BINUS UNIVERSITY
BINUS INTERNATIONAL

Assignment Cover Letter

(Individual Work)

Student Information:	**Surname**	**Given Names**	**Student ID Number**
1.	**Ambadar**	**Fiolisya Faustine**	**2101704615**

Course Code	**: COMP6335**	**Course Name**	**: Introduction to Programming**
Class	**: L1AC**	**Name of Lecturer(s)**	**:** 1. Bagus Kerthyayana 2. Tri Asih Budiono
Major	**: CS**		
Title of Assignment	: **Audio Visualizer**		
Type of Assignment	**: Final Project**		
Submission Pattern			
Due Date	**: 6-11-2017**	**Submission Date**	**: 6-11-2017**

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: (Name of Student)

1. Fiolisya Faustine Ambadar

“Audio Visualizer”

Name: Fiolisya Faustine Ambadar

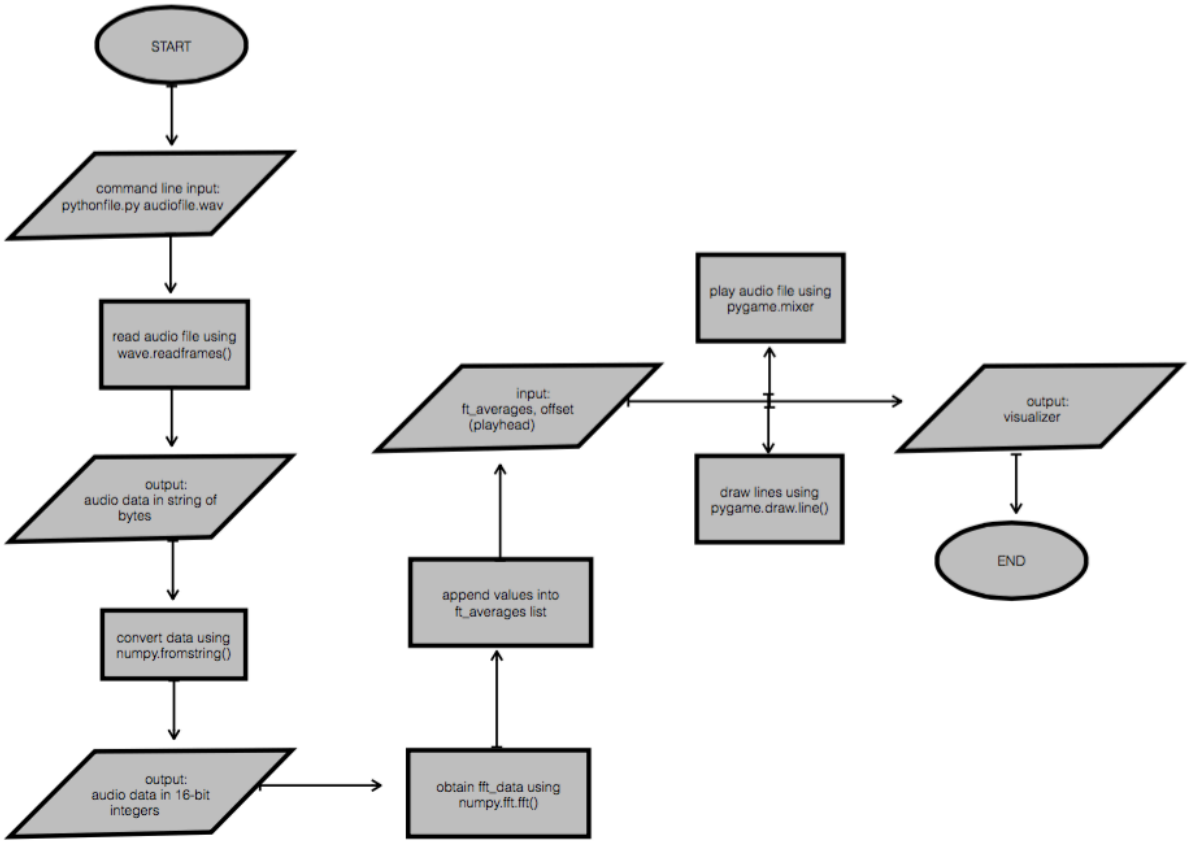
ID: 2101704615

I. Project Specification

This program extracts Fourier transform information from audio files and visualizes the low and high frequencies in the form of patterns. This is similar to visualizations provided in iTunes and Windows Media Player.

