University of Washington ECE Department

EE 235 Lab 1 – Introduction to Python and Signals Background Material

1.0 Introduction

While you are learning about mathematical tools for signal and system analysis in EE 235, it is useful to reinforce the concepts by working with actual signals and systems on the computer. In addition, understanding computer-based implementation of signal processing systems is important, because digital implementations are increasingly used due to the low cost, high reliability, and easy development. In this course, we will use the Python programming language for computer implementation, because Python is an open source project that is popular for scientific programming. The goal is to familiarize you with a language that will enable you to perform calculations and write applications that demonstrate course concepts and will also be useful in subsequent classes and in future employment. This document will serve to introduce you to the Python language and associated tools for working with signals, as well as to explain how the continuous-time (analog) signal processing concepts are realized with digital computing.

In the EE235 labs, you will use many of the basic concepts of programming that you learned in CSE 142 and 143, but you will need to learn how to express them in the Python language. For those students who have previously learned about scientific programming with Matlab, we will be using Python modules that have many of the same capabilities but again the expression will be a bit different. This document will walk you through the most important Python concepts needed for this course, but you will need to read documentation on your own to become more proficient in Python and to learn details related to different operating systems.

On the signal processing side, we will start with describing the digital representation of a signal, and constraints of that representation. By the end of the course, after you have learned about the frequency domain representation of signals, you will learn the theory behind these constraints.

We will be using an electronic notebook for our labs. Notebooks are useful for interactive development (you can run portions of your code and see the output immediately) and for integrating the code with presentation of the results in a single document. The notebook tool we are using is called Jupyter. When you submit code for your lab, you will submit it as a notebook, complete with documentation, results and discussion, as will be described further in this document.

This document is organized in sections based on the order that material is introduced in the labs. Summary information are provided in a separate document: **235py_ref**.

2.0 Getting Started with Python

Python has several tools and modules that will be useful for EE235 labs. For example, the *scipy* package is a powerful library that supports arrays and functions that are useful in signal processing. Additionally, *numpy* and *matplotlib.pyplot* are modules that we will use heavily in lab, for array manipulation and plotting, respectively. You need to import whatever module you use into your script or notebook with an import statement. We recommend that you install Anaconda, which puts together several useful modules for scientific computing. Most of the modules you need for the EE235 labs will be included in Anaconda, but some additional modules will be needed which we will walk you through. You can use Python's very useful *pip install* tool to install new modules from the command line.

A major shift in the Python world came with the transition from Python 2 to Python 3. All versions of Python 3 are syntactically similar. The EE235 lab assignments were developed using Python 3.6, and should be consistent with later versions of Python. Labs are a great opportunity to get familiar with reading documentation, because any of the programming languages where you'll apply your EE 235 training (e.g. MATLAB, Octave, Scilab, even C) can and will go through updates. The ability to think in terms of general programming principles in using specific software will keep you relevant throughout your career.

Setting up your laptop

You can run python scripts from the command line on your laptop (e.g. "python script.py"), but we will work with it interactively using Jupyter notebook. To make installation easy, we recommend you download Anaconda, which is a software distribution for Python and R. It is available for Linux, Mac OS X, and Windows. It includes most of the necessary Python modules, as well the notebook to write scripts.

- Download the Python 3.6 version of Anaconda here: https://www.anaconda.com/downloads
- Installation instructions may be found here: https://docs.anaconda.com/anaconda/install/

You can use the graphical interface associated with Anaconda Navigator, or you can use conda with command line commands. Guidelines for getting started with the different options are at:

- Navigator: https://docs.anaconda.com/anaconda/navigator/getting-started
- Conda: https://conda.io/docs/user-guide/getting-started.html

While Navigator is easy to use, the conda documentation is a bit more informative. Both starters have links to more complete documentation.

You can use conda (directly or via Navigator) to create separate environments with their own files and packages. If you are starting from scratch, you will only have the "base" environment. If you already have another Python installation on your machine besides Anaconda, you'll need to double check which environment you're in, particularly when installing packages. With Navigator, just go to the "Environments" section to see the list; at the command line type "conda info –envs". You can create new environments as needed. For example, it may be useful to have different environments for different courses. See the relevant getting started documentation for instructions (beware the biopy typo in the Navigator instructions). Changing environments in Navigator simply involves clicking on the one you want.

