Published in Towards Data Science · Follow

Animations with Matplotlib

Using the matplotlib library to create some interesting animations.

Rain Simulation with Matplotlib

the years, climate change over the past decade, seasonalities and trends since we can then see how a particular parameter behaves with time.

The above image is a **simulation of Rain** and has been achieved with Matplotlib library which is fondly known as the **grandfather of python visualization packages.** Matplotlib simulates raindrops on a surface by animating the scale and opacity of 50 scatter points. Today Python boasts of a large number of powerful visualization tools like Plotly, Bokeh, Altair to name a few. These libraries are able to achieve state of the art animations and interactiveness. Nonetheless, the aim of this article is to highlight one aspect of this library which isn't explored much and that is **Animations** and we are going to look at some of the ways of doing that.

. . .

Overview

Matplotlib is a Python 2D plotting library and also the most popular one. Most of the people start their Data Visualisation journey with Matplotlib. One can generate plots, histograms, power spectra, bar charts, error charts, scatterplots, etc easily with matplotlib. It also integrates seamlessly with libraries like Pandas and Seaborn to create even more sophisticated visualizations.

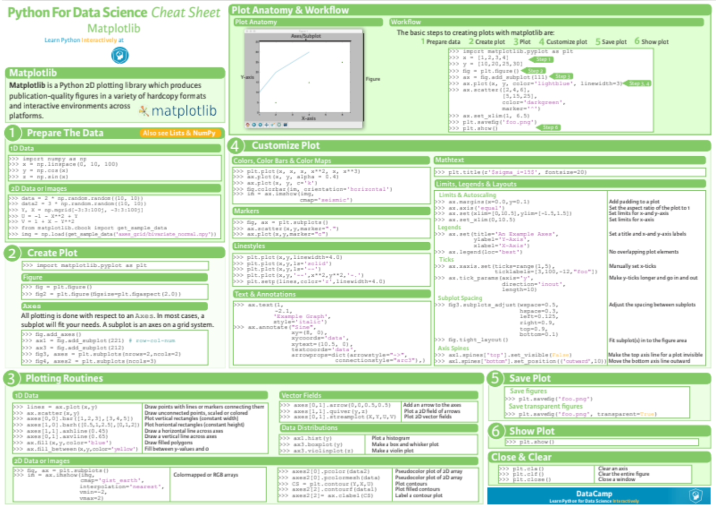
Some of the nice features of matplotlib are:

- It is designed like MATLAB hence switching between the two is fairly easy.
- Comprises of a lot of rendering backends.
- It can reproduce just about any plots(with a bit of effort).
- Has been out there for over a decade, therefore, boasts of a huge user base.

However, there are also areas where Matplotlib doesn't shine out so much and lags behind its powerful counterparts.

- Poor support for web and interactive graphs.
- Often slow for large & complicated data.

As for a refresher here is a Matplotlib Cheatsheet from **Datacamp** which you can go through to brush up your basics.



Python For Data Science *Cheat Sheet*

Matplotlib

Learn Python Interactively at

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.

matplotlib

1 Prepare The Data

Also see Lists & NumPy

1D Data

```
>>> import numpy as np
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))
>>> data2 = 3 * np.random.random((10, 10))
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample_data
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

2 Create Plot

```
>>> import matplotlib.pyplot as plt
```

Figure

```
>>> fig = plt.figure()
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

Axes

All plotting is done with respect to an Axes. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()
>>> ax1 = fig.add_subplot(221) # row-col-num
>>> ax3 = fig.add_subplot(212)
>>> fig3, axes = plt.subplots(nrows=2,ncols=2)
>>> fig4, axes2 = plt.subplots(ncols=3)
```

3 Plotting Routines

1D Data

`>>> lines = ax.plot(x,y)`	Draw points with lines or markers connecting them
`>>> ax.scatter(x,y)`	Draw unconnected points, scaled or colored
`>>> axes[0,0].bar([1,2,3],[3,4,5])`	Plot vertical rectangles (constant width)
`>>> axes[1,0].barh([0.5,1,2.5],[0,1,2])`	Plot horiontal rectangles (constant height)
`>>> axes[1,1].axhline(0.45)`	Draw a horizontal line across axes
`>>> axes[0,1].axvline(0.65)`	Draw a vertical line across axes
`>>> ax.fill(x,y,color='blue')`	Draw filled polygons
`>>> ax.fill_between(x,y,color='yellow')`	Fill between y-values and 0

2D Data or Images

