SE 3XA3: Test Report
xPycharts

Team 4, xPy
Hatim Rehman (rehmah3)
Louis Bursey (burseylj)
Sarthak Desai (desaisa3)

December 8, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	Version	Notes
Dec. 7, 2016	1.0	Revision 1

1 Functional Requirements Evaluation

1.0.1 Area of Testing 1

Requirement #1: The software shall read data given to it.
Requirements #4: The software will plot all the data points.

1. **Test ID #1.1**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (4, 4)]
   Expected Output: A window depicting a graph with plotted points at (1, 1), (2, 2), (3, 3), and (4, 4).
   Output: A graph with the points (1,1), (2,2), (3,3) and (4,4) was created.
   Result: PASS

2. **Test ID #1.2**
   Type: Functional, Dynamic, Manual
   Initial State: None Object.
   Input: Instantiate Graph(6, data = [(1, 1), (2, 2), (3, 3), (4, 4)])
   Expected Output: A window depicting a graph with plotted points at (1, 1), (2, 2), (3, 3), and (4, 4).
   Output: A graph with the points (1,1), (2,2), (3,3) and (4,4) was created.
   Result: PASS

3. **Test ID #1.3**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: The list object: [ ].
   Expected Output: A window depicting a graph with no plotted points.
   Output: An empty graph was outputted.
   Result: PASS

4. **Test ID #1.4**
   Type: Functional, Dynamic, Manual

Initial State: None Object.
Input: Instantiate Graph(6, data = [ ])
Expected Output: A window depicting a graph with no plotted points.
Output: An empty graph was outputted.
Result: PASS

1.0.2 Area of Testing 2

Requirement #2: The software will raise an exception if the data format cannot be plotted, and stop the program.

1. **Test ID #2.1**
   Type: Functional, Dynamic, Manual
   Initial State: A testing script that imports the method that validates data.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (4, 4)]
   Expected Output: A safe end to the execution (i.e, no exception raised).
   Output: The program completed execution.
   Result: PASS

2. **Test ID #2.2**
   Type: Functional, Dynamic, Manual
   Initial State: A testing script that imports the method that validates data.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (4)]
   Expected Output: The program raised an exception.
   Output: An exception was raised with output in terminal as "Exception: Inconsistent data"
   Result: PASS

3. **Test ID #2.3**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant..
   Input: Instantiate Graph(6, data = [(1, 1), (2, 2), (3, 3), (4)])
   Expected Output: The program raised an exception.

Output: An exception was raised with output in terminal as "Exception: Inconsistent data"
Result: PASS

1.0.3 Area of Testing 3

Requirement #3: The software will construct a coordinate system that will fit all the data points.

1. **Test ID #3.1**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (17, 77)]
   Expected Output: A window depicting a graph with max x axis value to be 17 and -17, and y axis to include 77 and -77.
   Output: Max y axis was still 6 and -6
   Result: FAIL

2. **Test ID #3.2**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (-17, -77)]
   Expected Output: A window depicting a graph with max x axis value to include 17 and -17, and y axis to include 77 and -77.
   Output: Max y axis was still 6 and -6
   Result: FAIL

   Two more test cases were added to validate the functionality the above test cases were trying to test.

3. **Test ID #3.3**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (17, 77)]
   Expected Output: A window depicting a graph with max x axis value include 17 and -17, and y axis to include 77 and -77.
   Output: A graph with all the points plotted, and an x axis from -18

to 18, and y axis from -78 to 87
Result: PASS

4. **Test ID #3.4**
   Type: Functional, Dynamic, Manual
   Initial State: None object
   Input: Instantiated Graph(6, [(1, 1), (2, 2), (3, 3), (-17, -77)]
   Expected Output: A window depicting a graph with max x axis value include 17 and -17, and y axis to include 77 and -77.
   Output: A graph with all the points plotted, and an x axis from -18 to 18, and y axis from -78 to 87
   Result: PASS

1.0.4 Area of Testing 4

Requirement #5: The software will connect a line that passes through all the data points if the data points are a function of x.

1. **Test ID #4.1**
   Type: Functional, Dynamic, Manual
   Initial State: An instantiated Graph object.
   Input: The default math.sin() function in python.
   Expected Output: A window depicting the sin() graph.
   Output: A graph with the sin wave plotted
   Result: PASS

2. **Test ID #4.2**
   Type: Functional, Dynamic, Manual
   Initial State: An instantiated Graph object.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (4, 4)] on the method plot_points_with_line()
   Expected Output: A window depicting a graph with the points plotted, and a line is connecting all points.
   Output: A line passing through the points in the input
   Result: PASS

3. **Test ID #4.3**
   Type: Functional, Dynamic, Manual

Initial State: An instantiated Graph object.
Input: The list object: [(1, 1), (2, 2), (3, 3), (3, 4)] on the method plot_points_with_line().
Expected Output: The software will plot the points but not connect a line through them.
Output: The points were plotted, but the line was not drawn.
Result: PASS

2 Nonfunctional Requirements Evaluation

2.1 Look and Feel

Description: These tests were conducted to ensure that the output graphs that are produced by the library are visually appealing to the user. This was a consideration as the library may be used to create graphs that may be used in presentations, reports and other academic or research environments. It is important that the graphs look professional and appealing.