Installing and Importing Packages

Pip is a tool that helps you download and install packages as necessary. You will need the *simpleaudio* package for EE235. First, make sure you are in the "base" environment. In your terminal or Anaconda prompt, type `pip install simpleaudio`.

Once installation is complete, you may exit your terminal or Anaconda prompt.

Documentation on the *simpleaudio* package can be found at:
https://simpleaudio.readthedocs.io/en/latest/

To use the capabilities of a package in programming with Python, you need to "import" it in your program, specifically:

```
import package_name as pckg
```

The "as pckg" is optional, but allows you to rename packages with long names for easier use in your program. For example, we will use the numpy package a lot, so you may want to use:

```
import numpy as np
```

3.0 Working with Jupyter Notebook

Having downloaded and installed Anaconda, we will write our first programs in Jupyter Notebook. Jupyter is a web-based environment, so you will need to make sure that your browser allows Javascript for full functionality. A useful tutorial is at:

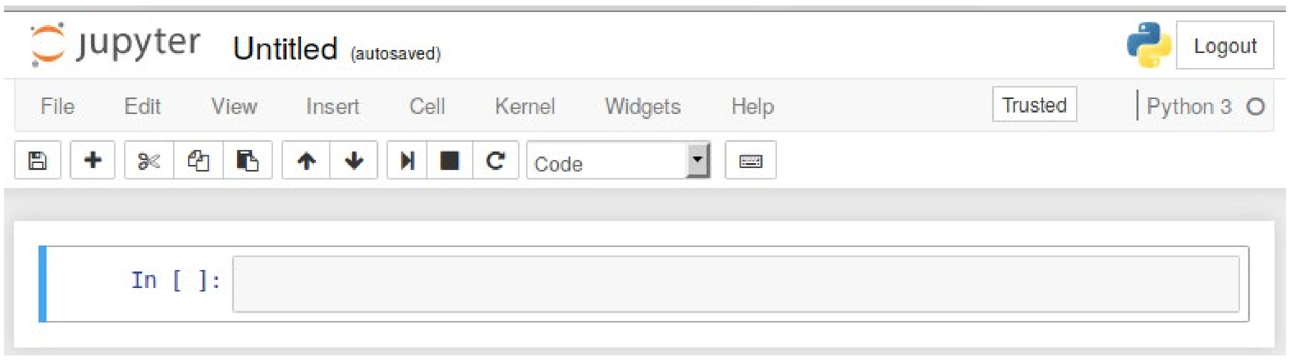
https://www.dataquest.io/blog/jupyter-notebook-tutorial/

To start Jupyter, you can use Anaconda Navigator or a terminal, start Jupyter Notebook. (For the latter, use `$ jupyter notebook`). A new instance of your default browser will open, showing your home directory.

It is useful to have separate folders for different projects and classes, so use the "new" dropdown menu to create a new folder called "ee235lab" – you will work out of this folder for all your lab files.

For the prelab, you need to create a notebook that demonstrates a few simple capabilities.

1. Click to enter ee235lab, and open a new Python3 Notebook. It should look something like this (depending on your browser):



Click on File and rename this notebook "prelab1" such that when you view the contents of the ee235 folder, "prelab1.ipynb" should appear.

2. The section to the right of "In [ ]" is called a **cell**. Cells can be either code (Python code to be executed) or Markdown (text to be displayed in the notebook). By default, new cells are code; the bar where it says "code" has a dropdown menu that allows you to change the cell type. (You can also change the cell type by clicking on "cell.") There can be multiple cells in a file, and cells can be executed individually or together. The first cell is often dedicated to import calls (more on this later). In this first cell, type print('Hello XXX') where XXX is the name of your TA. Run the cell by pushing the "run" button, and observe the output below the cell.

3. To verify that you can work with audio signals, download the file "train32.wav" included with the lab assignment and save it in the ee235lab folder. In order to use the *simpleaudio* package, you'll need to "import" it, as described in section 2. Click on "+" to add a new cell, and type the following into the cell

```
# Test the audio read and play functions
import simpleaudio as sa
wav_obj = sa.WaveObject.from_wave_file('train32.wav')
play_obj = wav_obj.play()
play_obj.wait_done()
```

and run. (Note that the first line is a comment.) You should hear a train whistle. This is just an initial test. Working with audio files is covered in more detail in section 5.

