CSCD01: Deliverable 4

Team Name: //TODO

Table of Contents:

User Guide

The new feature allows the *scatter* method in pyplot and _axes to accept a list of marker styles as a parameter. Before this feature was implemented, only one marker style was accepted by the method, but the new feature has altered the *scatter* method to accept a list of markers as well. By accepting lists, *scatter* allows users to alter data points to take on various shapes and sizes, as specified by the matplotlib marker style API. To use this class, the user must send in the type of markers they want. The *marker* parameter accepts a single marker as well as a list of markers. Below are some examples that demonstrate the usage of the *scatter* method and how users can alter the look of the data points through the marker parameter.

Type for markers: *marker* can be either a MarkerStyle object, the text shorthand (ex. ‘s’, ‘o’) for a particular marker or a list or array of MarkerStyle or text shorthands. Defaults to None, in which case it takes the value of rcParams["scatter.marker"] (default: 'o') = 'o'.

Throws: ValueError if *marker* is an empty list

Note:
Consult the Markers API for more information on markers and to view the type of marker styles: https://matplotlib.org/3.1.1/api/markers_api.html
Consult the Scatter API for more information on the parameters of the scatter method and the types of parameters it takes: https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.scatter.html

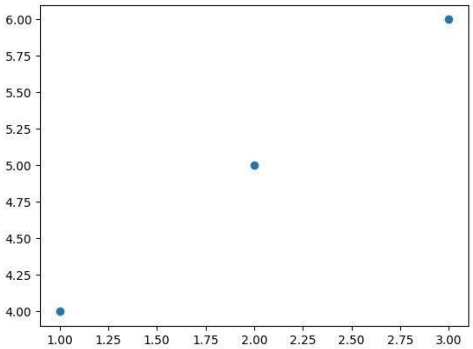
Code Examples:

Example 1: To make a simple scatterplot with three data points and default values for c, marker, edgecolors and linewidth, we use the code below.

Code:

```
import matplotlib.pyplot as plt
plt.scatter([1,2,3], [4,5,6])
plt.show()
```

Output:

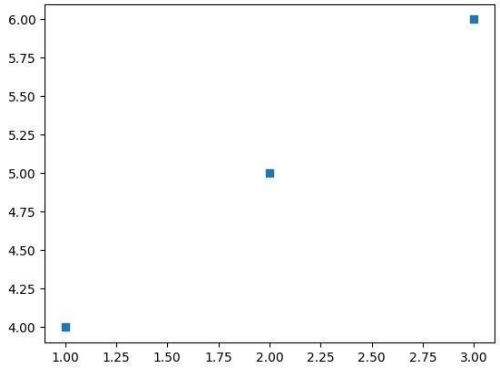


Example 2: If the user simply wants all the data points to be one shape, a list of markers is not required. Instead, the user must input a single marker, which was the way scatter was implemented before the new feature. This specific example makes all data points a square rather than the default circle.

Code:

```
import matplotlib.pyplot as plt
plt.scatter([1,2,3], [4,5,6], marker='s')
plt.show()
```

Output :

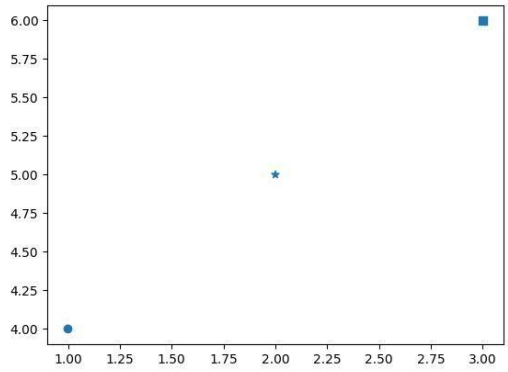


Example 3: To create a scatter plot with three points, where the symbol of the first marker is a circle, the second marker is a star and the third is a square, use the code below. This example uses the new implementation as the user can now input a list of marker styles for the marker parameter.

Code:

```
import matplotlib.pyplot as plt
plt.scatter([1,2,3], [4,5,6], marker=['o','*', 's'])
plt.show()
```

Output:

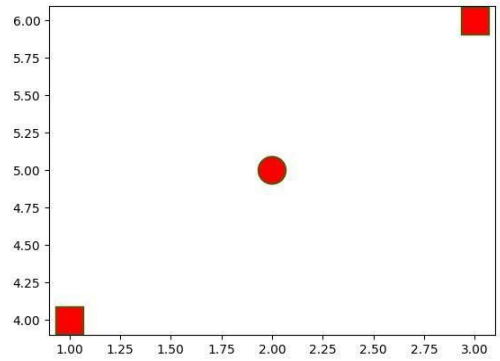


Example 4: This example uses the parameters color (c) and edgecolors. Color represents the color of the data points while edgecolor represents the border color of the data points. As seen below, the color has been set to red and edgecolors has been set to green and two different marker styles were given.

