Matplotlib

21

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

LO 1	Use Matplotlib library in two distinct ways: (1) By using helper functions and (2) Using the underlying classes
LO 2	Get insight of (1) the Figure class and (2) Axes class
LO 3	Grasp the concept of "containers" and "primatives"
LO 4	Use NumPy methods np.linspace and np.vectorize to create a plot
LO 5	Create subplots using subplot() and subplots()
LO 6	Create subplots using subplot2grid() method
LO 7	Understand the difference between add_axes() and add_subplot()
LO 8	Use log scales in Matplotlib

Additional LO in Beyond text section

LO 9	Understand 3-D plotting
LO 10	Use Matplotlib for animation and using "callback functions"
LO 11	Save a Matplotlib animation as a mp4 movie, using FFMpegWriter

21.1 Introduction

Matplotlib is a very extensive 2D plotting library.

Most introductory material on Matplotlib starts drawing of a single plot and then progresses to drawing multiple plots (or subplots) on a canvas or figure.

It is very easy to draw a single plot in Matplotlib because in Matplotlib there is a module pyplot. This module pyplot has a helper function plot(). So you can easily draw a plot using this function. But using a helper function does not tell you how the Matplotlib library actually works.

So this chapter does not start with the "helper" functions, rather it starts with understanding how plotting actually takes place in Matplotlib.

21.2 Basic Concepts (Part 1)

This section covers the very basic concepts used in Matplotlib for plotting. Topics like Figure and Axes class and concepts of "containers" and "primatives" are introduced here.

21.2.1 An Analogy to Understand Matplotlib

To understand how the pyplot module in Matplotlib is organized, consider an analogy:

- Consider the pyplot module (conventionally called plt) as a photo-frame. A photo-frame has not only the visible area for display but other parts also.
- Consider now the visible area or display area of the photo-frame and you may represent it by a class called Figure class. Conventionally when you create an object of this Figure class, you call it fig. So when you see a variable named fig in code, you can presume that it is an object of Figure class and represents the canvas on which the drawing is done.
- Consider individual subplots or parts of the visible area, which are shown as 1 to 4 in Figure 21.1. Each of these subplots are instances of class Axes and are conventionally represented by name ax. Further note that these subplots are arranged as rows and columns. So these subplots are in the form of a matrix of type m × n (where m stands for the number of rows and n is the number of columns). So ax is 2D (One dimension for rows and other for column). Of course in the special case when there is a single row, i.e., m = 1, then the ax variable will be 1D.
- Within these subplots, there are x- and y-axes. These are instances of Axis class (Note Axis class is for the axis while the Axes class is for the sub-plot. Note the difference in spelling.). Further there may be other objects like lines, rectangles, etc. which may be drawn on these subplots.

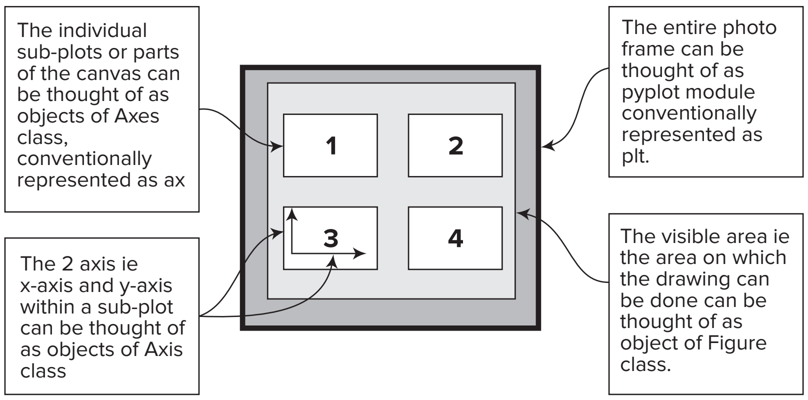

The individual sub-plots or parts of the canvas can be thought of as objects of Axes class, conventionally represented as ax
The 2 axis ie x-axis and y-axis within a sub-plot can be thought of as objects of Axis class
1
2
3
4
The entire photo frame can be thought of as pyplot module conventionally represented as plt.
The visible area ie the area on which the drawing can be done can be thought of as object of Figure class.

Figure 21.1 Layout of the "canvas" as plt; the "visible area" as Figure object; the "subplots" as Axes objects and the "x and y axis" as Axis objects

This chapter has been deliberately started from creating subplots first and using the plot() helper function later on in the chapter. The reason is that when this method of learning (where classes and objects of Matplotlib are understood before using the so called “helper functions”) is used, a student know that a drawing in Matplotlib consists of objects and each part of the drawing (be it the entire canvas, or the individual subplots or the x-axis/y-axis or the tick marks or lines, or other items like rectangles) are all objects. However on the other hand, if the so called “helper functions” are introduced directly without giving an insight into the object oriented nature of the Matplotlib library, then the student may not be able to effectively use the library.

21.2.2 The Figure Class and the Axes Class

The Matplotlib library has a Figure class (conventionally represented by fig object) and the Axes class (conventionally represented by ax object(s))

Note: On Jupyter notebook, there is a magic command: %matplotlib inline.
If you use this command in a Jupyter cell, then the resulting plot will appear in the notebook

The following points are noteworthy:

- When using the Matplotlib library, you import the pyplot module and traditionally it is called plt.
- In Matplotlib, the overall container or window or page on which the drawing is done is conventionally called fig and is an instance of Figure class.
- The Figure class is actually a container which can contain a single plot or many subplots.
- These plots or subplots are conventionally called ax and are instances of Axes class.
- It is this ax object on which all the drawings are done.
- So the Figure class is not a canvas, rather it is just a container. It is actually the ax object(s) on which the drawing is done.

This is shown in Figure 21.2.

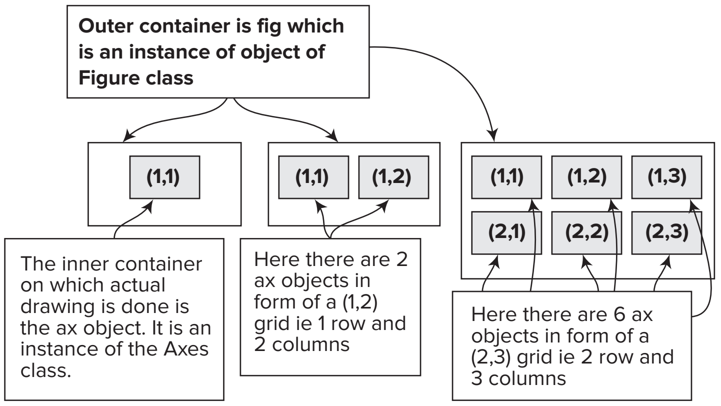

Outer container is fig which is an instance of object of Figure class
(1,1)
(1,1)
(1,2)
(1,1)
(1,2)
(1,3)
(2,1)
(2,2)
(2,3)
The inner container on which actual drawing is done is the ax object. It is an instance of the Axes class.
Here there are 2 ax objects in form of a (1,2) grid ie 1 row and 2 columns
Here there are 6 ax objects in form of a (2,3) grid ie 2 row and 3 columns

FIGURE 21.2 Three types of grids namely (a) 1 x 1, (b) 1 x 2 and (c) 2 x 3 are shown in the figure.

21.2.3 Method matplotlib.pyplot.subplot()

The matplotlib.pyplot module has a method subplots(). The method takes a number of attributes, which all have default values. The 2 most important attributes are (1) nrows: which specifies the number of rows in the grid of subplots, and (2) ncols: which specifies the number of columns in the subplots. The default values for both nrows and ncols is 1.

This method returns 2 objects (1) a Figure object, and (2) an array of Axes objects. So you can give the return value of this method to 2 variables. If you assign the return value of this method to 2 variables then the first variable on the left hand sight (LHS) of the assignment statement will get an Axes object while the second variable on the LHS of the assignment statement will get an "array of Axes objects". Don't worry if this is a bit confusing, it will become clear once you see some example code.

The signature of this method on Jupyter is given below:

```
1   import matplotlib.pyplot as plt
2   ?plt.subplots
```

```
3   # OUTPUT (Modified for clarity)
4   Signature: plt.subplots(nrows=1, ncols=1, sharex=False, sharey=False,
5   squeeze=True, subplot_kw=None, gridspec_kw=None, **fig_kw)
6   Parameters
7   nrows, ncols : int, optional, default: 1->Number of rows/columns of the subplot
8   grid.
9   sharex, sharey : bool or {'none', 'all', 'row', 'col'}, default: False
10  squeeze : bool, optional, default: True
11  Returns
12  fig : :class:`matplotlib.figure.Figure` object
13  ax : Axes object or array of Axes objects.-> ax can be either a single:class:
14  `matplotlib.axes.Axes` object or an array of Axes objects if more than one
15  subplot was created.
```

Explanation:

Line7-8: This line specifies the 2 attributes nrows and ncols. As mentioned earlier, the default values for both is 1.

Line9-10: This attribute specifies whether the subplots will share a common x-axis and y-axis. (This will be clear from example).

Line12-15: These lines specify the "return values" of the method. Note that normally most Python functions return a single object. However this method returns 2 objects namely (1) fig (which is an instance of Figure object) and (2) ax (which is an instance of Axes class). The important thing to note is that **the function returns two distinct objects** namely a Figure object and an **array of Axes objects** (Again note that it is not a single Axes object but an "array" of Axes objects). Further note that the Axes object returned may be a 1D array or a 2D array depending upon the way the plots are created.

If you create a single row say (1,n) subplots, then the array returned by the subplot() method will be a 1D array consisting of 1 row and n columns. But if the subplots created are of type (m, n) where m > 1, then the array returned will be a 2D array consisting of m rows and n columns. You can confirm this by checking the shape of the array returned.

