Deliverable 3

Team 28 - Big Software Energy 2.0
Muhammad Farooq, Jadin Luong, Vincent Teng, Aster Wu
March 19th 2020

Brief Feature Description 3

Implementation Investigation 5

Feature Selection 9

Customer Acceptance Testing 10

High Level System Design Revisited 14

Brief Feature Description

Briefly and clearly describe the selected features/bug fixes in your report.

1. ENH: Add legend toggle to figure options dialog box #11109

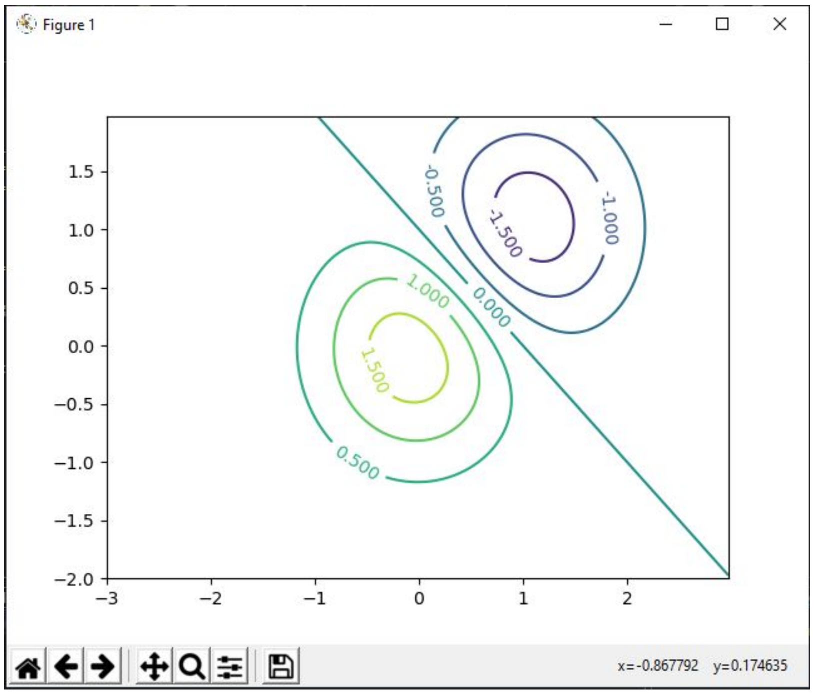
Issue: https://github.com/matplotlib/matplotlib/issues/11109

The user wants to be able to toggle the display of the legend as it can obstruct portions of the plot. The option to toggle the legend can be placed in the toolbar of the matplotlib window whenever a legend is asked to be displayed from the code. There is currently a workaround to viewing the obstructed portions by using the `plt.legend.draggable()` function. This will enable the legend to be draggable.

2. Feature Request: Show z-values for mouse position in contour plots

Issue: https://github.com/matplotlib/matplotlib/issues/14804

In normal plots, the x and y values are shown in the window at the bottom right hand corner:



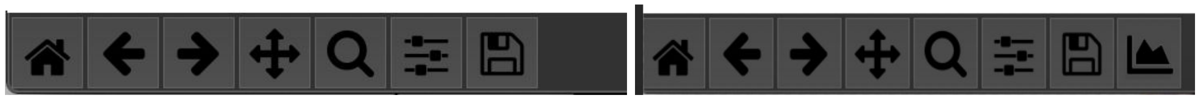
But with plots that also require a z-value, like when using contours, the user would also like to see this z-value shown on the graph. Right now, a potential workaround to this is to define your own mouse hover event which displays the z-value but ideally this z-value should be shown alongside the x and y values. Although the user wants this specifically for contour plots, the issue also extends this to other plots like pcolor, spy, etc. Incorporating these other plots can be considered “nice-to-have” features.