4. In another new cell, type the following

```
# Discussion
A comment of your choosing about the audio file
```

Change the cell type to "Markdown." Now click on the "run" button. You have now created documentation that would be part of a lab report. Notice that in Markdown cells, the "#" sign indicates that this is a heading. If you want a sub-heading or sub-sub-heading, use ## or ###, respectively.

Now click on the “file” button and download everything as a notebook (prelab1.ipynb), and upload the notebook to Canvas for your lab 1 prelab.

For formal lab reports, you will create documentation associated with each assignment in the lab, and you will start the notebook with an introduction paragraph. A sample template is provided for you for labs 1 and 2. After that, you will be expected to create your own following the same format. Jupyter notebook has various capabilities for creating nicely formatted documentation. In addition to the heading markers noted above, Jupyter accepts latex for rendering equations. If you want nicely formatted pdf, then you need to install latex on your laptop, but you can use the latex commands even if you don’t have it installed. Latex uses `$<command>$` for inline math and `$$<command>$$` for an equation that gets its own line, and {} are used for arguments of a command. Some simple latex commands include:

- Subscripts and superscripts, respectively: `$x_i$` and `$x^j$`
- Greek letters: `$\alpha$`, `$\beta$`, `$\gamma$` …
- Squareroot and fractions: `$\sqrt{x} = x^{1/2}$` and `$\frac{1}{x} = x^{-1}$`
- Relation operators: = < > \leq \geq \subset \subseteq
- Sum and integral with limits, respectively: `$\sum_{i=1}^n x_i$`, `$\int_0^{\infty} x(t)dt$`

There are a number of quick reference guides online, e.g.
https://www.sharelatex.com/learn/Mathematical_expressions
https://en.wikibooks.org/wiki/LaTeX/Mathematics

4.0 Variables and Arrays

Python does not use data types in the same way as Java. There is no need to declare variables as different types – the type is determined automatically based on the form of the assignment. For example:

```
length = 100          # integer assignment
value = 1000.0        # float assignment
name = “petunia”      # string assignment
```

The standard data types in Python are number (int, float, complex), string (character sequence), list (comma separated items in []), tuple (essentially a read-only list), dictionary (key-value pairs). Different “types” are allowed to comingle in a list. Variable names are case sensitive.

In signal processing, the most fundamental data structure is an array. In particular, for time signals, we’ll be working with 1-dimensional arrays (vectors), which is essentially a fixed-length list of numbers. One way to create an array is as a list:

```
x = [ 2, 3, 1, 0]
```

Since arrays are so important, we use the *numpy* package that provides tools that make working with arrays easier. Assuming that you have typed "import numpy as np" then you can use *numpy* to create arrays as follows:

```
x = np.array([2,3,1,0])
w = np.array([-1,1,4,2])
```

If you want to know the length of an array (or list), you can use len(x), which would return 4. If you want to access a value at a particular point in the array, you simply specify the element using a zero-based index, e.g. x[0] for the first element of x (value 2) and w[2] for the third element of w (value 4). (Confusing though this is at first, it is useful in working with signals.)

Basic math operations on arrays (+, -, *, /, etc.) are elementwise, and *numpy* gives you more operations. Using "#" to indicate a comment, here are some examples:

```
x+w                 # [1,4,5,2]
x*w                 # [-2,3,4,0]
2*x                 # [4,6,2,0]  (this won't work if you have defined x as a simple list)
w**2                # [1,1,16,4]  (** is the exponentiate operator)
np.abs(w)           # [1,1,4,2]    (absolute value or magnitude if w is complex)
np.concatenate([x,w]) #[2,3,1,0,-1,1,4,2])
```

Note that for elementwise operations combining multiple arrays, you need the arrays to have the same dimension (in this case, the same length).