```
>>> fig, ax = plt.subplots()
>>> im = ax.imshow(img,
                   cmap='gist_earth',
                   interpolation='nearest',
                   vmin=-2,
                   vmax=2)
```

Colormapped or RGB arrays

`>>> axes2[0].pcolor(data2)`	Pseudocolor plot of 2D array
`>>> axes2[0].pcolormesh(data)`	Pseudocolor plot of 2D array
`>>> CS = plt.contour(Y,X,U)`	Plot contours
`>>> axes2[2].contourf(data1)`	Plot filled contours
`>>> axes2[2]= ax.clabel(CS)`	Label a contour plot

Vector Fields

`>>> axes[0,1].arrow(0,0,0.5,0.5)`	Add an arrow to the axes
`>>> axes[1,1].quiver(y,z)`	Plot a 2D field of arrows
`>>> axes[0,1].streamplot(X,Y,U,V)`	Plot 2D vector fields

Data Distributions

`>>> ax1.hist(y)`	Plot a histogram
`>>> ax3.boxplot(y)`	Make a box and whisker plot
`>>> ax3.violinplot(z)`	Make a violin plot

Plot Anatomy & Workflow

Plot Anatomy

Workflow

The basic steps to creating plots with matplotlib are:

1 Prepare data 2 Create plot 3 Plot 4 Customize plot 5 Save plot 6 Show plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4]                        Step 1
>>> y = [10,20,25,30]
>>> fig = plt.figure()                   Step 2
>>> ax = fig.add_subplot(111)            Step 3
>>> ax.plot(x, y, color='lightblue', linewidth=3)   Step 3, 4
>>> ax.scatter([2,4,6],
               [5,15,25],
               color='darkgreen',
               marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png')
>>> plt.show()                           Step 6
```

4 Customize Plot

Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
                   cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,y,'--',x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

Text & Annotations

```
>>> ax.text(1,
            -2.1,
            'Example Graph',
            style='italic')
>>> ax.annotate("Sine",
                xy=(8, 0),
                xycoords='data',
                xytext=(10.5, 0),
                textcoords='data',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="arc3"),)
```

Mathtext

```
>>> plt.title(r'$sigma_i=15$', fontsize=20)
```

Limits, Legends & Layouts

Limits & Autoscaling	
`>>> ax.margins(x=0.0,y=0.1)`	Add padding to a plot
`>>> ax.axis('equal')`	Set the aspect ratio of the plot to 1
`>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5])`	Set limits for x-and y-axis
`>>> ax.set_xlim(0,10.5)`	Set limits for x-axis
Legends	
`>>> ax.set(title='An Example Axes', ylabel='Y-Axis', xlabel='X-Axis')`	Set a title and x-and y-axis labels
`>>> ax.legend(loc='best')`	No overlapping plot elements
Ticks	
`>>> ax.xaxis.set(ticks=range(1,5), ticklabels=[3,100,-12,"foo"])`	Manually set x-ticks
`>>> ax.tick_params(axis='y', direction='inout', length=10)`	Make y-ticks longer and go in and out
Subplot Spacing	
`>>> fig3.subplots_adjust(wspace=0.5, hspace=0.3, left=0.125, right=0.9, top=0.9, bottom=0.1)`	Adjust the spacing between subplots
`>>> fig.tight_layout()`	Fit subplot(s) in to the figure area
Axis Spines	
`>>> ax1.spines['top'].set_visible(False)`	Make the top axis line for a plot invisible
`>>> ax1.spines['bottom'].set_position(('outward',10))`	Move the bottom axis line outward

5 Save Plot

Save figures

```
>>> plt.savefig('foo.png')
```

Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

6 Show Plot

```
>>> plt.show()
```

Close & Clear

`>>> plt.cla()`	Clear an axis
`>>> plt.clf()`	Clear the entire figure
`>>> plt.close()`	Close a window

DataCamp
Learn Python for Data Science Interactively

Animations

Matplotlib's `animation` base class deals with the animation part. It provides a framework around which the animation functionality is built. There are two main interfaces to achieve that using:

`FuncAnimation` makes an animation by repeatedly calling a function *`func`*.