Code:

```
import matplotlib.pyplot as plt
plt.scatter([1,2,3], [4,5,6], c='red', edgecolors=['green'], marker=['s', 'o'], s=500)
plt.show()
```

Output:

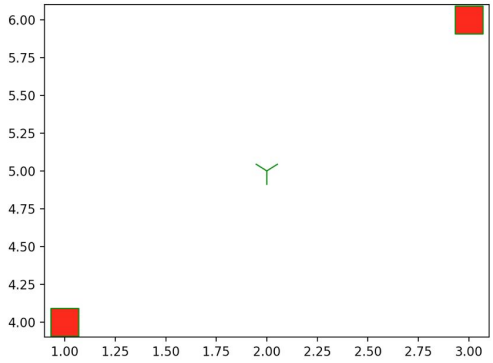


Example 5: Currently, there is a bug in the new code where when given an unfilled marker style such as ‘1’, and a color and edgecolor that differ, the color isn’t correctly set for the unfilled marker. In the example below, we see the color of the unfilled marker is green, but it is supposed to be red based on matplotlib specifications for unfilled markers.

Code (does not work as it should):

```
import matplotlib.pyplot as plt
plt.scatter([1,2,3], [4,5,6], c='red',edgecolors='green', marker=['s', '1'],s=500)
plt.show()
```

Output:



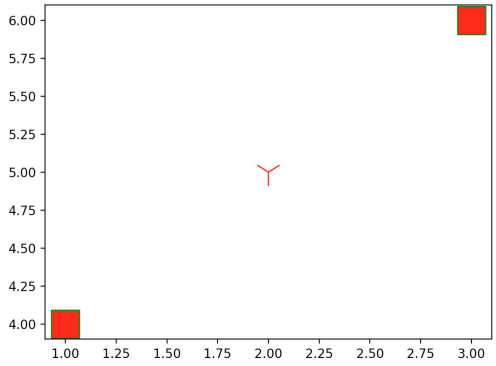
Workaround

For this issue, the user should use a workaround where they set the edgecolor and colour as the same value. This way, the unfilled marker will reflect the documentation.

Workaround Code:

```
import matplotlib.pyplot as plt
plt.scatter([1,2,3], [4,5,6], c='red',edgecolors=['green', 'red', 'green'], marker=['s', '1'],s=500)
plt.show()
```

Output:



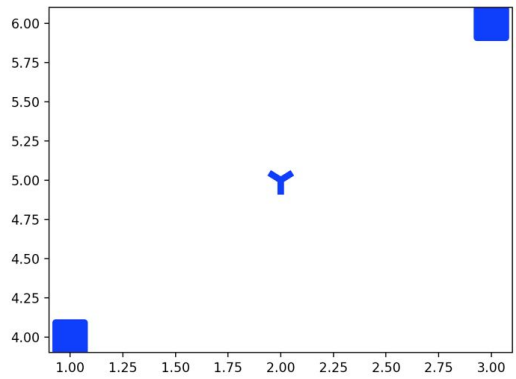
Example 6: When using linewidths, we also have a bug where when using an unfilled marker style, such as '1', when increasing the linewidth to a value greater than 1, we see the unfilled marker style's line width increase in size. Based on matplotlib specifications for unfilled

markers, no matter what size the linewidth is, the unfilled marker should always remain the same default size of 1.

Code (doesn't work as expected):

```
import matplotlib.pyplot as plt
plt.scatter([1,2,3], [4,5,6], c='blue', marker=['s', '1'],s=500, linewidths=5)
plt.show()
```

Output:

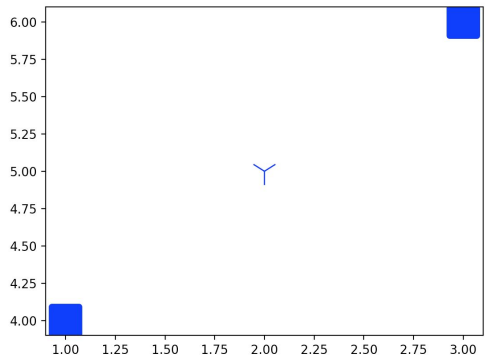


Workaround: Because the linewidth of the unfilled marker is increasing, even though it shouldn't be, a workaround for this issue is to always set linewidth=1 for the unfilled markers. This way, the unfilled marker will remain its default value.

