TEAM SQUAAD

Deliverable 4

Table of Contents

Issue #8818

Issue #8818 on GitHub || Occurs in Python 2 and 3 || Pull Request #10810

The Bug

This error occurs when making a structured array (using Numpy with names/labels for data) and plotting the data within the structured array by passing in the names and structured array using pyplot's plot function (pyplot.plot).

pyplot.plot calls axes.plot which has a _preprocess_data that is executed prior to axes.plot(). The _preprocess_data is set to call the _plot_args_replacer() function which determines the meaning for each input parameter (i.e. data points vs colour/appearance attributes), and parses the input parameters into the proper data points format to be plotted. It does so by comparing the second input parameter to see if it is listed to be one of the data points to be plotted. If the second input parameter is not found in the set of data points, then it is assumed to be a colour attribute - plot_args_replacer() returns with ["y", "c"] to indicate that the second input parameter should be interpreted as a colour/line/marker attribute; therefore, no parsing will take place for the second input parameter. This is an issue because in reality, the second input parameter can be a field name for a structured array instead of the actual data point value; in this case, the name will almost never match any data points in the data (e.g. the field name "abc" will never equal to "(1, 1)").

Due to the incorrect interpretation and parsing of input parameter during the preprocess stage, axes.plot() tries to parse the second input parameter (which is a field name for a structured array) as a colour attribute regardless of whether it can be properly processed as a colour attribute or not. Consequently, when _process_plot_format() is called later by axes.plot() to interpret the "colour attribute", a ValueError is raised because the input parameter (a field name) is not a proper colour/line/marker attribute.

The Fix

The final solution was to rearrange the order that _plot_args_replacer() checks what type of value is given.

- We've rearranged the checks within axes._plot_args_replacer() so that the function checks whether if the second input parameter can be properly parsed into a colour first.
- If the second input parameter can be properly interpreted as a colour attribute without raising any errors, then the method double checks that the parameter is not in data to confirm that it is a colour.
    - If the second input parameter can be interpreted as a colour and is also found in the list of data points, we favor the data interpretation and assume both parameters are data; a warning will be given to the user to highlight this behavior.
- If the second input parameter cannot be properly interpreted as a colour attribute without raising any errors, it is assumed that both parameters are data.

Tests Ran

- Tested scenario where two strings were given as field names to a Numpy structured array (e.g. pyplot.plot("field_name1", "field_name2", data=s_array); this is the original scenario in the bug report on GitHub
- Tested scenario where two strings were given as field names to a Numpy structured array, and an additional colour attribute (e.g. pyplot.plot("field_name1", "field_name2", "r-", data=s_array)
- Tested scenario where one string was given as a field name to a Numpy structured array (e.g. pyplot.plot("field_name", data=s_array)
- Tested scenario where one string was given as a field name to a Numpy structured array, and an additional colour attribute (e.g. pyplot.plot("field_name", "r-", data=s_array)
- Tested scenario pyplot.plot(s_array["field_name1"], s_array["field_name2"])
- Tested basic plot scenario pyplot.plot(t, s) where t is an array of x values, and s is an expression of t
- Ran matplotlib test suite

Confidence in Our Solution

We are confident that this solution works and will not affect and that it does not affect the other parts of Matplotlib.

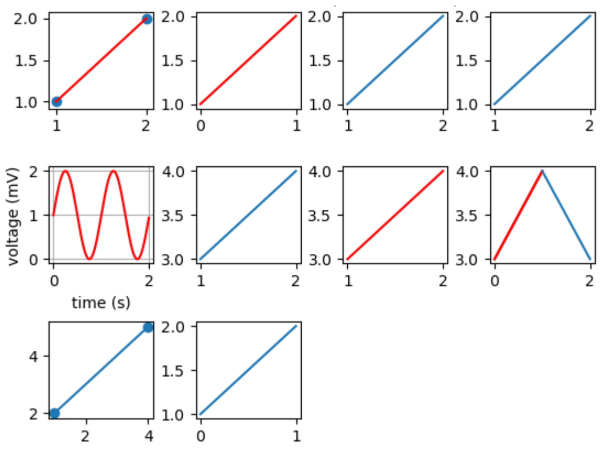
Not only did we create many tests (see: "Tests Ran") to compare our new solution with the old function with, but we also studied and learned how the existing plot() function works in Matplotlib in order to design a solution which was correct and efficient.

