University of Toronto – Scarborough

CSCD01 – Engineering Large Software Systems

**Deliverable 04 – Report**

Prepared by ***GoonSquad*** Team

Swarnajyoti Datta

Nikki L. Quibin

Junil Patel

Beiyang Liu

Laine London

Hajoon Choi

Leo Li

Date: March 16, 2018

**Table of Contents**

[**Bug #1: Legend Annotate (Fixed)** 1](#_Toc508883223)

[**Bug #2: Bbox Tight Legend (Fixed)** 3](#_Toc508883224)

[**Bug #3: Logscale (Fixed)** 3](#_Toc508883225)

[**Lessons Learned** 3](#_Toc508883226)

# **Bug #1: Legend Annotate (Fixed)**

**Bug/Issue:** [Legend does not show ‘annotate’ #8236](https://github.com/matplotlib/matplotlib/issues/8236)

**Estimated Hours:**

* Explore and create a solution (7 h)
* Implement solution (20 h)
* Testing/validation (12 h)
* Code Review (3 h)
* Documentation (1 h)
* *Total: 43 h*

**Description:**

Legend items for annotations are currently not operational despite inputs currently being legal for them. Boxing the bi-directional arrows above into a legend to denote amplitude and wavelength is an example of a desirable use case.

When the legend() method on an Axes object gets called, the program eventually adds items to a list of handles to be inserted into the Legend for that Axes (see legend.py:1308, 1313). Annotations, stored in the Axes field texts (as Annotations are a sub-class of the Text class), are not currently added to this list. Additionally, the handler to construct the legend items for Annotations and Texts do not currently exist in the file legend\_handler.py (and are subsequently not mapped in legend.py:805).

**Solution:**

[Updated Forked Repo](https://github.com/SwarnaD/matplotlib/tree/legend-annotation)

The affected files are ***legend.py***, and ***legend\_handler.py*** where the updated files are found in ***solutions/legend\_annotate/***.

The solution to this bug follows precisely the original plan for it from deliverable 3. A handler for the legend Annotation was added and it was mapped to Annotation objects like the other legend items. It additionally makes use of handlers for Texts and Arrows (that we also added), because annotations can be composed of texts and/or arrows which are themselves separate entities. Text objects (super class of annotations) were added to the list of objects to be added to the legend and import statements were added where necessary.

As for composing the actual legend item, all types of annotations have been broken down into 3 cases: blank, text, arrow (with or without text). Blank annotations show up as blank on the legend. Making the legend omit adding this item by default would require going outside of this self-contained solution and this presents a niche use case, anyways. A user making a blank annotation constitutes a very specific action and if they wanted to hide it, they could just add the appropriate argument label onto the annotation. Otherwise, it would likely be intended behaviour on their part. Texts, again, are only omittable by user action for the same reasons. The legend item will be a replication of the text in the original font, colour, and style (scaled to an appropriate size). Texts that exceed the width of the legend item field will simply be cut off. This was chosen because sizing down texts was deemed to be more useless to the end user if they cannot read it in the first place. Annotations with arrows are perfectly replicated into the legend area. Colour, style, and thickness of different components are respected.

There is a competing pull request with regards to this bug. As explained, an unrelated contributor worked on this a year ago and left it inactive after failing to pass error tests. Recently, it was picked up by who we suspect are fellow D01 students. As such, we spent time running and profiling their solution to see what we were up against in terms of winning the pull request.

Our solution differs in that texts are represented based on the actual text string even when cut off while the other one replaces the text with a base string set (“Aa” or similar) in the same styling as the original when exceeding a certain limit character limit (the limit is often premature and depends on a character limit rather than the actual space available). We do not like this because there is no way to differentiate between long texts that have the same style in every other respect other than the actual string. Additionally, we have a strict improvement when handling arrows. The “linestyle” property of FancyArrows is not respected in the other solution and thus does not reflect certain aspects of arrows (like whether it is dashed). The other solution chooses to make “text + arrow” annotations a special case and displays both the text (usually in base text form “Aa”) and arrow within the small confines of the legend. We do not believe this is useful for anyone for the same reasons we do not like how texts are handled and because it takes away emphasis from the arrow (reducing its size and identifiability). Another difference is that we also decided not to legend Text and Arrow items on the plot outside of actual Annotation objects (by not mapping the handlers for them to legend creation). This was not the purpose of the bug fix nor do we see any useful functionality for it that wouldn’t interfere with normal operation. The default nature of entries being added to the legend unless explicitly argued against can prove to be annoying for users who simply want to add simple texts and arrows without having them showing up on the legend. We think that making it so that Texts/FancyArrows in Annotations being added to the legend by default and raw Texts/FancyArrows not being added to the legend by default offers the most flexibility without having to resort to omission arguments on the part of the user. If a user wants to make an arrow/text to show up on the legend, they just need make it an Annotation. However, this functionality, should anyone choose to do so, can be trivially added onto our solution with the addition of 2 lines of code in the ***\_default\_handler\_map*** object in legend\_handler.py with appropriate import statements. The handlers we implemented in for Texts and FancyArrows as a side effect of the Annotation handler will already suffice for this simple addition. Overall, we think we have a more complete and well-thought out solution.