Workaround Code:

```
import matplotlib.pyplot as plt
plt.scatter([1,2,3], [4,5,6], c='blue', marker=['s', '1'],s=500, linewidths=[5, 1, 5])
plt.show()
```

Output:

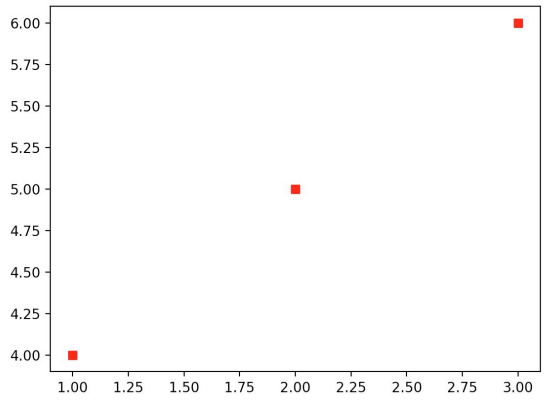


Example 7: The example shows when the marker is of type MarkerStyle. The original code allowed a single MarkerStyle and the new code allows a single MarkerStyle or a list of MarkerStyles.

Single MarkerStyle:

```
import matplotlib.pyplot as plt
import matplotlib.markers as m
marker = m.MarkerStyle('s')
plt.scatter([1,2,3], [4,5,6], c='red', marker=marker)
plt.show()
```

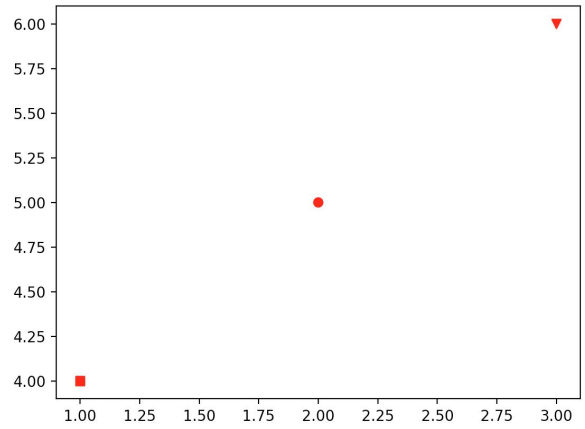
Output:



List of MarkerStyle:

```
import matplotlib.pyplot as plt
import matplotlib.markers as m
m1 = m.MarkerStyle('s')
m2 = m.MarkerStyle('o')
m3 = m.MarkerStyle('v')
plt.scatter([1,2,3], [4,5,6], c='red', marker=[m1, m2, m3])
plt.show()
```

Output:

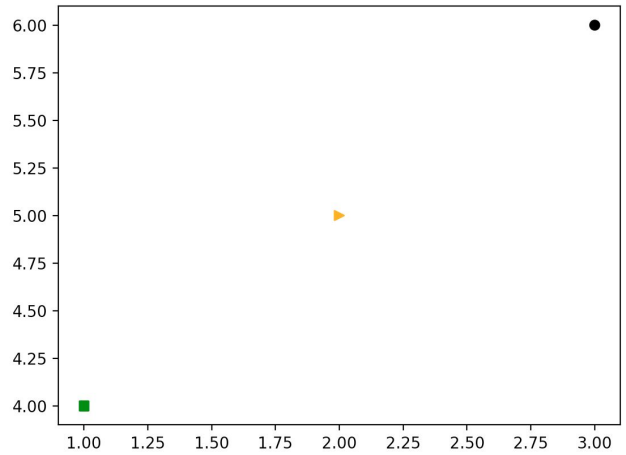


Example 8: This example is a combination of MarkerStyle and text shorthands for marker input.

Code:

```
import matplotlib.pyplot as plt
import matplotlib.markers as m
m1 = m.MarkerStyle('s')
m2 = m.MarkerStyle('o')
plt.scatter([1,2,3], [4,5,6], c=['green', 'orange', 'black'], marker=[m1, '>', m2])
plt.show()
```

Output:

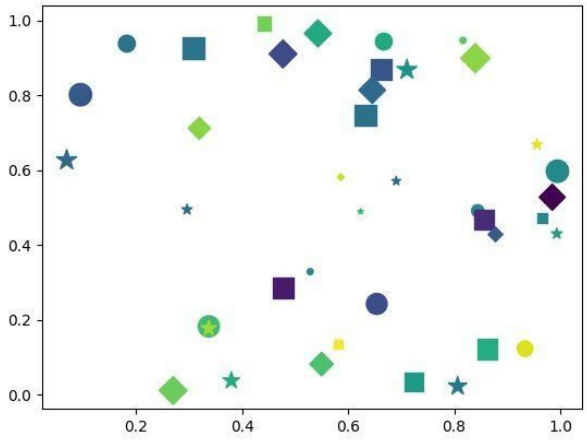


Example 9: The example below uses a marker of type numpy.ndarray. In this example, random styles have been given for a random assortment of data. Every time this is run, a different graph will appear with randomized points and marker styles.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
N = 40
x, y, c = np.random.rand(3, N)
s = np.random.randint(10, 220, size=N)
m = np.repeat(["o", "s", "D", "*"], N/4)
plt.scatter(x, y, c=c, marker=m,s=s)
plt.show()
```

Output: (will differ because of the random points)



Organization of New Code

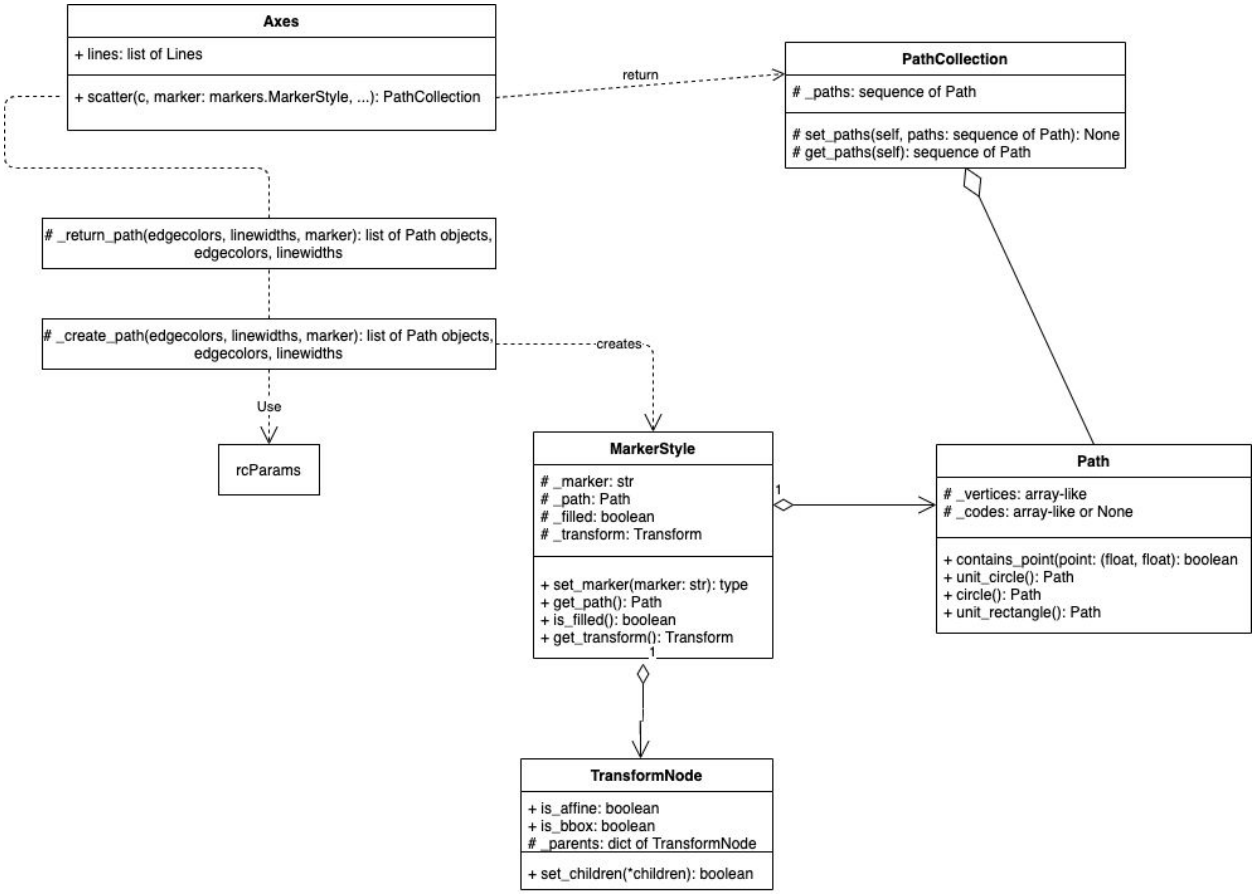
Implementation Details

The code changes we made were all in the _axes.py file. Two new functions were created called *_return_path* and *_create_path*, and three lines of code was added to the *scatter* function as well.

_return_path determines how to deal with the two kinds of markers in the type 'np.ndarray'. The first type is a list of floats that represents a single marker and the second type is a list of markers that is "repeated" using the *repeat* function in numpy. An example of this is the marker list given by the matplotlib devs. The two types must be dealt with differently because the first one must be placed into a list, but the second one doesn't need to be. *_return_path* was created to deal with this issue. Any other input type will be converted into a list using the method. Once the marker is converted as required, the *_create_path* method is called.

_create_path is a method that iterates through the marker list and creates a Path object for each marker. If there is only one marker provided, edgecolors and linewidths will be assigned with the correct values for filled and unfilled markers. That is, if the marker is unfilled, edgecolors will be assigned the value 'face' and linewidths will be assigned in the value in rcParams['lines.linewidth']. If the marker is filled, edgecolors and linewidths are not modified in this method.

Lastly, in the *scatter* method, the original code for the marker was on lines 4464 to 4477. This code created a Path object for the single marker to pass into PathCollection, where edgecolor and linewidth were modified if the marker was unfilled. This was replaced with a function call to *_return_path*, which returns a list of Path objects, the edgecolors variable, and the linewidths variable to pass into PathCollection. Additionally, in the beginning of the *scatter* method, there was an if statement added that checks if an empty list is given for the marker parameter. In this case, a ValueError is thrown to ensure that the user does not pass in an empty list.



Modifications in Implementation Plan

We made a few modifications to the original implementation plan from deliverable 3. Previously, we said we needed a method called *check_marker_list* that consisted of error checking to determine if the markers were valid and if the size of the marker list was the same as the number of points on the graph i.e. x.size or y.size parameters. After further investigation, we realized that this method is not required. The existing code checks for invalid markers when creating the MarkerStyle objects. So, we do not need another check for valid markers. Additionally, we realized that the size of the marker list does not have to be equal to the size of x.size or y.size parameters. Even if the size of the marker list is not the same as the x.size or y.size parameter, our feature will work as expected since the existing code provides the ability to cycle through the given markers. Therefore, we do not need the *check_marker_list* method. Instead, we needed the *_return_path* method to deal with the two kinds of markers of type 'np.ndarray'. Other than these changes, the implementation plan remained the same.