Fix Analysis

We wanted to ensure that the changes in behavior is minimal to avoid breaking any current working behaviors, and avoid modifying current expectations. Since the previous behavior favored data points over colour attributes, we kept this behavior by defaulting to data points interpretation when the input parameters fail to be interpreted as colour attributes. The previous behavior assumes that the second input parameter is a colour attribute immediately after failure to find it within the data points; we moved this check to be performed only after confirming that the input parameter can be interpreted as colour - this is to minimize the errors thrown by giving the parameter a “second chance” to be interpreted when the colour interpretation failed.

An alternative solution was to insert additional try and except blocks after the preprocess to catch ValueError raised by _process_plot_var_args(), then find alternative possible interpretations for the input parameter later within plot(); but this is insufficient because it might inhibit actual errors caused by bad colour attributes due to wrong error messages. Therefore, this proposed solution was abandoned.

Another alternative solution was to modify axes._plot_args() to include an additional check (try/except ValueError) prior to passing the second input parameter to _process_plot_format(). However, this solution would involve more process (additional parsing from later within plot_args(), and additional try/except block) to produce the same result as the final solution; therefore, this proposed solution was rejected as well.



These are the graphs produced from a few test cases, including the code given in the Github post.

Time Spent

3 hours: Analyzing where the bug is
4 hours: Discussing possible solutions
4 hours: Analyzing the amount of changes/consequences with different methods/solutions
5 hours: Implementing the fix
6 hours: Writing test cases to increase confidence and running these tests
2 hours: Running the entire py.test suite to ensure fix does not impact existing features
3 hours: Creating the pull request, writing a unit tests, PEP8, formatting the request as required
----
Total: ~27 hours

Lessons Learned

- It is better to implement a solution earlier on in the code to prevent the error from happening; rather than try/except to recover from an error that have already occurred.

- Continue to search for better solutions until the most efficient one is found.

- If there are more than one solutions produces the same results, the simplest solution should be preferred

Issue #9462

Issue #9462 on GitHub || Occurs in Python 2 and 3 on Ubuntu

The Bug

When drawing several identical subplots with bbox_inches='tight' and wspace=0 and outputting as a png with default DPI using the agg backend, the bounding boxes of the subplots will not line up correctly (they will be off by approximately one pixel).

Analysis of Fix Failure

In the end, we were unable to successfully fix this bug. We initially believed the bug to be caused by bounding boxes not being calculated correctly. Upon further inspection we decided that was not the case, as the bounding boxes were identical for each of the identical subplots. Next we believed the bug was in the section of code responsible for arranging subplots when bbox_inches='tight'. However, after extensive proding and testing the same render with all of the different backends, we discovered that the bug was specific to the agg backend. The agg backend calls it's renderer through a C++ interface where all rendering is done as a collection of paths to be rendered.

After extensive analysis of the path rendering algorithms, we decided that the algorithms were correct and that the issue was caused due to numerical precision errors caused by many floating point operations combined with the low default DPI when rendering PNGs. We evaluated and confirmed this by noticing that as the pixel density increased, the amount of error in the render decreased. Despite trying to reduce the amount of precision reducing operations in the algorithm, the issue was not sufficiently resolved.

The main source of failure to fix this bug was the amount of time and understanding needed to pinpoint the cause of the issue - we did not have any prior knowledge in regards to how image rendering worked, much less how image rendering worked in a large project such as Matplotlib. This greatly served as a roadblock for us, since a lot of additional time was required to understand the workings of each of the different backends in order to determine the difference (and cause of error, if any). Another roadblock was the fact that the file formats were all very different, so there was less common ground as a basis for comparison when investigating how the working file formats differed from the broken file format - this greatly increased the amount of learning we had to do for each backend/file type, increasing the amount of time spent investigating for the error in the code.

