CSCD01: Deliverable 3

Team Name: //TODO

Table of Contents:

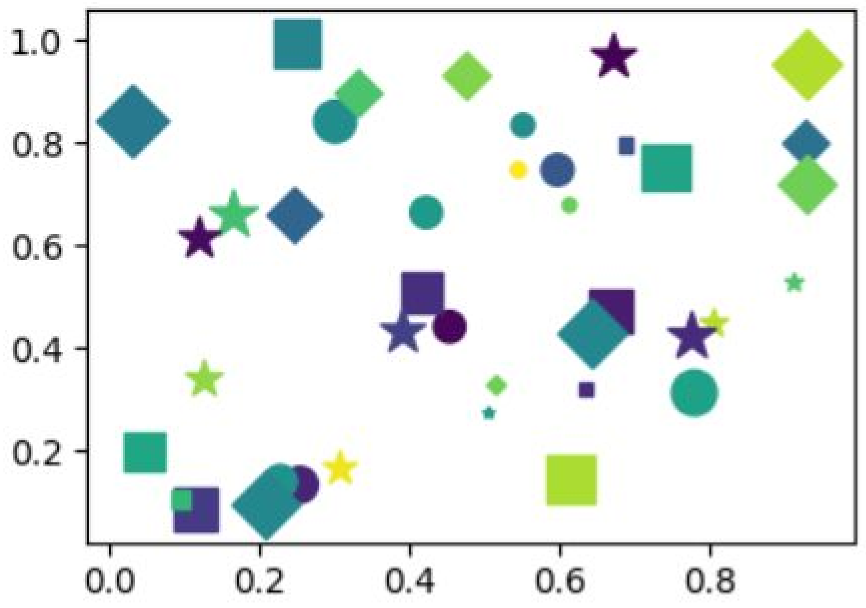
The 2 Issues Selected

We selected issue #11155 and issue #13440 for in-depth investigation.

Issue #11155 Description

This issue involves the implementation of a feature that allows the pyplot scatter method to accept a list of marker styles as a parameter - currently only one marker style is accepted by the method. By allowing a list of styles to be accepted, similarly to how individual labels can be designed with specific colours, it will allow the data points to take on various shapes and sizes as specified by the matplotlib marker style API.

Image of an Example of Intended Feature:



The above scatter graph depicts an example of a scatter graph that can be created when the intended feature has been implemented. This is done through the array [“o”, “s”, “D”, “*”] which are examples of four of the marker styles that can be used to modify the plot points. Each plot point is known as a marker. The symbols in the list above are text representations for some of the marker styles that can be applied; where “o” are circles, “s” are squares”, “D” are diamonds and “*” are stars.

The discussion in the comments contain three ways that the feature can be included in the project: either by including the code to create the above scatter as a gallery example; or adding a new parameter to the scatter method that accepts lists of marker styles; or by altering the implementation of the existing ‘marker’ parameter to accept lists as well as individual marker styles. Doing the former provides all users with a workaround to use locally when they feel the need to use this functionality. It is only a workaround and places the responsibility of implementation on individual users, but as the commenters discuss, it is the easiest form of ‘implementation’ requiring no actual coding or altering of the API. Meanwhile the latter two options include the functionality into the codebase itself. This will allow all users to use this feature when they want to, without having to rely on a gallery example. After some discussion the developers agreed that if the functionality was added to the codebase it should be done through the last option - altering the existing ‘marker’ parameter to accept either a single marker or a list of markers. Their reasoning is that adding a new parameter with an overlapping functionality to the current ‘marker’ parameter would lead to confusion and problems in the long-run. Therefore, the chosen method of implementation is the last - altering the existing ‘marker’ parameter.

The developers noted that plots created via the pyplot’s plot method are intended to have identically “markered” labels. Therefore, only the pyplot scatter method should be updated to include this functionality. Amongst all the ideas presented in the comments, it is noted (in the code example as well) that the number of marker styles used to plot the scatter graph must equal the number of points being produced on the graph. Otherwise, the functionality doesn’t work as intended as there won’t be enough markers to account for all the plotted points. This may cause some points to retain their default shape which would not display well in the final graph.