`ArtistAnimation:` Animation using a fixed set of `Artist` objects.

However, out of the two, **FuncAnimation** is the most convenient one to use. You can read more about them in the documentation since we will only concern ourselves with the `FuncAnimation` tool.

Requirements

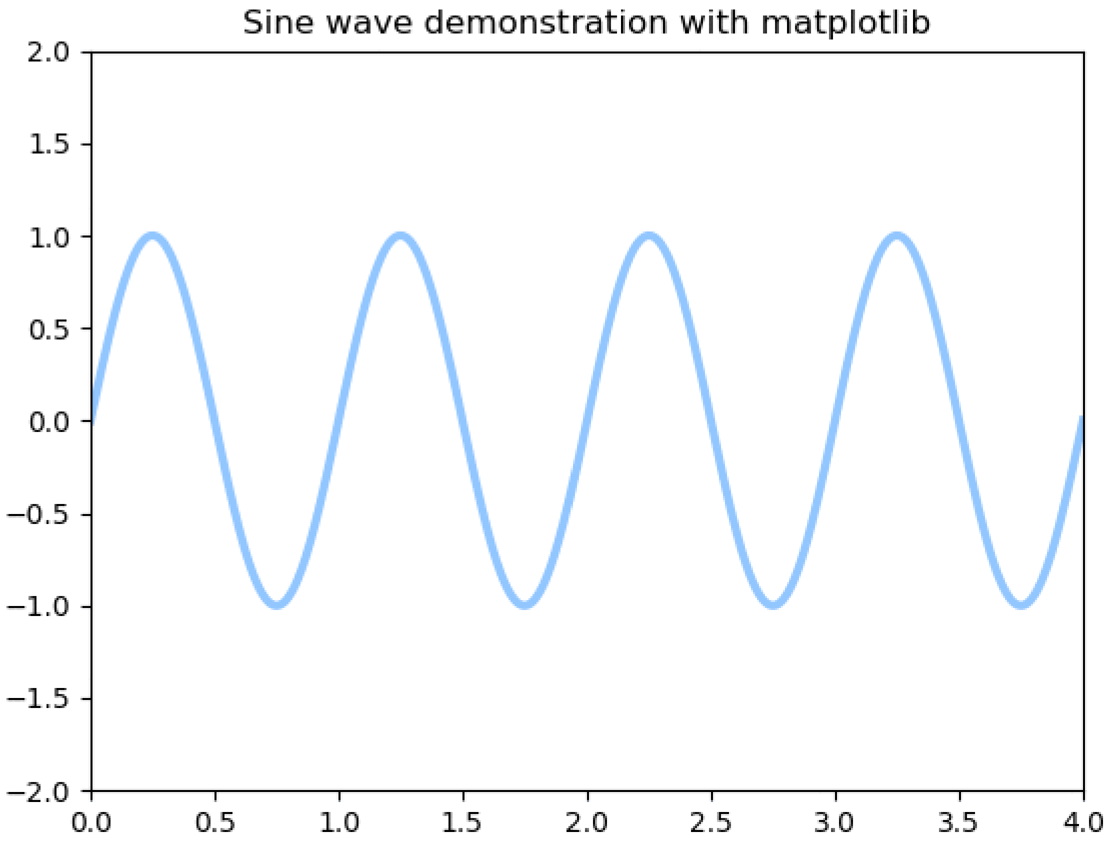
- Modules including `numpy` and `matplotlib` should be installed.
- To save the animation on your system as mp4 or gif, `ffmpeg` or `imagemagick` is required to be installed.