Some arrays that come in handy and are easy to create with *numpy* are:

```
np.ones(5)          # [1,1,1,1,1]
np.zeros(7)         # [0,0,0,0,0,0,0]
np.arange(5)        # [0,1,2,3,4]
np.arange(-0.2,1,0.2) # [-0.2,0,0.2,0.4,0.6,0.8]
```

The arguments for np.arange above are ([start,] stop [,step]), where the default start is 0 and the default step is 1. Because of the zero-based indexing, the start time is included and the stop time is not.

The start/stop concept works the same way for extracting elements of an array, so using w above, then w[1:3] corresponds to [1,4] and x[1:4] corresponds to [3,1,0].

5.0 Working with Signals on a Digital Computer

In class, we work with real-valued signals *x(t)* where the independent variable (time) is also real-valued, i.e. $x(t) \in \Re$ and $t \in \Re$ or $-\infty < t < \infty$. A signal represented on a digital computer has to have discrete time samples and be finite in length. This means we can only define a signal over a finite range like $-2 \le t \le 10$, and we can only keep time samples at discrete locations. We can only approximate continuous time systems in software.

The simplest approach is to use time samples taken at fixed intervals, which is referred to as the **sampling period**. Taking samples at fixed intervals over a finite time period means that our signals can be represented as arrays. It is important to keep track of the sampling period in order to relate the digital signal to the real world. Often it is more useful to keep track of the **sampling frequency**, which is inversely related to the sampling period. In Hertz (Hz), the sampling frequency is $f_s=1/T_s$ where T_s is the sampling period in seconds. A sampling frequency of 100 Hz means that we are taking 100 samples per second, which corresponds to a sampling period of .01 second. As you might expect, if samples are not spaced closely enough, then the approximation will not be good. There are theoretical results saying how fast we need to sample, but we'll learn that towards the end of the quarter. In the meantime, you need to be careful to use the specified sampling frequency (or sampling period).

5.1 Creating a Digital Version of a Time Signal

In creating digital versions of time signals, the first step is to create an array that is a sequence of time steps. The

numpy **arange** function is useful for defining a time samples vector

```
t = numpy.arange(start, stop, ts)          # ts = sampling period (step size)
```

Recall that the stop value is exclusive, so you need the stop time to be greater than the actual end time you want. In the examples below (as in most examples), we'll use *numpy* imported as np. Try these examples out for yourself.

1. Let's create the signal $x(t) = 2t$ over the range $0 \le t \le 5$ with a sampling period of 1. This sampling period means you will compute x(t) at t = 0, 1, 2, 3, 4, and 5. In this code, we take advantage of the default start and step size for the np.arange call.

```
# Signal as a function of time
# x(t) = 2t over the range 0 ≤ t ≤ 5 with a sampling period of 1.

t = np.arange(6)    # Time samples vector
x = 2 * t           # Signal vector
print('\n t = ', t)
print('\n x = ', x)
```

2. We now increase the time range from -2 to 10, maintaining a sampling period of 1.

```
t2 = np.arange(-2, 11)
print('t2 = ', t2)
```

3. Now let's use the time range from 0 to 2 with a sampling period of 0.5.

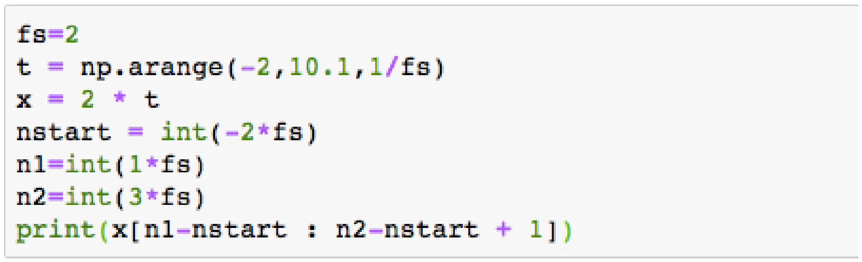
```
t3 = np.arange(0, 2.5, 0.5)
print('t3 = ', t3)
```

5.2 Accessing Value(s) of Signal at Specified Time(s)

The elements in a digital signal correspond to times *t=n*Ts* in the continuous-time signal, for integer n and sampling period *Ts*. In the example above, we have *Ts=0.5* and the signal is defined for *n = 0, …, 4*. If we want to find the digital time *n* that corresponds to a particular time *t*, you would use **n=int(t/Ts)** or **n=int(t*fs)**, where the **int** function rounds to the nearest integer. (Since we can't represent the whole continuous time signal, you can't get a result for arbitrary times.)