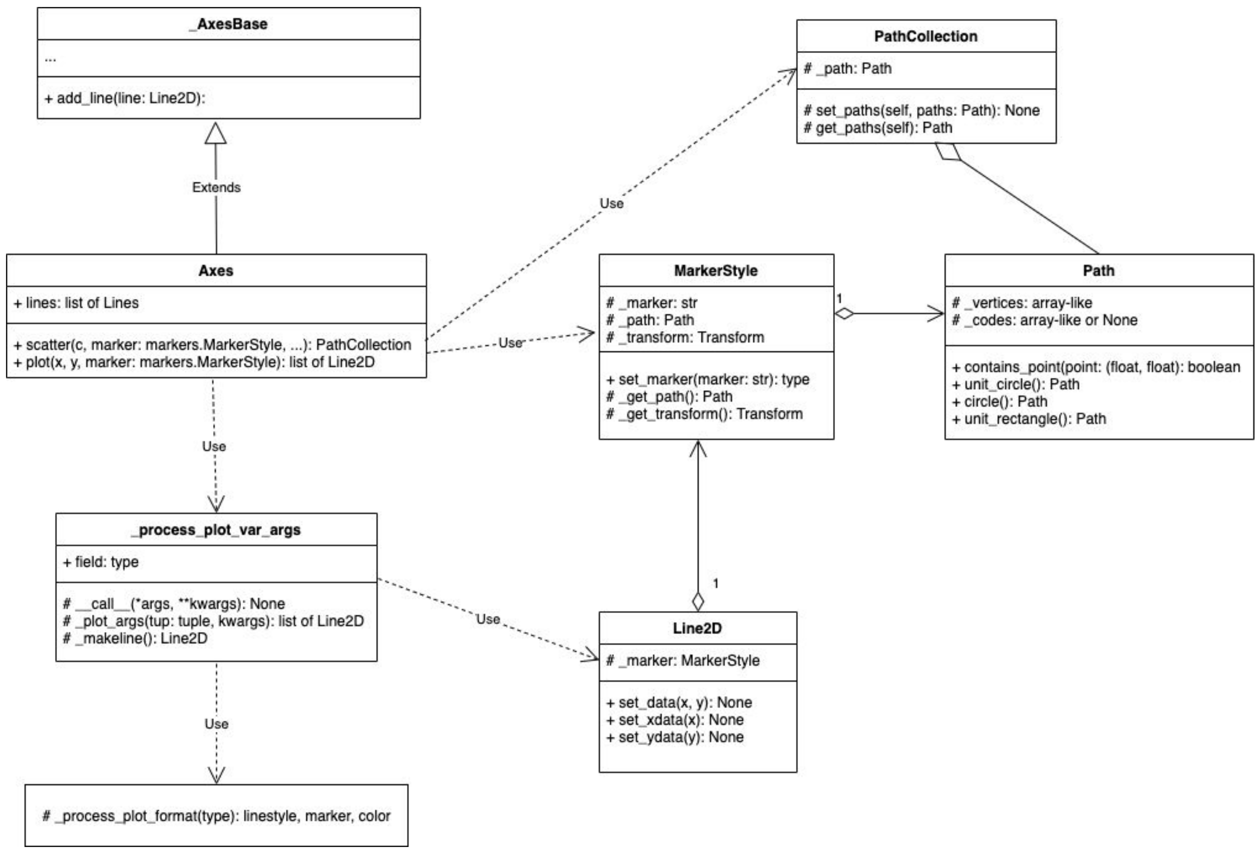
Code Investigation

The feature that was originally requested was for *plot* and *scatter* methods in pyplot to accept a list of markers as input. However, in the comments, a matplotlib developer suggested that this feature should not be added to the plot method because plot should not have the ability to accept multiple markers. Therefore, following the suggestion given by the developer, we are going to implement this new feature for the *scatter* method only. The *scatter* method in pyplot currently accepts a single marker. Once the new feature has been implemented, it will accept a list of markers and display them appropriately on the graph.

Through the investigation of the current code we determined the following: when the scatter method in pyplot is invoked it calls pyplot’s *gca* method to get the current Axes instance on the current figure. The *gca* method calls pyplot’s *gcf* method to get the current Figure instance and then calls the *gca* method in _axes.py to get the current Axes instance on the figure. Once the Axes object is obtained, pyplot’s *scatter* method calls the *scatter* method in _axes.py and passes in all the arguments without modification.