This will become clear from the following code, where three subplots in the form of 1 × 3 grid, i.e., 1 row and 3 columns have been created:

```
import matplotlib.pyplot as plt
#Create subplots in 1 row and 3 column
fig, ax = plt.subplots(1,3)
print('shape->',ax.shape)# shape-> (3,)
plt.show()
```

The output is shown in Figure 21.3.

shape-> (3,)

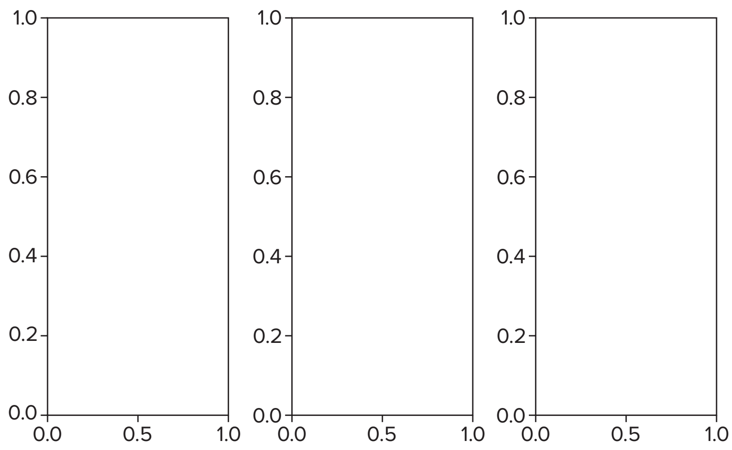


FIGURE 21.3 Three subplots in 1 x 3 grid

The following code is a slight modification of the above. It does the following:

- It creates 8 subplots in form of 2 x 4, i.e., 2 rows and 4 columns.
- These 8 subplots are stored in a 2D tuple named ax. This tuple is of shape (2, 4).
- Now each of these subplots has a property patch which gives the rectangle on which the plotting is done. You can use the set_facecolor() method to set the colour of the rectangle.

```
%matplotlib inline
import matplotlib.pyplot as plt
# Create subplots in 2 rows and 4 columns
fig, ax = plt.subplots(2,4)
print('type of fig->', type(fig))# type is Figure
print('type of ax->', type(ax)) #type is ndarray
print('type of ax[0][0]->', type(ax[0][0]))# type is AxesSubplot
print('shape->', ax.shape)# shape-> (2, 4)
#patch is a property of Axes and is the
# rectangle on which plotting is done
ax[1][2].patch.set_facecolor('black') # sets sub-plot background color as black
# Get current Figure and current Axes
print(fig.gca())# AxesSubplot object
print(plt.gcf()) # Figure object
plt.show()
```

```
# OUTPUT---
type of fig-> <class 'matplotlib.figure.Figure'>
type of ax-> <class 'numpy.ndarray'>
type of ax[0][0]-> <class 'matplotlib.axes._subplots.AxesSubplot'>
shape-> (2, 4)
plt.gcf()-> Figure(432x288)
fig.gca()-> AxesSubplot(0.731522,0.125;0.168478x0.343182)
```

Output figure is shown in Figure 21.4.

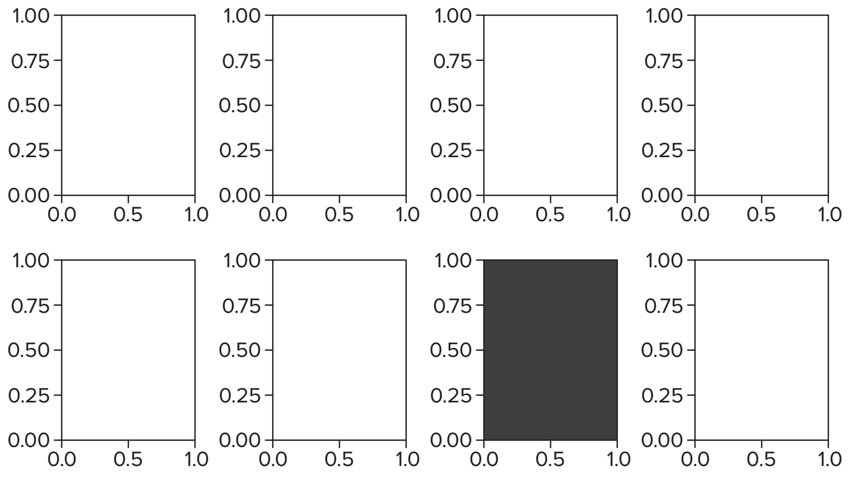


FIGURE 21.4 Creation of a 2 x 4 grid using subplot (2,4) function. Further it uses `ax[1][2].patch.set_facecolor('black')`, to set background color of one subplot to black

> ***Note:*** The index of subplots() method starts from 1 but the index of the array starts from 0. For example, if you have say subplots(1,2), then the array ax can be ax[0][0] or ax[0][1] only.

21.2.4 The Axis Container (under Axes container) and the Tick Container (under Axis)

It was pointed out that there is a Figure which contains Axes which are really subplots. The Axes class is represented conventionally by ax and they in turn contain the Axis class (Note the spelling difference between Axes and Axis). The Axis class are two in number namely X-Axis and Y-Axis and they represent the 2 Axis. The 2 Axis classes namely X-Axis and Y-Axis in turn contain Tick which are markers on the 2 Axis. Figure 21.5 gives the "hierarchy" of containers as well as the 4 types of Primatives used in Matplotlib.

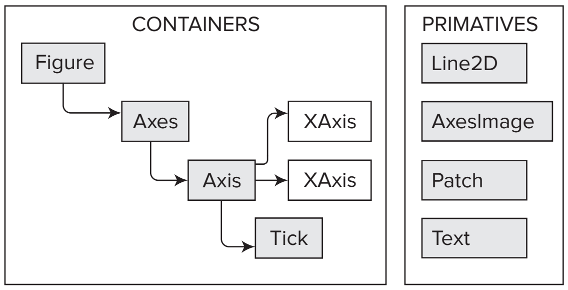


FIGURE 21.5 Layout of the various "Container objects" and "Primitive objects"

21.2.5 Primitives

Matplotlib has what is called primitives. The primitives are the objects that are drawn on the containers. Note that most objects are drawn on the Axes object ax. Primitives could be lines, circles, etc.

There are four drawing primitives in Matplotlib: Line2D, AxesImage, Patch and Text.

- A Line2D object uses a list of coordinates to draw line segments between them.
- An AxesImage class takes 2D data and coordinates and displays an image of that data with a colourmap applied to it.
- A Patch object is an arbitrary 2D object (generally a rectangle) that has a single color for its "face."
- Finally, the Text object takes a Python string, a point coordinate, and various font parameters to form the text that annotates plots.

21.2.6 Creating a Single Plot (no subplots)

In the above discussion, the Figure and Axis class along with other objects were discussed. However in many real-world examples you need to draw a single plot only. For this Matplotlib provides certain convenience functions like plot().

When working with a plot, there are two essential components:

- the figure and
- the axis. (***Note:*** axis is different from axes. Note the spelling difference.)
- The figure is the overall window or page on which the drawing is done and the axis are the two axis of 2D namely x-axis and y-axis.
- The most common function of pyplot is the plot() method, which is used to plot a set of points and then the .show() method which finally displays the plot.
- A method linspace(start, stop) is used.

The thing to note is that if you are using a convenience function like plot(), then you don't need to worry about objects of class Figure and Axes. The convenience function plot(), hides these details.

The following example code on Jupyter will make things clear:

```
1  %matplotlib inline
2  import numpy as np
3  import matplotlib.pyplot as plt
4  plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 16, 25])
5  plt.show()
```

Explanation

Line 1: %%matplotlib is a magic function in IPython. This line of code is required to display the plot in Jupyter notebook.

Line4: Here you are using the plot(x, y) method of the pyplot module. The first array, i.e., list to the plot function is the x-coordinates and the second list is the y-coordinates.

The screen shot of output on Jupyter is shown in Figure 21.6.

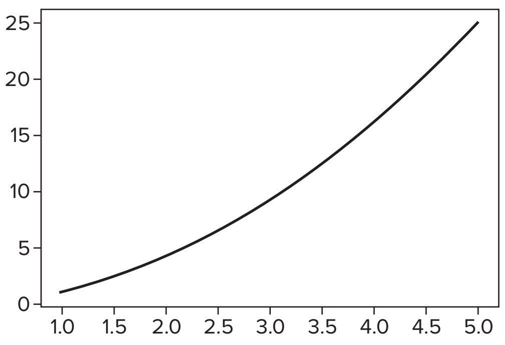


FIGURE 21.6 plot() method to plot a simple plot (on Jupyter)

21.2.7 Format String

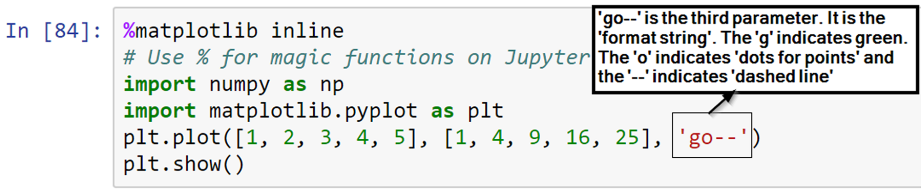
The concept of "format string" can be explained as follows:

- Note that for each x, y pair of arguments, it is possible to have an optional third argument. This "optional third" argument is called "format string".
- Through this "format string" you can vary various aspects of the point being plotted. For example, you can change "the color", "the line type", etc.
- Also note that this "optional 3rd argument" has a "default value" of "b-". Note that the "b" stands for color, i.e., "blue" and the "-" stands for "solid line". So if you don't give any 3rd parameter, then by default your graph will be plotted in "solid blue line".
- This "letter symbol" combination (for example "b-") used in Matplotlib has been inspired by Matlab.
- Similarly '—' stands for dashed line. Some common color abbreviations are 'b' for blue; 'g' for green' 'r' for red 'y' for yellow, 'k' for black and 'w' for white. Similarly 'o' is for circle marker; 'x' is for x marker and so on.

TABLE 21.1 Various options for "color" and "style" for a "format string"

Some options for color	Some options for style
'r' = red	'-' = solid
'b' = blue	'--' = dashed
'g' = green	':' = dotted
'y' = yellow	'-.' = dot-dashed
'c' = cyan	'.' = points
'm' = magenta	'^' = filled triangles
'w' = white	'o' = filled circles
'k' = black	

For example take the string 'go- -' as the third parameter to the plot() function in the above code. Then the screen shot of the code and the output plot are shown in Figures 21.7 and 21.8.


In [84]:
'go--' is the third parameter. It is the 'format string'. The 'g' indicates green. The 'o' indicates 'dots for points' and the '--' indicates 'dashed line'

```
%matplotlib inline
# Use % for magic functions on Jupyter
import numpy as np
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 16, 25], 'go--')
plt.show()
```

FIGURE 21.7 Screen shot of the script on Jupyter. Note the "third parameter" which is 'go--'. The 'g' stands for color green. The 'o' indicates that the points are to be marked as "o" and the '--' indicates that the joining line is to be a 'dashed line'.

A few other things to be noted are:

- You don't have to use a format string, you can also use named arguments. The names of the arguments are: color, marker, linestyle and markercolor
- You can draw more than 1 plot on the same figure.

You can draw sin and cos functions and use the named arguments form of the plot() method. Note here you have used an additional method, i.e., linspace(start, end, points). The linspace() method creates an array whose 'start' is first item, 'stop' is the last item and 'points' are the number of points generated. Note that the number of points generated will be 1 more than the number of intervals created.