Implementation Investigation

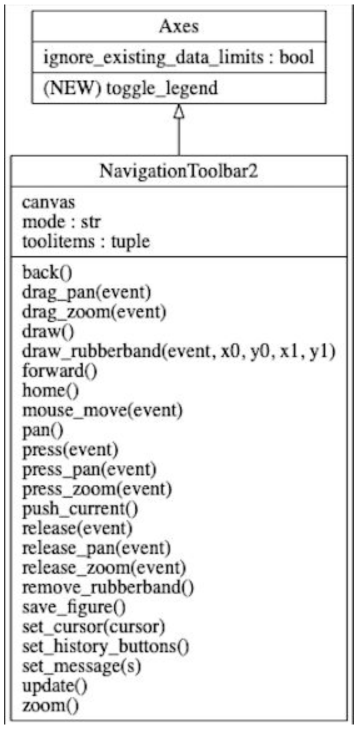
For each of the selected features/bug fixes, investigate what parts of the existing code base you would need to modify in order to implement the feature. Produce designs for the selected features, describing your plans for the organization of new code, as well as all interactions between new code and existing code. You guessed it: UML diagrams would be very helpful here.

1. Issue 11109

To fully implement this feature according to requirements, we will need to create a new button that allows the toggling of the legend.

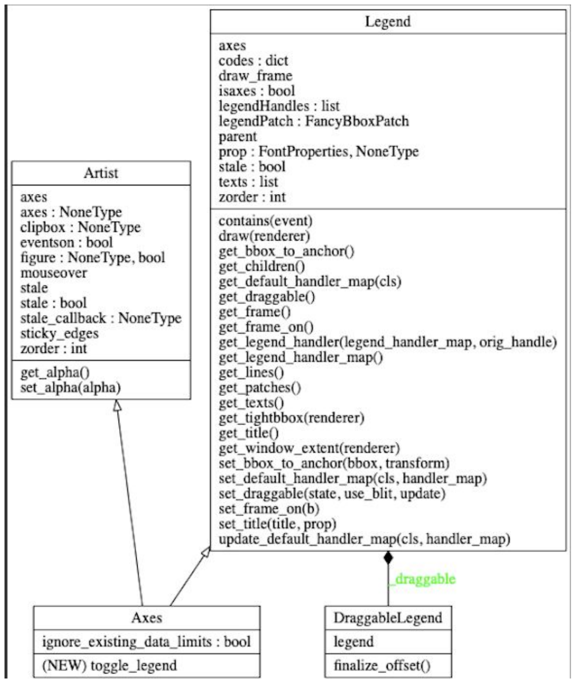

Original vs Anticipated Design

In order to implement this feature, we will need to modify the backend files (backend_bases.py and backend_tools.py + all of the backend files in the backend folder) to add the button. Specifically, the NavigationToolbar2 classes within them:



Since the pan and zoom buttons have similar functionality to the button we are implementing, we will use them to inform our code organization decisions. Notably, the functions drag_pan(event), pan(), press_pan(event), release_pan(event), press_zoom(), It may be challenging since there are numerous backends that are coded differently. Not all of them share the same methods and the number of backends may be an issue. We will need to find the code that directly affects the GUI. The button's icon will use the built-in FontAwesome icons.

In addition to the backend, the button will also interact with artist.py and legend.py. The button will use the method set_alpha from artist.py to change the opacity of the legend between 0 and 1. The actual functionality of the button will lie in _axes.py as that is the container where legends are initialized.