The marker parameter for _axes.py’s *scatter* method currently accepts either a MarkerStyle object or the string representation of a marker. If a string representation of a marker is given, it will be converted to a MarkerStyle object. Then, a Path object is created for the marker. A Path object is used to represent a number of line and curve segments that could be disconnected or closed. The PathCollection class in collections.py is then called by passing this Path object and kwargs that are passed to the scatter method. PathCollection is the most basic subclass of the Collections class. It is used to represent a collection of Path objects and offers functionalities such as getting and setting the object’s paths and creating labels and legend elements using its *legend_elements* method.

The PathCollection object contains a Path object for each marker that needs to be displayed on the scatter graph. Since the original functionality accepts a single marker, one Path object is created and passed to the PathCollection class in the original code. The new feature must allow the user to enter a list of markers. Therefore, the *scatter* method in _axes.py must be modified so that for each marker in the list, a Path object is created and this set of Path objects is passed to the PathCollection class while creating the PathCollection object.

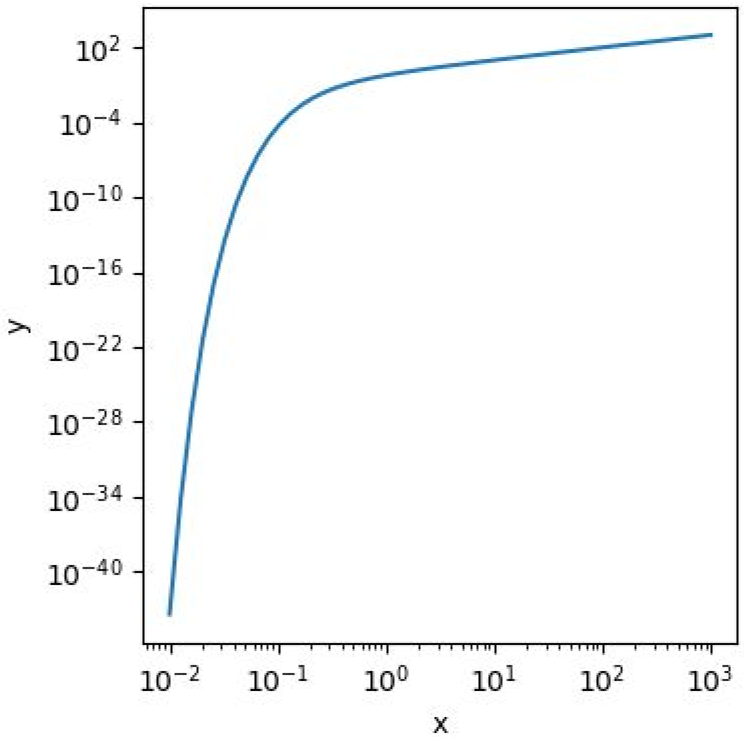


_AxesBase
...
+ add_line(line: Line2D):
Extends
Axes
+ lines: list of Lines
+ scatter(c, marker: markers.MarkerStyle, ...): PathCollection
+ plot(x, y, marker: markers.MarkerStyle): list of Line2D
Use
PathCollection
# _path: Path
# set_paths(self, paths: Path): None
# get_paths(self): Path
Use
MarkerStyle
# _marker: str
# _path: Path
# _transform: Transform
+ set_marker(marker: str): type
# _get_path(): Path
# _get_transform(): Transform
1
Path
# _vertices: array-like
# _codes: array-like or None
+ contains_point(point: (float, float): boolean
+ unit_circle(): Path
+ circle(): Path
+ unit_rectangle(): Path
Use
_process_plot_var_args
+ field: type
# __call__(*args, **kwargs): None
# _plot_args(tup: tuple, kwargs): list of Line2D
# _makeline(): Line2D
Use
1
Line2D
# _marker: MarkerStyle
+ set_data(x, y): None
+ set_xdata(x): None
+ set_ydata(y): None
Use
# _process_plot_format(type): linestyle, marker, color