Issues

When implementing the feature, two issues occurred. The first issue deals with edgecolors and how when working with unfilled markers in the marker list, the unfilled marker takes the colour of the edgecolor, rather than the colour of the color parameter (as seen in Example 5 of the User Guide). The second issue is that when working with an unfilled marker, if the linewidth is greater than 1, the unfilled marker increases in size, even though it's supposed to remain it's default size of 1 based on matplotlib specifications (as seen in Example 6 of the User Guide).

A great deal of investigation went into figuring out how and if we could solve these two issues. The fix we thought of for the edgecolors issue would require us to take the input from colors ('c' parameter in scatter) and use that value as the edgecolor if the marker was unfilled. This would make the unfilled marker have the same fill color and edgecolor, which is consistent with the matplotlib requirement. The problem with that approach is that colors accepts input of type numpy.ndarrays but edgecolor doesn't. Therefore, this implementation would not work if the user provides a numpy.ndarray for colors. We have looked into dealing with different inputs for colors and edgecolors but did not find any solutions that can solve the problem of converting float arrays to a different value that is accepted by edgecolors (ex. RGBA or hex). A similar issue came up with converting RGB or RGBA values in c into string colours in edgecolors if these are the formats that the user provides; since there are more RGB colours than named colours in matplotlib, the conversion wouldn't work. The solution may require us to work with the backend or change the way matplotlib deals with edgecolors, but none of our group members have enough knowledge in graphics to modify the backend or rendering of colors. This issue was too complex for us to solve within the given time frame.

The implementation for linewidths would require us to create multiple checks, such as checking whether the marker list and linewidths array are the same length because they have a one-to-one relationship, which we need to be able to set the unfilled marker's linewidth to the line width that matches the value in rcParams['lines.linewidth']. Assuming that the size of the input for linewidths is the same as x.size or y.size, for each unfilled marker in the marker list we would need to replace the corresponding element in linewidths with the value in rcParams['lines.linewidth']. This will ensure that the linewidth of all unfilled markers is the value in rcParams['lines.linewidth'], which is consistent with matplotlib requirements. To implement this solution, we would need to restrict the user input to make sure that the size of both lists (marker and linewidths) is the same. In fact, for either solution we would be restricting the user input to make sure they give the right size input for the marker, edgecolors, and linewidths. For edgecolors to work with our current knowledge, it would require restricting the user input types to avoid float arrays. The pre-existing code doesn't have those restrictions, which is why we decided it would be better if we didn't restrict the users input to fix these issues, as the original code did not restrict the length of edgecolors and linewidths. Instead, there is a work around that

