The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

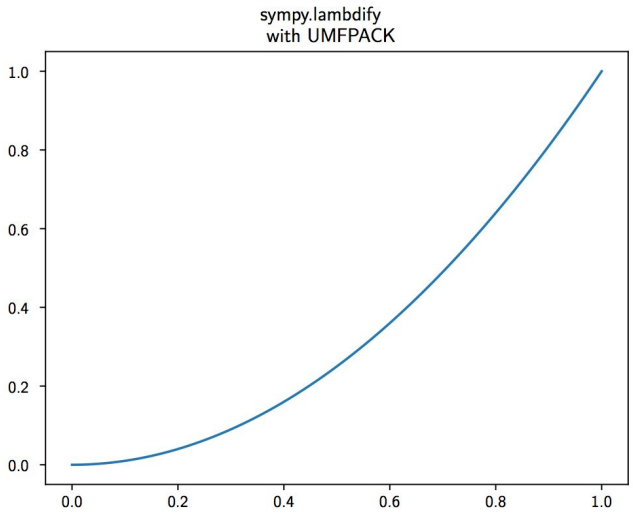
Issue #9747

https://github.com/matplotlib/matplotlib/issues/9747

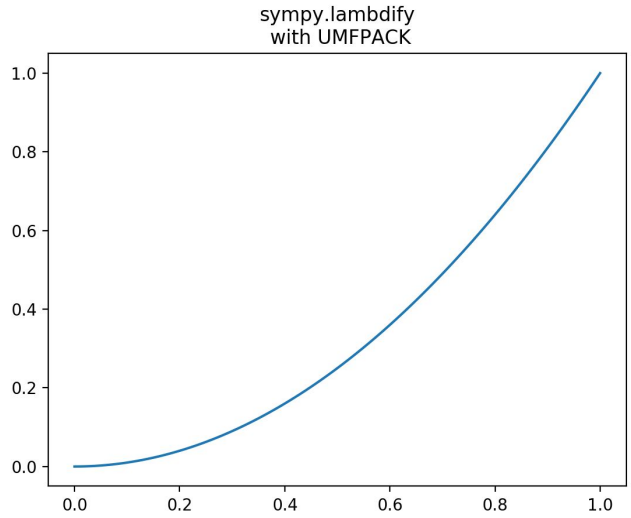
Issue description:

When saving to a PGF file, sometimes the title isn't centred properly, but with the format PNG, it seems to centre fine.

Here is an example of PGF



Here is an example of PNG



The problem is specific to titles with multiple lines, in the above example, the title looks like this in code

plt.title('sympy.lambdify\nwith UMFPACK')

When a single line title is created, it remains a text object, and retains layout information. However when multiple lines are created, they are no longer text objects, and instead are just passed as strings. And the text rendering functions can't handle multi-line text objects, so they are just oriented in based on default, aka left-justified.

Another observation is that the location of the title is based on font. The position is calculated using one font, but the render is done using another font. This is because one-line text and multiline text are handled differently. For a single-line, draw_text passes a Text object to the renderer, and that object retains all the layout information. PGF is clever and uses this.
For two lines however, the single text object doesn't work on most renderers, so it passes two separate lines to the renderer, and strips away the helpful Text object. The position it passes the renderer should be correct, but it's based on the font it was supplied with, and it loses any alignment information.

Code Base Modifications + Analysis:

Looking inside *pyplot.py*, the *title* method is just a wrapper for *Axes.set_title*, so the solution has to start there. The title is actually a *Text* object stored in the *Axes* (3 *Text* objects are stored for title, one for left, center, and right justified). Mainly, what *Axes.set_title* does is get the right *Text* object for the specified alignment (defaults to center if not specified), and sets the text inside that *Text* object.

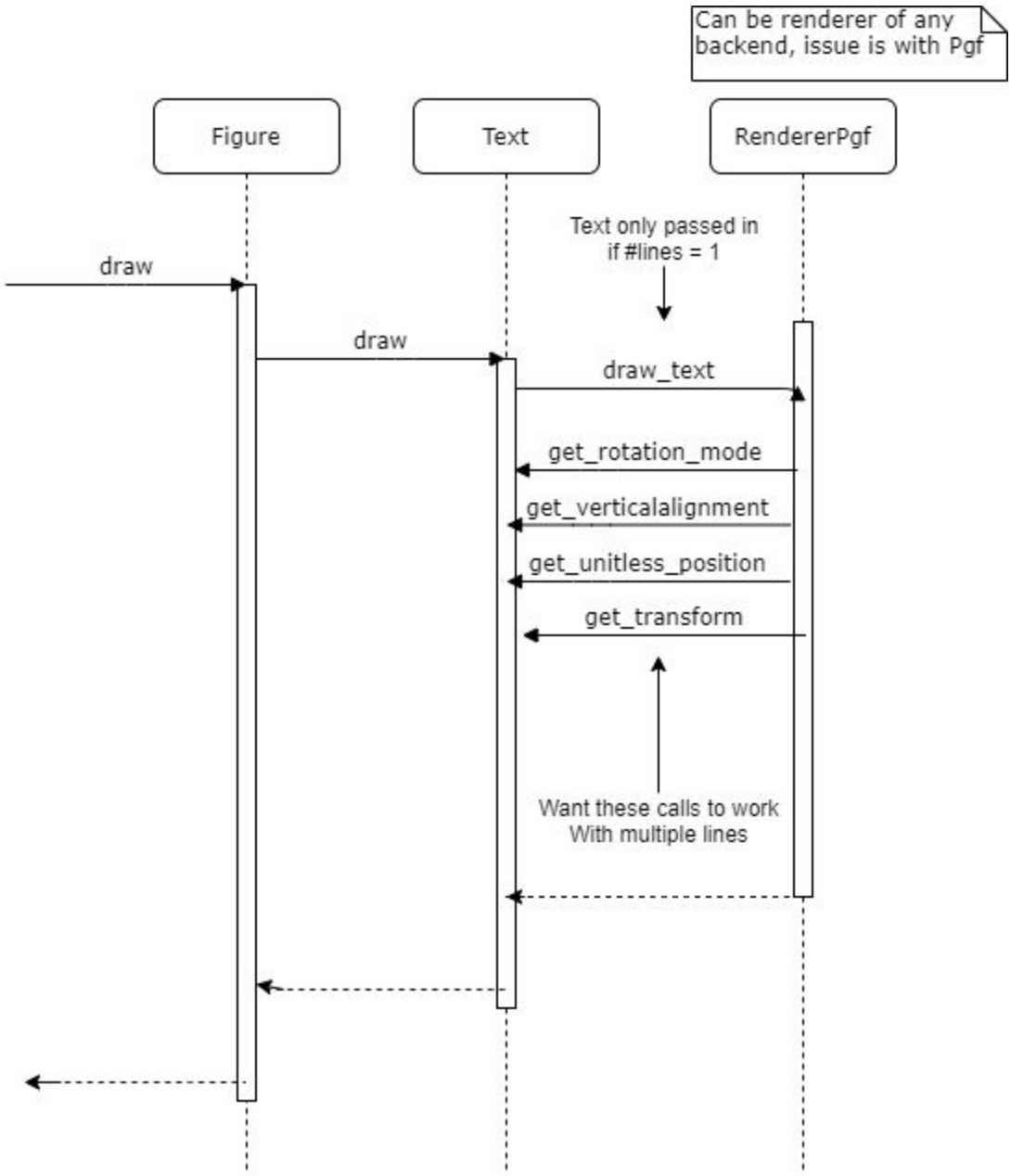
In the example for the issue, an alignment is not specified, so *Axes* will use *self.title*, the center aligned title *Text* object. When the *Text* object's *draw* method is invoked, it will use the renderer supplied to it, and not create its own (which it is able to do). The most important part of this draw method is the following line