**Testing:**

To test that the solution works, image comparison tests were used, as suggested by matplotlib when testing for changes to the graph figure. The associated tests are in ***solutions/legend\_annotate/tests/***. The file ***test\_legend\_annotate.py*** contains the specific test cases for annotations appearing in the legend. The result images are found in the folder ***/test\_legend/*** and should be copied over to ***lib/matplotlib/tests/baseline\_images/test\_legend*** in the actual matplotlib source directory when running tests. In addition, ***test\_legend.py*** is the formal way to test the legend, thus it includes the existing tests for legend and as well as the newly added test cases in ***test\_legend\_annotate.py***. As a result, when testing, this ***test\_legend.py*** should replace the existing matplotlib file.

Since the issue pertains to annotations not appearing in the legend, it is a rendering issue, so image comparison tests are used. Moreover, existing legend tests used image comparison tests for testing the labels, for example, ***test\_various\_labels()***.

There were six tests used. Below are the images used as the baseline images. The test cases cover all the existing linestyles, arrowstyles, couple of colours and texts, no arrow, text, or both, and a practical example of using annotations. Also, the existing tests were ran and have passed.

|  |  |
| --- | --- |
| *test\_all\_arrowstyles()* | *test\_all\_linestyles()* |
| *test\_annotation\_colours()* | *test\_annotation\_text()* |
| *test\_annotation\_no\_line\_text()* | *test\_simple\_annotation()* |

**Confidence in Solution:**

The proposed solution works very well since existing and new tests have passed. In addition, there are only two affected files and are only contained with respect to the legend, thus other parts of the matplotlib code are unaffected. Moreover, the implemented solution is a valid and practical fix as opposed to a “hack” fix. The solution consists of using the ***legend\_handler*** class to add the annotation to the legend, which was also the procedure for adding existing labels. Guidelines for contributing to matplotlib were also followed accordingly, such as proper documentation and following PEP8 guidelines. All in all, there is high confidence that the proposed solution works.

# **Bug #2: Bbox Tight Legend (Fixed)**

**Bug/Issue:** [Legend is not present in the generated image if I use “tight” for bbox\_inches #10194](https://github.com/matplotlib/matplotlib/issues/10194)

# **Bug #3: Logscale (Fixed)**

**Bug/Issue:** [Minor ticks on log-scale colorbar are not cleared #8358](https://github.com/matplotlib/matplotlib/issues/8358)

# **Lessons Learned**

There are a lot of lessons that our group has learned during our time working with matplotlib and fixing existing issues. Even though we’re not sure if any of our efforts will result in any accepted pull requests, it was a valuable experience working with open source projects. Listed below are just some of the various lessons we’ve learned and agreed upon.

* Working on an open source and large-scale project requires a lot of coordination. Therefore, it’s imperative that matplotlib had a guideline and standard for contributing.
* Quality of tests are important to validate that a solution works as opposed to just hoping it works.
* Fixing bugs in an open source project shouldn’t be a full-time thing, rather, something that should be considered a hobby or honing programming skills.
* Something that looks simple to fix can be really complicated; looks can be deceiving, just like the matplotlib code.
* Debugging is an unbelievably valuable skill to have as a developer, and it can be just as, if not more important than the developer's ability to write code.
* Having good test cases is very important to verify that the code functions according to the expectations and is also useful for finding errors and defects.
* Many of the issues have a common cause/background. (e.g. there were several bugs related to the legend and tight bbox option).
* Good in-depth analysis and documentation of code structures has an amplifying effect on development. The more specific the analysis was (as was the case with the legend annotation bug) the more immediately work was able to start, and progression speed was further amplified.
* An understanding of various design patterns is applicable to big projects and that it helps with the overall understanding of how the code structure works. This can be applied and transferred to future projects.
* If someone else works on the same issue, we must be quick and fast so that our update can be pulled first or be considered for other collaborators to work on. An example is the legend annotate bug pull request being closed because of another similar pull request addressing the same issue.