a file named SciKitLearnBasics.py containing the program described below. Additionally, in your lab report, copy & paste your Python code into a code box.
- In your lab report, respond to all Questions.

Directions:

1. The Anaconda Python distribution includes packages for a wide variety of computing tasks, such as **pandas** for data analysis, **statmodels** for statistical modeling, **networkx** for modeling complex networks, and **scikit-learn** for machine learning. In this exercise, we will use the scikit-learn package to estimate the model coefficients for a 100W mini-fridge using ordinary least-squares regression.
   To model the evolution of the temperature in the refrigerator, T, we will use the linear equation

$$\hat{T}^{k+1} = \theta_1 T^k + \theta_2 T_a^k + \theta_3 m^k$$

   Where $\hat{T}^{k+1}$ is the estimated fridge temperature at time step $k+1$, T^k is the fridge temperature at k, T_a^k is the ambient temperature at k, m^k is the compressor state at k, θ_1, θ_2, θ_3 are the model coefficients, and k is an integer valued time step.

   To identify the system, we want to select coefficient values that minimize the difference between the estimated fridge temperature and the measured temperature, $\hat{T}^k - T^k$, at each time step k. One approach for solving this problem is to use quadratic programming to find the coefficient values that minimize the sum of the square differences.

$$\min_{\theta_1 \theta_2 \theta_3} \sum_{n=1}^{N-1} \left(\left(\theta_1 T^k + \theta_2 T_a^k + \theta_3 m^k \right) - T^{k+1} \right)^2$$

Where N is the number of data points. This approach is known as ordinary least squares (OLS) linear regression. You will often see the OLS quadratic problem expressed using L2-norm notation.

$$\min_{\theta} \|X\theta - y\|_2^2$$

Where θ is a vector of coefficients, X is a matrix containing all input data points (e.g. columns are T^k, T_a^k, m^k and rows are k) and y is a vector containing all output data points (e.g T^{k+1}).

In this exercise, we will setup and solve the OLS problem in Python using the scikit-learn package.

2. Open the SciKitLearnBasics _skeleton.py file and save it as SciKitLearnBasics.py. At the top of the file, you will see that we are importing the numpy, datetime, and matplotlib modules. We are also importing the LinearRegression class from the sklearn.linear_model module.
3. Next, we will load simulated (i.e. fake) data for a 100W mini-fridge. We will store this data in an **X** and **y** array where **X** contains T^k, T_a^k, m^k and **y** contains T^{k+1} for $k = 0, \dots, N-1$.

```
# Load simulated fridge data from a csv file.
# Use genfromtxt().
# Set commas as the delimiter and skip the first row of the file.
sim = np.genfromtxt("SIM_FRIDGE_DATA.CSV",delimiter=",",skiprows=1)
rows, cols = np.shape(sim)
N = rows
# Fridge (degC), Ambient (degC), Compressor (On/Off)
X = np.array( sim[0:N-1,1:4] )
# Fridge (degC)
y = np.array( sim[1:N,1] )
```

4. We will create a LinearRegression object called **ols** and fit the linear model using the fit() method. See scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares for more about the LinearRegression class.

```
# Create OLS regression object.
# Set fit_intercept to false (no constant in linear model)
ols = LinearRegression( fit_intercept=False )
# Fit the linear model
ols.fit (X, y)
```

5. To see how well the model fits the data, we can use the **ols** object to generate a prediction for each data point (i.e. **y_est** is $\hat{T}^{k+1}$ for $k = 0, \dots, N-1$). We can also view the linear model coefficients (i.e. **ols.coef_** is $[\theta_1, \theta_2, \theta_3]$).

```
# Estimate model output
y_est = ols.predict( X )
```

```
# Print model coefficients
print "Sim. OLS Coef", ols.coef_
```

6. **Questions**:
    a. For the simulated fridge data, what is the root mean squared error (RMSE) between T^{k+1} and $\hat{T}^{k+1}$?
    b. Using matplotlib, generate a figure that shows T^{k+1} (i.e. **y**) and $\hat{T}^{k+1}$ (i.e. **y_est**) for the first 100 time steps. Include a legend, axis labels, and appropriate formatting. Save the figure as a png and include it in your lab report.
7. We will now repeat the OLS model regression using measured (i.e. real) data for a 100W mini-fridge. Again, we will store this data in an **X** and **y** array where **X** contains T^k, T_a^k, m^k and **y** contains T^{k+1} for $k = 0, \ldots, N-1$.

```
# Load measured fridge from a text or csv file.
# Use genfromtxt().
# Set commas as the delimiter and skip the first row of the file.
data = np.genfromtxt("FRIDGE_DATA.CSV",delimiter=",",skiprows=1)
data = data[0:1000]
rows, cols = np.shape(data)
N = rows
# Fridge (degC), Ambient (degC), Compressor (On/Off)
X = np.array( data[0:N-1,1:4] )
# Fridge (degC)
y = np.array( data[1:N,1] )

# Create OLS regression object.
# Set fit_intercept to false (no constant in linear model)
ols = LinearRegression( fit_intercept=False )
# Fit the linear model
ols.fit (X, y)

# Estimate model output
y_est = ols.predict( X )
# Print model coefficients
print "Meas. OLS Coef", ols.coef_
```

8. **Questions**:
    a. For the measured fridge data, what is the root mean squared error (RMSE) between T^{k+1} and $\hat{T}^{k+1}$?
    b. How do the model coefficients using measured fridge data compare to the model coefficients using simulated fridge data? Which set of coefficients makes more physical sense and why?
        i. Hint: In a physical sense, what do each of the parameters "mean"? How is energy entering and leaving the fridge and how does this impact the temperature?
        ii. Note: Generally, a linear model is a simple and potentially inaccurate way of modeling the dynamics of a system. Nonetheless, by filtering measurements,

scaling model inputs, and tuning a regression model, linear models are often very useful for representing a system.

iii. Another Note: Compared to the linear model above, you may find it easier to perform parameter estimation with the following models.

A. Linear in the states and parameters:

$$\hat{T}^{k+1} = \theta_1 T^k + (1-\theta_1) T_a^k + \theta_2 m^k$$

B. Linear in the states and nonlinear in the parameters:

$$\hat{T}^{k+1} = \theta_1 T^k + (1-\theta_1)\left(T_a^k + \theta_2 m^k\right) + \theta_3$$

```
# Repeat using model:
# T^(k+1) = θ_1 T^k + (1-θ_1) T_a^k + θ_2 m^k

# Load measured fridge from a text or csv file.
# Use genfromtxt().
# Set commas as the delimiter and skip the first row of the file.
data = np.genfromtxt("FRIDGE_DATA.CSV",delimiter=",",skiprows=1)
data = data[0:1000]
rows, cols = np.shape(data)
N = rows
# Fridge (degC) - Ambient (degC), Compressor (On/Off)
X = np.array([[0.0,0.0]]*(N-1))
X[:,0] = np.subtract(data[0:N-1,1],data[0:N-1,2])
X[:,1] = data[0:N-1,3]
# Fridge (degC) - Ambient (degC)
y = np.subtract(data[1:N,1],data[0:N-1,2])

# Create OLS regression object.
# Set fit_intercept to false (no constant in linear model)
ols = LinearRegression( fit_intercept=False )
# Fit the linear model
ols.fit (X, y)

# Estimate model output
y_est = ols.predict( X )
T_est_kplus1 = y_est + data[0:N-1,2]
T_kplus1 = data[1:N,1]
# Print model coefficients
print "Meas. OLS Coef", ols.coef_
```