The digital time is what you would want for labeling a graph in plotting, but the array index is what you would want for accessing signal values. If the digital signal start time is 0 (as in examples 1 and 3 above), then the digital time n will correspond to the array index. If the signal starts at a time other than 0 (as in example 2), then you need to account for that offset. Let **nt=int(t*fs)** be the desired digital time and **nstart=int(tstart*fs)** be the digital time corresponding to the array signal start time, then the index of the array you want for accessing the signal value is **narray=nt-nstart**.

Let's say you want to print the value of a signal x=2*t for t defined over [1,3], where the digital signal has a time range of [-2,10] with fs=2.



```
fs=2
t = np.arange(-2,10.1,1/fs)
x = 2 * t
nstart = int(-2*fs)
n1=int(1*fs)
n2=int(3*fs)
print(x[n1-nstart : n2-nstart + 1])
```

Note that you need the +1 in the ending time specification because the end point is exclusive. If the original signal time range started at t=0, then nstart=0, and you want x[n1:n2+1].

5.3 Reading, Writing and Playing Wav Files

There are multiple packages available for working with audio files. We will use both *simpleaudio* and *scipy.io*. Whatever package you use, there are two important issues to be aware of.

First, the digital file must always be associated with a sampling frequency fs=1/Ts. This means that when you read an audio file, you will get fs with it. Conversely, you need to specify fs when you save a file, or play the audio. If you play the file with a different fs than it was saved with, you will change the sound of the signal. (We'll try this in the lab.)

Second, audio files may be mono (1 channel) or stereo (2 channels). If the file is mono, then the audio data you read in will come in the form of a vector or 1-dimensional array. If it is stereo, then the data will have a pair of such vectors (one for each speaker), which together form a 2-dimensional array.

It is important to keep track of this information. For example, in creating new sounds, we might want to concatenate different sounds. In order to get the desired result, you need all concatenated components to have the same sample rate and the same number of channels.

You can read and write .wav files using the **wavfile** module from **scipy.io**, and you can read and play .wav files using the **simpleaudio** module. To do all three, we'll use both. With **simpleaudio,** the sampling frequency, channel and other information is bundled together with the data in a special WaveObject. With scipy.io, you have to keep track of the sampling frequency separately.

Let's start with **simpleaudio.** The easy way to read a file, which we gave you earlier, creates an object that has the data, the sampling frequency, number of channels, and bytes per sample as attributes. (The signal is digitized in amplitude as well as time, characterized by bytes per sample.) Assuming that you used import simpleaudio as sa, you can extract this information and play the file as follows:

```
# Test the audio read and play functions
wav_obj = sa.WaveObject.from_wave_file('train32.wav')
fs = wav_obj.sample_rate
channels = wav_obj.num_channels
print('Sampling rate =', fs)
print('Number of channels =', channels)
play_obj = wav_obj.play()
play_obj.wait_done()
```

If you have raw data that you created (or read the file using *scipy.io*), you can first create a WaveObject, or just use

```
play_obj =sa .play_buffer(data, channels, nbytes, fs)
```

where nbytes is 2 or 4, depending on data.dtype ('int16' is 2). In either case, you should follow the play command with:

```
play_obj.wait_done()
```

particularly if you are playing more than one file.

If you read .wav files using the **wavfile** module from **scipy.io,** then the sample rate is returned separately, as shown below for a stereo audio file. (Note that data.shape won't work for mono.)

```
# use scipy.io to read audio files
from scipy.io import wavfile as wav
fs1, data1 = wav.read('train32.wav')
print('Train whistle has: sampling rate', fs1, ', # of samples', len(data1), ', type', data1.dtype)
fs2, data2 = wav.read('tuba11.wav')
len2, ch2 = data2.shape
print('Tuba has: sampling rate', fs2, ', # of samples', len2, ', # of channels', ch2, ', type', data2.dtype)
```