mtext = textobj if len(info) == 1 else None

The length of *info* is the number of lines in the string to display, and if it is more than 1, *None* is used instead of an existing textobj to display the text. This is important because inside *RendererPgf.draw_text*, the method does different logic depending on whether or not *mtext* is *None*.

The parts of the codebase that would change would be that the *Text* object needs to be passed in regardless of the number of lines of text. However, the *Text* object would also need to be able

to properly align multiple lines. This means that likely the changes made would be local to the *Text* object. This does not imply the fix is easy, just that it is relatively local.

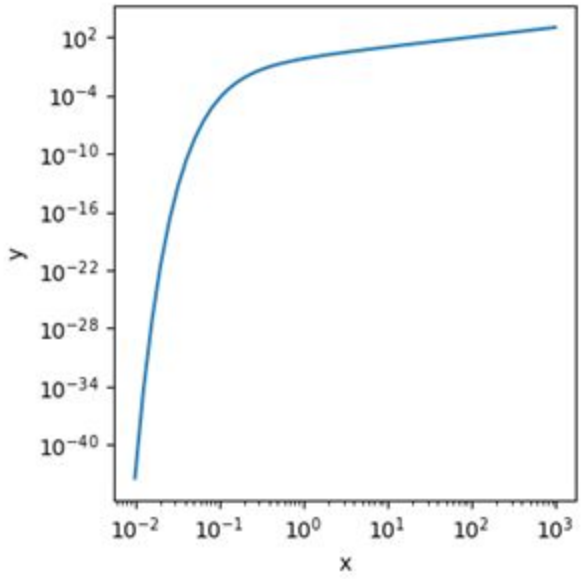


The calls from *RendererPgf* to *Text* is what happens when a *Text* object is passed in to the renderer. The issue is in passing this *Text* object even when the number of lines is more than 1, and also having these calls return proper results even for multiline text.

The Quick and the Studious

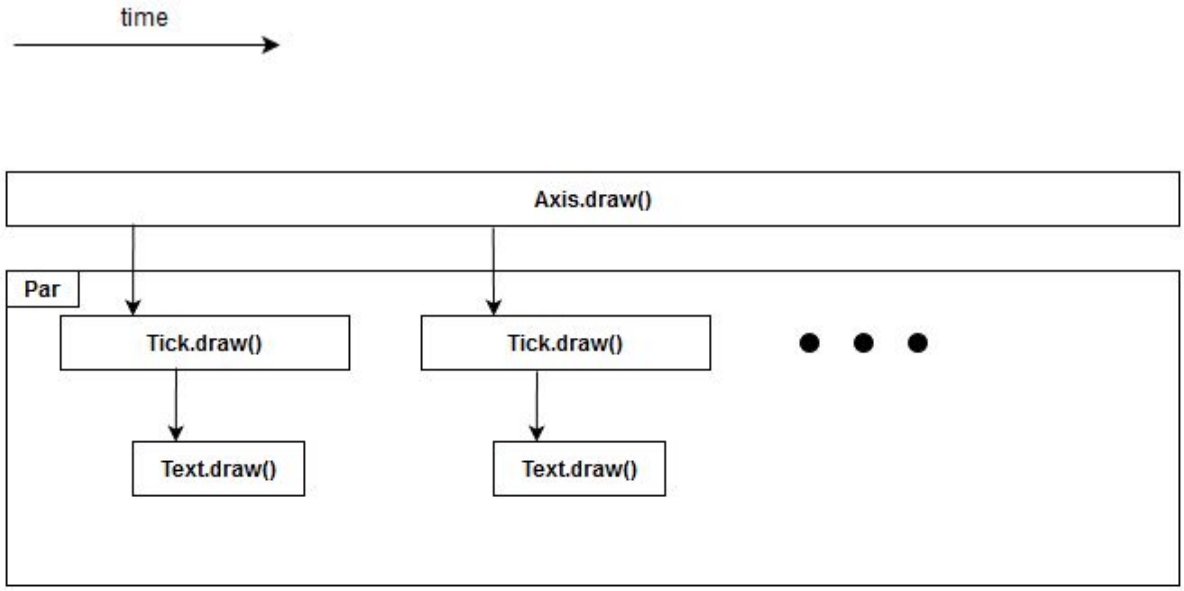
Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

Issue #13440 (We are picking this issue for D4)
https://github.com/matplotlib/matplotlib/issues/13440
Issue description:
When setting tick labels on an x or y-axis, there is currently no easy way to have the text align towards the left for y-axis or align towards the bottom for x-axis. Specifically, the user wants the text labels of the y-axis to align towards the left. When the user does **ax.yaxis.set_tick_params(horizontalalignment='left')**, the code crashes because setting horizontal alignment through **set_tick_params()** is no longer a supported method. This option had been removed some versions ago because setting horizontal alignment like this did not result in reasonable outputs. As can be seen in the example below, the user wants each of the labels on the y-axis to align left, but since each tick label is rendered individually without knowing the existence of their neighbours, implementing this will require the labels to somehow communicate with one another or at least when they are being rendered.