can resolve these bugs if the user provides the same edgecolor as color and linewidths to be 1 for unfilled markers.

Unit Tests (test_axes.py)

test_marker_list(self):

In order to limit the number of image comparison tests used several different functionalities are tested in this method in four different sub-plots (ax0, ax1, ax2, ax3).

1. On ax0 we test a list of length x.size of marker styles given as a list of style text shorthands.
2. On ax1 we test a shorter list to see if it cycles as expected
3. On ax2 we test linewidths and edgecolors on filled and unfilled markers
4. On ax3 we test a mixed list - a list with both text shorthands and MarkerStyle objects

test_marker_list_of_one(self):

Here we test using a list of marker styles with a length of 1. We use two subplots: on the first we test a list with a text shorthand; on the other a list with a MarkerStyle object

test_marker_nparray_list(self):

A test for (pseudo-) randomness and an nparray for the c parameter used with the marker lists - we are able to use an image comparison test since with the seed the numbers aren't really random and the image generated is the same each time.

test_marker_empty_list(self):

Here we test that an empty list gives an error and the correct error message.

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 feature-for-issue-11155.

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 by giving a list of marker styles in text shorthand or MarkerStyle objects.	The scatter method accepts a list of marker style shorthands or MarkerStyle objects through its 'marker' parameter.
02	The user wants to plot using the scatter method and just one marker style without putting it in a list.	The scatter method accepts a single marker style shorthand or MarkerStyle object.
03	The user wants to plot using the scatter method and just one marker style given in a list of length 1.	The scatter method accepts a list with a single marker style shorthand or MarkerStyle object in it.
04	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'].
05	The user wants to plot using edgecolors and linewidths that are visible on filled and unfilled markers.	The scatter method uses the provided edgecolors and linewidths on filled and unfilled markers alike.
06	The user wants to plot with edgecolors and linewidths but does not want them to appear on unfilled markers.	The scatter method uses the provided edgecolors and linewidths on filled and unfilled markers alike.

ID 01: Run this code in python3

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

np.random.seed(19680801)
n = 100
x, y = np.random.rand(2, n)
m = ['+', mmarkers.MarkerStyle('o', fillstyle='top'), 'o', mmarkers.MarkerStyle('s',
fillstyle='bottom')]

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

ID 02: Run this code in python3

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

np.random.seed(19680801)
n = 100
x, y = np.random.rand(2, n)
m = '+'
fig, (ax1, ax2) = plt.subplots(2)
ax1.scatter(x, y, marker=m)
ax2.scatter(x, y, marker=mmarkers.MarkerStyle(m))
plt.show()
```