If you have two-channel audio, you can extract the separate channels using data[:,ch] where ch is 0 or 1. Now let's create a new file that plays the two audio channels separately in sequence with a pause in between, and save the result as a new audio file.

```
pause = np.zeros(int(2*fs))                    # create a 2-sec pause
```

```
data0 = data[:,0]                                   # extract channel 0
data1 = data[:,1]                                   # extract channel 1
ptuba_data = np.concatenate([data, pause, data])    # insert pause between tubas
outfile = 'ptuba.wav'
wav.write(outfile,fs,ptuba_data.astype('int16'))    # write wav file
```

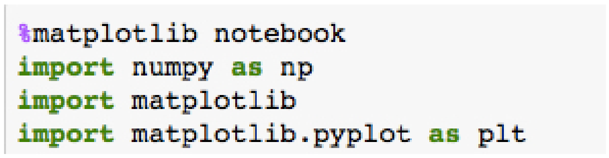
6.0 Plotting

We can use a library called *matplotlib* to visualize data. Within *matplotlib, pyplot* is a module that allows for a MATLAB-like plotting framework. By the end of this section, you should be comfortable with plotting a function, configuring plot settings, and creating subplots. You can find a summary of plotting commands in the **235py_ref** document. Further documentation is at:

Matplotlib Documentation: https://matplotlib.org/contents.html

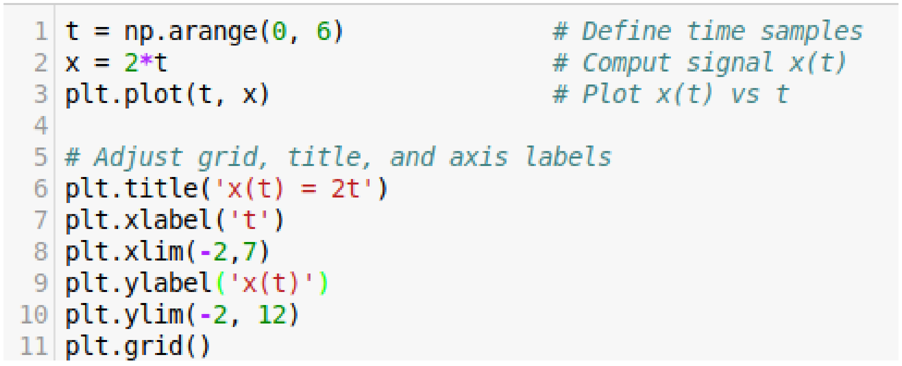
6.1 Basic Plots

To have your plots show up in-line in Jupyter, make sure to add the line `%matplotlib notebook` (or `%matplotlib inline` can be used with Python 2) to your import cell before importing matplotlib (remember, we put all our imports in the first cell of the notebook).



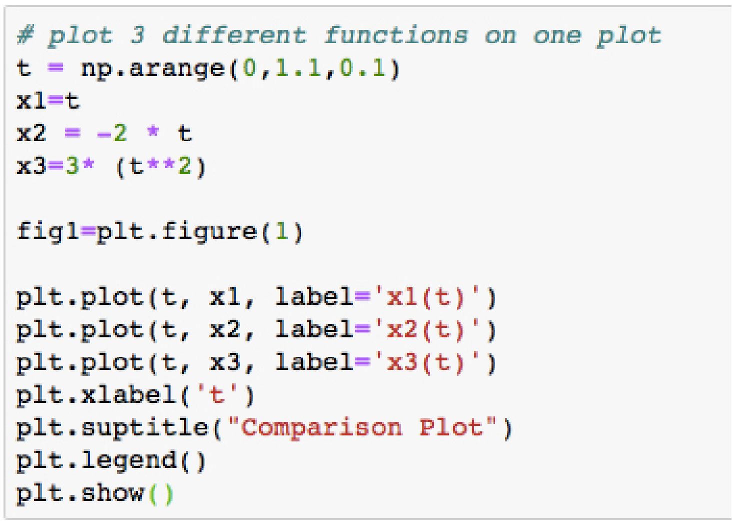
```
%matplotlib notebook
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

The example below illustrates how to plot **x(t) = 2t**, **0 ≤ t ≤ 5**, with a title and axis labels. It also shows how you can control the x and y ranges of the plot; here the x-axis limits are from **-2 to 7**; and y-axis, from **-2 to 12**.