Designs
Currently, the way that drawing works in matplotlib is that it is a series of cascading **draw()** method calls that start at the top level artist (for simplicity this will be our axis class, we will leave out the **Figure** which calls **draw()** on each axis because the Figure is unrelated to this issue). In the simplified UML diagram below, an axis' draw event is triggered by a call to its **draw()** method. And in turn, it calls its own internal methods to find all **Tick** locations, which is followed by a subsequent call to each of those tick's **draw()** method. Finally, each tick may be associated with one or more labels (which are **Text** objects) which has its own **draw()** method that will be called so that the renderer can draw it on to the display.

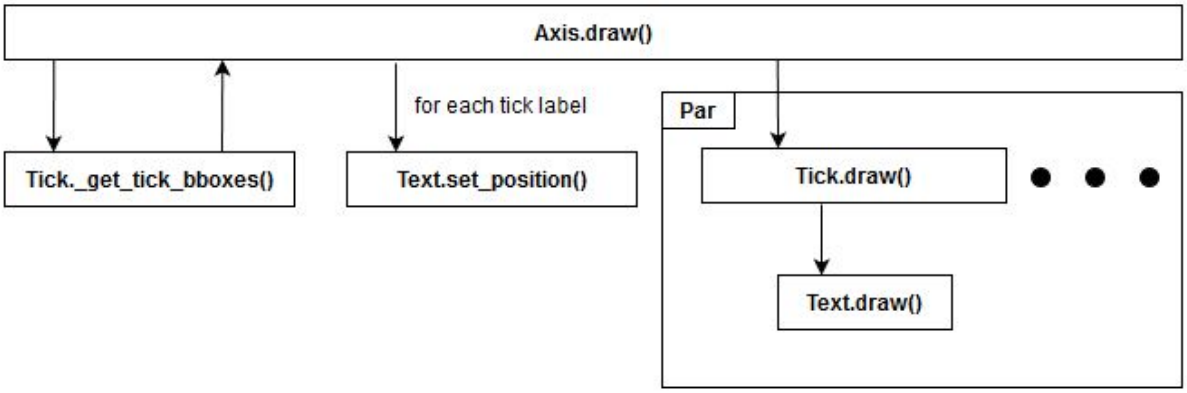
In this setup, each **Tick** label does not know of the existence of other tick labels, which means that aligning them to the left as it currently stands is impossible.



To remedy this, we first recognize that each tick label is just a **Text** object, and that all text objects have a bounding box property (bbox for short) which represent a rectangle used to wrap the text. The bbox property is useful to us because it contains a width property that we can leverage to our advantage. In short, the tick label with the longest width will be the furthest left text that appears on the screen, so we just need to obtain this width and compute the offset necessary to align all other tick labels to that position. Luckily for us, there is already an internal function in the **Axis** class called **_get_tick_bboxes()**, which returns 2 lists of tick labels bboxes associated with that axis. We will call on this method when the axis' **draw()** method is called before we tell each **Tick** to draw themselves, this will allow us time to set the tick label positions. A simplified UML sequence diagram can be seen below.

The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu



To keep the user experience consistent, we will re-implement this via an api call to **Axis.set_tick_params(horizontalalignment="left")** as wanted by the user initially. When **set_tick_params()** is called on an axis object, a set of kwargs are used to update an internal dictionary variable called either **_major_tick_kw** or **_minor_tick_kw**. We will utilize this to set an internal variable called **_horizontalalignment** in the **Tick** class object, so that when draw is called on each **Tick**, we can pass in the individual offset necessary for each **Tick** and as well if it is horizontal aligned to the left, then we shift it to the left by the offset.

Why we picked issue #13440

We think we are in a good position to tackle this issue because for our last deliverable, we fixed two bugs each dealing with **Text** objects and the other dealing with **Draw** methods in various backends. When reviewing the code for the **Axis** class, we noticed much similarity between how it is drawn compared to how a **canvas** is drawn by various backends. And because we have worked with the draw methods of various non-interactive backends with the last deliverable, we believe that we are well equipped to work on this issue which is similar in nature to the two bugs we worked on prior.

Expected timeline

Creating test cases (2-3 days)
Exploring solutions (5-7 days)
Testing/Debugging (5-7 days)
Documentation (1 day)

Acceptance Tests

This assumes one of the following conditions is true:

- The issue has been successfully merged into matplotlib, and user has installed the package
- User clones the fork, switches to appropriate branch, and sets up development environment as per:

The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

- https://matplotlib.org/3.1.3/devel/contributing.html#installing-matplotlib-in-developer-mode

The purpose of these tests is to verify that the call to our implementation of **Axis.set_tick_params()** is behaving as we expect it to across all scenarios. The major cases of concern are if the y-axis labels are displayed correctly when we set their *horizontalalignment*(to 'left') and if the x-axis labels are displayed correctly when we set their *verticalalignment*(to 'bottom'). While *horizontalalignment* and *verticalalignment* have other values (right and top, respectively), these are the default cases and it is how they are shown currently when their values are not set, so we are confident that these work as expected and do not need testing for this particular fix. The tests below show 2 graphs with different alignment settings for the labels on the x and y axes, to demonstrate that our implementation works correctly across the major cases.

Currently, the labels between the ticks cannot be centered. The ticks start at the beginning of the x-axis instead of starting at where the first x-label is. As a workaround, one can place a label on the minor ticks in between each major tick, and hide the major tick labels and minor ticks. Ever since tick_params came out, there have been cases where some properties are overwritten by a more general axis setting.