ID 03: Run this code in python3

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

np.random.seed(19680801)
n = 100
x, y = np.random.rand(2, n)
m = '+'
fig, (ax1, ax2) = plt.subplots(2)
ax1.scatter(x, y, marker=[m])
ax2.scatter(x, y, marker=[mmarkers.MarkerStyle(m)])
plt.show()
```

ID 04: Run this code in python3

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

np.random.seed(19680801)
n = 100
x, y = np.random.rand(2, n)
plt.scatter(x, y)
plt.show()
```

ID 05: Run this code in python3

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

np.random.seed(19680801)
n = 100
x, y = np.random.rand(2, n)
m = ['+', '*', '1', 'o']
ec = ['green', 'blue', 'cyan', 'lime']
lw = [7, 2, 5, 10]
plt.scatter(x, y, c='red', edgecolors=ec, marker=m, linewidths=lw, s=200)
plt.show()
```

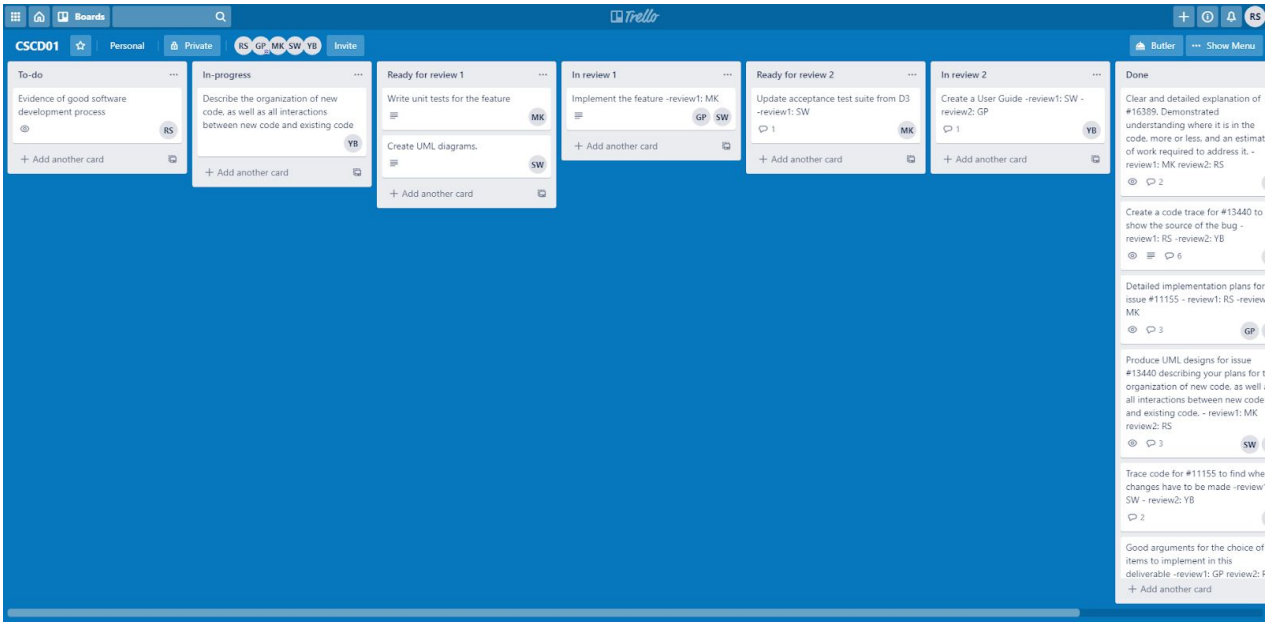
ID 06: Run this code in python3

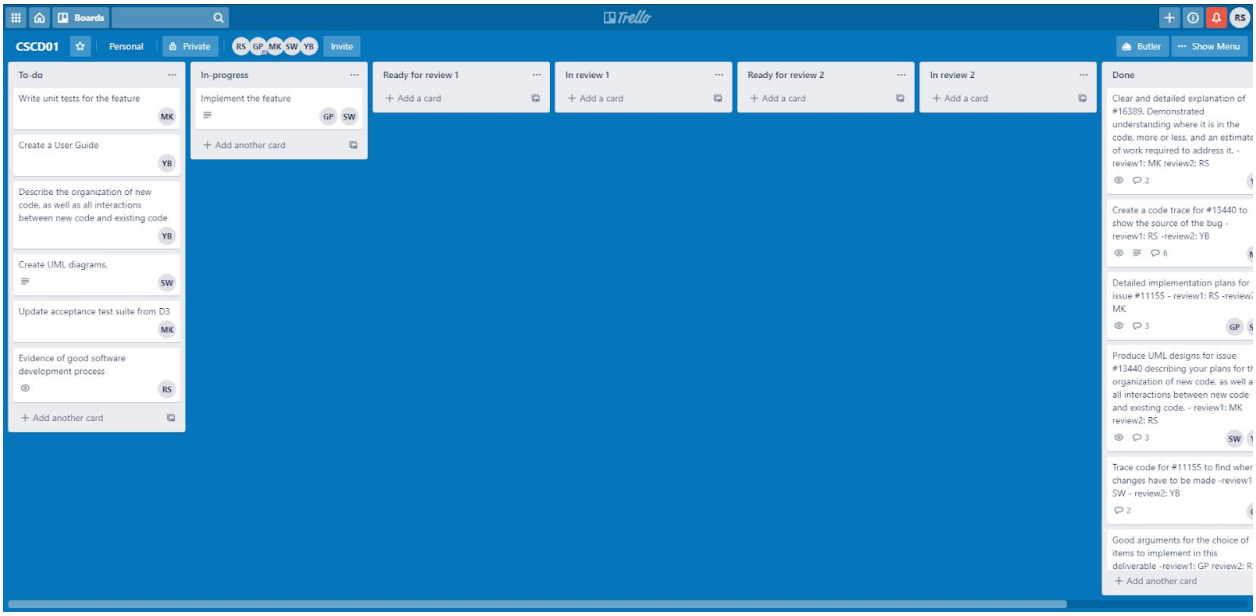
```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)
n = 100
x, y = np.random.rand(2, n)
m = ['+', '*', '1', 'o']
ec = ['red', 'blue', 'red', 'lime']
lw = [1, 2, 1, 10]
plt.scatter(x, y, c='red', edgecolors=ec, marker=m, linewidths=lw, s=200)
plt.show()
```

Software Development Process

As explained in Deliverable 1, our team decided to work with the Kanban development process. In order to achieve this we have been actively using Trello as our virtual Kanban board to track team and task progress. This is the invite link to the board: https://trello.com/invite/b/MP33R5lS/be30f092417270fb507cdae4910b5885/cscd01

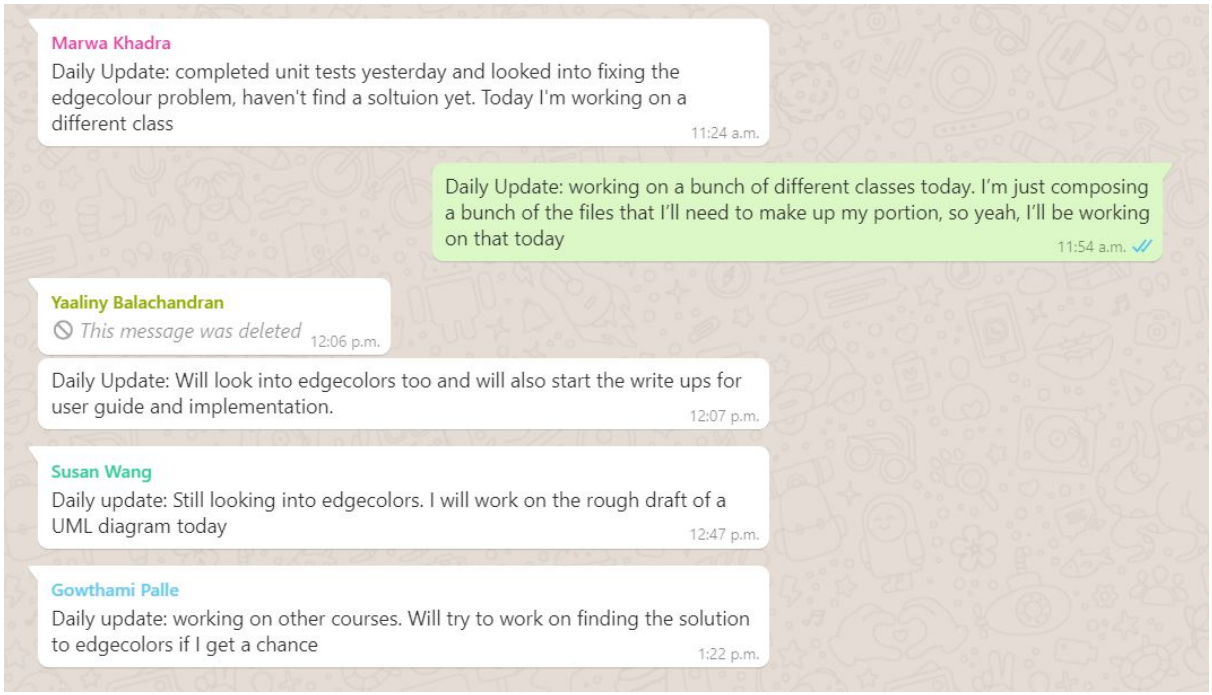
To show how the board typically progresses over some duration of work, here are a few snapshots of the Kanban board during our Deliverable 4 process.





As you can see, the Kanban board consists of seven columns, where three of them are the default “To-do”, “In-progress”, and “Done”. However, we added four columns to assist with our reviewing process as described at the beginning of the project. An item assigned to a member has their initials noted at the bottom right of the task, and items are moved along the columns according to their progress. Reviewers attach their initials to the task title to let other members know who is doing the review and whether they are performing review 1 or review 2. Any comments associated with the review are made on the task, so the task owner can read them and make changes based on any feedback that is given.