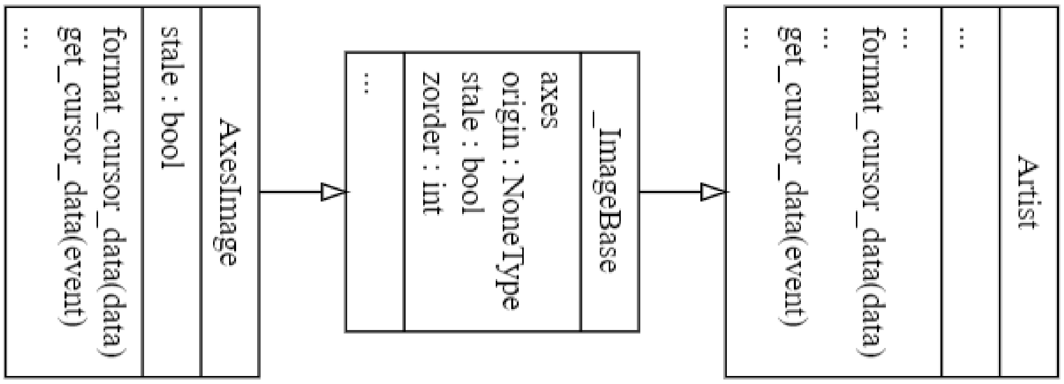
2. Issue 14804

May need to modify:
- cm.py
- contour.py

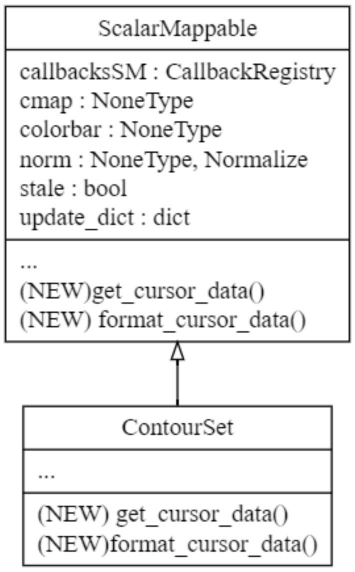
There already exists one type of plot, image.py, that shows the z-value:

There are two methods that are implemented from the Artist parent class (in artist.py) in the AxesImage subclass (in image.py):

get_cursor_data(self, event) -> Returns the cursor data for the given event
format_cursor_data(self, data) -> Returns a string representation of the data



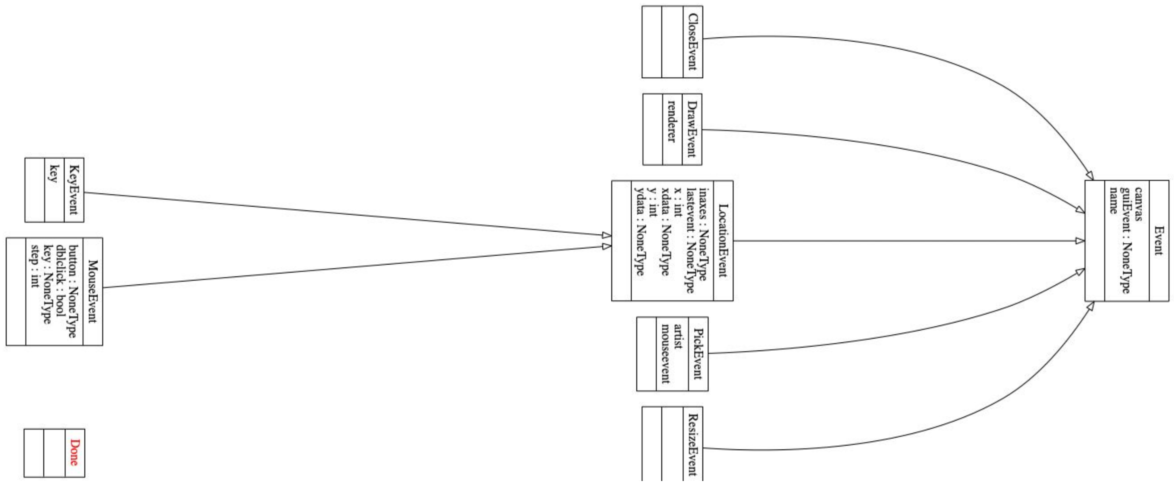
Similarly, in contour.py, the ContourSet class inherits from the cm.ScalarMappable parent class. Therefore, we will need add both get_cursor_data(self, event) and format_cursor_data(self, data) to ScalarMappable class in cm.py and write there implementation in the ContourSet class in contour.py:



The challenge with this is that a ContourSet is not a single object like image. Rather it is a collection of Line objects. So getting the cursor data will be difficult as we would have to loop through the collection line objects and find which of these Lines is the mouse currently hovering over. As mentioned in the comments for the issue, this is inefficient. We will need to find a better algorithm to identify how to obtain the mouse hover event.

Will need to find code which handles retrieval of X and Y value of the graph and displays them in the GUI on code execution.

With the mouse actions, the following UML illustrates the flow and structure of mouse actions



The mouse event function collection should yield the desired capabilities.

Additionally, the GitHub issues mentions

```
def mouse_move(self, event):
```

handling formatting of mouse movement actions
and

```
def get_cursor_data(self, event):
```

relevance to get_cursor_data and format_cursor_data as functions of interest. Further investigation into their relevance should yield a logical method of implementation.

Feature Selection

Select one of these features/bug fixes for implementation, and briefly explain your decision.