Test1:

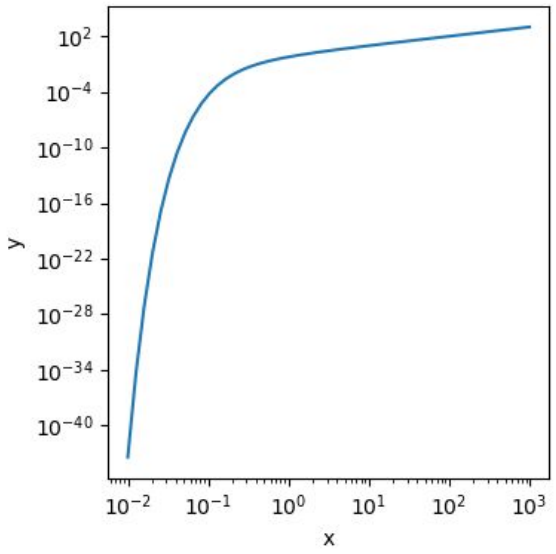
1. Import matplotlib.pyplot
2. Get x and y values through whatever means (can be a function, can be hardcoded)
3. Create a figure (eg. plt.figure(1, figsize=(3,3))
4. Add a x label
5. Add a y label
6. Set text labels of the y-axis to align towards the right. Ex)
7. Will see that it will be so

```
import matplotlib.pyplot as plt
import numpy as np
x = np.logspace(-2, 3)
y = 3 / (exp(3/x) -1)
fig = plt.figure(1, figsize=(4,4))
plt.loglog(x,y)
plt.xlabel("X")
plt.ylabel("Y")
plt.tight_layout()
ax = plt.gca()
ax.yaxis.set_tick_params(horizontalalignment='left')
```

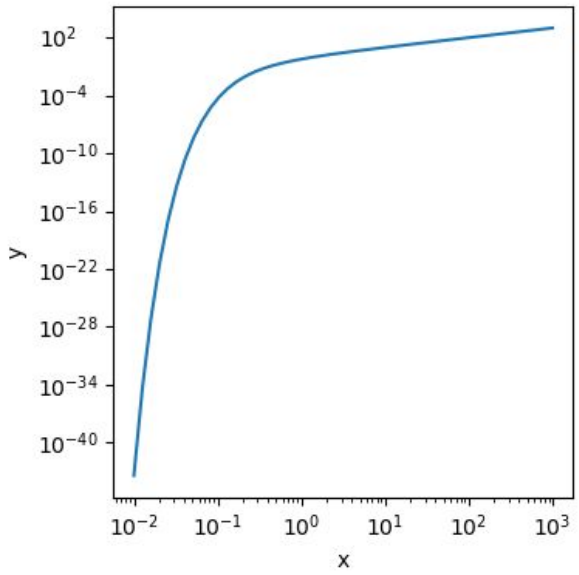
The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

Actual (with left alignment):



Expected (with left alignment):



The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

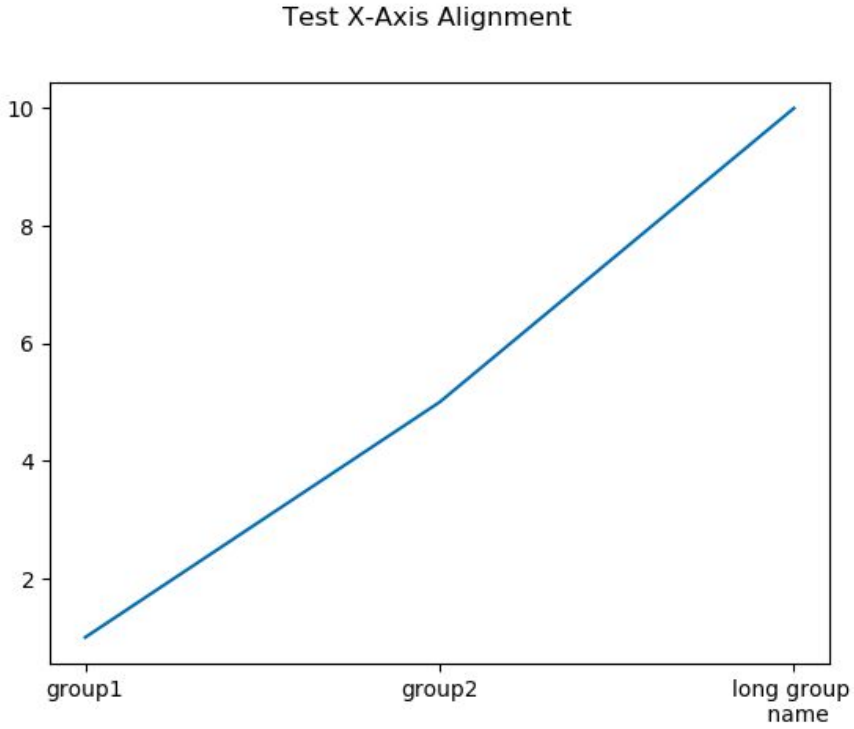
Test2: X-axis

To test that the *verticalalignment* works correctly for the x-axis labels, a user can take the following steps:

1. Create a new figure with any arbitrary data
2. Set (at least) one of the labels on the x-axis to have a name that spans longer than one line. For simplicity, this can be done by using a newline character
3. Call the implemented **Axis.set_tick_params()** on the x-axis, with the argument *verticalalignment='bottom'*
4. Verify that the figure that is generated when the plot is shown has the labels on the x-axis bottom aligned. Reference the figures below to see what the expected alignment should look like

Actual (with bottom alignment):

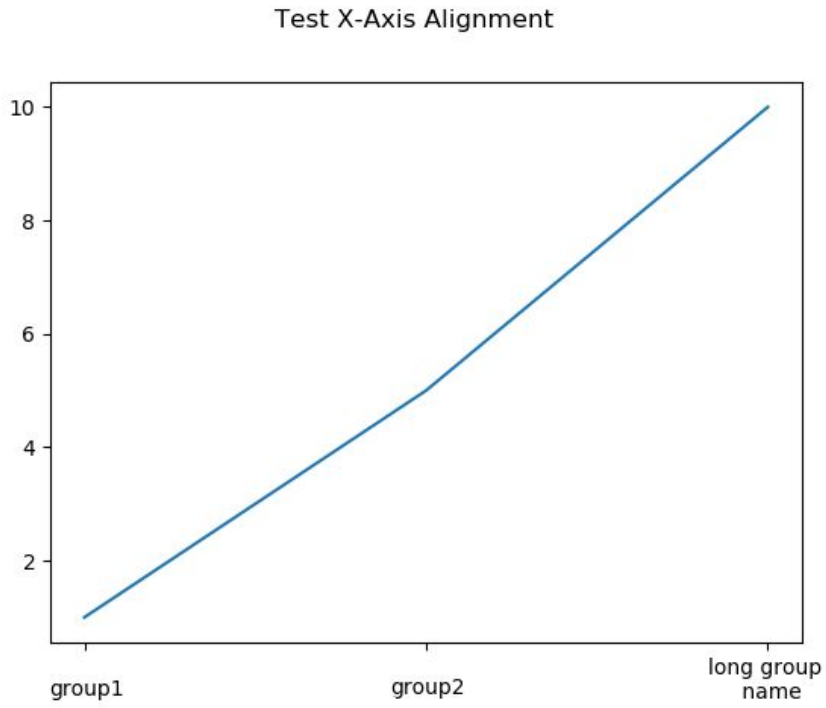
This is an example output of a figure that is generated when the x-axis label is bottom aligned on the current matplotlib code. Notice how all labels are aligned incorrectly towards the top of the x-axis