Issue #13440 Description

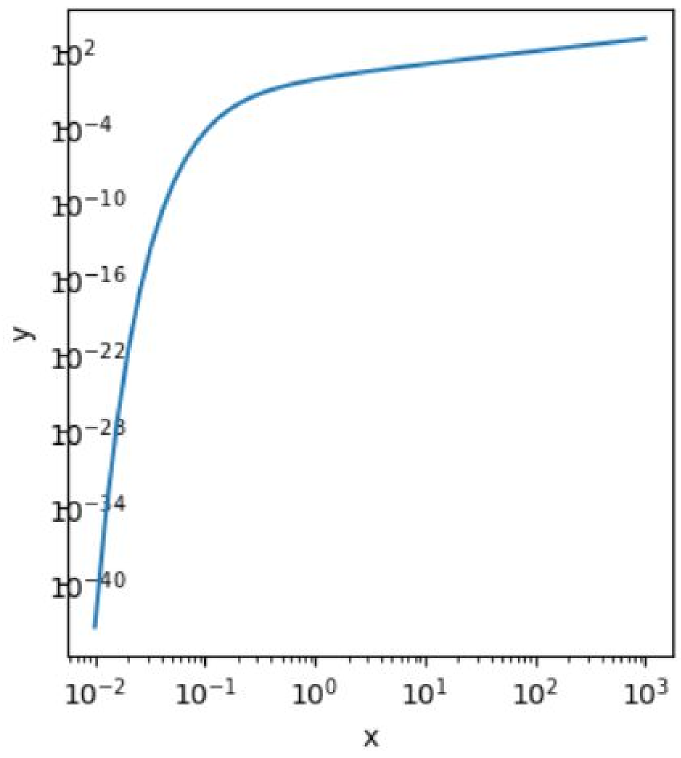
This task revolves around a bug that was reported regarding the label alignment for ticks on the y-axis. When the user attempts to align the values to the left using the set_tick_params method, the ticks remain right-aligned since the horizontal alignment parameter isn't included in the API for setting tick params on the axes.

Image of bug:



Commenters note that a horizontal alignment parameter doesn't currently exist for the set_tick_params() method and that the expected functionality would be hard to implement as each tick label acts as its own component and therefore, the individual tick labels need to be aligned relative to one another. The code provided in a comment shows that going through each individual tick label and setting its alignment only pushes the tick labels onto the axis of the graph, instead of successfully aligning them. Since the lack of support for a horizontal alignment parameter in the set_tick_params method is a separate issue (#13774) our team chose to focus on the current (faulty) method of alignment - setting each tick label alignment separately.

Image of faulty alignment:



Code for reproduction:

```
1:  import matplotlib.pyplot as plt
2:  import numpy as np
3:  import matplotlib as mpl
4:  print("Matplotlib version: %s" % mpl.__version__)

5:  x = np.logspace(-2,3)
6:  y = 1 / (np.exp(1/x) - 1)

7:  fig = plt.figure(1, figsize=(4, 4))
8:  plt.loglog(x,y)
9:  plt.xlabel('x')
10: plt.ylabel('y')
11: plt.tight_layout()

12: ax = plt.gca()

13: for label in ax.yaxis.get_ticklabels():
14:     label.set_horizontalalignment('left')
```

Code trace:

Statements 1-4:`import matplotlib.pyplot…("Matplotlib version: %s" % mpl.__version__)`

Result: Imported pyplot, numpy, and matplotlib. Printed matplotlib version.

Statement 5: `x = np.logspace(-2,3)`

Result: Set x = array([1.00000000e-02, 1.26485522e-02, 1.59985872e-02, …]

Statement 6: `y = 1 / (np.exp(1/x) - 1)`

Result: Set y = array([3.72007598e-44, 4.61838994e-35, 7.14821983e-28, …]

Statement 7: `fig = plt.figure(1, figsize=(4, 4))`

Result: Created a new figure.

Set fig = <Figure size 400x400 with 0 Axes>

Statement 8: `plt.loglog(x,y)`

Result: Created a plot with log scaling on both the x and y axis, using the x and y stored arrays as a base for each axis.

Statement 9: `plt.xlabel('x')`

Result: Set the label text for the x axis to ‘x’

Statement 10: `plt.ylabel('y')`

Result: Set the label text for the y axis to ‘y’

Statement 11: `plt.tight_layout()`

Result: Adjusted plot layout and padding.

Statement 12: `ax = plt.gca()`

Result: Set ax = the current Axes instance on the current figure

ax = <matplotlib.axes._subplots.AxesSubplot object at 0x000002289CD6CDF0>

Statement 13: `for label in ax.yaxis.get_ticklabels():`

Result: Called ax.yaxis.get_ticklabels() which called get_majorticklines() in turn which returned the major tick labels as a list of Line2D instances.

