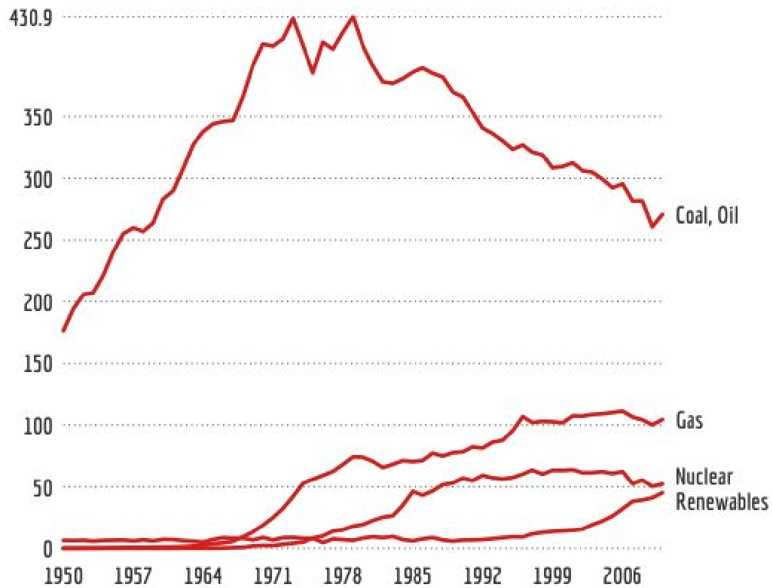
CSCD01 - Team I/O

Deliverable 3: Difficult Issue/Feature Analysis

Contents

Feature Selection

[Feature 12939]



Description

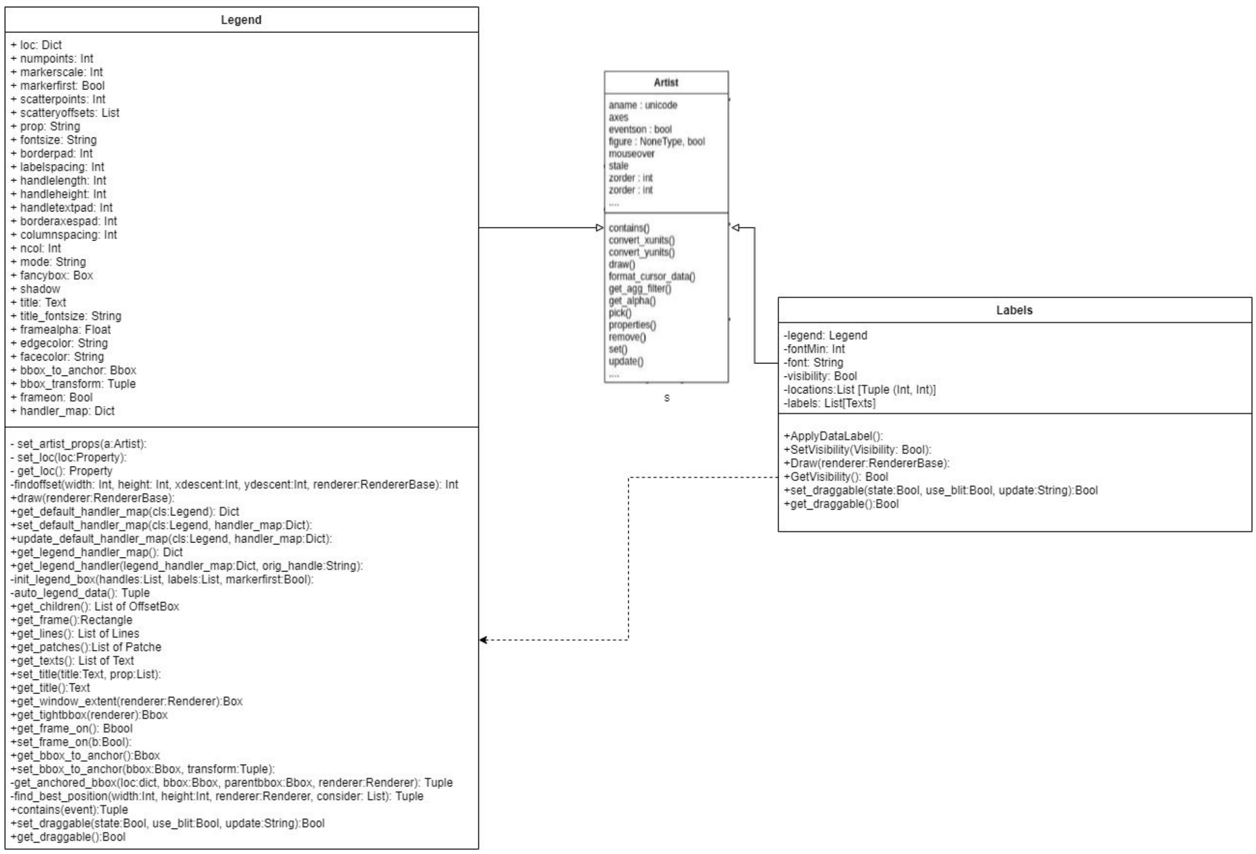
The first selected issue is a request for a new feature. The author asks for a feature which acts as a modified legend to the line graph. Rather than presenting the title of the line by the colour in a separate legend, the author asks for the ability to have the title of the line presented at the right side of the graph where the line ends. Through a little discussion by the lead developers it was decided that this will be a new feature and would have implementation separate from the legend of the graph.

Analysis & Architecture

Since the core contributors of Matplotlib have requested that the "Line labels" functionality be separated from the "Legend" functionality, our new feature would require:

- A new class in the Artist layer (ie. a "Labels" class inheriting from the "Artist" class). Moreover, since the requested labels are similar in functionality to the Legend, we will create and store a Legend object as a property of the Label class, thus, making Legend a dependency of Label.
- Appropriate methods in the scripting layer (ie. pyplot.py) to allow users to quickly add labels to the figure.
- Appropriate rendering in the backend layer (ie. figure.py, backend_bases.py) for the Label positioning code.