The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

Expected (with bottom alignment):
This figure was produced with the same code as the one above, notice how the labels are all aligned at the bottom



Here is the code that was used to produce the figures above. These can be used as samples for acceptance testing.

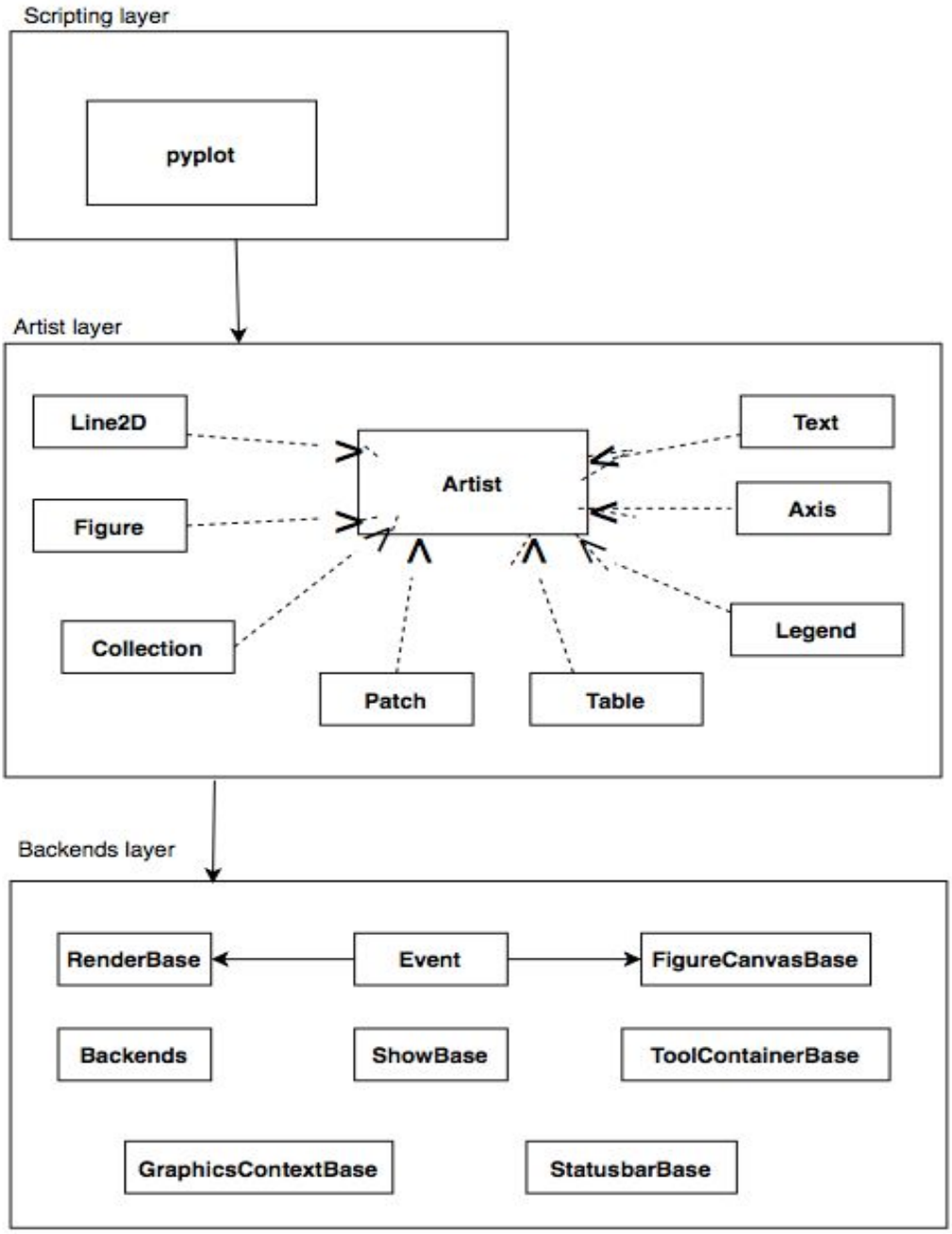
```
import matplotlib.pyplot as plt

names = ['group1', 'group2', 'long group \n name']
values = [1, 5, 10]

plt.figure()
plt.plot(names, values)
plt.suptitle('Test X-Axis Alignment')

ax = plt.gca()
ax.xaxis.set_tick_params(verticalalignment='bottom')

plt.show()
```

The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

Overall Architecture

Matplotlib is a 3-tiered closed layered architecture, meaning that it consists of layers of computation (i.e. presentation layer, interactive layer, backend layer). These layers provide the scalability advantage because each layer can be scaled independently (since each layer is separated). This independence means that there is a low degree of *coupling* between layers. Below is a high-level overview of Matplotlib:



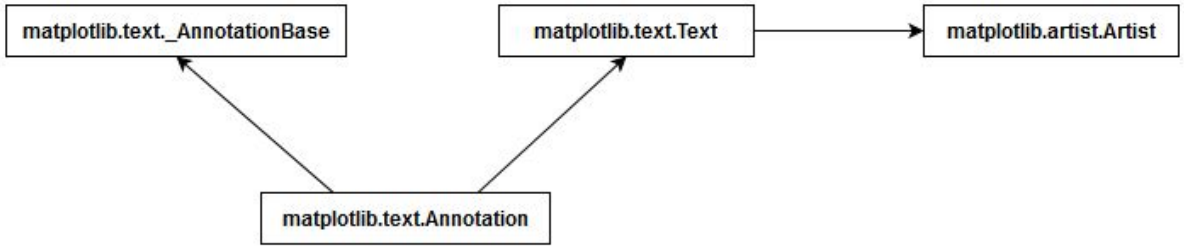
The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

Presentation Layer

The presentation layer is represented by the Artist class. This class processes data and manages all the drawing, styling, and plotting via an object-oriented API. In general, all visible elements that can be seen on a plot is a subclass of Artist. The Artist class itself is an abstract class and contains most of the functionalities that come with any visible object created by Matplotlib. All subclasses of Artist have a draw() method defined, this allows the Renderer to then draw that Artist on the Canvas (the canvas is where objects are drawn on the screen).