We chose to proceed with Issue 11109. We feel that we understood the requirements and the components of code related to this issue better compared to the other issue. This issue requires more addition rather than modification as compared to Issue 14804. When there is more modification, there is more risk as the changes may affect other components of the system. Also, 11109 requires us to work both the front end scripting component of matplotlib as well as the backend component which will be a better learning experience for us and will allow us to break up work easier as compared to 14804.

Another reason 11109 is a better choice is because we believe we have more freedom in our choice of implementation as compared to other potential issues. There is no specification from the user on how the legend should be togglable so this gives more creativity on how we want to accomplish this i.e. add a button in the toolbar, an onclick event on the legend itself, etc. Finally, since there is already a pan and zoom button in place, we can base our implementation of this new button based on the implementation of the pan and zoom.

Customer Acceptance Testing

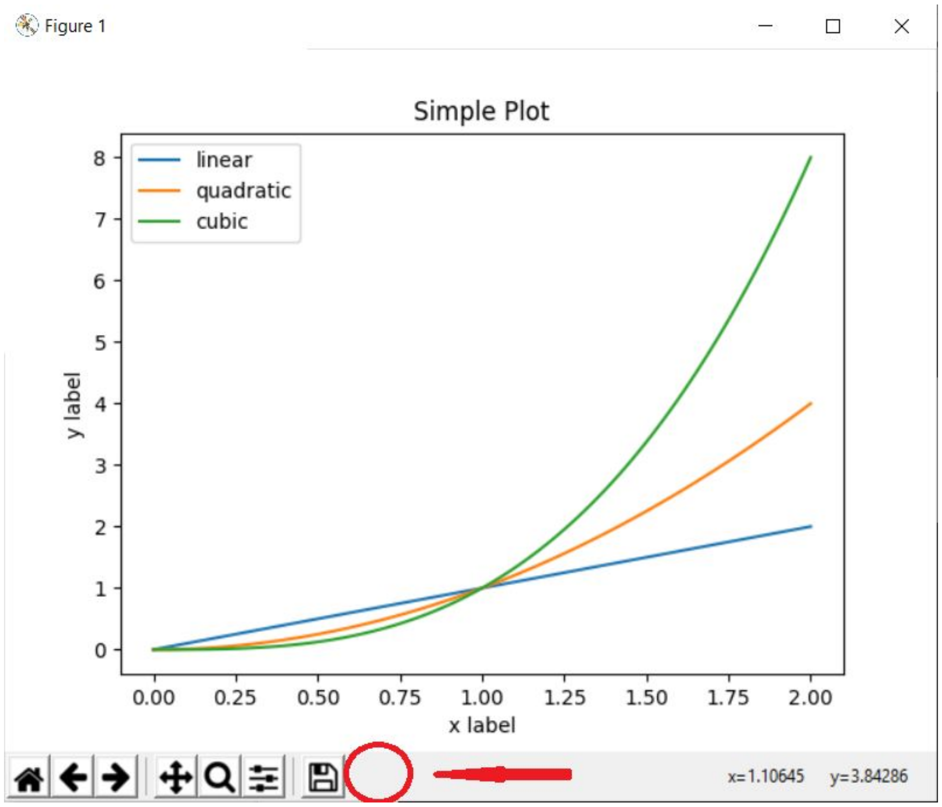
For the selected feature/bug fix, produce a suite of acceptance tests that will demonstrate that the feature/bug fix has been implemented correctly. Design these as “customer acceptance” tests: i. e. a description of the steps a user needs to carry out to check that the program works as expected.