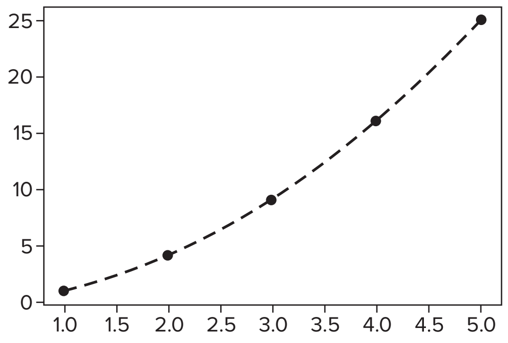


FIGURE 21.8 Output of the above script on Jupyter

You can see the "signature" of linspace() on Jupyter as follows:

```
import numpy
?numpy.linspace
```

```
# OUTPUT (Truncated and modified to save space and show only relevant portion)
Signature: numpy.linspace(start, stop, num=50, endpoint=True, retstep=False,
dtype=None)
Docstring:Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

Parameters (Only 4 parameters ie start, stop, num and endpoin are shown)
----------
    (1) start : scalar. The starting value of the sequence.
    (2) stop : scalar. The end value of the sequence, unless `endpoint` is set to
        False. In that case, the sequence consists of all but the last of ``num +
        1``    evenly spaced samples, so that `stop` is excluded.  Note that the
        step size changes when `endpoint` is False.
    (3) num : int, optional. Number of samples to generate. Default is 50. Must be
        non-negative.
    (4) endpoint : bool, optional. If True, `stop` is the last sample. Otherwise,
        it is not included. Default is True.

Returns:- Returns an ndarray. There are `num` equally spaced samples in the closed
interval ``[start, stop]`` or the half-open interval ``[start, stop)`` (depending on
whether `endpoint` is True or False).
```

For example see the following code and output:

```
import numpy as np
A = np.linspace(5, 8, 4)
print(A)
```

```
# OUTPUT
[5. 6. 7. 8.]
```

21.3 Basic Concepts (Part 2)

This is the second part of "basic concepts" in Matplotlib. Here topics like using methods like gcf() and gca() are covered. Also covered are topics like "spines and ticks" and using the plot() method of pyplot module.

21.3.1 Draw sin(x) and cos(x) on Same Plot Using plot()

The following code is used to draw a sin(x) and a cos(x) functions:

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

# Array A will hold 13 points ie [0, 30, 60.....360]
A = np.linspace(0, 360, 13)

# Round off all decimal places
X = np.around(A) #If  no second parameter, then number of decimal places is 0

#Array B will hold the corresponding angles of array A in radians
B = np.radians(A) # .radians(npArray) converts each item in nparray to radians
C = np.sin(B)

# Round off items in array C to 2 decimal places
Y1 = np.around(C, 2)
D = np.cos(B)
Y2 = np.around(D, 2)

#Plot graph of sin(x) in green dashed style using x markers and label of sin(x)
plt.plot(X,Y1,color ='green',linestyle ='dashed', marker = 'x', label = 'sin(x)')

#Plot graph of cos(x) in black solid style using o markers and label of cos(x)
plt.plot(X, Y2,color ='black',linestyle ='solid', marker = 'o', label = 'cos(x)')

# Title of the plot is 'Sin and Cos functions'
plt.title('Sin and Cos Functions')

#Label X-axis as x and Y-axis as y (sin(x)/ cos(x))
plt.xlabel('x (Angle in radians)')
plt.ylabel('y (sin(x)/ cos(x))')

#Draw X-axis in grey color at y = 0
plt.axhline(0, color = 'grey')

```

```
36 # Draw vertical line in yellow at x = 0
37 plt.axvline(0, color = 'yellow')
38
39 #Draw a 2nd vertical line in blue at the 7th point (index 6)
40 plt.axvline(A[6], color = 'blue')
41
42 # The legend location is lower left corner
43 plt.legend(loc = 'lower left')
44 plt.show()
```

Explanation:

Line 6: Here linspace(0, 360, 13) will create a list of 13 points including 0 and 360. Note that by default the end point is included unlike in arrange() function. Also note that with 13, the interval between 0 and 360 will be divided into 12 intervals.

Line 9: The around(A) function here is used on the array A but without giving any value for number of decimal places. So there will be no decimal places.

Line 16: Here the around(C,2) will convert all the values in array C to 2 decimal places.

Lines 21 and 24: Note the keyword label. This links the particular plot to the particular label.

Line 27: The .title() method creates the title of the plot.

Lines 30-31: Line 30 creates the label for the X-axis and Line 31 creates the label for the Y-axis.

Line 34: .axhline() creates X-axis at Y value of 0 in grey color.

Line 37: Creates vertical line at X = 0 in yellow color

Line 40: This creates a vertical line at the 7th point (i.e., index 6) which corresponds to 180^o or π radians.

Line 43: The legend() method of the pyplot module has a number of options. For loc = 'lower left', the legend is displayed at the lower left corner.

The output on Jupyter notebook is shown in Figure 21.9.

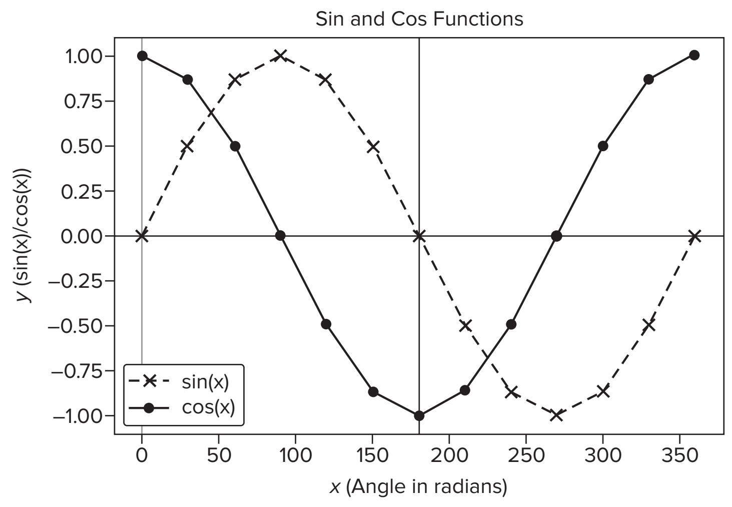


FIGURE 21.9 Plotting of sin and cos curves from the above script on Jupyter

21.3.2 Using Methods gcf() and gca()

Before proceeding further, there is an additional concept that you need to understand.

It had been pointed out that there are two methods of "drawing" used in the Matplotlib library.

- One method: Use the classes like Figure, Axes, etc.
- Another method: Use helper function like Matplotlib.pyplot.plot()

It has also been mentioned that if you use the "helper method plot()", then you don't have "access" to the underlying objects of the Figure class and the Axes class.

But Matplotlib provides a "**way of getting the underlying objects of the Figure class and the Axes class**". So even if you used the plot() helper method, but need to access the underlying Figure object or the Axes object, you may do so. For this purpose, Matplotlib provides two methods.

There two methods are: gca() and gcf(), respectively.

- gca() provides the current Axes object, i.e., the subplot on which the plotting is to be done.
- gcf() provides the current Figure, i.e., the overall container which contains the plot(s)/subplots.

21.3.3 The Spines and Ticks of a Plot

Many basic concepts of plotting have been explained in previous sections. However, two more concepts which need to be explained are of spines and ticks.

Spines are the lines outlining the boundary of the data area. They have tick marks on them. By default a figure will have four spines, i.e., at top, bottom, left and right.

So you may modify the above program so as to move the spines. The following script, makes the top and right spine "invisible" and shifts the bottom and left spines to the middle of the plot. This script also does not draw the axhline() and the axvline().

The code is as follows:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
A = np.linspace(0, 360, 13)
X = np.around(A) #If  no second parameter, then number of decimal places is 0
B = np.radians(A) # .radians(npArray) converts each item in nparray to radians
C = np.sin(B)
Y1 = np.around(C, 2)
D = np.cos(B)
Y2 = np.around(D, 2)
plt.plot(X, Y1, color = 'green', linestyle = 'dashed', marker = 'x', label =
'sin(x)')
plt.plot(X, Y2, color = 'black', linestyle = 'solid', marker = 'o', label =
'cos(x)')
plt.title('Sin and Cos Functions')
plt.xlabel('x')
plt.ylabel('sin(x)/ cos(x)')
plt.legend(loc = 'lower left')
ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.spines['left'].set_position(('center'))
ax.spines['bottom'].set_position(('center'))
# You can increase the axis by using axis([xmin, xmax, ymin, ymax])
plt.axis([-50, 400, -1.5, 1.5])
plt.show()
```

Explanation:

Lines 20-21: By setting the colors of "right" and "top" spines to "none", these spines will not be displayed.

Lines 22-23: By setting the positions of "left" and "bottom" spines to "center", these spines will appear at the centre of the plot.

Line 25: By default, the axis begins at the beginning of data and ends at the end of data. If you want the axis to go beyond the range of data, you may use the plt.axis([xmin, xmax, ymin, ymax]) methods.

The output is shown in Figure 21.10.

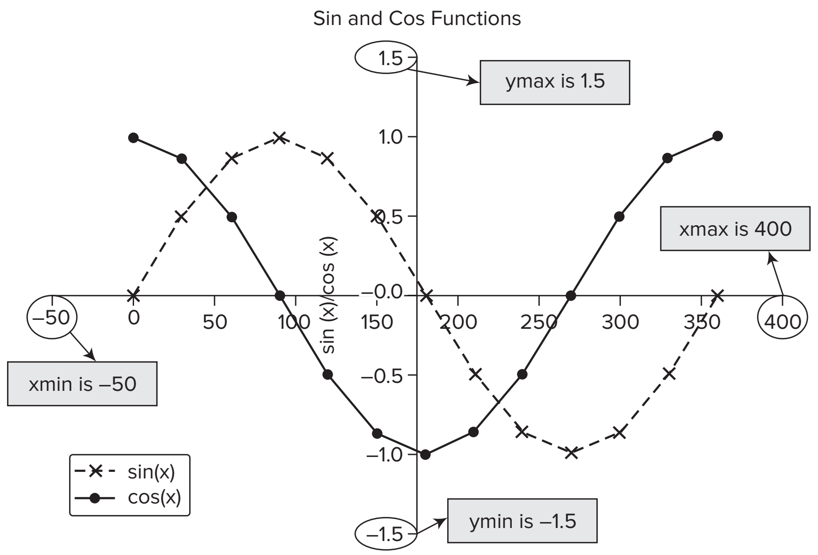


FIGURE 21.10 In these sin and cos curves, the "top" and "right" spines have been "discarded by setting there color to none". Further the left and bottom spines have been moved to the centre of plot.

