```
# --- GETS HTML ON SCROLLABLE --- #

chrome_options = webdriver.ChromeOptions()
chrome_options.add_argument("--incognito")
chrome_options.add_argument('headless')
#ChromeDriverManager().install()
driver = webdriver.Chrome()
driver.get('https://nextgenstats.nfl.com/charts/list/pass/team/2022/')
SCROLL_PAUSE_TIME = 1
last_height = driver.execute_script("return document.body.scrollHeight")
scroll_limit = 100

count = 0
while True and count < scroll_limit:
    driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
    sleep(SCROLL_PAUSE_TIME)
    new_height = driver.execute_script("return document.body.scrollHeight")
    if new_height == last_height:
        break
    last_height = new_height
    count += 1
sleep(2)
html = driver.page_source
driver.close()
soup = BeautifulSoup(html, 'lxml')
```

The https://nextgenstats.nfl.com/charts/list/pass webpage is scrollable and does not load all at once. The above code opens the link, scrolls to the bottom, and then copies the HTML containing the metadata for each image. As a result, all of the webpage can be scraped at once.

```python
# --- FUNCTIONS --- #
def line_intersection(line1, line2):
    xdiff = (line1[0][0] - line1[1][0], line2[0][0] - line2[1][0])
    ydiff = (line1[0][1] - line1[1][1], line2[0][1] - line2[1][1])

    def det(a, b):
        return a[0] * b[1] - a[1] * b[0]

    div = det(xdiff, ydiff)
    d = (det(*line1), det(*line2))
    x = det(d, xdiff) / div
    y = det(d, ydiff) / div
    return x, y


def PointsInCircum(start, end, n=10):
    return np.array([(np.cos(x), np.sin(x)) for x in np.arange(start, end, (end-start)/n)])


def smooth_convex(data, dist, num):
    # Store curve points
    smoothed_data = []
    for i in range(1, len(data) + 1):
        # Rename desired points
        p, pl, pr = data[i % len(data)], data[i - 1], data[(i + 1) % len(data)]
        # Calculate distances between points
        deltas = np.array((pl, pr)) - p
        # Calculate the perpendicular slopes
        perp_slopes = -deltas[:, 0] / deltas[:, 1]
        # Calculate the ratio of the distance between the points to start the curve
        t = dist/np.sum(deltas**2, axis=1)**0.5
        if any(t > 0.5):  # Limit the curve to start no further than half the line
            t /= np.max(t)*2
        # Find the new end points of the straight lights
        new_ends = np.array((pl*t[0], pr*t[1])) + np.array((p*(1-t[0]), p*(1-t[1])))
        # Calculate the intersection of the perpendicular slopes at the new end points
        center = line_intersection((new_ends[0], new_ends[0] + np.array([1, perp_slopes[0]])),
                                   (new_ends[1], new_ends[1] + np.array([1, perp_slopes[1]])))
        # Calculate the radius from the center to new end points
        radius = sum((new_ends[0] - center)**2)**0.5

        # Calculate radians of the start and end of the curve
        rad_s, rad_e = np.arctan(perp_slopes)
        # Adjust depending on placement of the points
        if deltas[1, 1] < 0:
            rad_e += np.pi
        if deltas[0, 1] > 0:
            rad_s += np.pi
        if deltas[0, 1] > 0 and deltas[1, 1] > 0:
            rad_e += 2*np.pi
        # Find the points along the curve in the determined radian bounds
        curve = PointsInCircum(rad_s, rad_e, n=num) * radius + center
        # Store curves
        smoothed_data.append(curve)
    return np.reshape(smoothed_data, (-1, 2))

def truncate_colormap(cmap, minval=0.0, maxval=1.0, n=100):
    new_cmap = mcolors.LinearSegmentedColormap.from_list(
        'trunc({n},{a:.2f},{b:.2f})'.format(n=cmap.name, a=minval, b=maxval),
        cmap(np.linspace(minval, maxval, n)))
    return new_cmap
```

The above code was written by my friend Kyle Hassold. He can be reached at:

https://www.linkedin.com/in/kyle-hassold/

This code's function is to round the sharp edges on the topographic maps. As you can see by the produced images, it works. Fortunately, Kyle does a much better job at commenting his code than I do. Please refer to the image above (or to Kyle directly) for commentary on this portion of the project.

```
# --- COLLECT DATA FROM PICTURES --- #

data = []
dataDict = {}
start = []
end = []
chartsHTML = []
# --- TEXT FROM IMAGE --- #
f = open('nextGenHTML.txt', 'r')
lines = f.read()
parse = BeautifulSoup(lines, "lxml").prettify()
for r in re.finditer('<ul class="chart-stats-line">|<!-- -->\\n\s*<div class="chart-image">', parse):
    start.append(r)
for r in re.finditer('<p class="player-bio">|NFL Enterprises LLC', parse):
    end.append(r)
end.pop(0)

# --- GETS CHARTS --- #
for i in range(len(start)):
    chartsHTML.append(str([parse[start[i].span()[0]:end[i].span()[1]]]))
chartsHTML = chartsHTML[26:]
```