UML Diagrams

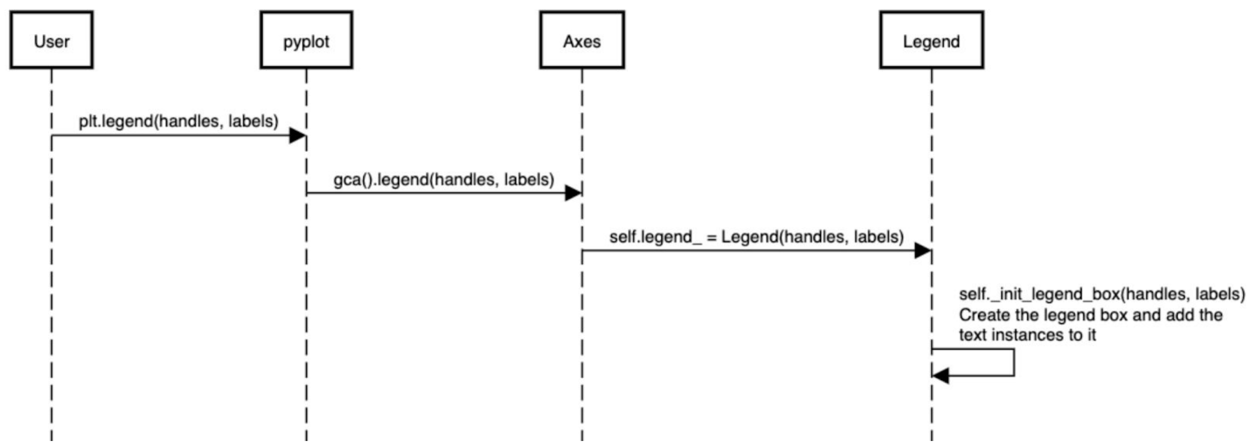


Sequence Diagram

Before

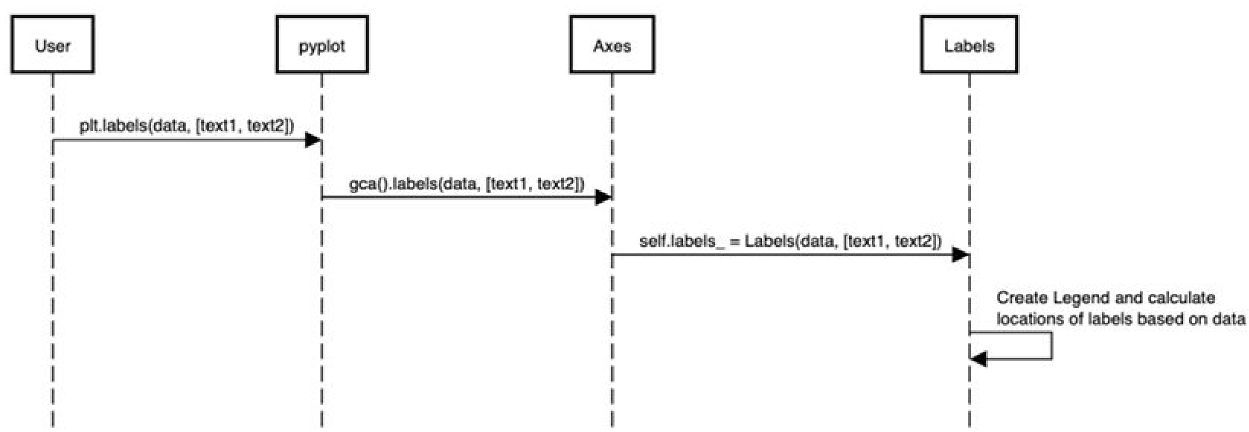
In this figure, we show how it is currently possible to achieve the desired image by using a legend and manually positioning the text vertically in the diagram.
We took inspiration from the following code:

```
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
red_patch = mpatches.Patch(color='red', label='The red data')
plt.legend(handles=[red_patch])
plt.show()
```



After

In this figure, we describe how our implementation of the Labels class will be accessible from the User layer.



Feature 15105

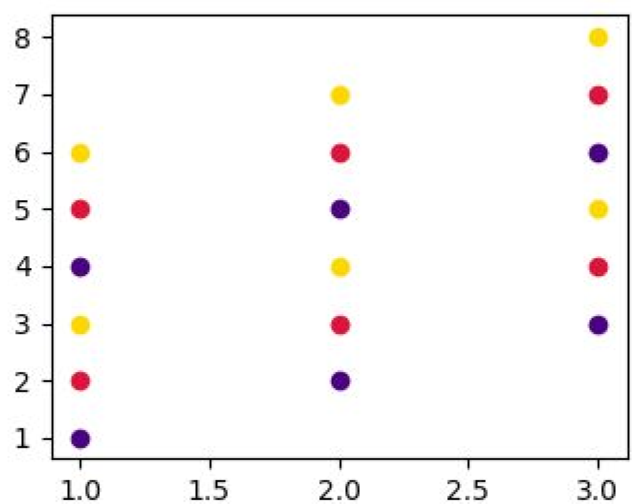
Feature Description

Currently the scatter function only cycles through colors in default props_cycle. This feature suggests that scatter could also cycle through markers and more things in general (ie, size).

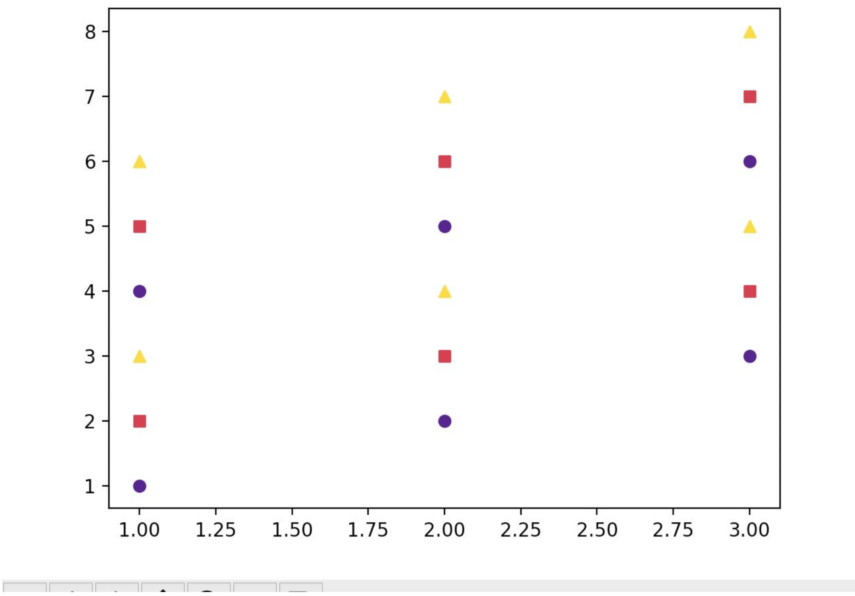
Example:

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

c = plt.cycler(color=["indigo", "crimson", "gold"],marker=["o", "s", "^"])
plt.rcParams["axes.prop_cycle"] = c

fig, ax = plt.subplots(1, 1)

x= np.array([1, 2, 3])

for i in range(6):
    ax.scatter(x, x+i)