Note that there are four spines (top/bottom/left/right), but the top and right have been discarded by setting their colour to none. Further the "bottom" and "left" spines have been moved to coordinates (0,0), i.e., at the centre of the plot.

21.3.4 Using np.linspace and np.vectorize to Create a Plot

When you plot a function, you need points on the x-axis, which you can create using the np.linspace method. But how do you create the corresponding y-point? To do this, NumPy provides a numpy.vectorize(pyfunc) method. Here pyfunc is a python function. The following code shows how the np.vectorize[1] method works:

[1] For details on vectorize method see: https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.vectorize.html

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import math
u = np.linspace(-4, +4, 20)
def f(u):
    return np.sin(u)
f2 = np.vectorize(f)
print(np.around(f2(u), 2))
plt.plot(u,f2(u))
```

```
# Output. . .
[ 0.76  0.42  0.02 -0.39 -0.74 -0.95 -1.   -0.87 -0.59 -0.21  0.21  0.59
  0.87  1.    0.95  0.74  0.39 -0.02 -0.42 -0.76]
```

The output figure is shown in Figure 21.11

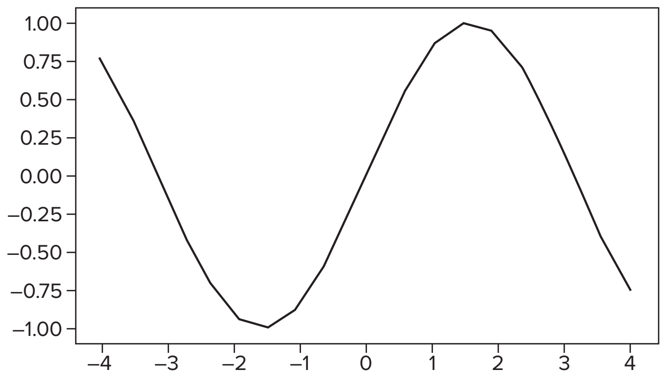


FIGURE 21.11 Use of np.linspace and np.vectorize to create a sin curve (output on Jupyter)

21.3.5 The plot() Method of pyplot

In previous examples the plot() method of pyplot module was used. The signature of this method is as follows:

```
Signature: plt.plot(*args, **kwargs)
*args* is a variable length argument, allowing for multiple *x*, *y* pairs with an
optional format string.
Return value is a list of lines that were added.
```

The important points to note are:

- You can give a number of pairs of x-y values and for each pair of values a line will be plotted. There is no limit on the number of pairs you may give, so you could plot any number of lines on a single plot.

- The return value of the method is a list of lines. These lines are objects of class Line2D. You need to understand that the plot() method returns a container or collection of objects. The container here is a list and the object here is an object of class Line2D.

These concepts can be clarified with the following example. The example script draws three simple lines say of functions $y_1 = x$, $y_2 = 2*x + 2$ and $y_3 = 3*x + 5$. The plot() function is used to draw them. So you will have a collection, i.e., a list of three objects of class Line2D.

```
1  %matplotlib inline
2  import numpy as np
3  import matplotlib.pyplot as plt
4  # x is a list of 11 points from 0 to 10
5  x = np.linspace(0, 10, 11)
6  y1 = x
7  y2 = 2*x + 2
8  y3 = 3*x + 5
9  # plot() returns a list of Line2D objects.
10 line_list = plt.plot(x, y1, 'bo', x, y2, 'r+', x, y3,'y*')
11 # You can iterate over the collection of lines
12 for line in line_list:
13     line_color =line.get_color()
14     line_marker = line.get_marker()
15     print(line, 'color->', line_color, 'line style->', line_marker)
```

```
16 # Output. . .
17 Line2D(_line0) color-> b line style-> o
18 Line2D(_line1) color-> r line style-> +
19 Line2D(_line2) color-> y line style-> *
```

Lines 13-14: Each of the Line2D objects has a color and a marker property which can be found using get_color(0 and get_marker() methods.

The plot is shown in Figure 21.12

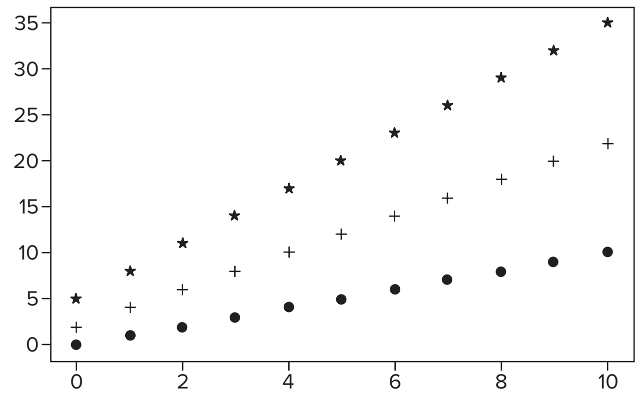


FIGURE 21.12 Plotting of multiple lines in different styles on the same plot.

Note that in the above script, you get the list of lines as a return value from the plot() method. You can also get the list of lines from the Axes object because the Axes object has a property Axes.lines. Suppose you have an Axis object say ax. Then you can write:

```
1 # Add following lines to code in above example
2 # First get current Axes object
3 ax = plt.gca()
4 # From Axes object, you can get the list of lines
5 print(ax.lines)
6 # Output. . .
7 [<matplotlib.lines.Line2D object at 0x0986C2F0>, <matplotlib.lines.Line2D object at
8 0x0986CD30>, <matplotlib.lines.Line2D object at 0x0986C3F0>]
```

Line 3: Here you got the current Axes object and call it ax.

Line 5: Every Axes object has a property lines which is a collection of the Line2D objects present on the Axes container.

Lines 7-8: Note that the output is a list of three Line2D objects.

21.3.6 Draw a Chess-board using Matplotlib

This section gives an example script of how to draw a "chess board". The script uses a method of Matplotlib called matshow().

The signature of matshow() is:

```
matplotlib.pyplot.matshow(A, fignum=None, **kwargs)
A : array-like(M, N)→ The matrix to be displayed.
```

The script to create a chess-board is:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
A= np.zeros((8,8),dtype=int)
A[1::2,::2] = 1
A[::2,1::2] = 1
print(A)
plt.matshow(A)
```

The output on Jupyter Notebook is shown in Figure 21.13.

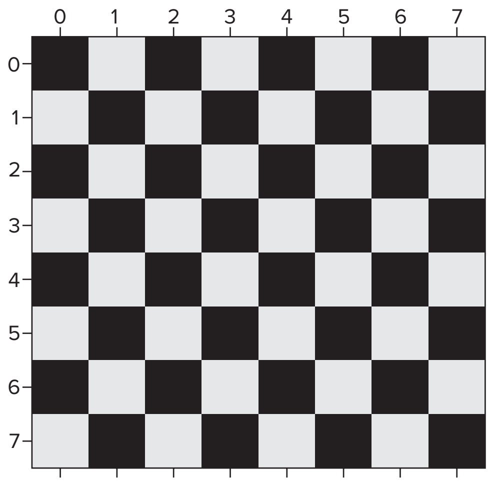


FIGURE 21.13 Output of script for drawing a “chess board” on Jupyter

21.4 CREATING MULTIPLE SUBPLOTS

Sometimes, you may want to draw more than 1 plot in the same figure. You can do this by dividing the entire plotting area into subplots. There are different methods for creating subplots. Some ways of creating subplots are discussed in the following section.

21.4.1 Creating Subplots using subplot() and subplots()

The pyplot module has two functions subplot() and subplots().

Note that subplot() is different from add_subplot(). The difference is that subplot() is a function of the pyplot module where add_subplot() is a method of the Figure class. So if you want to use the add_subplot() method you need to first create a Figure object. This is explained in later section. Here you need to focus only on plot() and plots().

The signature of subplot() method is given below. Note that the subplot() method returns a subplot Axes object. So it returns a single Axes object.

```
Signature: plt.subplot(*args, **kwargs)
Return a subplot axes positioned by the given grid definition.
Typical call signature::
subplot(nrows, ncols, plot_number)
Where *nrows* and *ncols* are used to notionally split the figure
into ``nrows * ncols`` sub-axes, and *plot_number* is used to identify
the particular subplot that this function is to create within the notional
grid. *plot_number* starts at 1, increments across rows first and has a
maximum of ``nrows * ncols``.
In the case when *nrows*, *ncols* and *plot_number* are all less than 10,
a convenience exists, such that the a 3 digit number can be given instead,
where the hundreds represent *nrows*, the tens represent *ncols* and the
units represent *plot_number*.
For instance:: subplot(211) produces a subaxes in a figure which represents the
top plot (i.e. the first) in a 2 row by 1 column notional grid (no grid actually
exists, but conceptually this is how the returned subplot has been positioned).
Keyword arguments:
*facecolor*: The background color of the subplot
*polar*: A boolean flag indicating whether the subplot plot should be a polar
projection.  Defaults to *False*.
```

On the other hand the subplots() method returns a Figure object as well as an array of Axes objects. The advantage of using this function is that you can create a common layout for the subplots and you can have a reference to each of the subplots in a single call. The signature is:

```
Signature: plt.subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True,
subplot_kw=None, gridspec_kw=None, **fig_kw)
Docstring: Create a figure and a set of subplots
This utility wrapper makes it convenient to create common layouts of
subplots, including the enclosing figure object, in a single call.

Parameters
nrows, ncols : int, optional, default: 1
    Number of rows/columns of the subplot grid.
sharex, sharey : bool or {'none', 'all', 'row', 'col'}, default: False
    Controls sharing of properties among x (`sharex`) or y (`sharey`)
    axes:

Returns
fig : :class:`matplotlib.figure.Figure` object
ax : Axes object or array of Axes objects. ax can be either a single
:class:`matplotlib.axes.Axes` object or an array of Axes objects if more than one
subplot was created.
```

There may be confusion regarding the methods plot(), plots(), add_subplot() and add_axes(). Table 21.2 clarifies them.

TABLE 21.2 Comparison of the three methods namely subplots(), add_subplot() and add_axes()

	subplot()	subplots()	add_subplot	add_axes(rect)
1	It is a method of the pyplot module.	It is a method of the pyplot module.	It is a method of the Figure class. So to use this you should first create a Figure object conventionally called fig.	It is a method of the Figure class. So to use this you should first create a Figure object conventionally called fig.