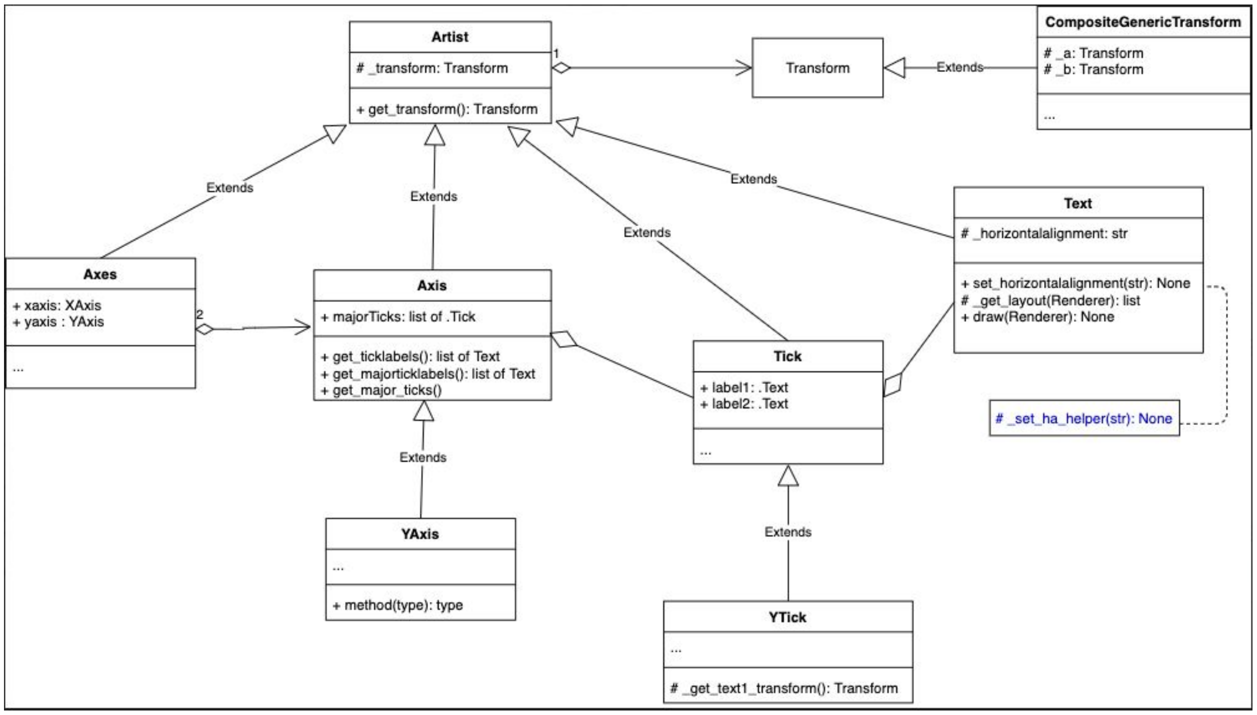
Statement 14: `label.set_horizontalalignment('left')`

Result: For each major tick label set the _horizontalalignment value of the label to ‘left’. This controls the x position of the text and aligns it to the left of the bounding box. The bug occurs because the bounding boxes are not aligned with one another.

Implementation Plan for #13440:

Since the alignment of the axis ticks are relative to their own bounding boxes, there are multiple factors that are considered. Some of these factors include the font size of the labels, the number of ticks on the axis, and the placement of the axis title. A potential fix involves working with a TextCollection class to produce multiple labels with similar properties (including alignment) and later attaching them to the graph according to the user’s specification. This would allow multiple tick labels to be produced at once to the liking of the user and it will create the graph accordingly. However, since the tick labels and text class are used throughout the code base, any changes that come with this solution would run the risk of change propagation. Therefore, attempting this fix could present issues in other areas of the code.