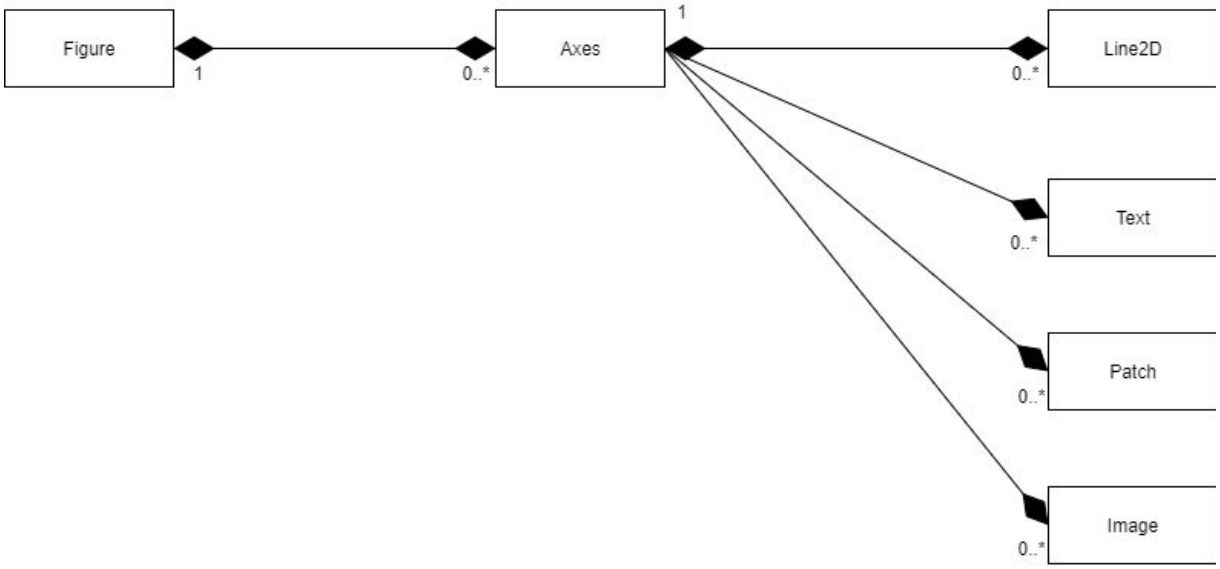
Artist objects can be grouped into two main types, primitives and containers. A primitive type Artist are objects that we actually draw such as the lines (Line2D class), rectangles (Rectangle class), and text (Text class). A container type Artist is the area where a user draws other objects such as the x and y axis of a graph (Axes class) and figures (Figure class). Typically, a user will not directly create an Artist object. Instead, it is the Axes object which acts as a wrapper for plotting functions (ex. plot(), text(), hist(), imshow()) and will return the common primitives (ex. Line2D, Text, Rectangle, Image). Seen in the UML diagram below is an example of how a user would create an annotation on a plot. An annotation object is created which is a subclass of the Text object, which in turn is a subclass of Artist. Once an instance of an Artist object has been created, a user can freely modify its properties. This is usually done by directly modifying the primitive itself (i.e. by changing its color and width), or by changing the property of the container (i.e. changing the axis scale from linear to logarithmic).



For instance, a user would create a Figure through pyplot's APIs and proceed to add subplots to that figure by defining axis in the figure, the number of columns or rows in the figure, or by drawing lines in the figure. The relationship of the Figure object to other Artist objects can be seen in the UML diagram below. Every class shown on the diagram is a subclass of Artist. Recall that a Figure is a container type of Artist object where you can draw other Artist objects into. In this case, a user would define the axes of this figure and then proceed to add in lines, text, or images as wanted. A Figure is then just an aggregation of Axes, which itself is a container that is responsible for determining the shape, background, and border of the items within. Some of these items as seen in the diagram below are lines, text, patch, and images.

In general, when a container type Artist is called upon to draw itself, it will also call each sub-artists that it contains to also draw themselves. This will trigger a cascading series of draw events where each Artist is responsible for drawing themselves and then calling on any sub-artist to do the same.

It is worth noting that the classes in this layer have a high level of cohesion. Each object has a clear purpose and does only what it was designed to do. This design has the advantage that each object is highly reusable and that changing any one component requires relatively few changes in others.



Link to matplotlib documentation on Artist class
https://matplotlib.org/3.1.3/tutorials/intermediate/artists.html

Interactive Layer

The interactive layer is constituted largely by the PyPlot module. The PyPlot module in matplotlib is an interactive command-line style way of drawing plots. This is the ideal module to use to draw simple plots programmatically. However, this is achieved by abstracting away a lot of the Presentation Layer which means that it is not as flexible as the object-oriented API provided by the Artist class.