II. Solution Design

Flow Chart



III. Explanation of Functions

Class Audio:

1. def __init__(self)
    - Initializes the class attributes:
        - wave_file = opens an audio file through command line
        - audio_size = uses getnframes() from wave module to get the total number of frames

- sample_rate = uses getframerate() from wave module to get the number of frames per second
- audio_data_x = uses readframes() from wave module
- audio_data = uses fromstring() from numpy module to convert bytes from audio_data_x into integers.
- audio_time = uses a calculation of audio_size/sample_rate to get total length of audio
- fourier_rate = set as a constant, higher values result in more reactive patterns.
- fourier_spread = defined to be 1 divided by fourier_rate
- sample_size = fourier_spread multiplied by the sample_rate
- fft_averages = an empty list to store fft data later

2. def play_audio(self)
   - Using pygame.mixer, this function plays the audio file that is opened using sys.argv[].
     As initialized in def __init__(self), the audio file is passed as an argument through a command line, so the code can be used to play different audio files. This is better than instantly having the audio file included in the code.

Class Equalizer(Audio):

1. def __init__(self)
   - Initializes the attributes of inherited class (Audio).
     The equalizer is meant to continue processing the audio data that have been read in class Audio. Data processed in this class are stored in the following class attributes:
   - Initializes attributes for class Equalizer:
     - offset = this refers to the position of the data on the file that is being read (the location of the playhead). It is initialized at 0.
     - freqBands = it is initialized to 12 as it refers to the 12 frequency bands in an octave.
     - length_to_process = This is defined as one less than audio_time
     - total_transforms = this is calculated by multiplying length_to_process with fourier_rate. Because fourier rate refers to the fourier transforms per second, the total transforms would be the

amount of fourier transforms per second multiplied by the total time to process the data.

- sample_range = this is defined to be equal to audio_data

2. def avg_fftBands(self, fft)

   In this method, the fft_averages list declared previously will have some data appended to it. This function first defines the low and high frequencies from the freqBand (range=12), and using the freqIndex method, creates a high and low bound. For I in the range of low and high bound, 'avg' will increase by an index of fft (fft is the parameter which will be passed in the next class). 'avg' is then divided by the range between high and low bound before being appended to the fft_averages list.