Another proposed solution is using hard-coded padding on the individual tick labels to push them into their specified alignments. However, hardcoding values is considered bad coding practice - and rightly so. Instead of using a fixed value for the padding, a value could be calculated based on the various factors of the graph and used in a similar manner. Factors to consider would be the size of the labels and their proximity to the surrounding graph elements and the placement of the other bounding boxes. It is important to consider the location of all of the bounding boxes so they can be aligned relative to one another. This solution requires less work and contains a lower risk of change propagation; therefore, this is the solution we would implement had we chosen this issue. In order to reduce the risk of breaking pre-existing code and the risk of change propagation we would add a helper method to the Text class called _set_ha_helper(str). This helper method would take in the alignment type ('right', 'left', or 'center') and calculate and apply the necessary padding. The helper method would be called by the set_horizontalalignment(str) method after the current alignment (within the bounding boxes) had been applied.


Artist
# _transform: Transform
+ get_transform(): Transform
1
Transform
Extends
CompositeGenericTransform
# _a: Transform
# _b: Transform
...
Extends
Extends
Extends
Extends
Text
# _horizontalalignment: str
+ set_horizontalalignment(str): None
# _get_layout(Renderer): list
+ draw(Renderer): None
# _set_ha_helper(str): None
Axes
+ xaxis: XAxis
+ yaxis : YAxis
...
2
Axis
+ majorTicks: list of .Tick
+ get_ticklabels(): list of Text
+ get_majorticklabels(): list of Text
+ get_major_ticks()
Tick
+ label1: .Text
+ label2: .Text
...
Extends
YAxis
...
+ method(type): type
Extends
YTick
...
# _get_text1_transform(): Transform

The Issue to Implement

We selected the issues #13440 and #11155 for in-depth investigation because they seemed interesting and suitably challenging. After investigation we realised that issue #13440 is not ideal for this project for two main reasons. First, the bug described in #13440 involves code that is used in multiple places throughout the codebase. This raises concerns of change propagation. Any implementation we apply would have to be carefully considered in order to reduce the chance of change propagation and would require a level of familiarity with the codebase that we are unlikely to develop within the next two weeks. Furthermore, the fix for issue #13440 requires calculations based on multiple factors including tick size, tick label size, placement of the axes, and the locations of the other bounding boxes. While these calculations are possible they would be complex and would require time and dedication that our team would be unable to provide within the next two weeks. For these reasons we were extremely hesitant to pursue an implementation for this issue. Luckily, our instincts held true for issue #11155. #11155 requires an isolated implementation that has a low risk of propagation change. In addition, it is the implementation of a requested feature allowing our team to fulfill our goal of contributing meaningfully to the codebase. Therefore, we chose issue #11155 for implementation.

Implementation Plan

Two modifications to the original code would be necessary to implement this issue: to create a Path object for each marker in the list; and to pass this list of paths, instead of a single path, by creating a PathCollection object.

Our team will add two new helper methods to _axes.py:
check_marker_list: this method handles error checking. If a list of marker styles is passed, this method is called to check if the size of the list is the same as the number of points to be displayed. The method will ensure the size of the marker style list is equal to the size of the x parameter (which is an array of the x-coordinates of the points). In addition, it will check if the values in the list are valid MarkerStyle objects or valid marker style shorthand strings. If the size is equal and the values are valid, this method returns true and scatter will continue implementing the next instructions. Otherwise, a ValueError, for wrong length or value respectively, will be returned.

create_marker_paths: this method will be called to create the PathCollection object. It will use a for loop to go through each marker style. If the marker style is formatted as one of the string shorthands, it will be converted into the corresponding MarkerStyle object. If the marker style is given as a MarkerStyle object, this step is ignored. A Path object is then created for this MarkerStyle object. This Path is added to a list of Path objects that will be maintained by the method. As done in the existing code, the edgecolor and linewidth for each MarkerStyle object will be set. After iterating over the marker list, a PathCollection object is created. The list of Path objects created in the for loop is passed to the PathCollection class as one of the parameters along with scatter method's kwargs. Each Path in the PathCollection object will now be displayed as a marker on the graph.

Acceptance Test Suite

Instructions: Fork the repo https://github.com/CSCD01-team23/matplotlib and build matplotlib in a virtual environment as described in https://matplotlib.org/3.1.1/devel/contributing.html#installing-for-devs. Checkout the feature branch.

Issue #11155

User Story: As a matplotlib user, I want to be able to give a single marker style or a list of marker styles to the pyplot scatter method through the 'marker' parameter.

ID	WHEN	THEN
01	The user wants to plot with multiple marker styles on the same graph	The scatter method accepts a list of markers through its 'marker' parameter.
02	The user wants to plot using the scatter method and just one style of marker.	The scatter method accepts a single text representation of marker or MarkerStyle object.
03	The user wants to plot using the scatter method without specifying the marker style.	The scatter method uses a default marker style i.e. the value in rcParams['scatter.marker'].