2	The return is a single Axes object.	There are two return values, i.e., a Figure object and a tuple of Axes objects. But in case a single subplot is created the second return is not a tuple but a single Axes object.	The return is a single Axes object. (This method is discussed later in the book.)	The add_axes method takes a list named rect consisting of four numbers. These are (1) left, (2) bottom, (3) width, and (4) height for the subplot. Here "left" and "bottom" are coordinates of the lower left corner of the subplot. Further "width" and "height" are the width and height, respectively of the subplot. Note that all values specified in relative units (where 0 is left/bottom and 1 is top/right). Both add_axes() and add_subplot() methods return an Axes object. The difference is that add_axes() starts from lower left corner ie it uses "absolute coordinates". So with add-Axes(), you can put a subplot "anywhere" even "inside" another subplot.

Following example shows use of subplots():

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
# create 2 x 3 grid of subplots
fig, ax_arr = plt.subplots(2, 3)
#ax_arr is a 2D array
print('shape of ax_arr->', ax_arr.shape)
#use sharex and sharey.Subplots on same row share y-axis
#subplots on same column share x-axis. Looks neater
fig, ax_arr = plt.subplots(2, 3, sharex='col', sharey='row')
```

```
# Output. . .
shape of ax_arr-> (2, 3)
```

Figure 21.14 shows the subplots when they don't share the axis, here they look a bit untidy. Figure 21.15 shows the subplots when they share their axis, they are much neater.

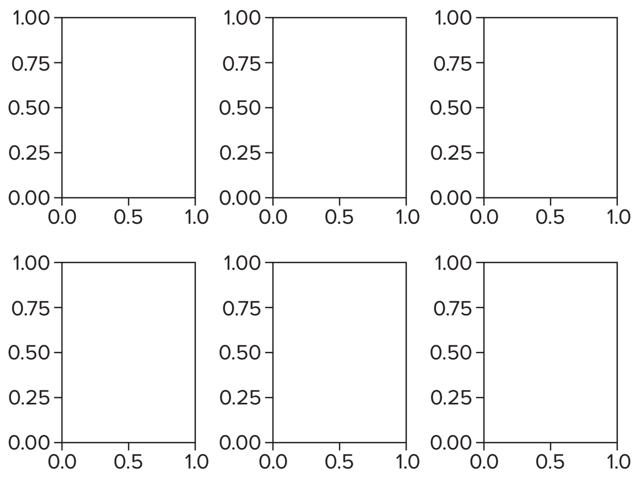


FIGURE 21.14 Subplots when they "don't share" the axis

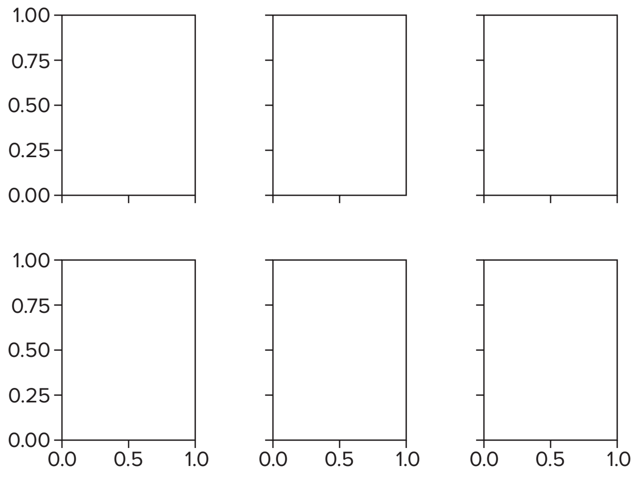


FIGURE 21.15 Subplots when they "share their axis"

The following code shows how subplot() is used.

Note: When you create a subplot, it will delete any previously existing subplot over which it occurs on the screen. But it can "share a boundary" with a pre-existing subplot.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.subplot(2, 2, 1)
plt.subplot(224)
```

The output is shown in Figure 21.16.

```
Out[54]:  <matplotlib.axes._subplots.AxesSubplot at 0xa8459f0>
```

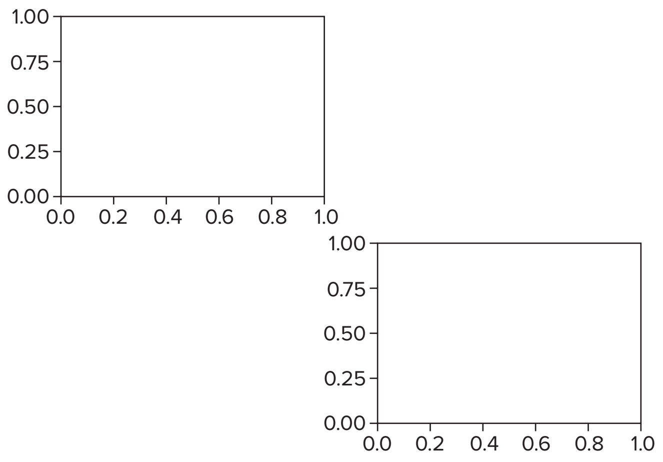


FIGURE 21.16 Creation of subplots using (i, j, k) notation and also (ijk) notation where i is for nrows, j is for ncols and k is the index

21.4.2 Creating Multiple Figure Objects

Note that in general you create a single Figure object (conventionally called fig) and then plot various subplots on it (conventionally denoted by ax).

But you can always create more than one Figure objects. For this you may use a method of the pyplot module called figure(). Its signature is as follows:

```
Signature: plt.figure(num=None, figsize=None, dpi=None, facecolor=None,
edgecolor=None, frameon=True, FigureClass=<class'matplotlib.figure.Figure'>,
clear=False, **kwargs)
Parameters
num : integer or string, optional, default: none
figsize : tuple of integers, optional, default: None
Returnsfigure : Figure
```

21.4.3 Using add_subplot()

Note that the method called subplots() has already been explained. Note that subplots() is a method of the pyplot module.

On the other hand add_subplot() is a method of the Figure class. So if you first create a Figure object and then want to add a subplot to this Figure object, you may do so by using the add_subplot. It will be clear from the following subplot.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
#create 2 Figure containers fig1 and fig2
fig1 = plt.figure()
fig2 = plt.figure()
# fig1 is an object of type Figure
print(type(fig1))
#Each figure has a number attribute
print('number of fig1->', fig1.number)
print('number of fig2->', fig2.number)
# pyplot has a method to get all figure numbers
print('list of figure numbers->', plt.get_fignums())
#You can always create Axes objects on each Figure object
ax1 = fig1.add_subplot(111)
# ax2 will have a polar sub-plot
ax2 = fig2.add_subplot(111, polar = True)
```

```
# Output. . .
<class 'matplotlib.figure.Figure'>
number of fig1-> 1
number of fig2-> 2
list of figure numbers-> [1, 2]
```

The output is shown in Figure 21.17

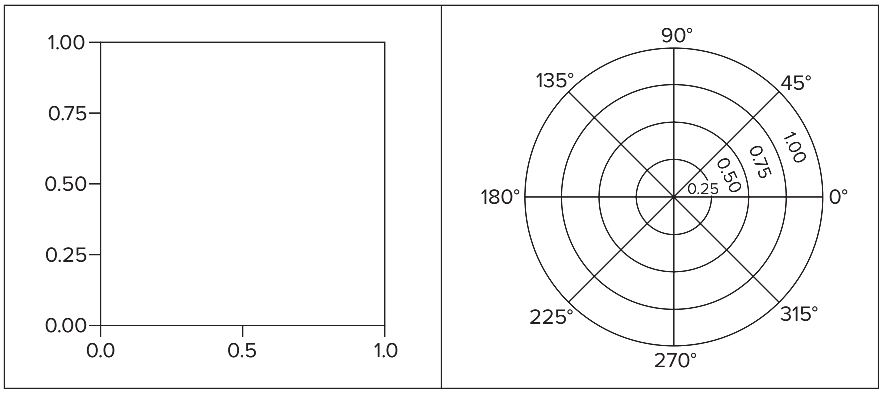


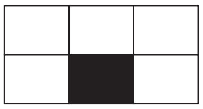
FIGURE 21.17 Output of add_subplot

Note that the signature of this method is `add_subplot`(**args*, ***kwargs*).

The two common formats of the above method are:

- add_subplot(nrows, ncols, index, **kwargs). Note here *args stands for nrows, ncols, index. Here nrows is the number of rows in the grid, ncols is the number of columns in the grid and index is which particular element in the grid is taken (starting from top left and increasing to right and then down). For example, add_subplot(2, 3, 5) means you divide the entire plot in to a matrix of 2 x 3, i.e., 2 rows and 3 columns and take the subplot number 5 (counting from top left corner).
- add_subplot(pos, **kwargs) . Note that pos is a three digit integer. The first digit (i.e., leftmost digit) stands for the number of rows. The second digit (i.e., the middle digit) stands for the number of columns. The third digit, (i.e., the right most digit) stands for index of the subplot. For example you could have add_subplot(235), which is same as add_subplot(2, 3, 5). Note that all integers must be less than 10 for this form to work.

The method add_subplot(2, 3, 5) or add_subplot(235) will produce grid somewhat like this (The darkened box has an index of 5 counting from top left):



Note that add_subplot(ijk) means a grid of i rows, j columns and then selecting the kth subplot. Note that the return value of this function is the axes of the subplot. This will become clear from the following example.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure(figsize = (8,4))
ax1 = fig.add_subplot(221)
ax1.text(0.5, 0.5, '221')
ax2 = fig.add_subplot(222)
```

```
8  ax2.text(0.5, 0.5, '222')
9  ax3 = fig.add_subplot(223)
10 ax3.text(0.5, 0.5, '223')
11 ax4 = fig.add_subplot(224)
12 ax4.text(0.5, 0.5, '224')
13 plt.show()
```

The output is shown in Figure 21.18.

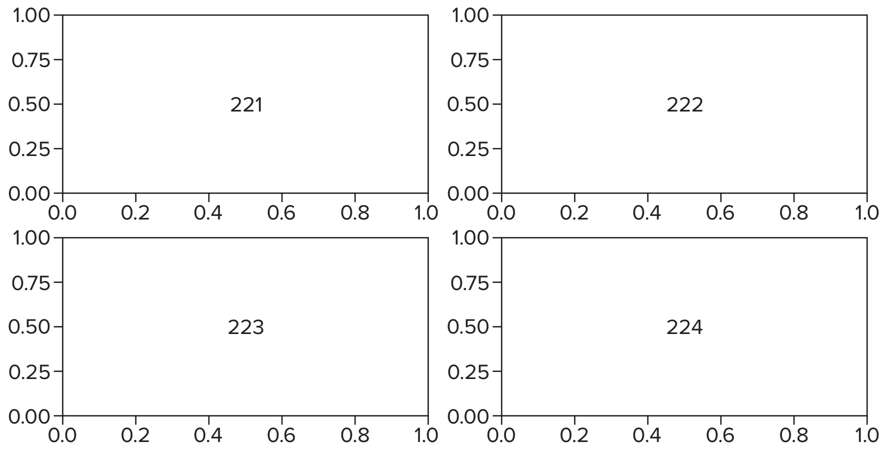


FIGURE 21.18 Output of the above script. Four subplots are created.