plt.show()
```

Output



Expect output



Analysis & Architecture

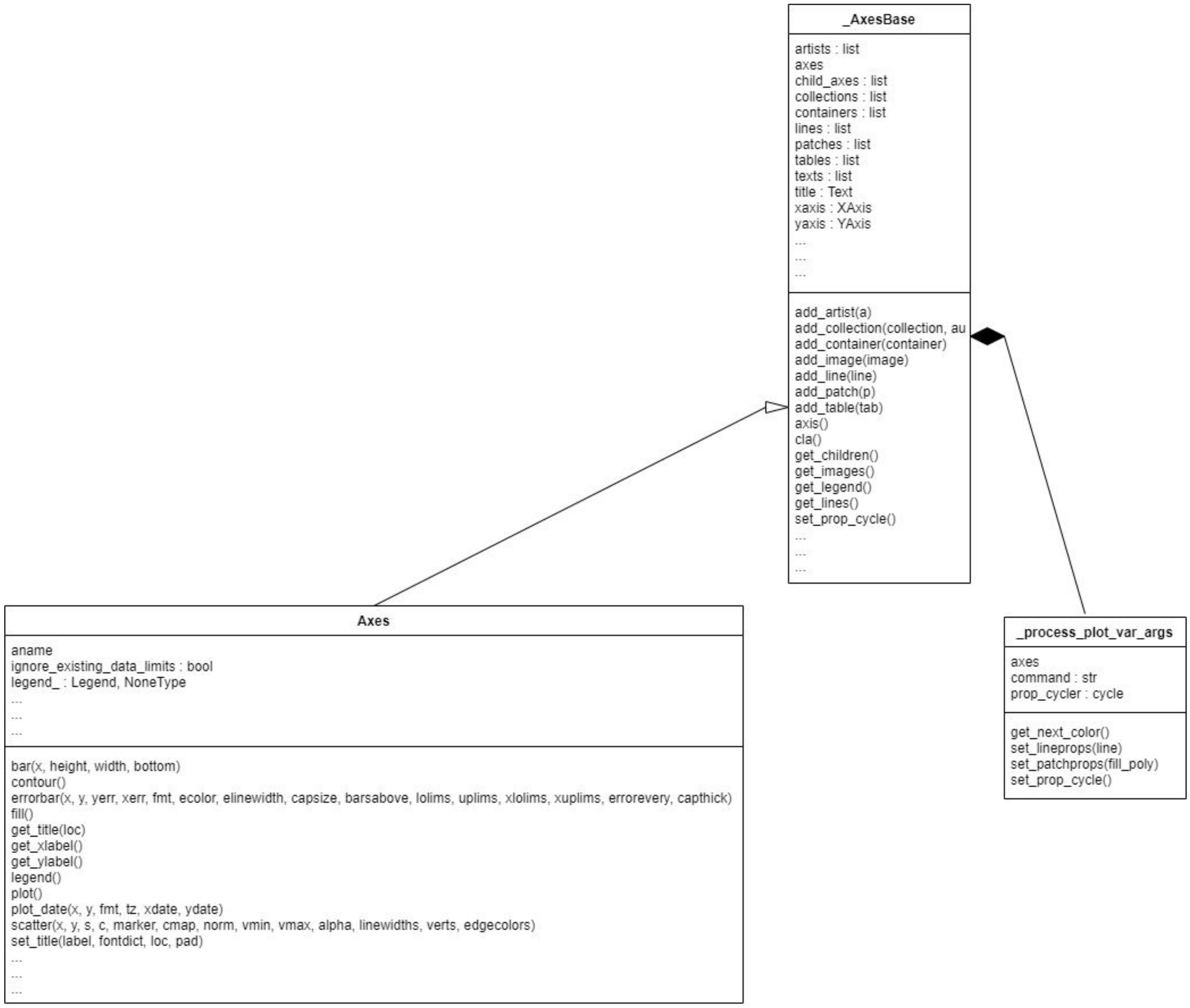
Currently the Scatter function does not read Marker given in default props_cycle, if the marker was not given when building the scatter function, scatter will assign mark ‘o’ to scatter points in the graph. The implementation of this feature would need to make the scatter read the marker from default props_cycle if marker was not given, if marker was also not given in props_cycle, then assign ‘o’ as marker to scatter points.

Code changes proposal:

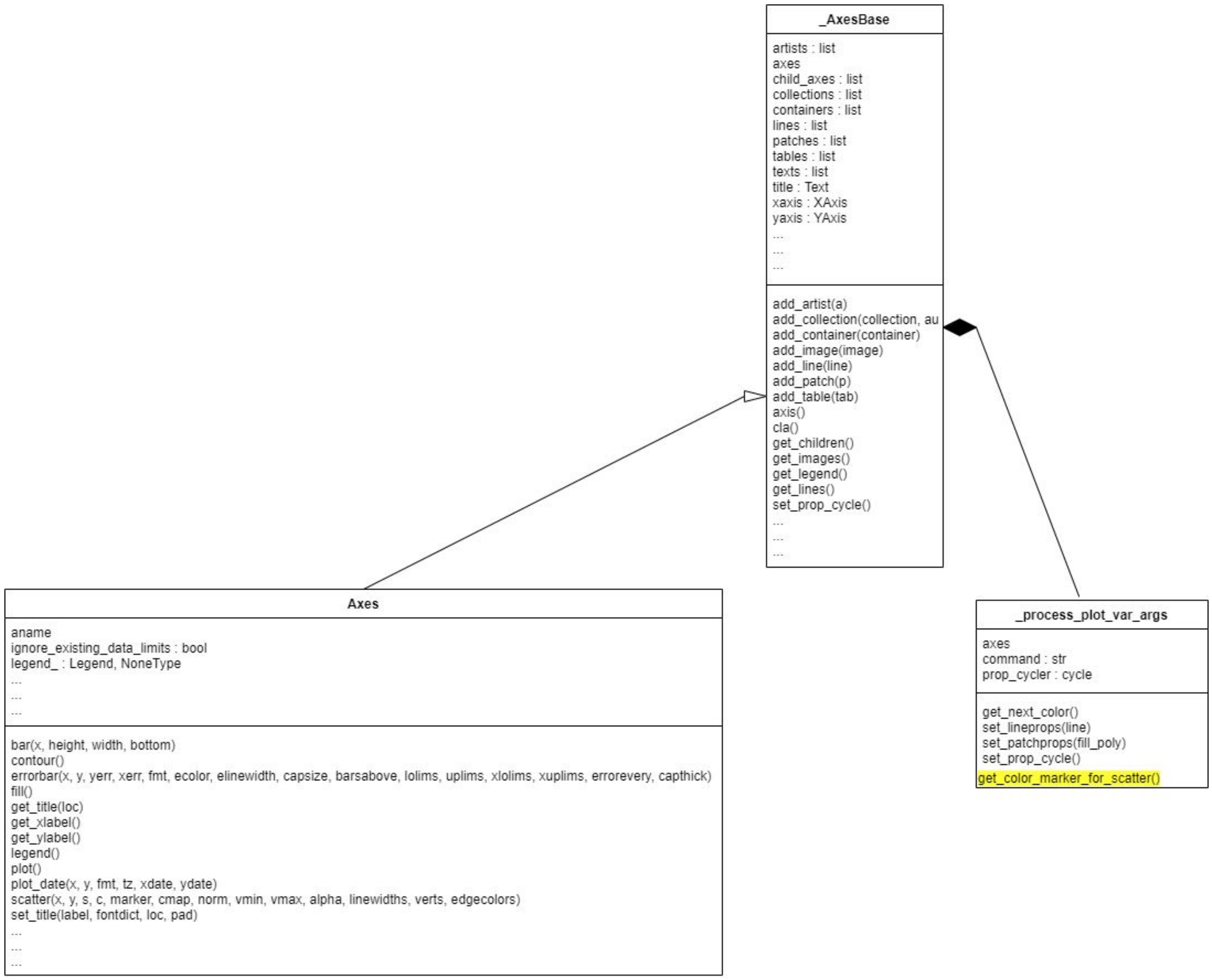
- _base.py:
  - Add new function to iterate through cycler to get parameters for scatter
- _axes.py:
  - Add additional function call to get parameters from cycler, and set up necessary properties for scatter

UML Diagrams

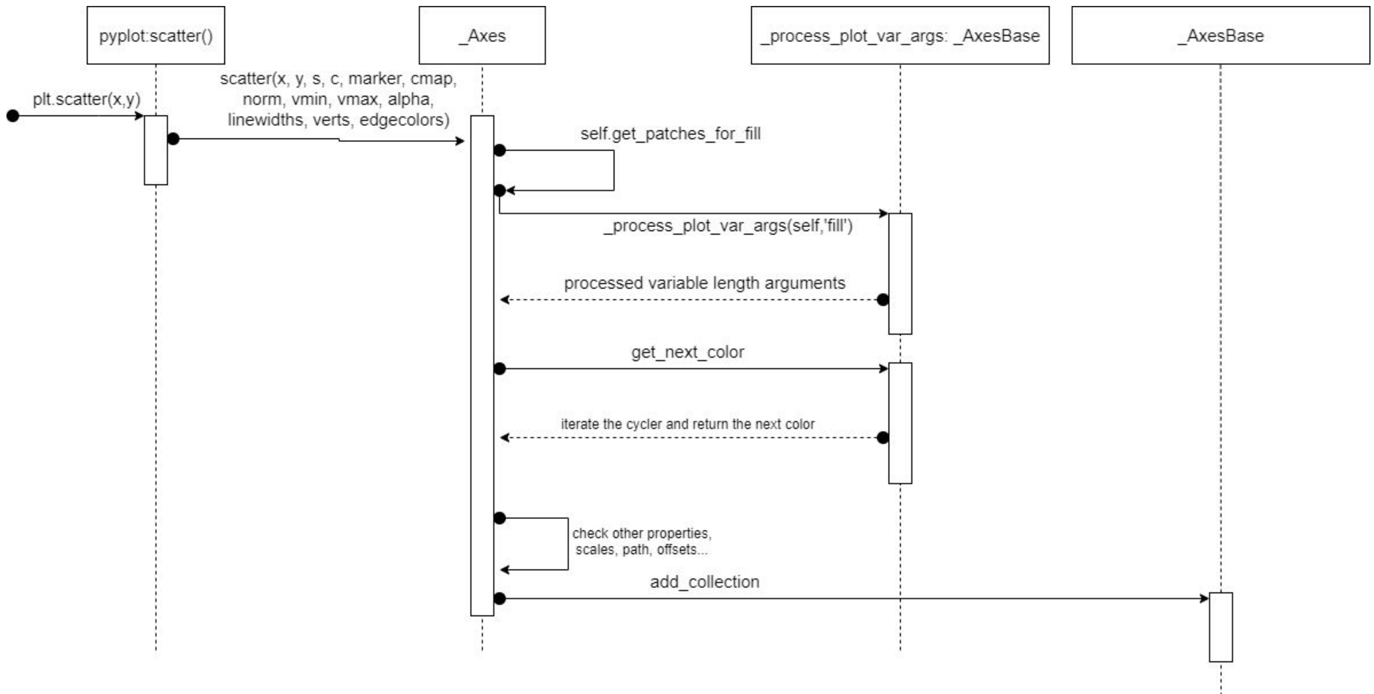
Class UML - before the change:



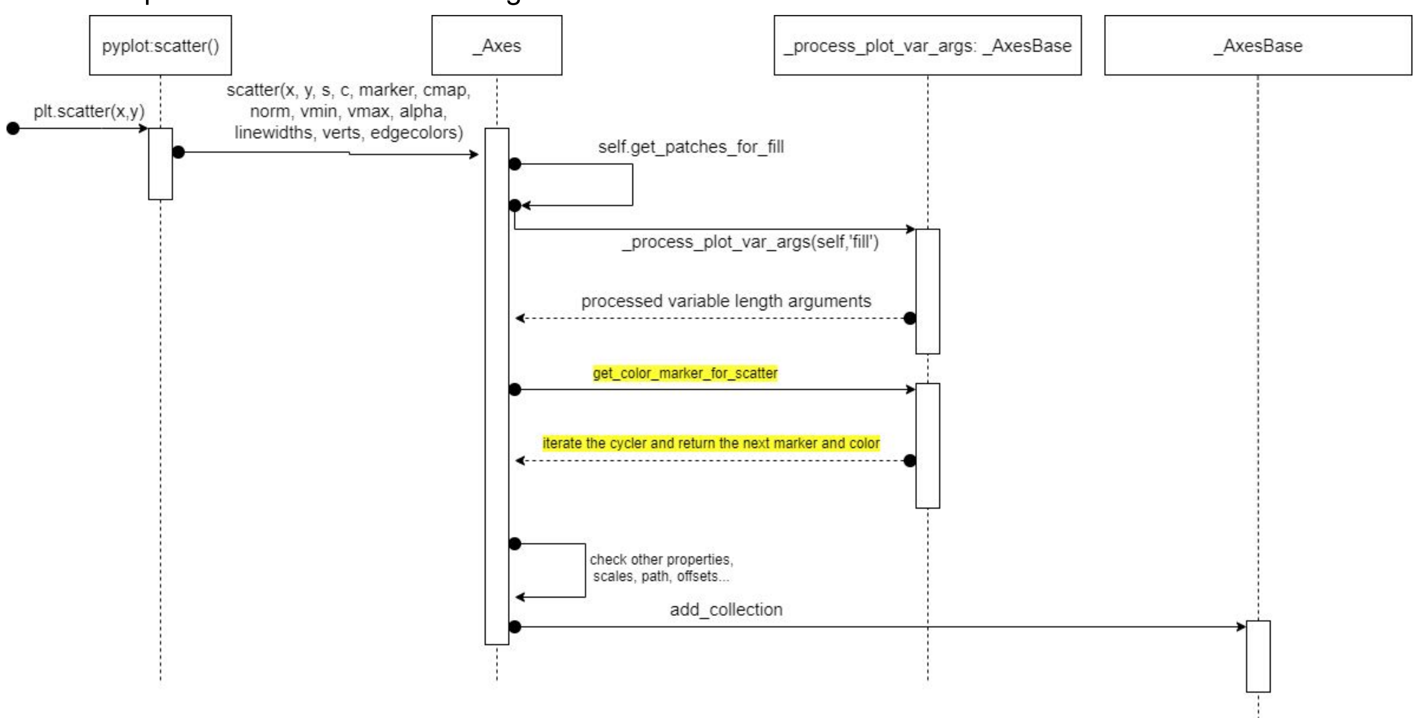
Class UML - after the change


_AxesBase
artists : list
axes
child_axes : list
collections : list
containers : list
lines : list
patches : list
tables : list
texts : list
title : Text
xaxis : XAxis
yaxis : YAxis
...
...
...
add_artist(a)
add_collection(collection, au
add_container(container)
add_image(image)
add_line(line)
add_patch(p)
add_table(tab)
axis()
cla()
get_children()
get_images()
get_legend()
get_lines()
set_prop_cycle()