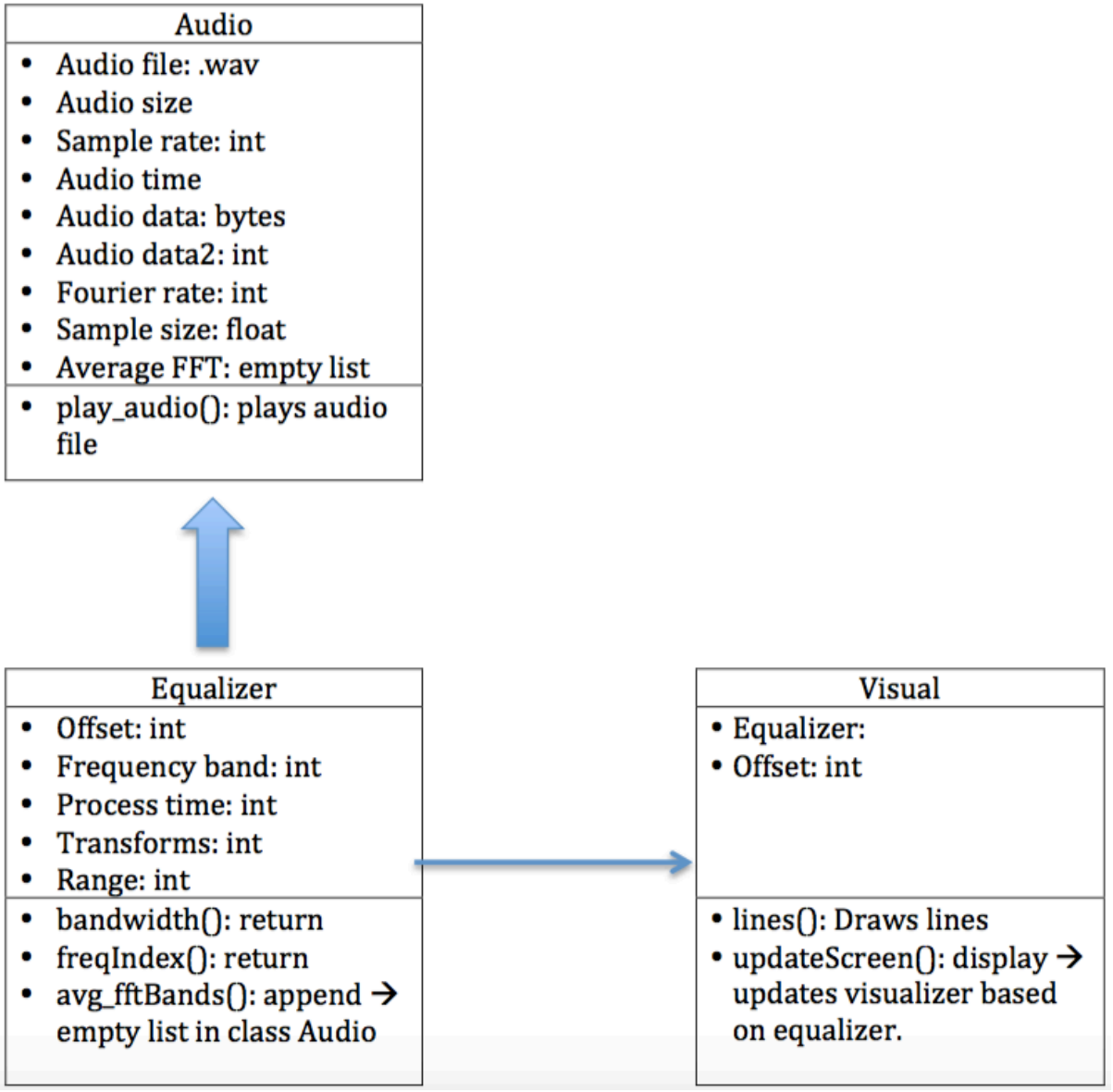
Class Visual():

1. def __init__(self, Equalizer)
   - Sets class Equalizer as a class parameter
   - Initializes class attributes
     - equalizer = Equalizer
     - offset = 0
2. def lines(self, start, end)

   This method first defines the colors 'red', 'green', 'blue' as random.randint(0, 255). This allows the RGB of the lines to be randomized and constantly change on updates. It then uses pygame.draw.line() to allow the rendering of lines once the method is called, using the parameter start and end.
3. def updateScreen(self)

   first it sets global variables 'first', 'second' and 'third' which will later be used to create coordinates for the lines in def lines. It then uses the offset attribute (which as a playhead, continues to move as the audio file is being read), while offset has not reached total transforms, avg_fftBands method from class Equalizer is called with fft_data as a parameter. The variables 'first', 'second' and 'third' are defined to be affected by the fft_averages (this will then influence the visualization to reflect fft). Line coordinates are defined using 'first', 'second', 'third' and 'offset'. The method def lines is called with the coordinates as the parameters. Pygame then displays update.