Time Spent

3 hours: Compiling Matplotlib from source, getting/fixing all dependencies, and running py.test
2 hours: Identifying potential lines in the code where error could occur based off D3
4 hours: Learning the behaviour of bounding boxes in Matplotlib when ‘tight’ is specified
3 hours: Investigating the bounding boxes for different image formats (png, svg, pdf, etc.)
5 hours: Learning about the backend renderers of Matplotlib
5 hours: Learning how the agg rendering library works
9 hours: Investigating rendering algorithms for each file format and correctness
3 hours: Investigating the what causes only agg backend formats to break
2 hours: Studying the C++ wrapper and interfaces that matplotlib uses to interface with agg
8 hours: Attempting to improve path rendering algorithm to reduce error introduced from many floating point operations
----
Total: ~43 Hours

Lessons Learned

Don’t take on bugs relating to backend renderers when time is limited, as a lot of time was spent researching the underlying 3rd party rendering library and familiarizing oneself with how the Matplotlib backend interfaces with the rendering library. More importantly we learned that the more libraries/backends/coupling a project/feature/module has, the more difficult it is the trace down and fix a bug - especially if the coupled libraries/backends are nearly separate from the main project itself, since it will take a lot more time to learn and investigate the structure and identify where the bug could appear.

Another lesson learned is that the implicit type system of Python can make tracing bugs difficult due to functions and classes being stored in variables without being labeled as such. Without explicit types, identifying which variables are data and which variables contain functionality takes a non-trivial amount of time.

Issue #4354

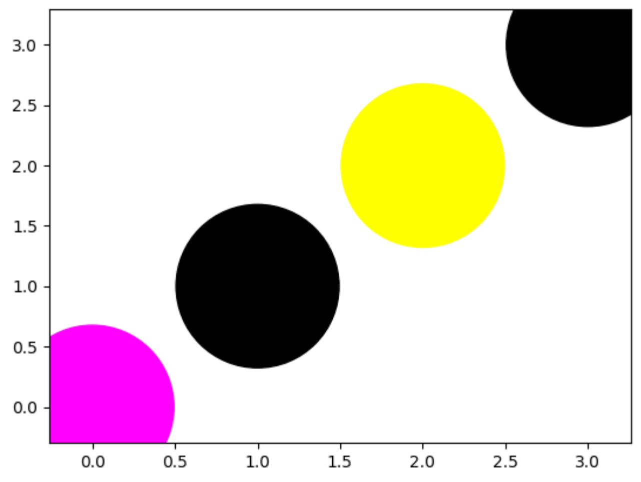
[Issue #4352](#) on GitHub || Occurs in Python 2 and 3 on Ubuntu || [Pull Request #10809](#)

Description

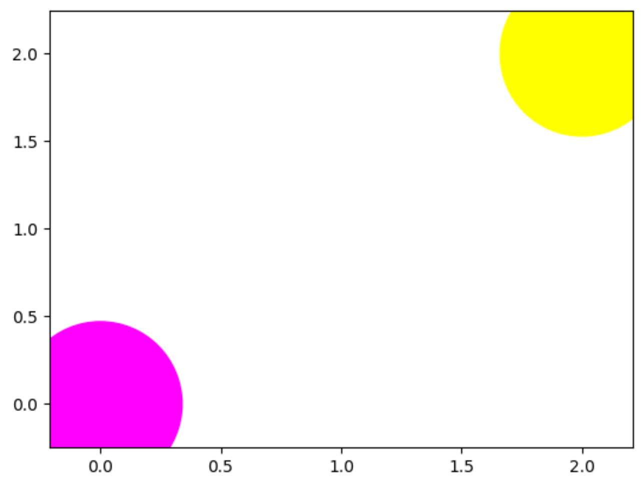
When using set_bad to designate what a bad colour should be, and plotting valid points with invalid colours, scatter should plot the points regardless of the invalid colour.