Once ready, we can begin with our first basic animation in the Jupyter Notebooks. The code for this article can be accessed from the associated **Github Repository** or you can view it on my binder by clicking the image below.

tutorial. Let’s first see the output and then we shall break down the code to understand what’s going under the hood.

```python
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation
plt.style.use('seaborn-pastel')


fig = plt.figure()
ax = plt.axes(xlim=(0, 4), ylim=(-2, 2))
line, = ax.plot([], [], lw=3)

def init():
    line.set_data([], [])
    return line,
def animate(i):
    x = np.linspace(0, 4, 1000)
    y = np.sin(2 * np.pi * (x - 0.01 * i))
    line.set_data(x, y)
    return line,

anim = FuncAnimation(fig, animate, init_func=init,
                               frames=200, interval=20, blit=True)


anim.save('sine_wave.gif', writer='imagemagick')
```

Moving Sine Wave.py hosted with ❤ by **GitHub** **view raw**



- In lines(7–9), we simply create a figure window with a single axis in the figure. Then we create our empty line object which is essentially the one to be modified in the animation. The line object will be populated with data later.
- In lines(11–13), we create the `init` function that will make the animation happen. The init function initializes the data and also sets the axis limits.
- In lines(14–18), we finally define the animation function which takes in the frame number(i) as the parameter and creates a sine wave(or any other animation) which a shift depending upon the value of i. This function here returns a tuple of the plot objects which have been modified which tells the animation framework what parts of the plot should be animated.
- In line **20**, we create the actual animation object. The `blit` parameter ensures that

. . .

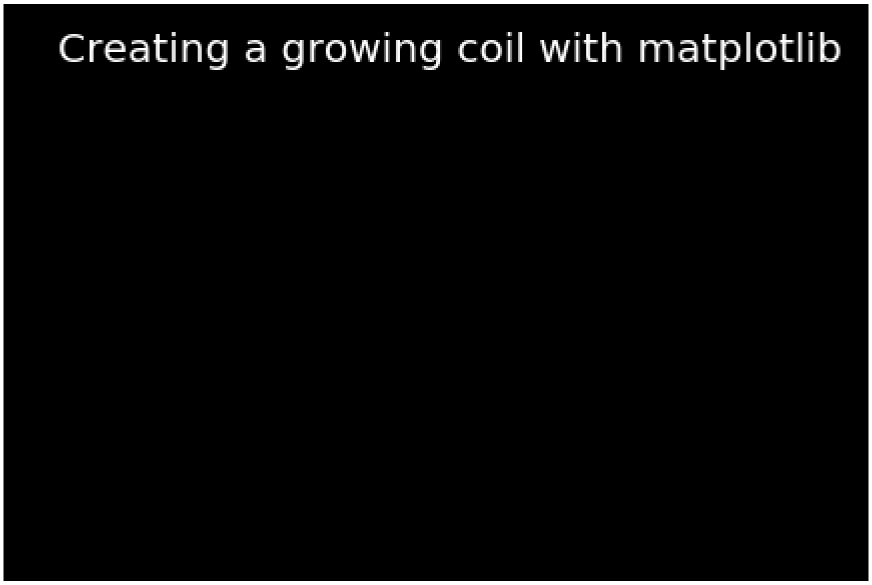
A Growing Coil

Similarly, there is a nice example of creating shapes at GeeksforGeeks. Let's now create a moving coil that slowly unwinds, with the help of `animation` class of matplotlib. The code is quite similar to the sine wave plot with minor adjustments.

```

import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
plt.style.use('dark_background')

fig = plt.figure()
ax = plt.axes(xlim=(-50, 50), ylim=(-50, 50))
line, = ax.plot([], [], lw=2)

# initialization function
def init():
        # creating an empty plot/frame
        line.set_data([], [])
        return line,

# lists to store x and y axis points
xdata, ydata = [], []

# animation function
def animate(i):
        # t is a parameter
        t = 0.1*i

        # x, y values to be plotted
        x = t*np.sin(t)
        y = t*np.cos(t)

        # appending new points to x, y axes points list
        xdata.append(x)
```

```
plt.title('Creating a growing coil with matplotlib!')
# hiding the axis details
plt.axis('off')

# call the animator
anim = animation.FuncAnimation(fig, animate, init_func=init,
                                                        frames=500, interval=20, blit=True

# save the animation as mp4 video file
anim.save('coil.gif',writer='imagemagick')
```

coil.py hosted with ❤ by **GitHub** **view raw**



. . .