However, if you run the following code, you get:

```
1  %matplotlib inline
2  import numpy as np
3  import matplotlib.pyplot as plt
4  fig = plt.figure(figsize = (8,4))
5  ax1 = fig.add_subplot(221)
6  ax1.text(0.5, 0.5, '221')
7  ax2 = fig.add_subplot(222)
8  ax2.text(0.5, 0.5, '222')
9  ax3 = fig.add_subplot(212) # treating the second row as having only 1 column
10 ax3.text(0.5, 0.5, '212')
```

Lines 5-6: These two lines treat the figure as a 2 x 2 grid with 2 rows and 2 columns and plot the first plot and second plot of this 2 x 2 grid.

Line 7: Note here the figures are treated as a 2 x 1 grid (Not 2 x 2 grid). So to select the bottom plot of this 2 x 1 grid, the plot number is 2 (Not 3 or 4).

The output is shown in Figure 21.19.

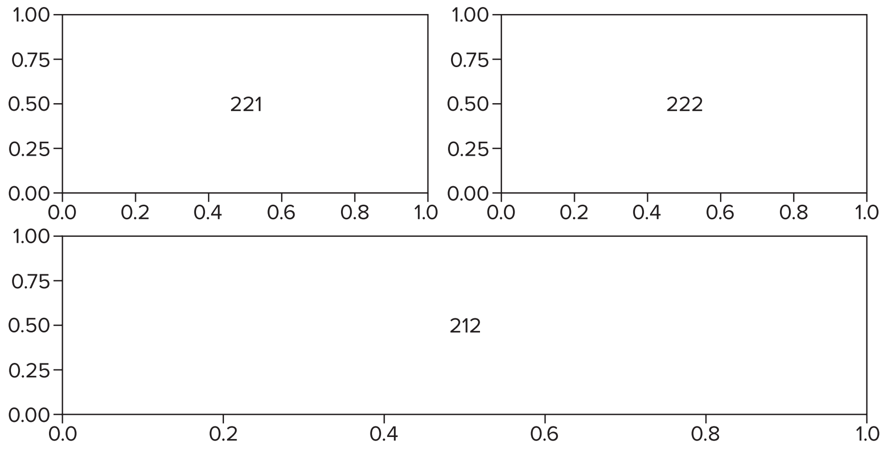


FIGURE 21.19 Output of three subplots using 221, 222 and 212. Note that in 212, there are 2 rows but only 1 column so the third plot spans the entire width.

21.4.4 Creating Subplots using subplot2grid() Method

You can also create subplots using subplot2grid() method. The signature of the method is as follows:

```
subplot2grid(shape, loc, rowspan=1, colspan=1, fig=None, **kwargs)
shape->a tuple giving the shape ie the number of rows and columns in the grid. It
is the shape of grid in which to place axis. First entry is number of rows, second
entry is number of columns.Thus shape = (3,3) means you want a grid of 3 rows and
3 columns.
loc-> It is a tuple of the starting point in terms of row and column of the starting
point. Note the origin is (0,0).
rowspan-> indicates the number of rows (ie vertical height in terms of rows)
colspan-> indicates the width ie horizontal width in terms of number of columns.
```

The use of this method will become clear from the following example (note use of \n in the strings):

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
ax1 = plt.subplot2grid((3, 3), (0, 0))
ax1.text(0.05, 0.5,'ax1 Starts at (0,0)\nspans 1 row 1 col')
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax2.text(0.1, 0.5,'ax2 Starts at (0,1)\nSpans 1 row 2 col')
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=1, rowspan=2)
ax3.text(0.05, 0.5,'ax3 Starts at (1,0) \nSpans 2 rows 1 col')
ax4 = plt.subplot2grid((3, 3), (2, 2), rowspan=1)
ax4.text(0.05, 0.5, 'ax4 Starts at (2,2)\n spans 1 row 1 col')
plt.show()
```

Line 4: This creates a subplot in a grid of 3 x 3 starting at (0,0) with default values for rowspan and colspan of 1 each.

Line 6: This creates a subplot again in 3 x 3 grid starting at row 0 and column 1. The rowspan defaults to 1 while the colspan = 2 so this subplot is 1 row high and 2 columns wide.

Similarly, all the other subplots can be understood. The output is shown in Figure 21.20.

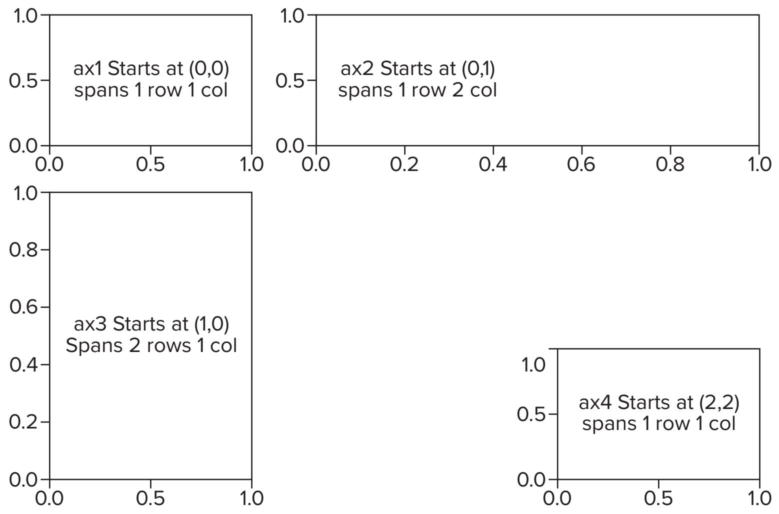


FIGURE 21.20 Output when subplot2grid() method is used

21.4.5 Creating Subplots using the axes() Function of pyplot Module

You can also create subplots using the axes() function of pyplot module.

There are many variations of this method but only the following are considered:

```
axes(rect, facecolor='w')
*rect* = [left, bottom, width, height] in normalized (0, 1) units.
*facecolor* is the background color for the axis, default is white.
```

This will become clear from the following code

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.figure('A')
# Rect starts at origin and has height/ width 0.4 of plot ht/width
ax1 = plt.axes([0.00, 0.00, 0.4, 0.4])
ax1.text(0.1, 0.5, 'Starts at (0, 0)\nWidth =0.4,\nheight = 0.4')
# Rect starts at (0.5, 0.5) and has height/ width 0.4 of plot ht/width
ax2 = plt.axes([0.50, 0.5, 0.4, 0.4])
ax2.text(0.1, 0.5, 'Starts at (0.5, 0.5)\nWidth =0.4,\nheight = 0.4')
plt.show()
```

Its output is shown in Figure 21.21.

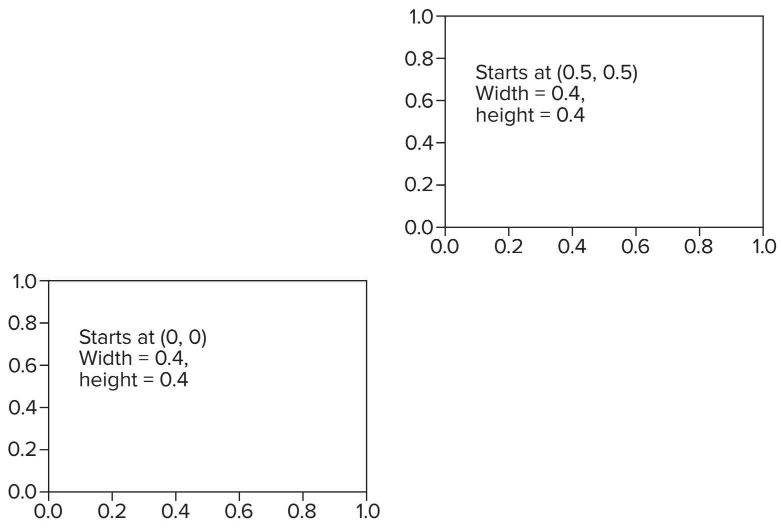


FIGURE 21.21 Output plots using the axes() method on Jupyter

21.4.6 Difference between add_axes() and add_subplot()

There is a clear difference in working of add_axes() and add_subplot(). The signature of add_axes() is:

```
add_axes(rect_list)
```

where rect_list is a list of type [x_left, y_bottom, width, height].

For example, you can have:

```
fig = plt.figure()
# width = 1 and height =1 means entire canvas
ax = fig.add_axes([0,0,1,1])
```

On the other hand, in add_subplot(), you cannot place the subplot at a predefined position. Rather you can place the subplots as per the subplot grid. One common way of doing this is using a three integer notation. The following code explains it:

```
fig = plt.figure()
# create a 2 x 3 grid and place the subplot at row= 1 column = 1
ax = fig.add_subplot(231)
```

21.4.7 Summary of Ways to Create Figure Object

Note that "Figure objects" contain "Axes objects" and Axes objects in turn contain "other objects"

There are three common methods to create containers, i.e., Figure and Axes. First you may create a Figure object which is conventionally called fig and then you may create an Axes object which is conventionally called, ax.

Table 21.3 summarizes the process of creation of figure objects and Axes objects. (Note that in method 3, both figure and axes objects are created by a single method call).

TABLE 21.3 Comparison of the steps in creating plots using three different methods: fig.add_sublot(); fig.add_axes() and plt.subplots()

Steps	Method1	Method2	Method3
1	Use method fig = plt.figure() This will create a Figure object referenced by variable fig.	fig = plt.figure(). This will create a Figure object fig just like in Method1.	fig, axarr = plt.subplots(n) Here steps 1 and 2 are combined. The method subplots(n) returns a tuple of two things. The first is an axes object which gets assigned to the variable fig and the second is a list of axes. Note that fig and axarr are separated by a comma indicating that the return is a tuple.
2	Use ax = fig.add_subplot(u,v,w) This will create an Axes object of plot w in u x v matrix with u rows and v columns.	Use ax = fig.add_axes([x1, y1, x2, y2]). Where x1, y1, x2, and y2 are all between 0 and 1. This creates an axes ax for the plot	

The following example code shows how each of the three methods are used:

Method 1:

```python
from matplotlib import pyplot as plt
# Have to first create a Figure object
fig1 = plt.figure()
# From fig1 create an Axes object ax1
ax1 = fig1.add_subplot(1, 1, 1)
# Show
plt.show()
```

Figure 21.22 shows the output of above script.