For example, when spring.set_bad('k', 1) is added, it sets all invalid colours to be black. So when using scatter([0,1,2,3], [0,1,2,3], c=np.array(1, np.nan, 5, np.nan), cmap=spring), it will not display the points with invalid colours, i.e. points 1 and 3 in x and y. The expected of course is where the valid points and invalid colour = black is plotted in. Demonstrated behaviour as seen in the following images/example.

Expected Behaviour



Actual Behaviour



As seen from the actual behaviour image, (1,1) has no colour designated, and (3,3) is just completely gone. Note that although the 2nd point, i.e. (1,1) is still there though not coloured, this is most likely a designed/expected behaviour due to the 3rd point being valid in both point and colour.

The Bug

Points with a masked/invalid color are expected to be drawn with a default color (white) and when set_bad() is called, the program is expected to display these points with a specified color. However, when scatter is called with valid coordinates but invalid colors, scatter deletes ignores points with invalid color instead of drawing them.

The Fix

The function scatter in _axes.py makes a call to a delete_masked_points helper function. This helper function is what is responsible for the bug; it takes several arrays of the same size and and looks for nan, infs, and masked values in all of the arrays. If it finds an invalid value in any of the arrays, it removes the element at that index from all of the arrays. So points with valid coordinates but invalid color were being removed.

Removing this helper function moved us closer to the intended behaviour. We discovered that this call is redundant and the arrays are being passed to areas of the program that are able to handle invalid values. However, this change broke the coloring of the points.

We tracked the issue down to the colors array being passed to color.Normalize. If NaNs/infs are in the array passed to color.Normalize, it outputs nonsense. The solution was to use np.ma.masked_invalid to convert NaNs/infs in the colors array to masked values which color.Normalize is able to ignore. We did not decide to modify color.Normalize because it dealt with floating point precision and mathematics that we did not have time to understand.

We removed some uses of np.asarray because they were unmasking the colors array and were redundant.

Finally, we decided to add an additional parameter to scatter to give the user the choice of not rendering invalid colors, adding the delete_masked_points back under an if statement.

Tests Ran

Note: spring is used only for the tests where masked=True and for each case, it tests for when masked=None and masked=True.

- For scatter:
    - x and y parameters both have valid values/points, but a few colour values in c are invalid; the last index, and the 3rd index from the right
    - masked=True and masked=None.
- For scatter:
    - x and y parameters both have valid values/points, but a few colour values in c are invalid; the last index, and 3rd index from the right.
    - s parameter is an array with an invalid/nan value
    - masked=None and masked=True.
- For scatter:
    - x and y parameters both have valid values/points, but a few colour values in c are invalid.
    - linewidth parameter is an array with valid values and an invalid/nan value.
    - edgecolor is black.
    - cmap=spring where spring.set_bad(‘b’, 1)
    - masked=None and True.
- For scatter:
    - x and y parameters both have valid values/points, but a few colour values in c are invalid.
    - third parameter is a masked array where all but 1 index is a valid value.
    - cmap=plt.get_cmap(‘jet’, 16).
    - linewidth is 1, edgecolor is ‘black’.
    - and masked=None and True.
- Matplotlib’s tests.py file was also used for regression testing.

Confidence in Our Solution

We are confident in not only the solution, but also that it will not affect the other parts of matplotlib.

Time Spent

Analyzing code: 3 hours
Hours to fix: 8 hours for:
- making rough fixes
- exchanging solutions
- selecting best solution
- refactoring the rough best fix to a better one
- finalizing the final fix

Testing: 8 hour for:
- coming, making, and setting up different tests cases
- making regression test cases and testing them
- testing using matplotlibs tests.py

Documenting code: 3 hours

Total: ~22 hours

Lessons Learned

No matter how simple a fix seems, it can always lead you down a rabbit hole and become much more complex than originally thought. In our case removing delete_masked_points started producing wrong colors in our plot images and we had to track down the reason why.