Class Diagram (UML)



IV. What is Implemented (Built in Modules) + Lessons that Have Been Learned

25th October:

Originally, my project was going to be a game. I have made quite the progress, and I created the game obstacle that consists of two lines going in random directions to reduce the player's lives. Because of the dynamism of these lines, I got inspired instead to create an audio visualizer as shown in iTunes/Windows Media Player. However, to create an aesthetically pleasing visualizer, the movements of the lines are too random (originally it was random.randint() for both x and y coordinate). I modified the coordinates of the starting and ending position for the pygame.draw.line() function, so that there is a constant value for the x position and a y position that follows the y-values of the sin-cos-tan waves.

```
+        x, y = 0, abs(math.sin(currentSate)) * 600
+        j, k = 400, abs(math.sin(currentSate + 15)) * 600
+        m, n = 800, abs(math.sin(currentSate)) * 600
+
+        x1, y1 = 0, abs(math.tan(currentSate)) * 600
+        j1, k1 = 400, abs(math.tan(currentSate + 15)) * 600
+        m1, n1 = 800, abs(math.tan(currentSate)) * 600
+
+        x2, y2 = 0, abs(math.cos(currentSate)) * 600
+        j2, k2 = 400, abs(math.cos(currentSate + 15)) * 600
+        m2, n2 = 800, abs(math.cos(currentSate)) * 600
+
+        line1 = pygame.draw.line(screen, (random.randint(0, 255), random.randint(0,
255), random.randint(0, 255)), [x, y], [j, k], 5)
+        line2 = pygame.draw.line(screen, (random.randint(0, 255), random.randint(0,
255), random.randint(0, 255)), [j, k], [m, n], 5)
```

In the image above, the variable ‘currentSate’ starts at 0 then increases by 0.01 every time. This constant increase results on the sin-cos-tan wave movement.

```
+filename = sys.argv[1]
+p = pyaudio.PyAudio()
+#pygame.mixer.music.play(filename)
+
+if __name__ == '__main__':
+    # Read wave file and get sound info
+    wave_file = wave.open(filename, 'r')
+    audio_size = wave_file.getnframes()
+    sample_rate = wave_file.getframerate() #frames per second
+    print(wave_file.getsampwidth())
+    duration = audio_size/float(sample_rate)
+
+    audio_data = wave_file.read(audio_size)
+
```

Since making an audio visualizer is unfamiliar to me, I had to search how to read audio files and extract data. As I searched, I learned about several modules including pyaudio, sys, struct, wave, numpy and sched.

The module that confused me the most at first was sys. As shown in the image above, sys.argv[1] is shown on line 12. This is a method to open a file using a command line / terminal. sys.argv[0] is the script name (name of python file) and sys.argv[1], for this project would be the audio file. This is a new method for me, as for a previous assignment I simply included the audio file name in the code. Using this method allows users to open different audio files without changing the code.

Throughout the process of creating this project, I continued to experiment with these modules to see which ones work best.

30th October:

After doing some research, it turns out that wave and pyaudio both do not play audio files. So I figured that the best solution would be to use pygame.mixer to play the audio while wave/pyaudio processes the audio data.

```
22 +         audio_data = self.wave_file.readframes(fftLen) #results in bytes
23 +         self.audio_data = numpy.fromstring(audio_data, 'Int16') #converts bytes
   into 16-bit integers
24
25 +     def play_audio(self):
26 +         pygame.mixer.Sound(self.filename).play(-1)
27
28 +     def plot_signal(self):
29 +         plt.plot(self.audio_data)
30 +         plt.pause(1)
31 +         plt.show()
```

By trying both wave and pyaudio, I noticed that both can perform the same tasks. In the image above I am using the wave module. As seen on line 22, I used .readframes() to get audio data in the form of a string of bytes. The same task can be done in pyaudio using .read(). Both then need to be converted from the string of bytes into a numpy array. Most commonly is the format of 16-bit integers, which is what “Int16” means. To do this process I also tried using the function ‘unpack’ from the ‘struct’ module, which has a main purpose of converting values. However I was unable to continue using struct.unpack() because it was unable to accept the format I was passing as a parameter.