Issue 11109

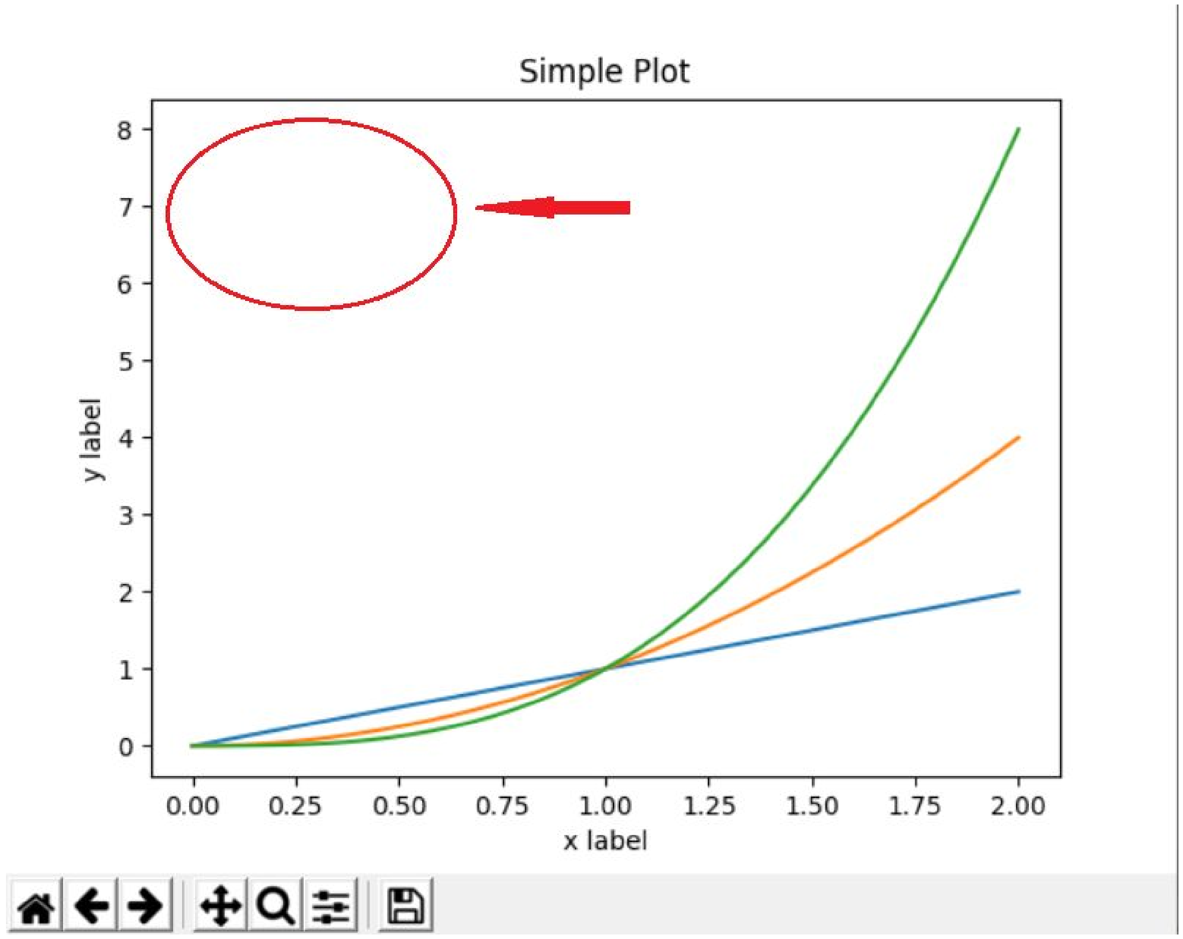
1. Create a random figure/plot. For the sake of this feature, create a figure that plots more than one function. In the following example, the figure contains the plot of a linear, quadratic and cubic function. We will run this file using python3 **filename**.py.

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

x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')

plt.title("Simple Plot")

plt.legend()

plt.show()
```

2. Ensure that there is a toggle button to show/hide the legend of the figure.



3. Ensure that clicking the toggle button for the legend will actually show or hide the legend. Image below shows the legend being hidden after toggling the legend off.



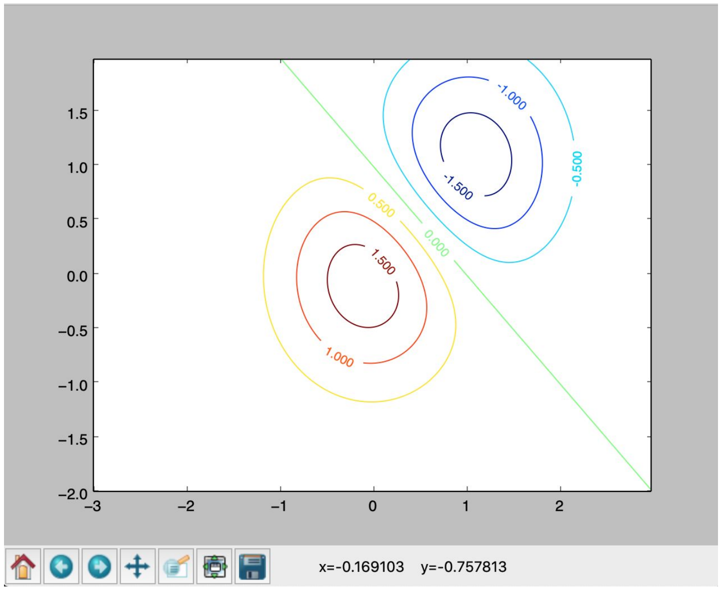
Issue 14804

1) Create a contour plot, this example is from the GitHub issue. Run the file from terminal with python3 FILE_NAME.py

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



delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
```

```
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)
ax.clabel(CS, inline=1, fontsize=10)
plt.show()
```