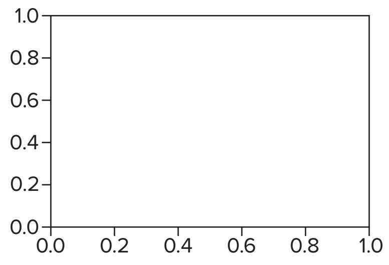


FIGURE 21.22 Output on Jupyter when add_subplot(i, j, k) is used with I = j = k = 1.

Method 2:

```python
from matplotlib import pyplot as plt
fig2 = plt.figure()
ax2 = fig2.add_axes([0, 0, 1, 1])
plt.show()
```

The output is shown in Figure 21.23.

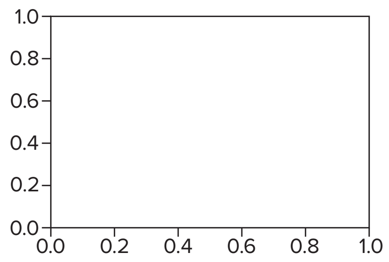


FIGURE 21.23 Creation of subplot using the add_axes() method

Method 3:

```python
from matplotlib import pyplot as plt
# axarr is a 2-D numpy array
fig, axarr = plt.subplots(2, 4)
# Sub plot on top left is (1, 1)
axarr[0][0].text(0.2, 0.5, '(1, 1)')
# Sub plot on bottom right is (1, 4)
axarr[1][3].text(0.2, 0.5, '(1, 4)')
plt.show()
```

Line 5: Note that the axarr object returned by the subplots() method is a 2D NumPy array. So you have to use indexes to access its individual Axes objects. Here it is a 2 x 4 array (With index starting as usual from 0).

The output is shown in Figure 21.24.

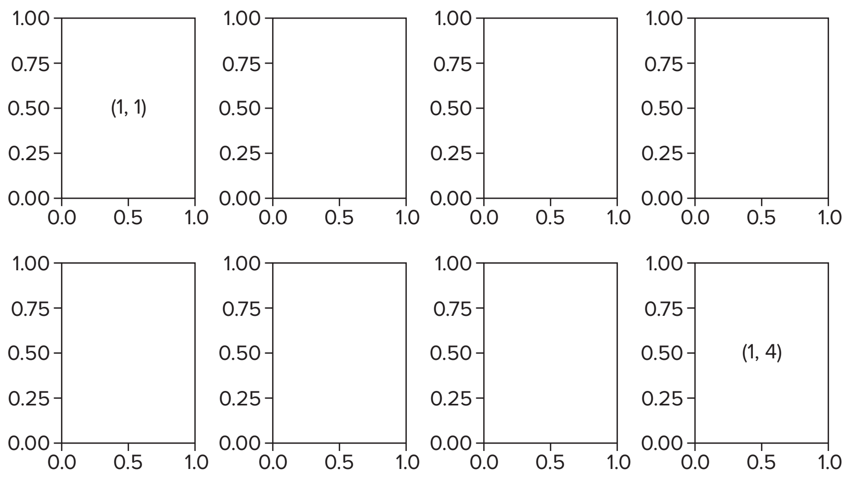


FIGURE 21.24 Creation of plots using the subplots() method

The steps to be followed in drawing are summarized as follows:

- **Step1:** Create a Figure object, for example fig = plt.figure()
- **Step2:** Use the figure object created in Step1 to create Axes object. (You may do this by using the add_axes() or add_subplot() methods of the figure object. In step1 above, fig is an object of class

Figure. Now you can use fig.add_axes() or fig.add_subplot methods on this fig object. In either case the object returned will be an object of type axes.). Note that the Figure object and the Axes object form the "container" on which objects can be drawn.

- **Step3:** Apart from the "container object", there are also "Primitive objects". These include objects like Line2D (for drawing a line), Rectangle (for drawing a rectangle), Text (for writing text), etc. So in the third step you may use one of these "Primitive objects" to draw on the "Containers" created in Steps 1 and 2.

The 3 steps outlined above are shown in form of Figure 21.25 below:

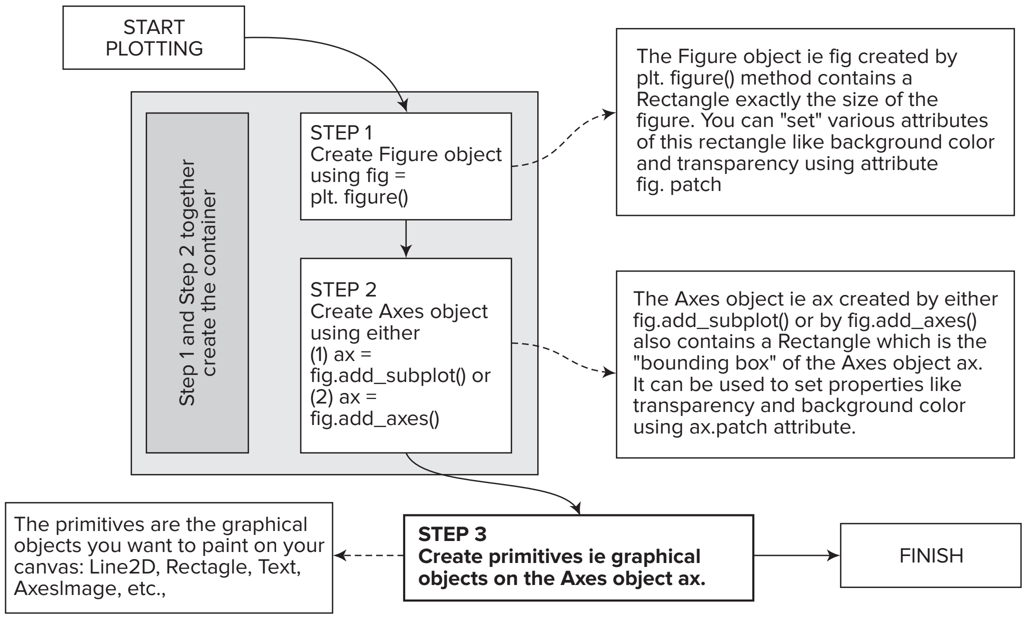

START PLOTTING
Step 1 and Step 2 together create the container
STEP 1 Create Figure object using fig = plt. figure()
The Figure object ie fig created by plt. figure() method contains a Rectangle exactly the size of the figure. You can "set" various attributes of this rectangle like background color and transparency using attribute fig. patch
STEP 2 Create Axes object using either (1) ax = fig.add_subplot() or (2) ax = fig.add_axes()
The Axes object ie ax created by either fig.add_subplot() or by fig.add_axes() also contains a Rectangle which is the "bounding box" of the Axes object ax. It can be used to set properties like transparency and background color using ax.patch attribute.
The primitives are the graphical objects you want to paint on your canvas: Line2D, Rectagle, Text, AxesImage, etc.,
STEP 3 Create primitives ie graphical objects on the Axes object ax.
FINISH

FIGURE 21.25 The three steps in drawing in Matplotlib: (1) Create Figure object, (2) Create Axes object and (3) Drawing with "Primitive objects"

21.5 SOME ADDITIONAL TOPICS

In this section, certain additional topics are covered. The topics covered include: drawing scatter plots, histograms, etc.; log-scales; 3D plotting; using meshgrid;() method and doing "animation".

21.5.1 Creating Scatter Plots, Histograms, Bar-plots and Contour Plots

- **Scatter plot:** A scatter plot is used to visualize relationships between variables. Unlike a line plot, a scatter plot does not connect the points with lines.
- **Histograms:** In a histogram, you plot "quantitative data". The entire range of 1 variable is divided into "intervals" or "bins" and then the other variables are plotted. In a histogram, the X-axis will have a "low end" and a "high end" because the data is quantitative.
- **Bar charts:** Unlike histograms (which plot quantitative data), bar charts plot "categorical data". In bar charts, each column represents a group or a "category", which may have no relationship with

one another. Note that in a bar-chart, the X axis cannot have a "high" or a "low" because the data being represented is "categorical" and not "quantitative".

Quantitative versus categorical data: Quantitative data is that data which falls on a "continuum". So quantitative data will have a "high point" and a "low point". One good example of categorical data is "grades in a school exam". For example you may say score from 91-100 is grade "A"; from 81-90 is grade "B"; and so on.

On the other hand, "categorical" data has no "high or low point". For example, you have three languages say L1, L2 and L3 and you need to plot the number of speakers of each language. Now you cannot place the three languages on a "continuum", i.e., you cannot say which will come first on X-axis because you can arbitrarily decide which one to place first.

For creating a scatter plot you may use the method plt.scatter(). The scatter() method has many parameters, but the following simpler version is used:

```
plt.scatter(x, y, s = None, c = None, marker = None, alpha = None)
Docstring:
Make a scatter plot of `x` vs `y`.
Marker size is scaled by `s`and marker color is mapped to `c`.
Parameters
----------
x, y : array_like, shape (n, )
s-> For size of points
c-> For color of points
alpha->The alpha blending value, between 0 (transparent) and 1 (opaque)
```

The following code shows the use of scatter() function to create two scatter subplots:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
# Create 2 scatter sub plots with axes ax1 and ax2
ax1 = plt.axes([0, 0, 0.4, 0.9])
ax2 = plt.axes([0.5, 0, 0.4, 0.9])
N = 60
# X1 and Y1 are random integers between o and 100
X1 = np.random.randint(0, high = 101, size = N)
Y1 = np.random.randint(0, high = 101, size = N)
#s1 is for size of each point. It is sum of 2 random integers
s1 = (X1 + Y1)
#col1 is a list of 4 * 15 = 60 colors Red Green Blue Black
col1 = ['r', 'g', 'b', 'k'] * 15
ax1.scatter(X1, Y1, s = s1, c = col1, alpha = 0.3)
ax1.set_title("Plot1")
ax1.set_xlabel('Red Green Blue Black')
# X1, Y1 random integers from 0 to 100 for second plot
```