```
t = np.arange(0, 6)                 # Define time samples
x = 2*t                             # Comput signal x(t)
plt.plot(t, x)                      # Plot x(t) vs t

# Adjust grid, title, and axis labels
plt.title('x(t) = 2t')
plt.xlabel('t')
plt.xlim(-2,7)
plt.ylabel('x(t)')
plt.ylim(-2, 12)
plt.grid()
```

Sometimes you want to plot multiple functions on the same graph with a legend comparing them. This is easy: just use the plot command with labels, as in:



```
# plot 3 different functions on one plot
t = np.arange(0,1.1,0.1)
x1=t
x2 = -2 * t
x3=3* (t**2)

fig1=plt.figure(1)

plt.plot(t, x1, label='x1(t)')
plt.plot(t, x2, label='x2(t)')
plt.plot(t, x3, label='x3(t)')
plt.xlabel('t')
plt.suptitle("Comparison Plot")
plt.legend()
plt.show()
```

6.2 Multiple Plots at Once (Subplots)

You'll use the **subplot** function to graph multiple plots together as one figure. From the documentation link above, you can find the usage is **subplot(nrows, ncols, index, **kwargs)**

The first two arguments specify the layout of the plots, with **nrows** and **ncols** corresponding to the number of plots vertically and horizontally, respectively. For example (1,2) will have 2 plots side by side, and (3,1) will have a stack of 3 plots. The **index** indicates which plot is being created, counting from left to right, top to bottom (as one would read). The notation ****kwargs** represents optional qualities you can add to plot, like background or line color. We'll focus on the basics, but you can check the documentation if you want to make fancier plots.

The example below shows how subplots are made and indexed for a 2x2 subplot figure.

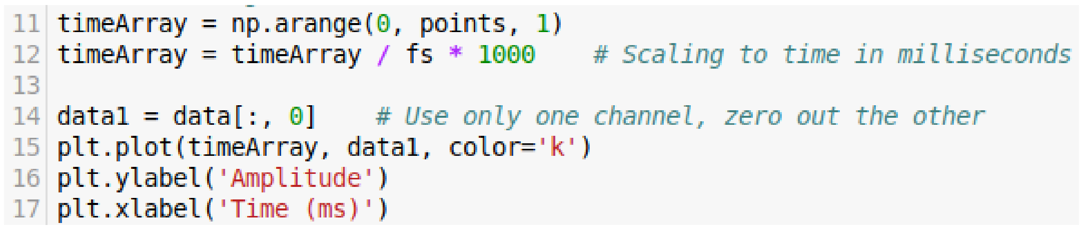
```python
# Plot 4 different signals together in a figure with separate subplots
fs=2
t = np.arange(0,1.1,0.1)
x1=t
x2 = -2 * t
x3=3* (t**2)
x4=-4*(t**2)

# configure subplots
fig2=plt.figure(2)                            # create a new figure
fig2.subplots_adjust(hspace=0.5,wspace=0.2) # set distances between plots

# create each subplot, index counts left to right starting from the top
plt.subplot(2,2,1)     # subplot(nrows,ncolumns,index)
plt.plot(t,x1)
plt.title('x1(t)')
plt.ylim(-4,4)         # control the y limits so x1 and x2 are the same

plt.subplot(2,2,2)
plt.plot(t,x2)
plt.title('x2(t)')
plt.ylim(-4,4)        # control the y limits so x1 and x2 are the same

plt.subplot(223)
plt.plot(t,x3)
plt.title('x3(t)')

plt.subplot(224)
plt.plot(t,x4)
plt.title('x4(t)')
plt.grid()           # this shows how to add a grid
```

Note that each plot's title, axes limits and labels, use of a grid etc. must be modified individually.

6.3 Plotting Sound Files

To plot the sound samples, we have to map each sample to time. Remember that **ts** = 1/**fs**. We make a vector the length of our audio sample vector, and then scale based on the sample period.

Add the code below to your code from **4.1**, and plot the signal:



```python
timeArray = np.arange(0, points, 1)
timeArray = timeArray / fs * 1000    # Scaling to time in milliseconds

data1 = data[:, 0]    # Use only one channel, zero out the other
plt.plot(timeArray, data1, color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
```