Test: 7.1 - Pass

Results: 75% of the testers said that the graphs produced by the library were visually appealing in the user survey that was conducted. This satisfied the requirement to have at least 70% of users who find the graphs visually appealing. Since, the requirement was satisfied no changes were required to the implementation

2.2 Usability

Description: This test was used to check of the usability of the library in through a timed task provided to the testers. The goal was to have testers create all the graphs independent of any assistance from the developers except a user guide document.

Test: 8.1 - Pass

Result: The goal of being able to create all graphs with only the assistance of the user guide in 20 minutes was achieved by majority of the testers. 83%

of the testers completed the task within the given time frame surpassing the requirement of 80%.

Description: This test was to access whether the error messages produced by the library were sufficient in helping the testers identify the error.

Test: 8.2 - Fail

Result: The goal of having at least 80% of users understand the error messages was not achieved as only 75% of the testers were clearly able to understand the errors and correct them. This means a revision of the error messages must be done and the messages need to be made more clear for user understanding.

2.3 Performance

Description: This test for performance uses a Sine Function that plots 12000 points to produce a Sine function graph. The standard time library was used to time how long it takes to produce the graph. The aim was to have the graph produced in under 5 seconds

Test: 9.1 - Pass

Result: The test returned results that were that suggested that the graphs were created quickly and within the time frame. The tests original expectation suggested that each graph should be produced in no more than 5 seconds however it only took approximately 1.7 seconds for the graph to be plotted.

2.4 Robustness

Robustness tests are tested alongside the comparison to existing library tests. Specifically, sections 11.1, 11.2, 11.4. These tests existed to see how jCharts, the original library would deal with erroneous data, which made sense to do.

2.5 Operational and Environmental

Description: The purpose of this test was to ensure that the library is functional on different operating systems given that they have Python 2.7 in-

stalled onto the system. On each of the operating systems a test program (that works on the original platform) run on Python.

Test: 10.1 - Pass

Result: All operating systems were able to handle the task and in each case graphs were produced without any errors or issues.

3 Comparison to Existing Implementation

Test: 11.1

Description: This test was compare how the XPYCharts and jCharts libraries deal with infinite range inputs. It was expected that the XPYCharts would not be able to deal with this input and an error will result. jCharts library was expected to catch this exception and output a message.

Result: PASS. The test shows that the XPYCharts library cannot deal with such inputs and so changes must be made to the how it deals with unexpected inputs. When jCharts was provided with the same input it was able to catch the exception and gave back a message.

Test: 11.2

Description: Goal was to compare the results of entering an empty list of data into the library and compare whether XPYCharts deals with this input in a similar manner to the jCharts library.

Result: PASS. Both graphing libraries produced an empty graph when given as input an empty list of data. This means that XPYCharts faithfully recreates this aspect of the jCharts library.

Test: 11.3

Description: This test was to compare how the two libraries deal with a long list of repeated inputs. It is expected that a good library would be able to ignore such repetitions and not take a long time by just graphing one of the inputs. In this case it is expected that jCharts would be able to deal with such a situation and would simply ignore the repetitions and not take a long time to make the graph. On the other hand it is expected that XPYCharts will take a considerably long time according to the current implementation.

Result: PASS. jCharts was able to quickly graph the input without being stalled out due to the long list of repeated inputs. XPYCharts took considerably longer to graph the same set of inputs. This was the expected output, however it means that changes must be made to the XPYCharts library to deal with such a situation.

Test: 11.4

Description: This test is to compare how the two libraries deal with erroneous inputs where some of the values are (x,y,z) coordinates and not just (x,y). Recall the scope of XPYCharts only covers the XY plane. It was expected that in this case XPYCharts would simply avoid the z value and graph the tuple as a XY tuple. jCharts is expected to catch this exception return a message.

Result: PASS. XPYCharts graphs the tuple as a XY tuple as expected. However this is not ideal as an error message should be produced letting the user know that a (x,y,z) tuple cannot be expected. jCharts gives an error message as expected.

4 Unit Testing

4.0.1 _get_translated_point(self, coord)

1. **Test ID #1.1**
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: Dictionary with $x := int \vee float$, $y := int \vee float$
   Assertion: Function returns an int or float

   Table 2: Test cases

Input
$[(1,-1),(2,-1),(3,-2),(4,-3),(5,-5)]$ $[(-4,4),(-2,1),(-1,1),(1,1),(2,2),(4,5),(5,33)]$

   Result: PASS

2. **Test ID #1.2**
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: Dictionary with $x := int \vee float$, $y := int \vee float$
   Assertion: Function appropriately scales given coords
   Result: PASS

4.0.2 _Lagrange(self, x)

1. **Test ID #2.1**
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Assertion: Function returns a function
   Result: PASS

2. **Test ID #2.2**
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.