...
...
...
Axes
aname
ignore_existing_data_limits : bool
legend_ : Legend, NoneType
...
...
...
bar(x, height, width, bottom)
contour()
errorbar(x, y, yerr, xerr, fmt, ecolor, elinewidth, capsize, barsabove, lolims, uplims, xlolims, xuplims, errorevery, capthick)
fill()
get_title(loc)
get_xlabel()
get_ylabel()
legend()
plot()
plot_date(x, y, fmt, tz, xdate, ydate)
scatter(x, y, s, c, marker, cmap, norm, vmin, vmax, alpha, linewidths, verts, edgecolors)
set_title(label, fontdict, loc, pad)
...
...
...
_process_plot_var_args
axes
command : str
prop_cycler : cycle
get_next_color()
set_lineprops(line)
set_patchprops(fill_poly)
set_prop_cycle()
get_color_marker_for_scatter()

Sequence UML - before the change:



Sequence UML - after the change:



Feature Selection

For the next deliverable, we have decided to work on **Feature 12939.** We believe that this feature will provide an excellent opportunity for the team to work across the multiple layers (scripting, artist, and backend) in the layered architecture of Matplotlib. Likewise, this feature request is deceptively simple and requires our team to handle intricate cases of label collision and separation. Moreover, this feature -- despite only being demonstrated with a line graph -- can be extended to additional graphs.