31st October:

One of the things I accomplished today was converting the line of codes for reading audio file into a class. The way the lines was drawn on pygame was previously too long horizontally, hence is not in accordance to the python style guideline (Python Crash Course, pg. 72).

```
64 -       line3 = pygame.draw.line(screen, (random.randint(0, 255), random.randint(0,
   255), random.randint(0, 255)), [x1, y1], [j1, k1], 5)
65 -       line4 = pygame.draw.line(screen, (random.randint(0, 255), random.randint(0,
   255), random.randint(0, 255)), [j1, k1], [m1, n1], 5)
66
67 -       line5 = pygame.draw.line(screen, (random.randint(0, 255), random.randint(0,
   255), random.randint(0, 255)), [x2, y2], [j2, k2], 5)
68 -       line6 = pygame.draw.line(screen, (random.randint(0, 255), random.randint(0,
   255), random.randint(0, 255)), [j2, k2], [m2, n2], 5)
```

```
97  +       visualize = Visual()
98
99  +       visualize.lines([x, y], [j, k])
100 +       visualize.lines([j, k], [m, n])
101 +       visualize.lines([x1, y1], [j1, k1])
102 +       visualize.lines([j1, k1], [m1, n1])
103 +       visualize.lines([x2, y2], [j2, k2])
104 +       visualize.lines([j2, k2], [m2, n2])
```

As shown in the image above, the new code on the right is simpler than the one on the left, because it simply calls the class and method to draw the lines and have the coordinates as the parameters.

```
52
53 -       x1, y1 = 0, abs(math.tan(currentSate)) * 600
54 -       j1, k1 = 400, abs(math.tan(currentSate + 15)) * 600
55 -       m1, n1 = 800, abs(math.tan(currentSate)) * 600
56
57 -       x2, y2 = 0, abs(math.cos(currentSate)) * 600
58 -       j2, k2 = 400, abs(math.cos(currentSate + 15)) * 600
59 -       m2, n2 = 800, abs(math.cos(currentSate)) * 600
```

```
77  +
78  +       x1, y1 = first, abs(numpy.tan((freqLine/100) * currentSate)) * 600
79  +       j1, k1 = second, abs(numpy.tan((freqLine/100) * currentSate + 15)) * 600
80  +       m1, n1 = third, abs(numpy.tan((freqLine/100) * currentSate)) * 600
81
82  +       x2, y2 = first, abs(numpy.cos((freqLine/100) * currentSate)) * 600
83  +       j2, k2 = second, abs(numpy.cos((freqLine/100) * currentSate + 15)) * 600
84  +       m2, n2 = third, abs(numpy.cos((freqLine/100) * currentSate)) * 600
85
86
```

Previously, I imported both numpy and math. Later I learned that numpy has similar functions to the math module, so it is possible for me to simply use numpy and importing math is unnecessary.

3rd November:

Most examples I found on how to get frequency information from an audio was rather complicated, so I decided to follow the method from the Python Playground e-book. The example uses pyaudio to open the audio file and numpy to get the data.

```
117 +# # stream = p.open(format = pyaudio.paInt16,#reading data in 16-bit integers
118 +# #                 channels = wave_file.getnchannels(),
119 +# #                 rate = sample_rate,
120 +# #                 frames_per_buffer = fftLen,
121 +# #                 input = True,
122 +# #                 output = True)
123 +#
124 +# audio_data = wave_file.readframes(fftLen) #this comes out in bytes
125 +# audio_data = numpy.fromstring(audio_data, 'Int16') #change data format into 16-
    bit integers
126 +# plt.plot(audio_data)
127 +#
128 +# # data_array = numpy.frombuffer(audio_data, dtype=numpy.int16)
129 +# #
130 +# # fft_values = numpy.fft.rfft(data_array) *2.0/fftLen
131 +# # real_fft_values = numpy.abs(fft_values)
```

As seen in the image above, unlike using the wave module, it is more complicated to open a file using pyaudio, as more parameters are required. Here, pyaudio.PyAudio() is stored in a variable ‘p’, and ‘open’ is a method from the PyAudio() class.

From the Python Playground e-book I learned that numpy has an FFT method, which could instantly get the Fourier transforms from the audio data. This returns complex numbers (in the form a + bi). numpy.abs() is then used to get the magnitude of the complex numbers so the result is real.

```
158 +s.enter(1000/audio_file.sample_rate, 1, updateScreen())
159 +s.run()
160
```

It has been defined s = sched.scheduler(time.time, time.sleep). This module (sched) is used to make the visualization synchronized with the playing audio file. Before including line 158 in the image above, the visualization is not in sync with the audio file. 1000/audio_file.sample_rate tells the program how many times it should update per second, and by using the audio_file.sample_rate, it is able to be in sync with the audio file.