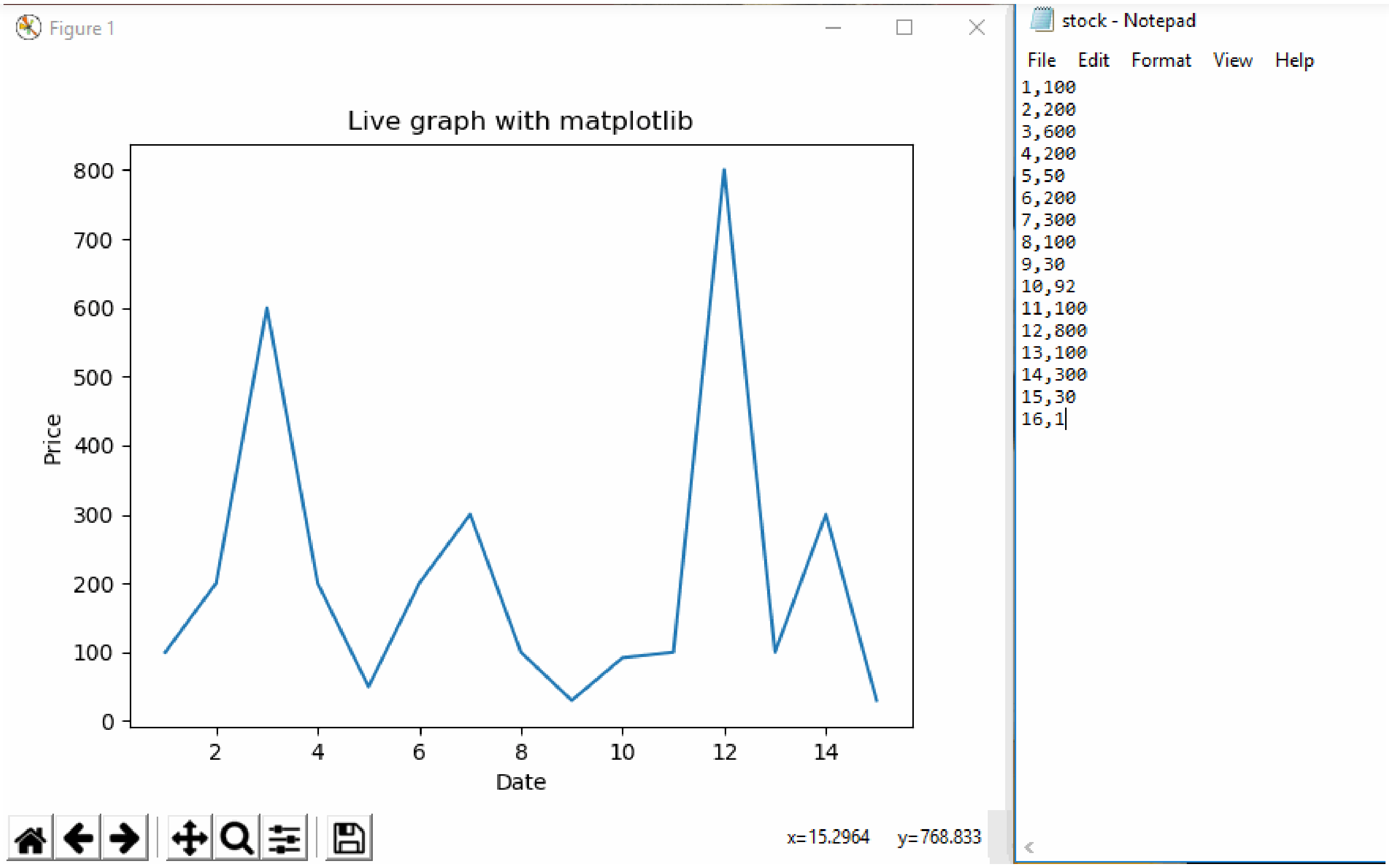
Live Updating Graphs

Live updating graphs come in handy when plotting dynamic quantities like stock data, sensor data or any other time-dependent data. We plot a base graph which automatically gets updated as more data is fed into the system. This example has been taken from **sentdex**. Be sure to visit this youtube channel for some awesome tutorials.

```python
import matplotlib.animation as animation


fig = plt.figure()
#creating a subplot
ax1 = fig.add_subplot(1,1,1)

def animate(i):
    data = open('stock.txt','r').read()
    lines = data.split('\n')
    xs = []
    ys = []

    for line in lines:
        x, y = line.split(',') # Delimiter is comma
        xs.append(float(x))
        ys.append(float(y))


    ax1.clear()
    ax1.plot(xs, ys)

    plt.xlabel('Date')
    plt.ylabel('Price')
    plt.title('Live graph with matplotlib')


ani = animation.FuncAnimation(fig, animate, interval=1000)
plt.show()
```

live_graph.py hosted with ❤ by **GitHub** **view raw**

Now, open the terminal and run the python file. You will get a graph like the one below which automatically updates as follows:



```
1,100
2,200
3,600
4,200
5,50
6,200
7,300
8,100
9,30
10,92
11,100
12,800
13,100
14,300
15,30
16,1
```

Here interval is 1000 milliseconds or one second.