Issue #8768

Issue #8768 on Github || Occurs in Python 2 and 3

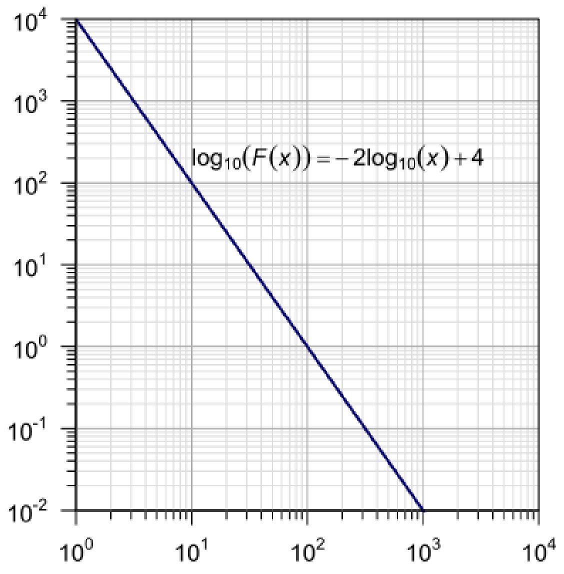
The Bug

This bug occurs when plotting a graph using the loglog function from a pyplot object. When the points given are spatially close together, the resultant graph will sometimes only show one or zero ticks on each axis. This is important because both axes are logarithmically scaled instead of being linear; without at least two ticks as reference points, it is difficult to determine the true formation of the graph.