ID 01: Run this code in python3

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

np.random.seed(19680801)
n = 750
x, y = np.random.rand(2, n)
labels = np.random.randint(0,3, size=len(x))
m = np.repeat([“+”, “*”, “o”], len(labels)/3)

plt.scatter(x, y, marker=m)
plt.show()
```

ID 02: Run this code in python3

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

np.random.seed(19680801)
n = 750
x, y = np.random.rand(2, n)

m = “+”
plt.scatter(x, y, marker=m)
plt.show()
```

ID 03: Run this code in python3

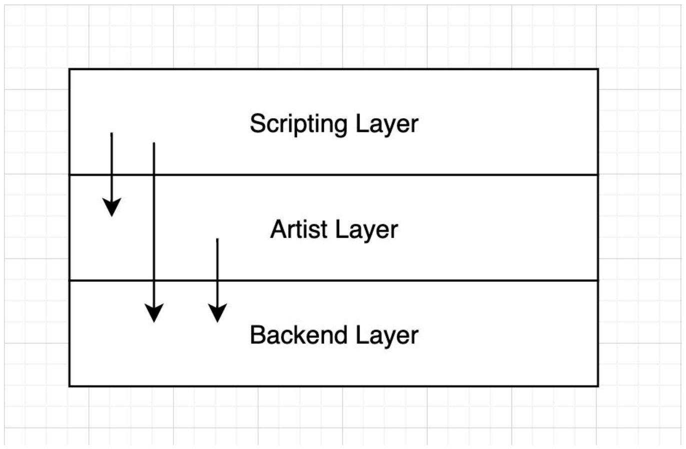
```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)
n = 750
x, y = np.random.rand(2, n)

plt.scatter(x, y)
plt.show()
```

Architecture Document

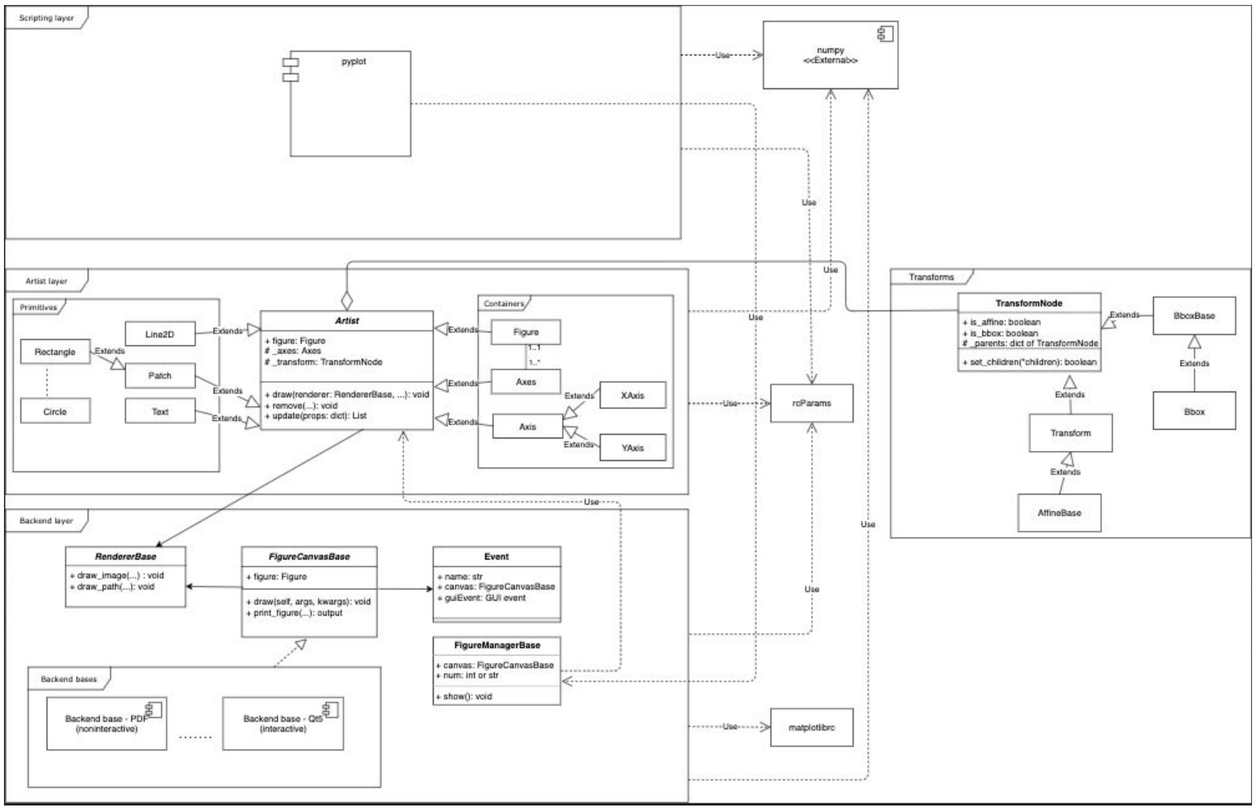
As described in the previous deliverable, matplotlib has three general layers in their architecture: the backend layer; the artist layer; and the scripting layer. The backend layer deals with rendering and outputting the graphs to the appropriate backend. The artist layer is concerned with all the elements used for a graph (e.g. Axes, Figure, Text, Axis). The scripting layer, also known as the pyplot interface, is where users can use the defined API to draw simple graphs. For more complex graphs, it is recommended that the user avoids using the pyplot API and directly manipulates Artist objects in the Artist Layer via the object-oriented API. These layers are stacked up in the order, as seen below.