Similar to what a user would do in the object-oriented API, the PyPlot module provides simplified static methods for users to create a figure with the pyplot.plot() function. A user would then modify that plot by subsequently calling other static methods such as pyplot.axis() to change the numbers displayed on the xy-axis, or pyplot.xlabel() to change the x-axis label in the plot. It is worth noting that this module is not recommended for complex plots

Link to matplotlib documentation on the PyPlot module
https://matplotlib.org/3.1.3/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py

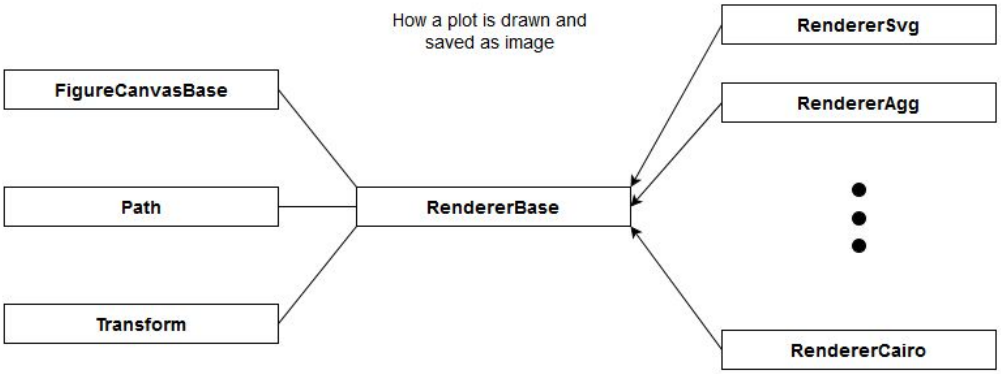
Backend Layer

The backend layer is composed of everything in the matplotlib/backends directory, the backend_bases, backend_tools, and backend_managers modules. This layer contains all the tools necessary for matplotlib to render an image, save it to a file, or embedded it into another application.

The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

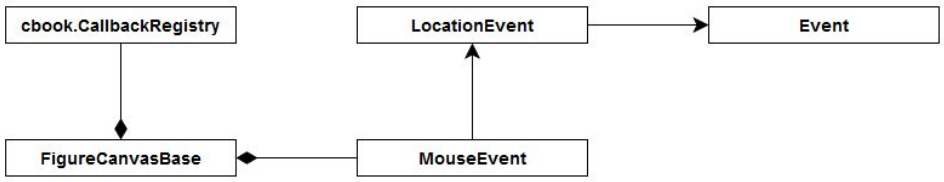
The RendererBase is a base class that defines all the functions for how an image is to be drawn and displayed on the screen. Several of the Renderer classes that inherit from this base class have to do with the image's specific format (e.g. RendererSvg, RendererAgg, RendererCairo). The diagram below shows an example of how the Renderer classes interact with the Artist class to create an image.



The Renderer takes a Path object that denotes the line that is to be drawn on the canvas, a Transform object which represents the geometric transformation that is used to determine the final position of all elements drawn on the canvas, and a FigureCanvas object which is the canvas to which a Figure object would be rendered into.

In addition to writing images to files, the backend also enables matplotlib to be embedded into various other applications. For example, the backends_gtk3 module defines all methods that utilizes the GTK rendering engine to render images based on the Artist objects.

Another major component to the backend is the Event class. In general, an event is an object that is created when a user interacts with matplotlib. Once an event has been created, there will be an appropriate event handler which in most cases is a callback that is there to handle the event. For instance, as seen in the UML diagram below. When a user clicks on the screen, a MouseEvent object will be created in the FigureCanvas class. The FigureCanvas class will first look up the callback function associated with the event that is stored in a CallbackRegistry object. Then, it will attempt to process the event based on the parameters generated by the mouse click (i.e. the location that was clicked and the mouse button that was clicked). In addition to pre-defined events that come with matplotlib, users can also create their own event and connect it to their generated plot.



Link to matplotlib documentation on backend_bases and event handling

The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

https://matplotlib.org/3.1.1/api/backend_bases_api.html
https://matplotlib.org/3.1.1/users/event_handling.html

External libraries used

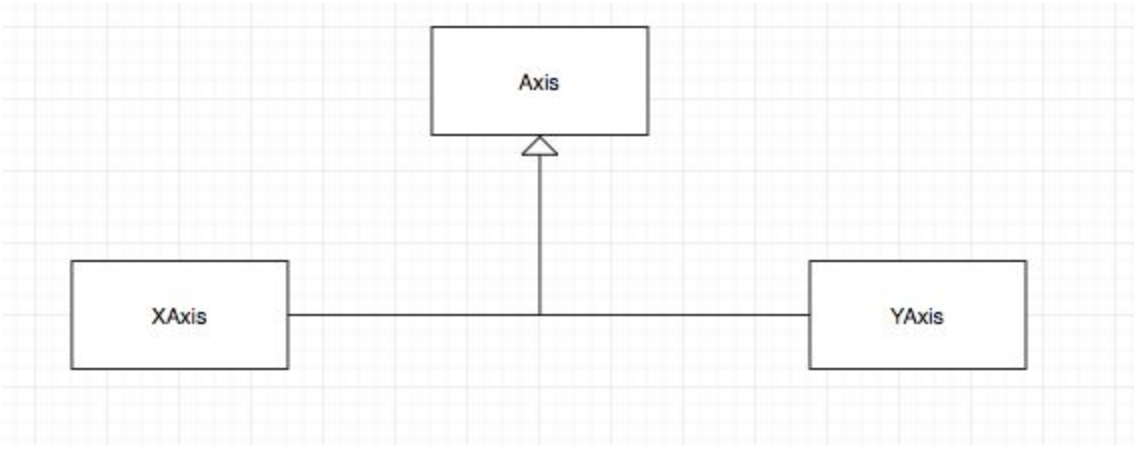
Matplotlib uses the NumPy library to manage the numbers and to perform calculations like converting between polar and cartesian coordinates (seen in /projections/polar.py). For example, it uses the ndarray (n-dimensional array) in NumPy to create and manage the data in a plot as can be seen in Matplotlib.pyplot, Matplotlib.transform, Matplotlib.axis. It also uses NumPy.arange() often to create evenly spaced intervals which is particularly useful for creating evenly spaced tick marks on a graph's axis.