Now that the HTML has been collected, it needs to be parsed. The TEXT FROM IMAGE section separates all of the charts' HTML from the rest of the code. Each charts' HTML is stored separately from one another. The GETS CHARTS section removes the 26 playoff charts stored in the website, as they do not contain metadata like the rest.

```
for number in [z for z in range(len(chartsHTML)) if z != 374]:
    print(number)
    url = 'images/{0}.jpeg'.format(number)
    white_counts = []
    green_counts = []
    blue_counts = []
    red_counts = []
    white_qualifier = 0
    green_qualifier = 0
    blue_qualifier = 0
    red_qualifier = 0
    text = []
    name = []
    green_kmeans = 0
    white_kmeans = 0
    blue_kmeans = 0
    red_kmeans = 0

    # --- GETS METADATA --- #
    for r in re.finditer('\d+', chartsHTML[number]):
        text.append(int(r.group(0)))
    text = text[:5]
    # print(text)
    for r in re.finditer('<img alt=".+ Pass Chart" ', chartsHTML[number]):
        name.append(str(r.group(0)))
    # print(name)
    name = name[0].split('"')
    name = name[1].split(' ')
    week = name[3]
    name = name[0] + ' ' + name[1]
```

Now the images are cycled through by number. Their order in the HTML matches with the order that they are saved. The images themselves could not be scraped like the HTML as the NFL blocks these actions. Image 374 is skipped, as it is just blank. One image at a time, the numbers are removed from the HTML, followed by the player's name. This collected data is then separated stat-by-stat into an array.

```
# --- OPEN IMAGE IN OPENCV --- #
im = cv.imread(url, 1)
im = cv.cvtColor(im, cv.COLOR_BGR2RGB)
# plt.imshow(im)
w, h = im.shape[0], im.shape[1]

# --- RESHAPE INTO RECTANGLE AND UNWRAP --- #
src = np.float32([(115, 0), (485, 0), (-35, 325), (635, 325)]) #NW, NE, SW, SE
dst = np.float32([(0, 600), (456.857142857, 600), (0, 0), (456.857142857, 0)])
h, w = im.shape[:2]
M = cv.getPerspectiveTransform(src, dst)
im = cv.warpPerspective(im, M, (w, h), flags=cv.INTER_LINEAR)
im = cv.flip(im, 0)
im = im[0:600 ,0:457]
im = cv.rectangle(im, (0, 507), (20, 493), (0, 0, 0), -1)
im = cv.rectangle(im, (457, 507), (435, 493), (0, 0, 0), -1)
```

The image associated with the present number is opened. First the original images are cropped. They also have a warped perspective which must be taken care of via a keystone correction.

```
# --- GREEN THRESHOLDING --- #
if (text[0] - text[3]) != 0:
    green_lower = np.array([60, 80, 40])
    green_upper = np.array([130, 255, 75])
    green = cv.inRange(im, green_lower, green_upper)
    green_text = text[0] - text[3]

    # --- GREEN ELIMINATE NOISE --- #
    rows,cols = green.shape
    green_threshold = []
    for y in range(rows):
        for x in range(cols):
            if green[y, x] == 255:
                green_threshold.append([x, y])
    green_threshold = np.array(green_threshold)
    if len(green_threshold) != 0:
        db = DBSCAN(eps=4, min_samples=10).fit(green_threshold)
        labels = db.labels_
        green_dbscan = green_threshold[labels != -1]

        # --- GREEN K-MEANS CLUSTERING --- #
        if len(green_dbscan) != 0:
            green_kmeans = KMeans(n_clusters = green_text, init = "k-means++").fit(green_dbscan)
```