There is an open layered architecture where layers on top can directly access services from any lower layer. This creates more compact code, but breaks the encapsulation of layers. When the user uses pyplot (the scripting layer), a FigureManagerBase from the backend layer is created. The declaration for a FigureManagerBase is defined in the backend_bases.py file. The FigureManager is a backend-independent abstraction of a figure container and controller. Although the scripting layer can go directly to the backend to access the FigureManager, the dependency between these layers is not too significant.

Good architecture will minimize coupling between modules and will maximize the cohesion between each module. This is generally true for matplotlib as the modules do not know much about one another for interaction. For example, in the artist layer, there is a method called “draw” that takes in a Renderer from the backend and uses it to draw the objects. The artist layer does not need to know the details of how the rendering takes place, which minimizes coupling. In addition, the sub-components of each layer are strongly correlated, which maximizes cohesion. In the implementation of matplotlib good architectural decisions were made to create low coupling and high cohesion.

The UML displaying the overall architecture and major components is shown below. It is mostly the same diagram as the one we submitted previously, however, some additions were made. After working with matplotlib we noticed that the code is very dependent on “rcParams”, which is a global dictionary object that contains default values and includes validation functions. There are default settings defined in rcParams that are used if the user does not specify certain values like colors or linewidths. The user may also modify the value in rcParams as described in matplotlibs documentation. In addition, the configuration file “matplotlibrc” may also be used to customize properties for graphs created. Lastly, an important framework that was not included in the previous UML diagram is Transforms. The classes in Transforms belong to the Artist layer and are used to determine the positions of all the elements drawn onto the canvas. Each Artist object has a field “transform” which is a tree of Transform node objects.


Scripting layer
pyplot
numpy
<<External>>
Use
Artist layer
Primitives
Rectangle
Circle
Line2D
Patch
Text
Extends
Artist
+ figure: Figure
# _axes: Axes
# _transform: TransformNode
+ draw(renderer: RendererBase, ...): void
+ remove(...): void
+ update(props: dict): List
Containers
Figure
Axes
Axis
XAxis
YAxis
1..1
1..*
rcParams
Transforms
TransformNode
+ is_affine: boolean
+ is_bbox: boolean
# _parents: dict of TransformNode
+ set_children(*children): boolean
BboxBase
Bbox
Transform
AffineBase
Backend layer
RendererBase
+ draw_image(...) : void
+ draw_path(...): void
FigureCanvasBase
+ figure: Figure
+ draw(self, args, kwargs): void
+ print_figure(...): output
Event
+ name: str
+ canvas: FigureCanvasBase
+ guiEvent: GUI event
FigureManagerBase
+ canvas: FigureCanvasBase
+ num: int or str
+ show(): void
Backend bases
Backend base - PDF
(noninteractive)
Backend base - Qt5
(interactive)
....
matplotlibrc