2) Hover your mouse over the graph verify that the z value is displayed in addition to the X and Y



x=-0.169103 y=-0.757813

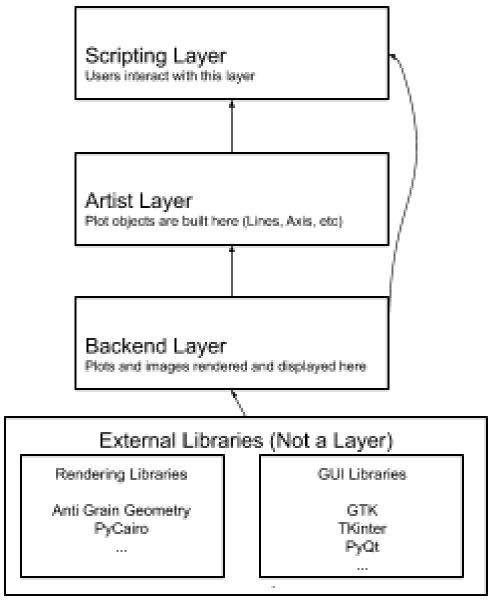
3) Try this with a number of other graphs capable of Z values for assurance (especially ones with great number of contours to verify concern mentioned in GitHub comment was addressed)

High Level System Design Revisited

In addition, now is your chance to revisit the architecture of the project! Draw a higher-level diagram to show the overall architecture of the system. Use any appropriate UML notation (e.g. packages, components, interfaces, etc). Be sure to show clearly where the classes belong in this architecture, and what external packages the system interacts with. As always, the exact UML notation that you use is much less important than the modelling decisions you make: what is important enough to include, what to omit, and how to structure your diagrams. Now that you know more about software architecture, make sure to describe all patterns, design principles followed, and interesting solutions you notice in the architecture.

<u>Layered Architecture</u>

Matplotlib use a layered architecture model:


Scripting Layer
Users interact with this layer
Artist Layer
Plot objects are built here (Lines, Axis, etc)
Backend Layer
Plots and images rendered and displayed here
External Libraries (Not a Layer)
Rendering Libraries
Anti Grain Geometry
PyCairo
...
GUI Libraries
GTK
TKinter
PyQt
...

The lower layers of matplotlib generally don't know about the upper layers i.e. the backend layer works independently of the Artist and Scripting Layer but both the Artist and Scripting Layers depend on the backend layer. In particular, although matplotlib only uses 3-layers, it does use an open-layered architecture as the Scripting Layer can interact with both the Artist and Backend Layer.

Each layer acts as its own independent unit, is highly cohesive and there is minimal coupling between other layers in the hierarchy. The coupling between the Artist and Backend layers comes from the draw() method although the Artist layer does not know what it is actually drawing to (SVG, PDF, etc.). The coupling between the Scripting Layer and the Artist Layer happens only through the pyplot object.

<u>Backend Layer</u>

The Backend layer consists of three key components:

1. FigureCanvas – represents the surface/area in which figures will be drawn on
2. Renderer – responsible for drawing and rendering the figure onto the canvas
3. Event – handles user input/events

This is the bottom layer among the three layers in the stack of matplotlib's architecture and is essentially responsible for formulating/drawing figures onto the canvas and allowing users to interact with the figures. Since this layer is at the bottom of the architecture, it is not aware of the upper layers within the architecture, thus, it does not directly interact with those layers.