Now the more complex image analysis begins. Above is how the green dots (completions) are located. Other colors follow similar processes. First, a check is run to make sure that any green dots exist at all. A color threshold is then run on the image, selecting all green pixels. Green pixels are defined as any pixel with an RGB value such that red is 60-130, green is 80-255, and blue is 40-75. A clustering algorithm is then run on all the pixels. It detects groups of green pixels, eliminating noise and extraneous pixels. Next, using the meta data collected, a k-nearest neighbors algorithm is run where k is the total number of completed passes minus the touchdowns. Colors that need to be double checked due to a potentially incorrect k value then undergo a variance calculation, however green does not need this.

```
    if green_kmeans != 0:
        for i in range(len(green_kmeans.cluster_centers_)):
            data.append((name, week, green_kmeans.cluster_centers_[:, 0][i], green_kmeans.cluster_centers_[:, 1][i], 'Complete'))
    if white_kmeans != 0:
        for i in range(len(white_kmeans.cluster_centers_)):
            data.append((name, week, white_kmeans.cluster_centers_[:, 0][i], white_kmeans.cluster_centers_[:, 1][i], 'Incomplete'))
    if blue_kmeans != 0:
        for i in range(len(blue_kmeans.cluster_centers_)):
            data.append((name, week, blue_kmeans.cluster_centers_[:, 0][i], blue_kmeans.cluster_centers_[:, 1][i], 'Touchdown'))
    if red_kmeans != 0:
        for i in range(len(red_kmeans.cluster_centers_)):
            data.append((name, week, red_kmeans.cluster_centers_[:, 0][i], red_kmeans.cluster_centers_[:, 1][i], 'Interception'))

# plt.gca().invert_yaxis()
for item in data:
    if (item in dataDict):
        dataDict[item] += 1
    else:
        dataDict[item] = 1
```

After all the colors undergo their data collection processes, they can optionally be plotted on a scatter plot. The data is then stored for topographic mapping.

```
for nme in names:
    file1 = open("DictionaryDataPickle.txt","rb")
    DataDict = pickle.load(open("DictionaryDataPickle.txt", "rb"))
    file1.close()
    df = pd.Series(DataDict).reset_index()
    df.columns = ['name', 'week', 'x', 'y', 'type', 'instances']

    df = df[(df['name'] == nme)].sort_values(by=['name', 'type', 'x', 'y'])
    df['y'] = (-(df['y'] * (65/600)) + 65) - 10
    df['x'] = (df['x'] * (53.333333/457)) - (53.333333/2)

    colors = {'Complete': 'limegreen', 'Incomplete': 'white', 'Touchdown': 'royalblue', 'Interception': 'red'}
    pic = plt.imread('images/field.png')
    fig, ax = plt.subplots()
    ax.imshow(pic, extent=[-(53.333333/2), (53.333333/2), -10, 55])
    ax.scatter(df['x'], df['y'], c = df['type'].map(colors))
    ax.set_title(df['name'].iloc[0])
    path = 'test/' + str(count) + 'a.png'
    ax.figure.savefig(path)

    layers = list(range(3, 19, 2))
    j = 0
    dist = 1.75
    cmap = cl.LinearSegmentedColormap.from_list("", ["darkgreen", "limegreen", 'palegreen', 'plum', "mediumorchid", "darkmagenta", 'indigo'])
    colors = cmap(np.linspace(0, 1, len(layers)))

    pic = plt.imread('images/field.png')
    fig, ax = plt.subplots()
    ax.imshow(pic, extent=[-(53.333333/2), (53.333333/2), -10, 55])
    ax.set_title(df['name'].iloc[0])
```

The names array stores the names of any quarterbacks whose mappings you wish to display. First, the collected data is opened and stored in a dataframe. The X and Y data is scaled and reflected across a custom y-axis (the line of scrimmage). The same is done with a blank image of the next gen statistics' football field background, which is to double as a background for these new graphics. A custom color mapping is set as the map's title is set to the player's name.

```
for i in list(range(0, len(layers), 1)):
    locations = df[['x', 'y']]
    X = locations
    #--------------------------------------------------------#
    db = DBSCAN(eps=dist, min_samples=layers[i]).fit(X)
    #--------------------------------------------------------#
    clusters = pd.DataFrame(db.fit_predict(locations))
    labels = db.labels_
    unique_labels = set(labels)
    core_samples_mask = np.zeros_like(labels, dtype=bool)
    core_samples_mask[db.core_sample_indices_] = True
    for k in unique_labels:
        if k != -1:
            class_member_mask = labels == k
            xy = X[class_member_mask]
            if len(xy) >= layers[i]:
                hull = ConvexHull(xy)
                if hull.volume > 0.0015:
                    xy_smooth = smooth_convex(xy.iloc[hull.vertices].to_numpy(), 1, 10)
                    ax.fill(xy_smooth[:, 0], xy_smooth[:, 1], c = colors[i], alpha=1)
path = 'test/' + str(count) + 'b.png'
ax.figure.savefig(path)
count += 1
```

Finally, the data is graphed topographically. A clustering algorithm is run at each layer, with increasing density. A convex hull is calculated on each hull. Convex hulls take a cluster of points and form a perimeter around them, turning them into a shape. Kyle's algorithm rounds the sharp vertexes, giving the shapes a smooth look. They layers are plotted as they are calculated, from largest (least dense) to smallest (most dense). Finally, the images are stored and the charts are completed.