On top of using the board, each team member also updates the others asynchronously via WhatsApp in place of having daily meetings to discuss progress. An update is typically a summary of any tasks being worked on, and any completed work. An example of a daily update can be seen below.



In addition, we conduct two meetings each week to discuss any problems or updates in more detail. For each of these meetings, meeting minutes are noted down by Marwa for later use and review. These notes allow members to get a general summary of what was discussed, and what needs to be done for the next meeting to keep on track. All of the meeting minutes are stored here, and a link to view them is provided below:
https://drive.google.com/drive/folders/1muOZpbDDL-th3TuzMf5DIq0MsJjeVWiw?usp=sharing

This software process proved to be the right choice for our team as it has helped us improve team efficiency via consistent communication and keeping tasks organized and on track. The daily updates allowed each team member to be aware of what others were working on and to

provide any assistance needed. By ensuring no team member struggled with the work they were assigned, the development process went smoothly with minimal issues. The Trello board assisted in this process by providing a visual aid of the work needed to be done and the different stages of the process each task was in. This allowed team members to identify any bottlenecks in order to get them cleared up. By assigning work and keeping track, no work was duplicated and it allowed the team to stay organized. The board also provided an idea as to which tasks needed more focus and effort to get the project done in a timely manner. Ultimately, the Kanban software process kept us on-track with the project to ensure every task was completed properly whilst keeping the progress clean and organized for minimal stress on each team member.