<u>Artist Layer</u>

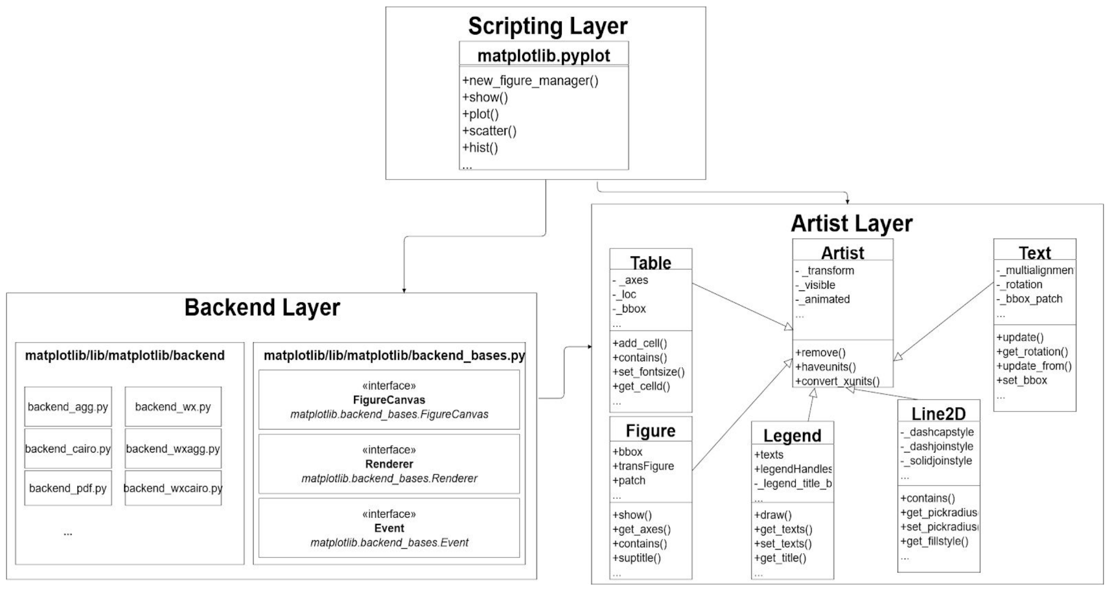
The Artist layer represents the middle layer of matplotlib's architecture. This layer consists of one main component which is the Artist class/object. Being the middle layer of matplotlib's architecture means that it is aware of bottom layers, in this case, it is aware and able to interact with the Backend Layer which is exactly what it does. The Artist layer tells the Backend layer how to draw/render on the canvas.

There are two main types of Artists:

1. Primitive Artists - basic building blocks of matplotlib (i.e. lines, rectangles, text)
2. Composite Artists - collections of primitive artists that form a complete object (i.e. axis, graphs, plots, etc.)

<u>Scripting Layer</u>

The scripting layer is simply the Matplotlib.pyplot interface and it allows people who are not necessarily professional programmers to analyze and interact with the data portrayed within each constructed figure. The scripting layer interacts with both, the Backend Layer and Artist Layer as it acts like a controller that connects both layers with each other giving the layers the ability to communicate with one another. Overall, this layer allows people to easily visualize their data, making data analysis processes easier.


Scripting Layer
matplotlib.pyplot
+new_figure_manager()
+show()
+plot()
+scatter()
+hist()
...
Backend Layer
matplotlib/lib/matplotlib/backend
backend_agg.py
backend_wx.py
backend_cairo.py
backend_wxagg.py
backend_pdf.py
backend_wxcairo.py
...
matplotlib/lib/matplotlib/backend_bases.py
«interface»
FigureCanvas
matplotlib.backend_bases.FigureCanvas
«interface»
Renderer
matplotlib.backend_bases.Renderer
«interface»
Event
matplotlib.backend_bases.Event
Artist Layer
Table
- _axes
-_loc
-_bbox
...
+add_cell()
+contains()
+set_fontsize()
+get_celld()
...
Artist
- _transform
-_visible
-_animated
...
+remove()
+haveunits()
+convert_xunits()
Text
-_multialignmen
-_rotation
-_bbox_patch
...
+update()
+get_rotation()
+update_from()
+set_bbox
...
Figure
+bbox
+transFigure
+patch
...
+show()
+get_axes()
+contains()
+suptitle()
...
Legend
+texts
+legendHandles
-_legend_title_b
...
+draw()
+get_texts()
+set_texts()
+get_title()
...
Line2D
-_dashcapstyle
-_dashjoinstyle
-_solidjoinstyle
...
+contains()
+get_pickradius
+set_pickradius
+get_fillstyle()
...