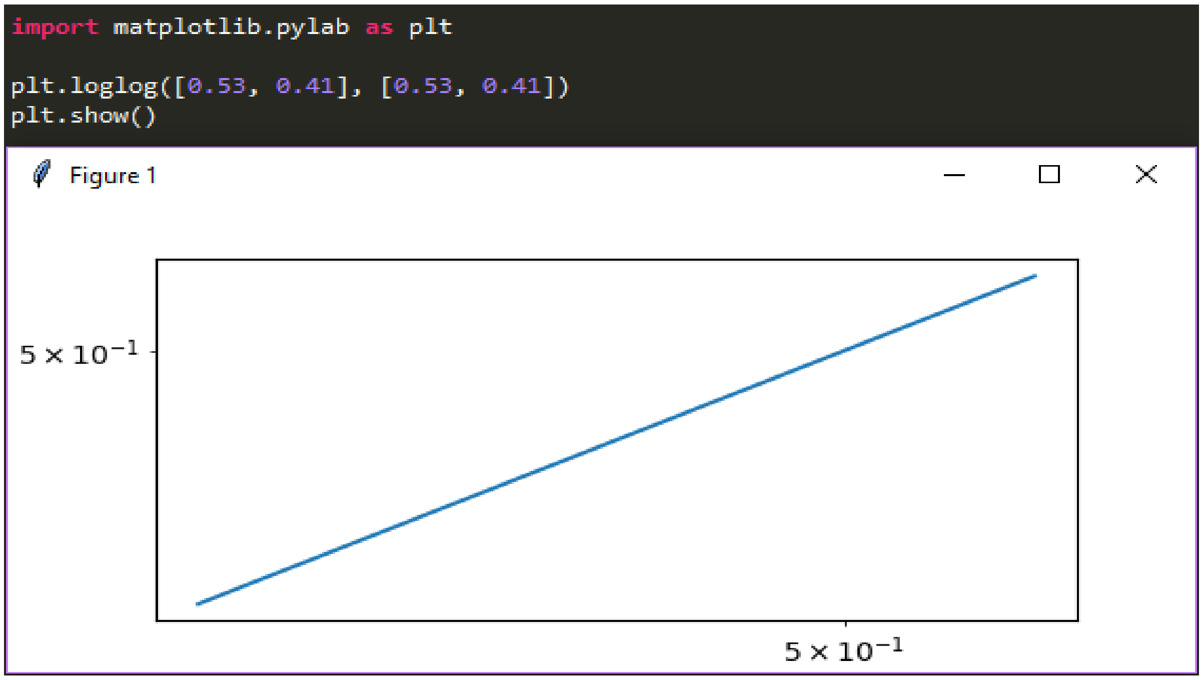
Inspection

Initially, we looked into and experimented with other Locator classes within ticker.py to obtain any sort of comparable information to LogLocator. The variance between the implementations did not help much and ended solely in attempting to redesign parts of LogLocator. The difficulty of this task stems from the fact that the code was working as intended, and as such was difficult to modify without inadvertently affecting its functionality. We thought of adding optional parameters similar to MaxNLocator, but ultimately decided that we wouldn't want these new parameters to be used for other purposes. Looking further into the code, we weren't able to fully access the actual tick placement from just within LogLocator, so we couldn't add additional tick endpoints without branching out into other code. Modifying relevant code within axis.py would most likely result in affecting other functionalities of matplotlib.

We now think that this issue is a problem that comes from the nature of the logarithmic graphs stemming from their logarithmic x and y axes. Due to the fact that logarithmic scales are non-linear, separate equal-distance regions on the axis do not represent the same range, as demonstrated by the following logarithmic scaled graph. Notice how the minor ticks are separated by non-equal intervals.



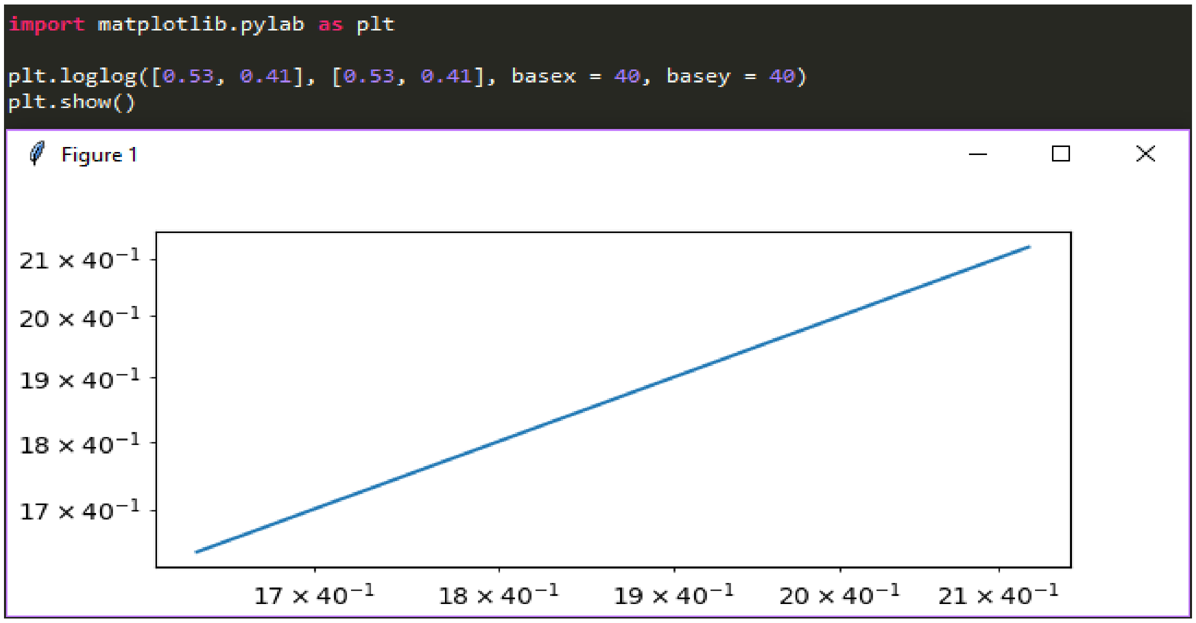
Graphs produced by the loglog function by default only show ticks of a integer multiple of a power of 10. In the buggy example, the (x, y) coordinates of (0.53, 0.53) and (0.41, 0.41) were shown, these points create a line that lies strictly on one base 10 tick and therefore the adjacent ticks do not show since they are out of the range of the graph.



```
import matplotlib.pylab as plt

plt.loglog([0.53, 0.41], [0.53, 0.41])
plt.show()
```