Assertion: Function returns appropriate function
Result: PASS

4.0.3 checktype(x, y)

1. **Test ID #3.1**
   Initial State: Empty script
   Input: Set of dictionaries with $x := int \vee float$, $y := int \vee float$
   Assertion: Function raises no exception
   Result: PASS

2. **Test ID #2.1**
   Initial State: Empty Script.
   Input: Set of dictionaries with one member not respecting $x := int \vee float$, $y := int \vee float$
   Assertion: Function raises exception
   Result: PASS

4.0.4 _is_function(data)

1. **Test ID #4.1**
   Initial State: Empty script
   Input:
   Assertion: Returns true
   Result: PASS

4.0.5 clean_data(data)

1. **Test ID #5.1**
   Initial State: Empty script
   Input: Non tuple-list variable
   Assertion: Throws Inconsistent Data error
   Result: PASS

2. **Test ID #5.2**
   Initial State: Empty script
   Input: Valid data set
   Assertion: Returns formatted list of dictionaries
   Result: PASS

4.0.6 scale(data_set, round_to = 0)

1. **Test ID #6.1**
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: Set of dictionaries with $x := int \vee float$, $y := int \vee float$
   Assertion: Returns max of x and y respectively
   Result: PASS

2. **Test ID #5.2**
   Initial State: Empty script
   Input: Set of dictionaries with $x := int \vee float$, $y := int \vee float$. round_to $\neq 0$
   Assertion: Returns max of x and y respectively. Return values are multiples of round_to
   Result: PASS

5 Changes Due to Testing

As seen in the Area of Testing 3.1 and 3.2 for Functional Requirements Evaluation, the test cases had failed to produce the expected output.

After manual code evaluation, it was realized by the developers that this was because an instantiated graph defaults to drawing the axes under the assumption of the data given to it. Because the test case was instantiating the Graph object with NoneType data, the graph would default to drawing the axes with a scale of 1 (therefore, the max values of magnitude 6, for the 6 markings). When calling the plot_...() methods, it would draw on a canvas already painted in the initializer.

To fix this, a few extra lines were added to the plot_...() methods where they would call the init() method to reinitialize the graph canvas and paint

it with the appropriate axes. **The two test cases passed once these changes were complete.**

Additionally, according to the results of test case 8.2 changes were made to the input parser to output better error messages so that the users can understand what they did wrong while inputting the data.

6 Automated Testing

Because of the timeframe of the project, the extent of the automated testing was unit testing, and testing of source code (seen in the test case below) using the online tool Pylint.

6.0.7 Area of Testing 5

Testing of source code.

1. **Test ID #5.1**
   Type: Functional, Static, Automated
   Initial State: Completed source code.
   Input: Source code.
   Expected Output: Review of source code.
   Output: log-file.
   Result: N/A

It would not have been feasible to do bitmap comparison automated testing for this project because to do so correctly and accurately would have been out of the scope of the project. The developers verified the outputs of the graphs manually to realize that the behavior was up to expectations, and the challenge in the nature of this sort of automated testing would not have been worth the value compared to the time dedicated to it, especially to realize an outcome that was already manually verified.

7 Trace to Requirements

8 Trace to Modules

9 Code Coverage Metrics

Code coverage metrics are important in showing the extent of the testing that the project underwent. We decided to use line coverage as our coverage metric. This is easiest to implement given the nature of the project. Manually examining the execution of our tests determined that 99% line coverage was achieved. This surpasses our original goal of 90% code coverage, and should be considered a success. The few lines that were not covered were trivial try/except blocks that do not currently require testing. As the project expands, these sections may become more relevant, and code coverage should be reexamined in future iterations.

More in depth code coverage analysis, with the python coverage.py application , was not possible due to the reliance of this project on the TKinter library, and the graphical nature of the project.

References

Test ID	**Requirement**
Functional Requirements Testing	
1	R1, R4
2	R2
3	R3
4	R5
Non-functional Requirements Testing	
7	RNF1
8.1	RNF2, RNF3, RNF4, RNF5
8.2	RNF5
9	RNF6, RNF9, RNF10
10.1	RNF11, RNF12, RNF13, RNF15
Comparison to Existing Implementation	
11.1	R1, R2
11.2	R3
11.3	RNF6, RNF8
11.4	RNF6, RNF8, RNF10

Table 3: Trace Between Tests and Requirements

Test	**Module**
Functional Requirements Testing	
1	M1, M3.1, M3.3, M2.2, M1.1
2	M2.1
3	M3.3
4	M3.2, M1.1
Non-functional Requirements Testing	
7	M1.1
8.1	M1, M2.1, M3
8.2	M2.1
9.1	M1, M2.3, M3
10.1	M1
Comparison to Existing Implementation	
11.1	M3.1
11.2	M3.1, M2.1
11.3	M2.2, M3.2, M3.3
11.4	M2.1

Table 4: Trace Between Modules and Tests