. . .

Animation on a 3D plot

Creating 3D graphs is common but what if we can animate the angle of view of those graphs. The idea is to change the camera view and then use every resulting image to create an animation. There is a nice section dedicated to it at The Python Graph Gallery.

Create a folder called **volcano** in the same directory as the notebook. All the images will be stored in this folder which will be then used in the animation.

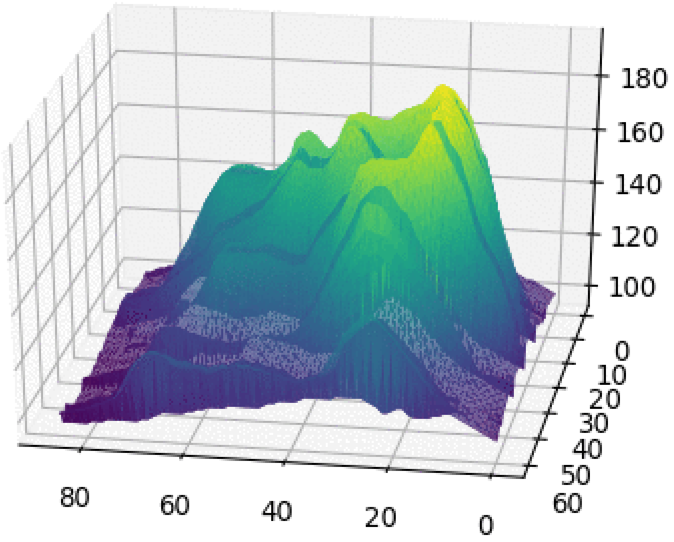
```
# library
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
```

```python
data = pd.read_csv(url)

# Transform it to a long format
df=data.unstack().reset_index()
df.columns=["X","Y","Z"]

# And transform the old column name in something numeric
df['X']=pd.Categorical(df['X'])
df['X']=df['X'].cat.codes

# We are going to do 20 plots, for 20 different angles
for angle in range(70,210,2):

# Make the plot
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_trisurf(df['Y'], df['X'], df['Z'], cmap=plt.cm.viridis, linewidth=0.2)

    ax.view_init(30,angle)

    filename='Volcano/Volcano_step'+str(angle)+'.png'
    plt.savefig(filename, dpi=96)
    plt.gca()
```

3d_animation.py hosted with ❤ by **GitHub** **view raw**

This will create multiple PNG files in the Volcano folder. Now, use ImageMagick to transform them into animation. Open Terminal and navigate to the Volcano folder and enter the following command:

```
convert -delay 10 Volcano*.png animated_volcano.gif
```



· · ·

Animations using Celluloid Module

Celluloid is a Python module that simplifies the process of creating animations in matplotlib. This library creates a matplotlib figure and creates a `Camera` from it. It then reuses figure and after each frame is created, take a snapshot with the camera. Finally, an animation is created with all the captured frames.

Installation

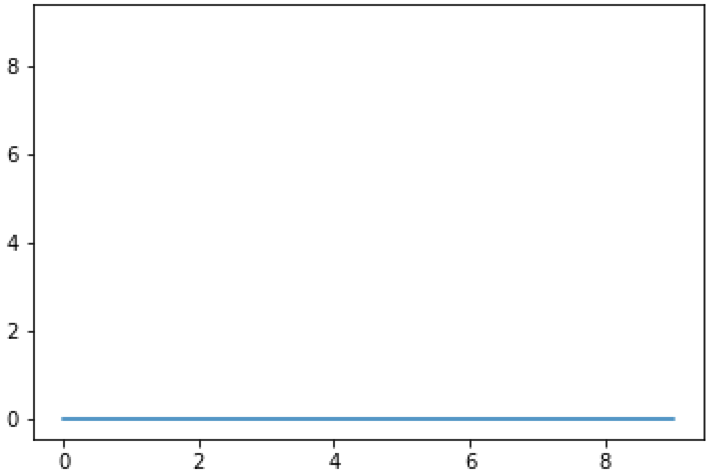
```
pip install celluloid
```

Here are a few examples using the Celluloid module.