While this feature is seemingly simple, there are numerous factors that must be accounted for when rendering the line labels that make a strong implementation of this feature a challenge. Label positioning in the best possible scenario (see image) may be simple; yet, there will be cases in which a line graph will have lines whose y-values are closer together than the sample image provided. For these cases, our team must choose an appropriate handling mechanism, including shrinking up until a minimum size (if specified) and repositioning.

In addition to this, the extensibility of this feature makes it a great selection -- despite that the feature requirements only indicated line graphs, this feature can be extended to bar graphs, pie charts, and even scatter plots (regression lines). Thus, this increases the overall complexity of this feature request. Overall, our team has agreed that the selection of **Feature 12939** over **Feature 15105** will provide a greater challenge and a better opportunity to explore the codebase and its multi-layered architecture, while being an excellent feature to add on to the Matplotlib library.

Acceptance Tests

Basic Functionality Test:

Step 1:

Generate line graph

Step 2:

Apply Line Labels

Step 3:

Assess all labels are properly colour coordinated and placed to the right of their respective line

.

Non-Interference Test:

Step 1:

Generate line graph displaying the legend

Step 2:

Apply Line Labels

Step 3:

Assess that the legend is not affected by the creation of this new label feature

Same Endpoint Test:

Step 1:

Generate line graph where multiple lines end at the same value

Step 2:

Apply Line Labels

Step 3:

Assess all labels are within the minimum font size parameter as well as, that the labels are nicely formatted at lines which share multiple endpoints

Movement Test:

Step 1:

Generate line graph where multiple lines end at the same value

Step 2:

Apply Line Labels

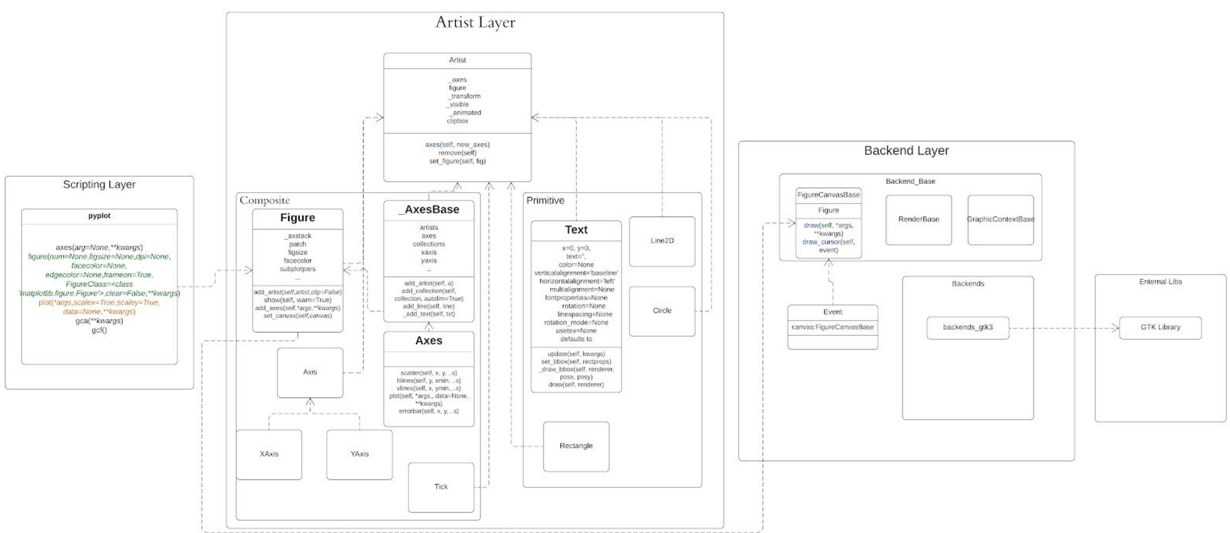
Step 3:

Move the chart to ensure that when the window is affected, the labels move accordingly.

Higher level diagram of matplotlib

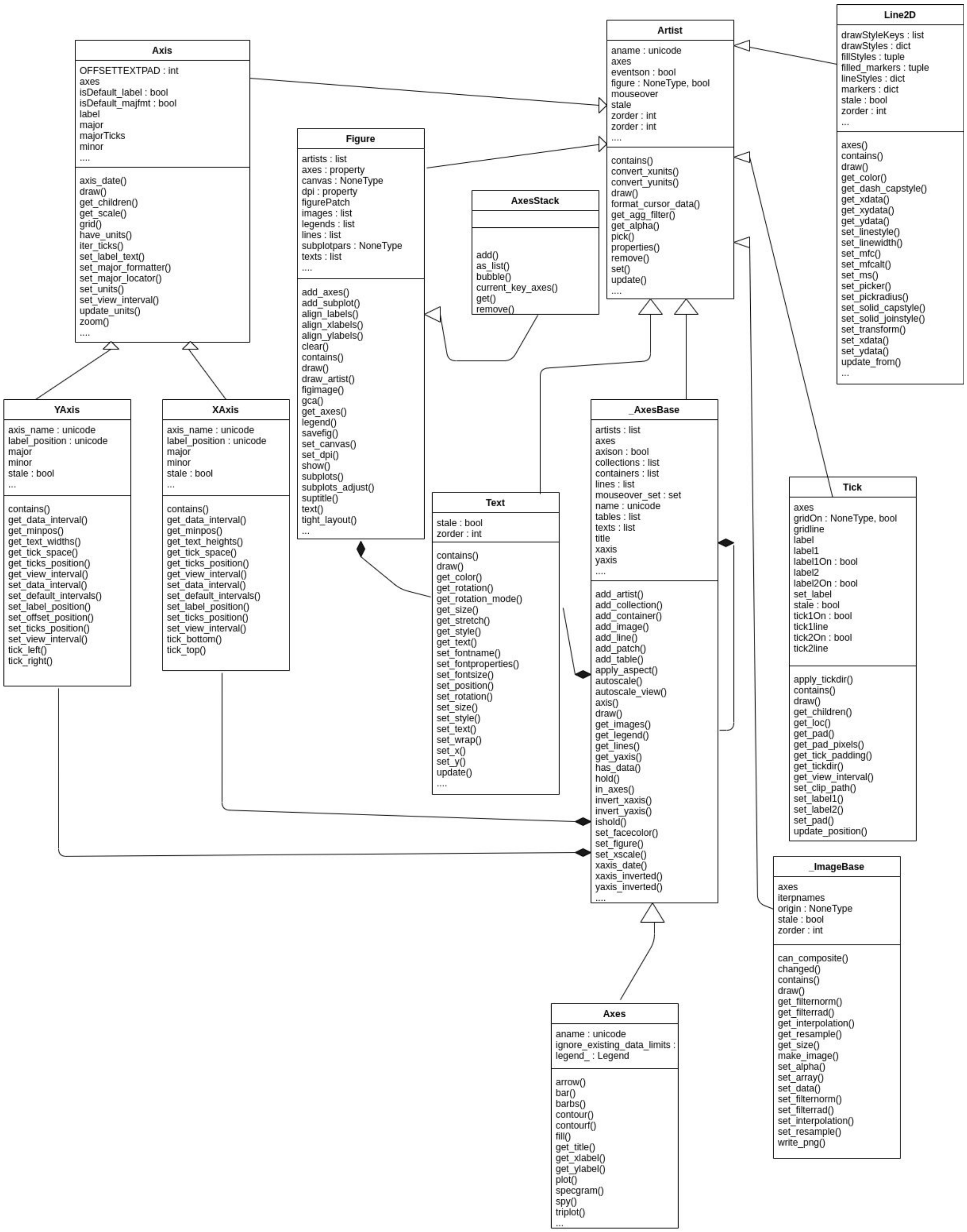
1. Three Layer Architecture of matplotlib

   Matplotlib uses a closed layers architecture design. Script layer acts as a wrapper of Artist layer and parses user commands to corresponding classes in Artist layer, Artist layer updates states based on user's command, when graphical output is requested, Artist layer then sends corresponding commands to Backend layer. Backend layer will parse the command and produce corresponding graphics output.
   Since layers can only interact with the layer below them, it's a closed layer architecture.

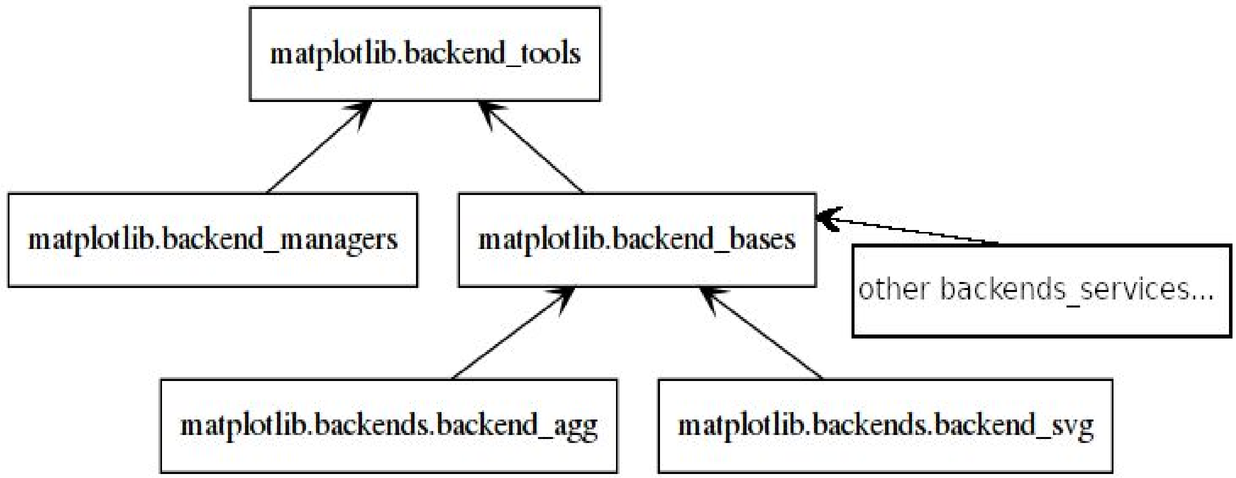


(Original size)

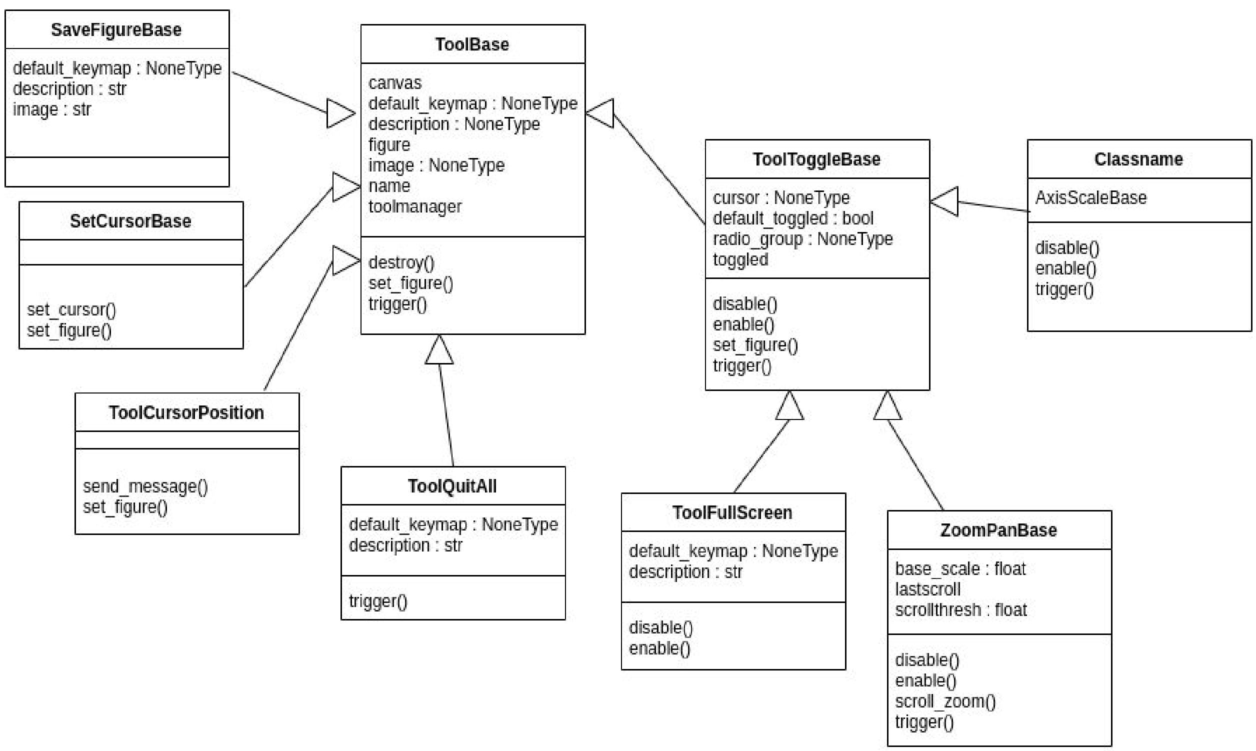