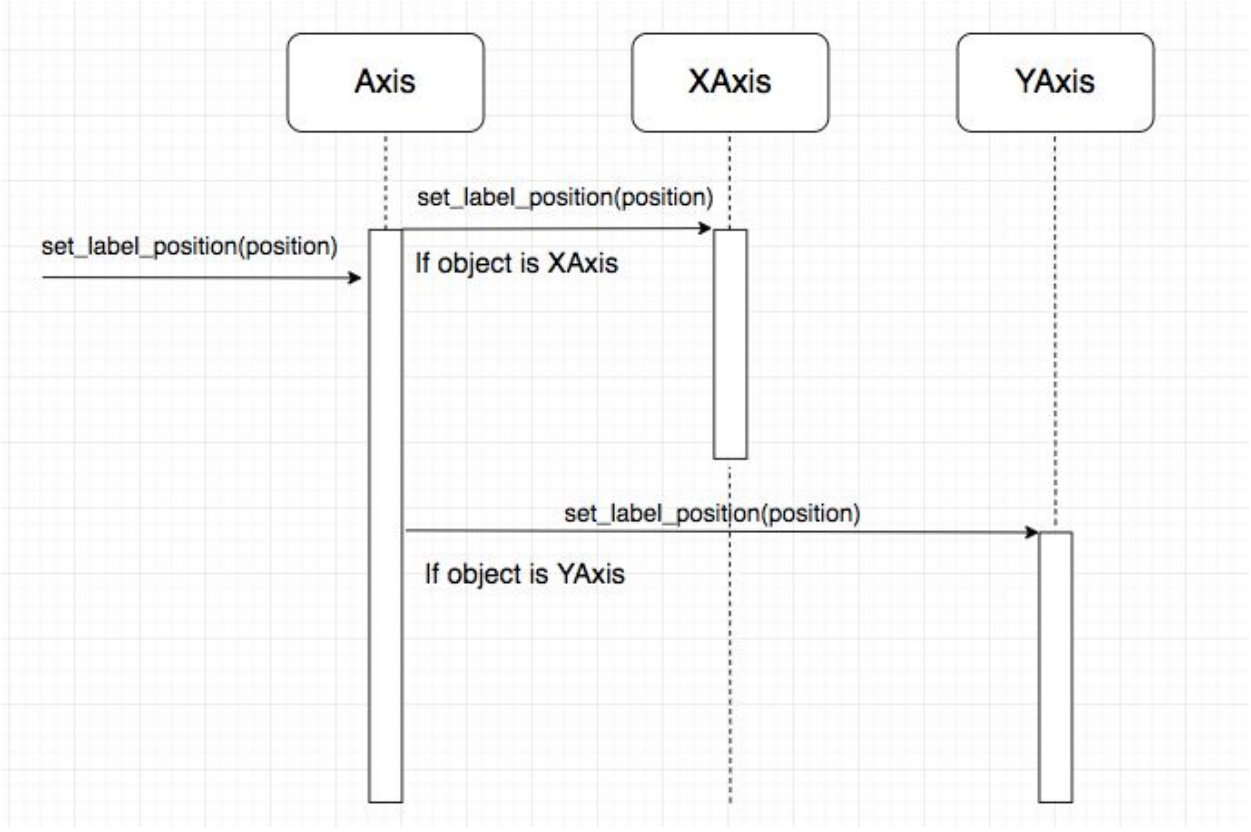
The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

Design Patterns

1) **Strategy Design Pattern**



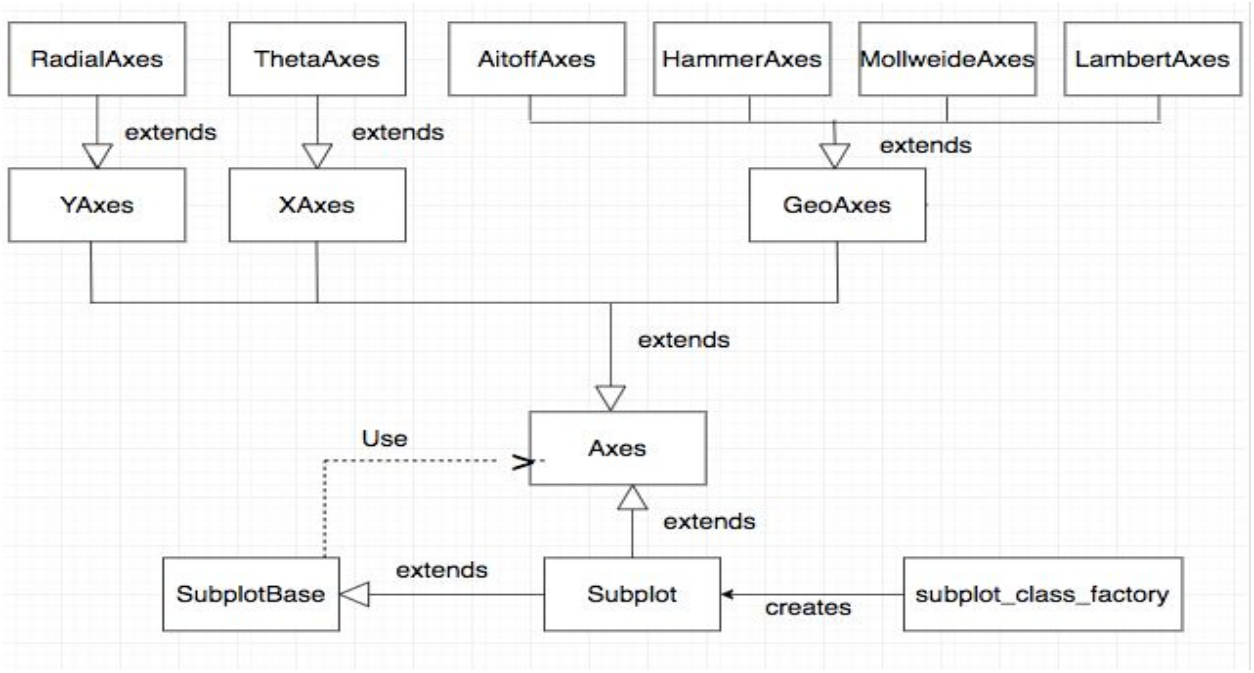


When set_label_position is called on an XAxis object, the set_label_position of XAxis is called (similar logic for YAxis). There are many other examples.

The Quick and the Studious

Chris Ling, Faris Ally, Harrison Fok, Vili Milner, Winston Zhu

2) **Factory Design Pattern**



subplot_class_factory takes in an Axes and creates a new class that inherits from SubplotBase and the given axes_class (which is assumed to be a subclass of Axes). This allows a Subplot of any type of axes to be created without having to create a new Subplot class for every Axes. There are many types of Axes and each have their own transform algorithms (eg. some invertible, some not), which will make them look different on the subplots. Once a Subplot is created, it is returned so that the calling function can instantiate it.