Minimal

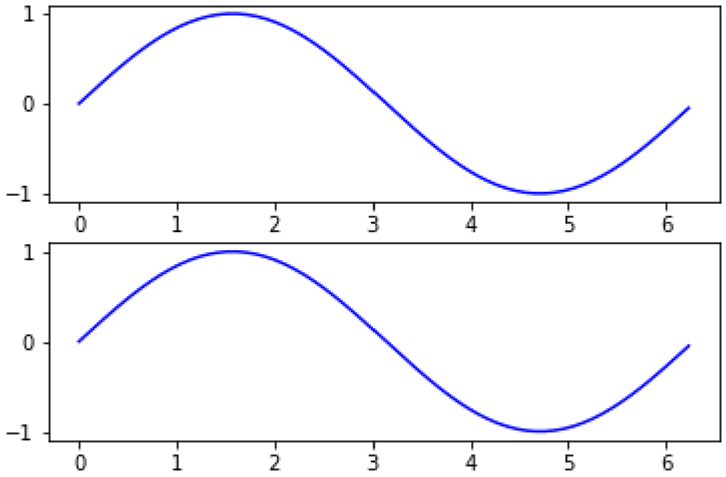
```python
camera = Camera(fig)
for i in range(10):
    plt.plot([i] * 10)
    camera.snap()
animation = camera.animate()
animation.save('celluloid_minimal.gif', writer = 'imagemagick')
```

celluloi_minimal.py hosted with ❤ by **GitHub** **view raw**



Subplots

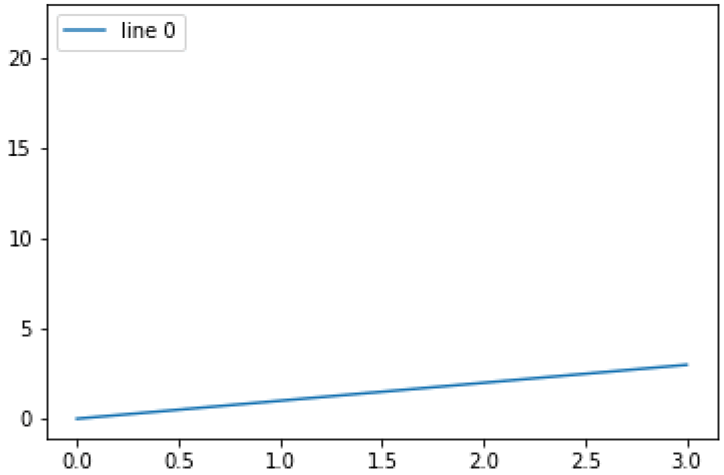
```python
import numpy as np
from matplotlib import pyplot as plt
from celluloid import Camera

fig, axes = plt.subplots(2)
camera = Camera(fig)
t = np.linspace(0, 2 * np.pi, 128, endpoint=False)
for i in t:
    axes[0].plot(t, np.sin(t + i), color='blue')
    axes[1].plot(t, np.sin(t - i), color='blue')
    camera.snap()

animation = camera.animate()
```



Legends

```python
import matplotlib
from matplotlib import pyplot as plt
from celluloid import Camera

fig = plt.figure()
camera = Camera(fig)
for i in range(20):
    t = plt.plot(range(i, i + 5))
    plt.legend(t, [f'line {i}'])
    camera.snap()
animation = camera.animate()
animation.save('celluloid_legends.gif', writer = 'imagemagick')
```

celluloid_legends.py hosted with ❤ by **GitHub** **view raw**



. . .

Wrap Up

Animations help to highlight certain features of the visualisation which otherwise cannot be communicated easily with static charts. Having said that it is also important to keep in mind that unnecessary and overuse of visualisations can sometimes complicate things. Every feature in data visualisation should be used judiciously to have the best impact.