Generated and simplified UML for Artist layer:



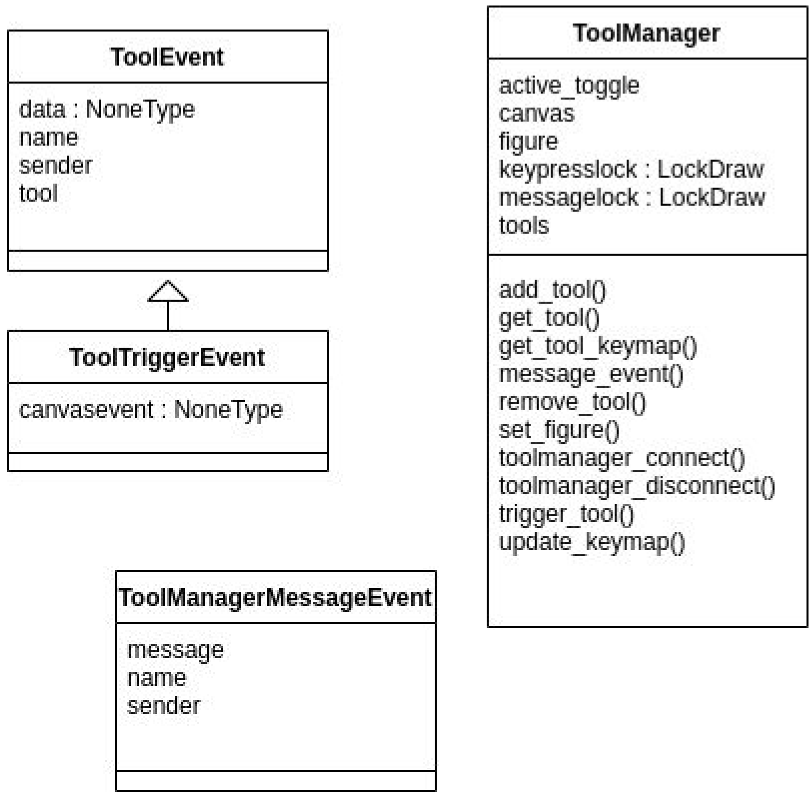
Generated and simplified Component UML for Backend Layer:



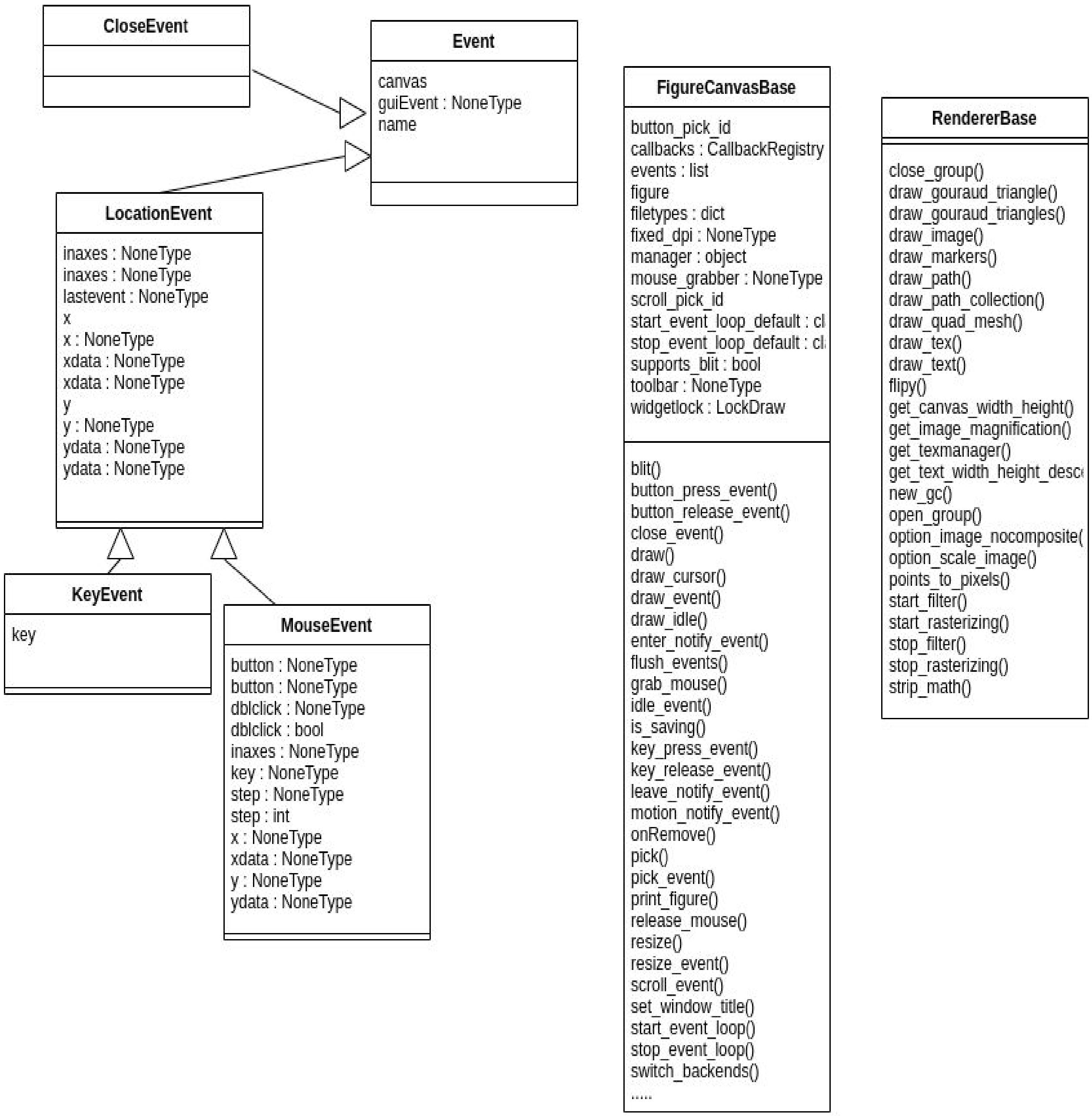
Generated and Simplified UML for backend_tools:



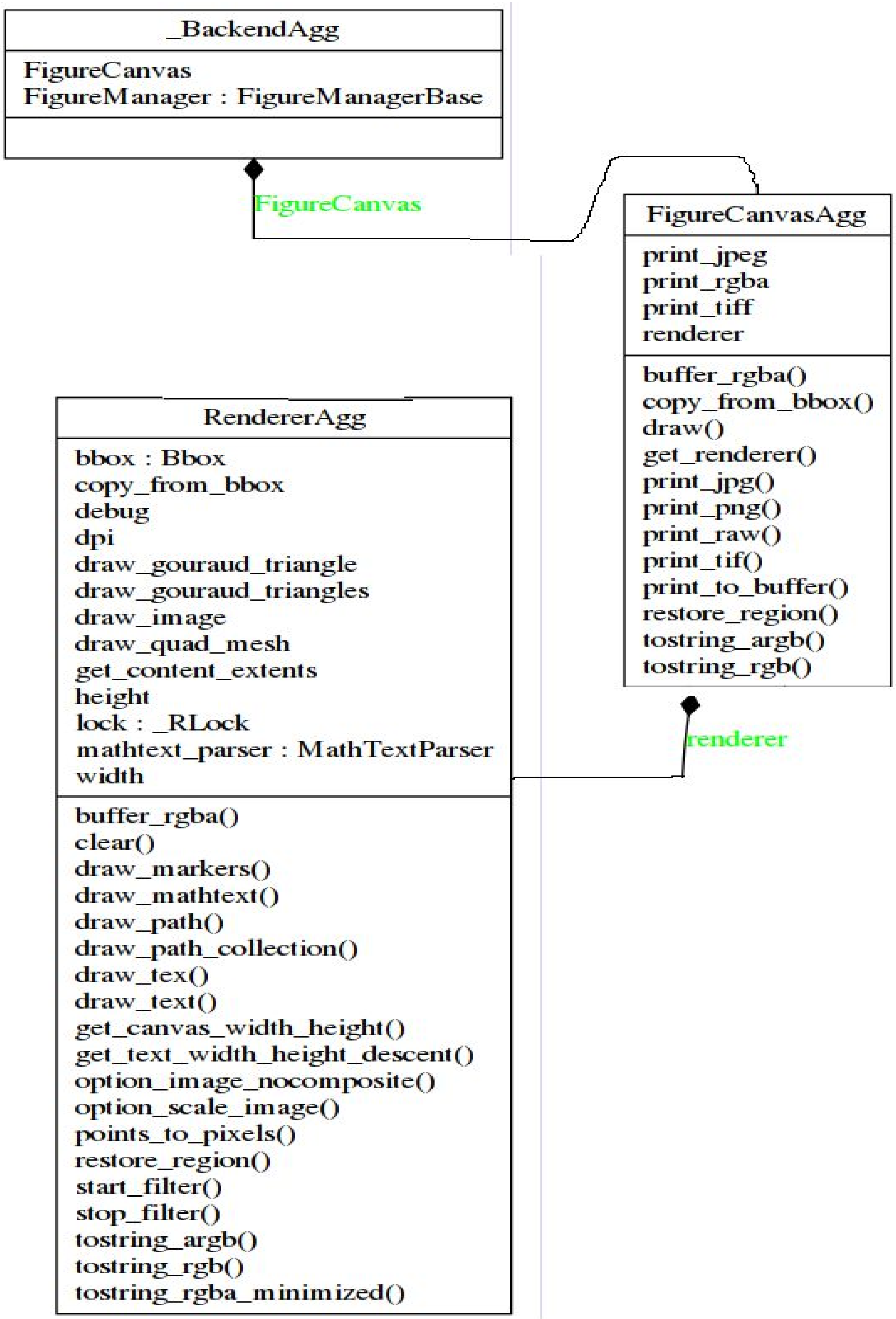
Generated and Simplified UML for backend_managers:



Generated and Simplified UML for backend_bases:

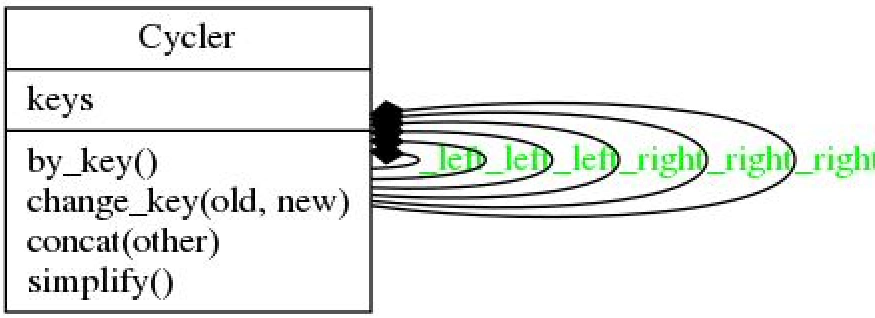


Generated and Simplified UML for Backend Layer AGG implementation:



Design Patterns of Matplotlib:

Iterator



Matplotlib uses Cycler to cycle through properties and apply them when plotting graphs.
Cycler can be created for infinite cycling or finite cycling. Infinite cycling will start from the first time again after iterating the last item, whereas finite cycling will stop once iterating the last item.

Observer design pattern

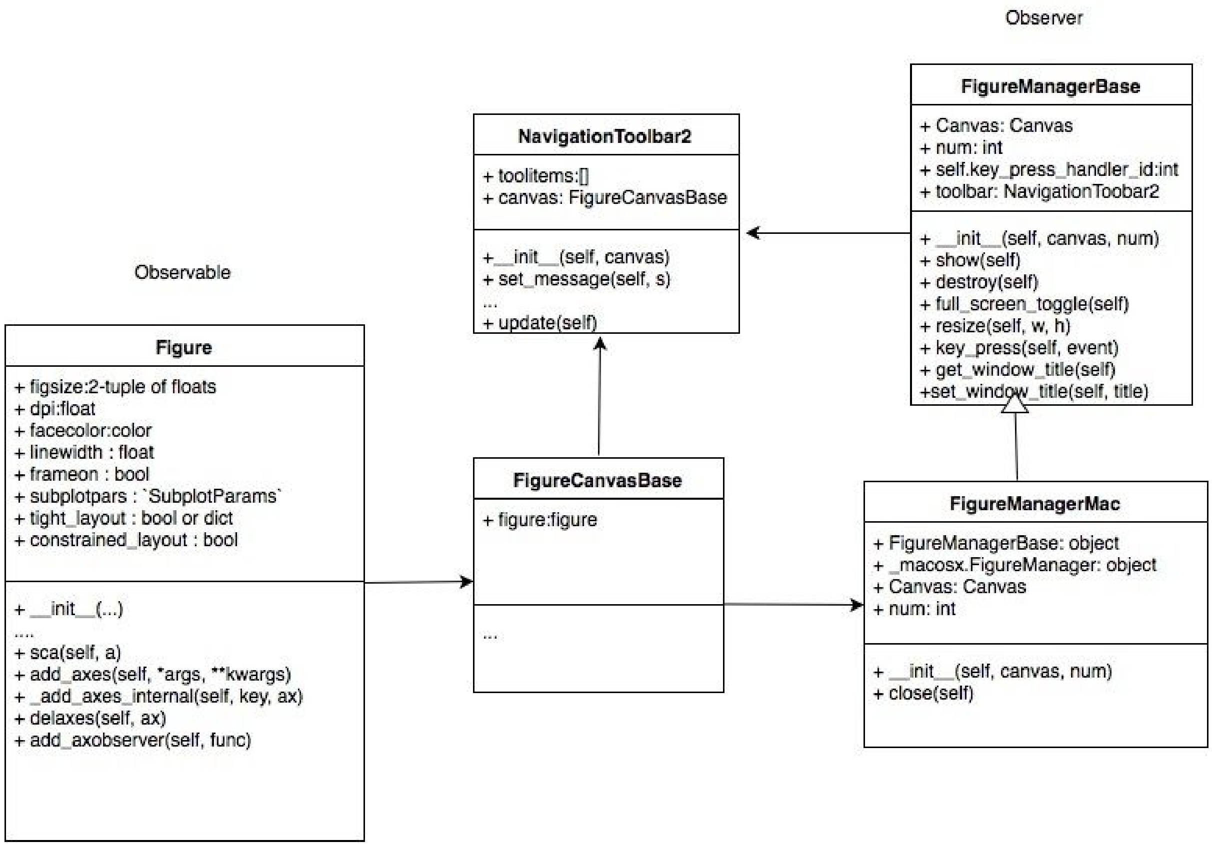
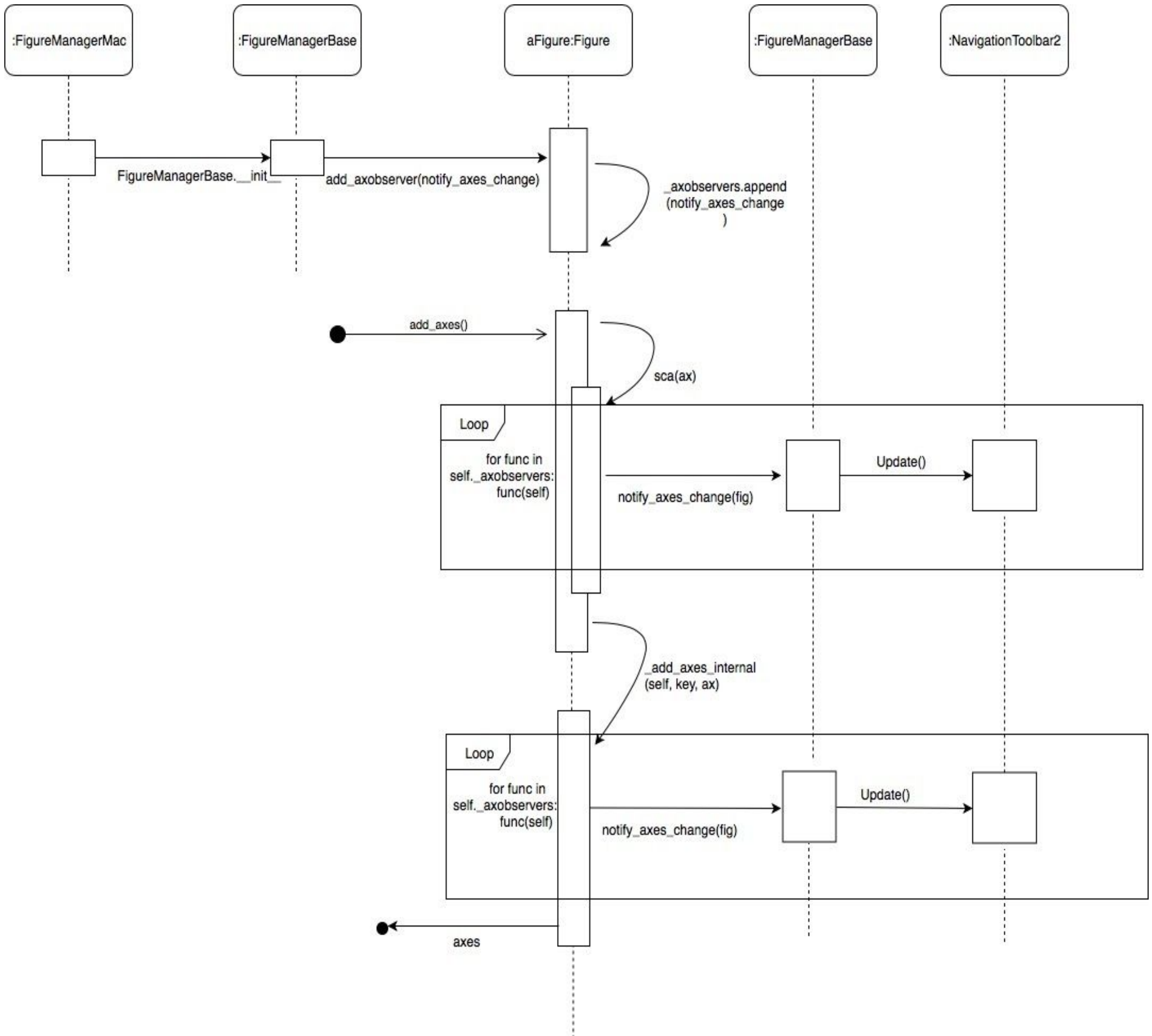


Figure Object implements observer design pattern by setting up a list of observers and notifies the registered observers when changes in Figure's axes happens _axobserver field stores a list of functions, figure's axes notify its observers by running all functions listed in _axobserver field



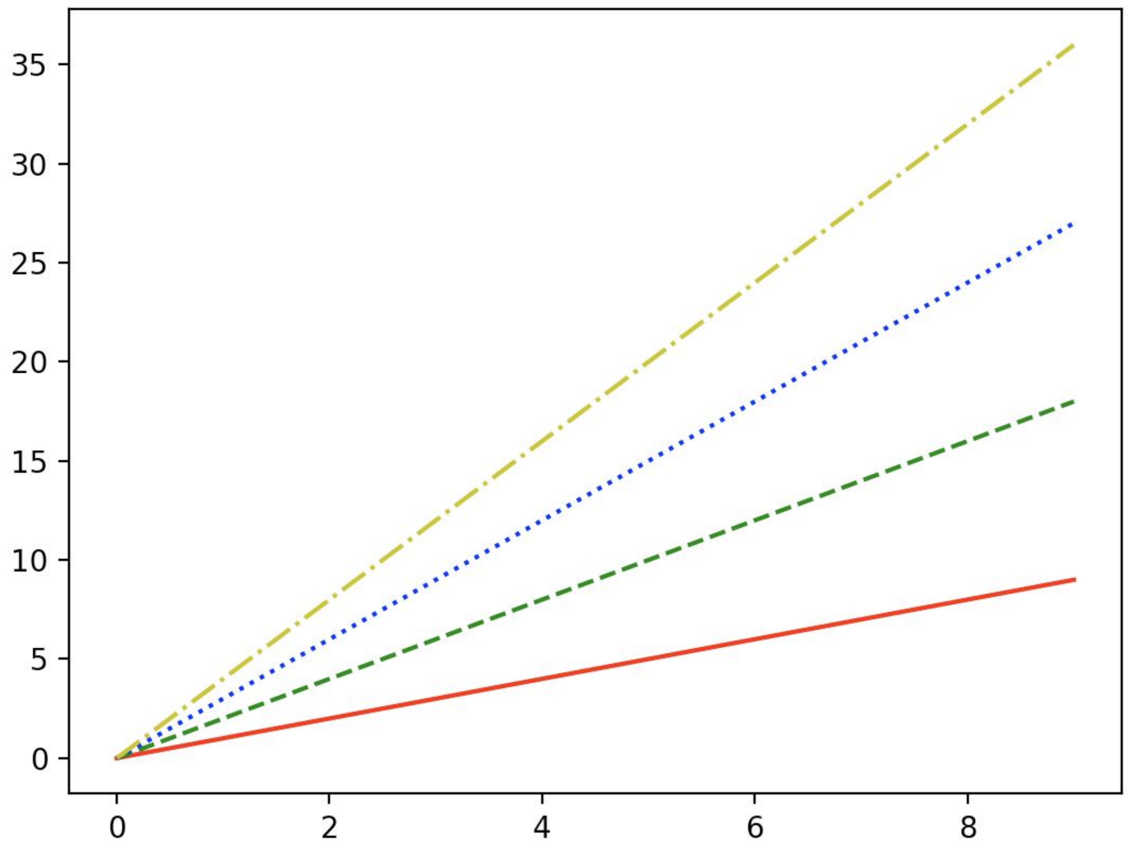
Interesting Solutions

Matplotlib is passing a `Cycler` object to `props_cycle`.

For example