We can see ticks if we increase the base as seen below. We used the same coordinates as in the example but set the x and y axes to logarithmic base 40, and it can be clearly seen that the ticks show up in base 40 because the ticks are closer together and falls within the line created.



```
import matplotlib.pylab as plt

plt.loglog([0.53, 0.41], [0.53, 0.41], basex = 40, basey = 40)
plt.show()
```

This means that the tickers are not absent from the graph in the buggy case, the base was just too low and the graph created has too little range to show some of the ticks.

Solution Analysis

We did not redesign the code to accommodate for additional ticks. Logarithmic plotting such as this was designed with the intent for use with only log-friendly numbers and intervals. As such, this oversight is just an unfortunate downside of the logarithmic graph due to its characteristics. It could technically be fixed with a higher base value, but this number can be arbitrarily high and since base 10 is what humans are most comfortable with, we kept it at that. We could have also used non-integer exponents to show additional ticks, but we felt like this over complicated the graph and also the exponent would be forced to be decimal floats due to graphs displaying tick values in decimal. Overall, we felt that this issue cannot be helped if ticks can only be multiples of a certain base and the range of coordinates are sufficiently small; however, loglog graphs are still lacking information if each axis only contains one or zero ticks.

We figured one fix to this issue is to add the maximum and minimum x and y values to the axes as extra ticks so that each graph will have at least two ticks on each axis in every situation. This does fix the issue of not enough ticks but it might not be as useful in graphs that do not care for axis values since the added ticks might interfere with the default ticks and produce a non-uniform axis and a messy graph. The user, however, can easily fix this by changing the base values to a more appropriate number.

Tests Ran

Tests were mainly done on running the loglog function with different parameter values to figure out the where bug originates from. They include:

- Running *loglog* function in *pyplot.py* with zero coordinates (axes produced with no clear graph)
- Running *loglog* graph in *pyplot.py* with one coordinate (axes produced with no clear graph)
- Running *loglog* graph in *pyplot.py* with negative value coordinates (error)
- Changing parameter *base* in initializer of *LogLocator* class from 10 to 2 (no change)
- Changing parameter *subs* in initializer of *LogLocator* class from (1.0,) to None (no change)
- Changing parameter 'numticks' in initializer of LogLocator class from None to 2 (no change)
- Changing parameter of *LogLocator* class initialization in function *set_default_locators_and_formatters* in *LogScale* class from self.base to base = 2 (no change)
- Changing parameter of *axis.set_major_formatter()* in function *set_default_locators_and_formatters* in *LogScale* class from LogFormatterSciNotation(self.base) to LogFormatter(self.base) (error)

Confidence in Our Solution

We stand confident that this issue is not a result of a coding error, but a mere inaccuracy of logarithmic scales and how they are represented as a decision of the developers as well as a way to keep the graph clear and precise to the average user.

Time Spent

Total time spent researching and testing for this bug was around 27 hours. The breakdown is roughly as follows:

2 hours: Experimenting a bit more with Matplotlib
5 hours: Learning about and experimenting with the behaviour of tickers
3 hours: Learning about other relevant classes based on D3
5 hours: Complete tracing and retracing through different scenarios of buggy code
6 hours: Attempted redesign of LogLocator using additional parameters
6 hours: Modifying other various classes and additional attempts to using existing variables within LogLocator to plot the endpoint ticks.

Total: ~27 hours

Lessons Learned

- It is not always better to make it easier to add additional classes, but also to modify existing classes which have been added.

- A bug may not always reside within the class you think it's in, but may actually be a bug that is shared amongst multiple areas of code.

- One should not assume that a program won't be used in a certain way just because it is not the norm.