5th November:

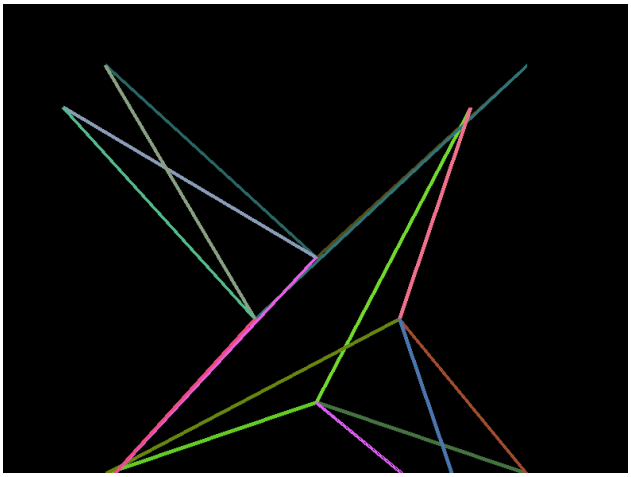
```
93        def update(self):
94  +         global first
95  +         global second
96  +         global third
97  +
98  +         first = 200 - (int(self.fft_averages[11]) * 5)
99  +         second = 400
100 +         third = 600 + (int(self.fft_averages[11]) * 5)
101
```

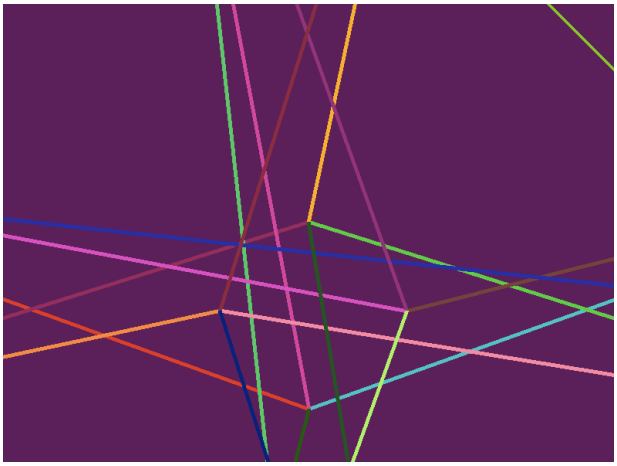
```
129 +         #change the color of the screen based on fft
130 +         screen.fill(( int(y_axis[9]) % 255, int(y_axis[10]) % 255, int(y_axis[9]) %
    255))
```

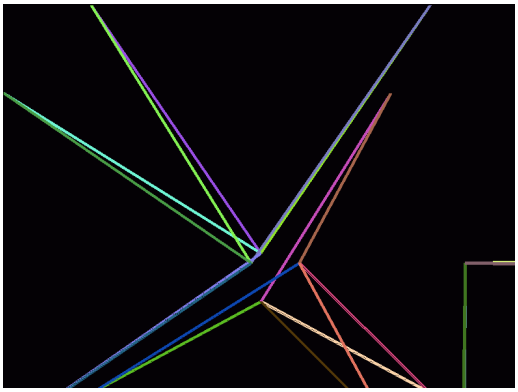
I added finishing touches to the visualizer. The updates for changing frequency will be shown by the action of the lines, as their start and end position will be affected by the fft_averages. Also to add to the aesthetic, the screen fill will be affected by a certain index of fft, so that a random color could be generated for the screen fill.

V. Evidence of Working Program

These images below are 3 samples of updates from the visualizer.







VI. Reference

Shepherd, A. (2012, September 7). *python-visualizer*. Retrieved October 25, 2017, from Github: https://github.com/n00bsys0p/python-visualiser/blob/master/plotvals.py