```
default_cycler = (cycler(color=['r', 'g', 'b', 'y']) +
                  cycler(linestyle=['-', '--', ':', '-.']))
```

```
fig, ax1 = plt.subplots()
ax1.set_prop_cycle(default_cycler)

x = np.arange(10)

plt.plot(x, x)
plt.plot(x, 2 * x)
plt.plot(x, 3 * x)
plt.plot(x, 4 * x)
plt.show()
```



Matplotlib would call the `_getdefaults` function in `_process_plot_var_args` to read the colors and linestyle from `props_cycle`.

```
    def _getdefaults(self, ignore, kw):

        prop_keys = self._prop_keys - ignore
        if any(kw.get(k, None) is None for k in prop_keys):
            default_dict = next(self.prop_cycler).copy()
            for p in ignore:
                default_dict.pop(p, None)
        else:
            default_dict = {}
        return default_dict
```

The interesting thing is that Matplotlib converts a Cycler object into an iterable Cycler object by calling `self.prop_cycler = itertools.cycle(prop_cycler)`, then it would work in the same way as Iterator. Furthermore, Cycler object has dictionary functionality, it links a key to values, calling `next` would move the pointer to the next value in all keys.

```
For example:
default_cycler = (cycler(color=['r', 'g', 'b', 'y']) +
                  cycler(linestyle=['-', '--', ':', '-.']))

iterator = itertools.cycle(default_cycler)
print(next(iterator)) #{'color': 'r', 'linestyle': '-'}
print(next(iterator)) #{'color': 'g', 'linestyle': '--'}
print(next(iterator)) #{'color': 'b', 'linestyle': ':'}
print(next(iterator)) #{'color': 'y', 'linestyle': '-.'}
```

Using Cycler in this case would be the best solution. Without Cycler, Matplotlib must create an iterator for each props in props_cycle cycle and call `next`method for each iterator to move the pointer to ensure all iterators are in the correct position.