```
20 X2 = np.random.randint(0, high = 101, size = N)
21 Y2 = np.random.randint(0, high = 101, size = N)
22 #s2 is size of point for second scatter plot
23 s2 = 200 - (X1 + Y1)
24 # col2 is list of 3 * 20 = 60 colors Yellow Magenta Cyan
25 col2 = ['y', 'm', 'c'] * 20
26 ax2.scatter(X1, Y1, s = s2, c = col2, alpha = 0.9)
27 ax2.set_title("Plot2")
28 ax2.set_xlabel('Yellow MagentaCyan')
29 plt.show()
```

Explanation:

Lines 6-7: Two subplots are created.

Lines 10-11: randint() signature is randint(low, high=None, size=None, dtype='l') and its return is an int or an ndarray of ints. Low must be given. High is the maximum integer up to which but not including it will be generated. So if low = 0 and high is 101 then integers from 0 to 100 will be generated. Size indicates the size of the array generated. Here size = N which is 50 so an array of 50 integers from 0 to 100 will be generated.

Line 13: Here s1 is the size of the point generated. Here the size s1 has been taken as s1 = X1 + Y1, so the points with large X1 and large Y1 will be bigger.

Line 15: col1 is a list of 4 colors. Note 'r' is red, 'g' is green, 'b' is blue and 'k' is black

Line 16: Here you may use the scatter() method. Method used is: `scatter(X1, Y1, s = s1, c = col1, alpha = .5)`. Here s is the area of the point to be plotted and alpha indicates transparency of the point. Higher the value of alpha, less transparent it will be. The output is shown in Figure 21.26.

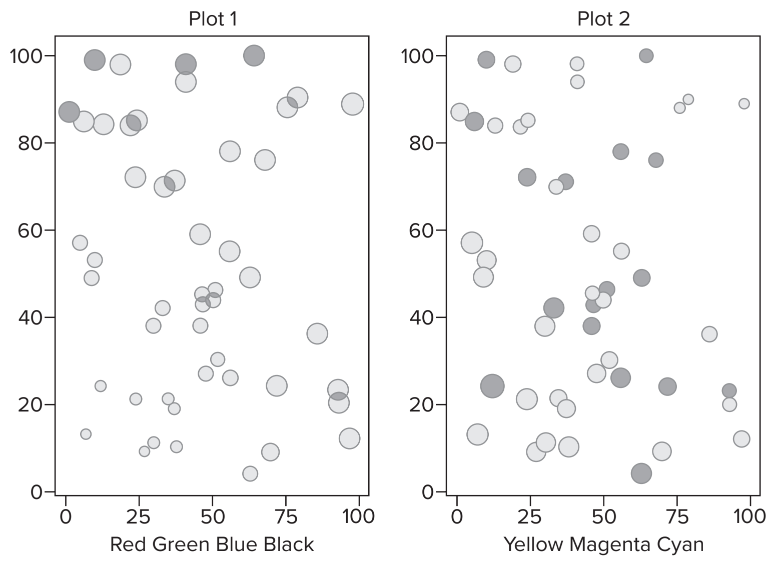


FIGURE 21.26 Output of previous script in the form of two scatter plots (on Jupyter). ***Note:*** The actual output will display all the colors which may not be visible in above screen shot.

For plotting histograms, you may use the .hist() function. The complete details of the hist() function on Jupyter can always be seen by using the help(plt.hist) function. The signature of the hist() function is:

```
Signature: plt.hist(x, bins=None, range=None, density=None, weights=None,
cumulative=False, bottom=None, histtype='bar', align='mid',
orientation='vertical', rwidth=None, log=False, color=None, label=None,
stacked=False, normed=None, hold=None, data=None, **kwargs)
```

The three important arguments for this discussion are respectively: x, bins and range. Only the first argument, i.e., x is mandatory. It should be an array or sequence. The second argument, i.e., bins is the number of bins in which the data is divided. The range argument is a tuple of lower and upper limit. If no value is given for range, the python automatically extracts the lowest and highest values as the limits and divides it into the given number of bins. For example, suppose the marks of students in a class are: x = [11, 12, 14, 21, 22, 23, 24, 27, 31, 33, 33, 34, 34, 34, 39, 44, 45, 45, 46, 47, 49] and you may want to divide it into bins starting from 10 to 50, i.e., 4 bins. Then the code will be as follows:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
m = 30 + np.floor(70 * np.random.rand(100))
x = [11, 12, 14, 21, 22, 23, 24, 27, 31, 33, 33, 34, 34, 34, 39, 44, 45, 45, 46,
47, 49]
b = 7
r = (30, 100)
plt.hist(m, bins = b, range = r)

plt.xlabel('Marks')
plt.ylabel('Number of students')
plt.title('Class result')
t = np.arange(25) #t is [0,1,2....24] is for yticks
plt.yticks(t)
plt.grid(True)
plt.show()
```

The output is shown in Figure 21.27.

21.5.2 Log Scales in Matplotlib

Many functions in scientific computing require log scales. In Matplotlib, you have the option of plotting either X or Y or both X and Y on a log scale. The following script shows the graph for three functions, i.e., $y = 10^x$, y = x and y = logx. These three functions are plotted on four different graphs. In graph 1, both X and Y are linear. In graph 2, X is linear while Y is log. In graph 3, X is log while Y is linear and in graph 4, both X and Y are log scales. The script is as follows:

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

```
x = np.linspace(0, 3, 100)
fig = plt.figure(figsize = (6,4))
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223) # treating the second row as having only 1 column
ax4 = fig.add_subplot(224)
ax1.set_title("x-linear/ y-linear")
ax1.plot(x, 10**x,x, x, x, np.log(x))
ax2.set_title("x-linear/ y-log")
ax2.set_yscale("log")
ax2.plot(x, 10**x,x, x, x, np.log(x))
ax3.set_title("x-log/ y-linear")
ax3.set_xscale("log")
ax3.plot(x, 10**x,x, x, x, np.log(x))
ax4.set_title("x-log/ y-log")
ax4.set_xscale("log")
ax4.set_yscale("log")
ax4.plot(x, 10**x,x, x, x, np.log(x))
```

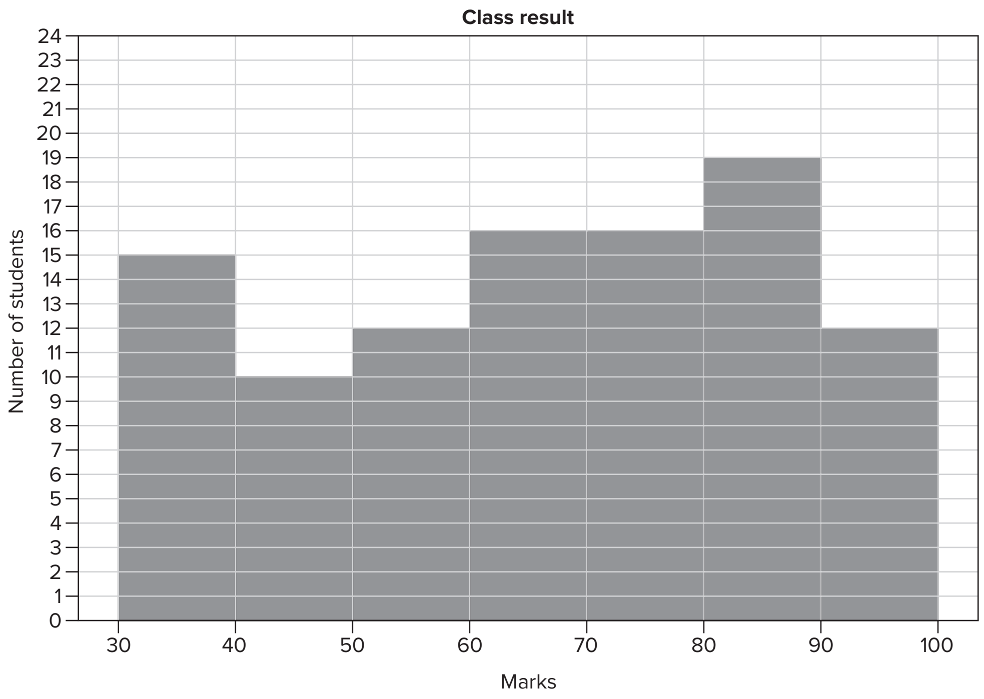


FIGURE 21.27 A histogram made by the hist() method. Student marks are on X-axis , while number of students scoring a particular range of marks are on Y-axis.

The output on Jupyter is shown in Figure 21.28.

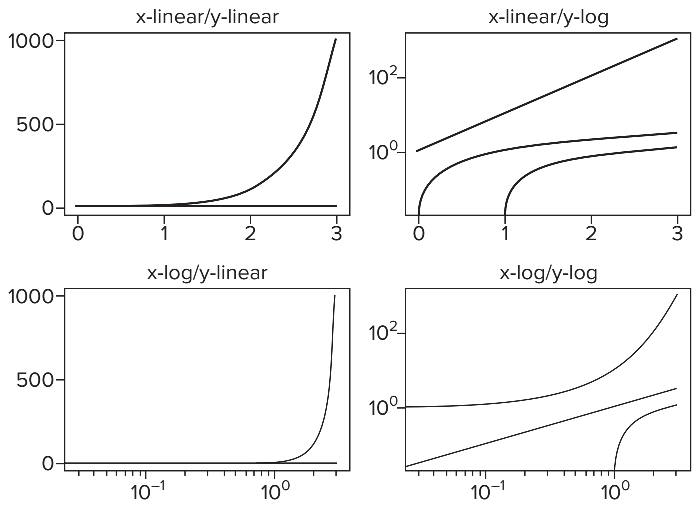


Figure 21.28 Four types of scales using three types of functions. The four types of scale are: (a) X (Linear) Y (Linear), (b) X (Linear) Y(log), (c) X (log) Y(Linear) and (d) X (Log) Y(Log). The three functions are (A) $y = 10^x$ (B) $y = x$ and (C) $y = \mathrm{Log}(x)$

Conceptual Questions

1. What is the difference between an object of class Figure and an object of class Axes?
2. What are the methods gcf() and gca()? Which one gets the Figure object and which one gets the Axes object?
3. What is the difference between the object of class Axes and class Axis?
4. What is the difference between Matplotlib and pyplot? Which is a package and which one is a module?
5. Questions on plt.subplots() method:
   - Read the following statement: The plt.subplots() method returns 2 objects namely an object of class Figure (conventionally called fig) and an object of class Axes (conventionally called ax)'. Explain this statement.
   - The ax object (of class Axes) returned by the plt.subplots() method may be 1D or 2D. What does this mean?
   - The plt.subplots() method has two parameters sharex and sharey with default values False. What is the role of these parameters? What will happen if you give values as True to these two parameters?
   - How many subplots will the code plt.subplots(3,4) create? There will be how many rows and how many columns?
6. What are primitives in Matplotlib? Name the four drawing primitives.
7. In plt, objects of the Axes class are required if you want to create subplots. However, if you intend to create a single plot, then plt provides a convenience function plot().
8. When you use the plot() method to draw a plot, you have a x-y pair of arguments. Apart from these x-y pair of arguments, you may also pass as parameter a "format string". What is a "format string"? What is its role?

9. What are “spines” and “ticks”?
10. What is the difference between the subplot() and the add_subplot() methods? It is said that subplot() is a “function of the pyplot module” while add_subplot() is a method of the Figure class. What does this mean?

Exercise

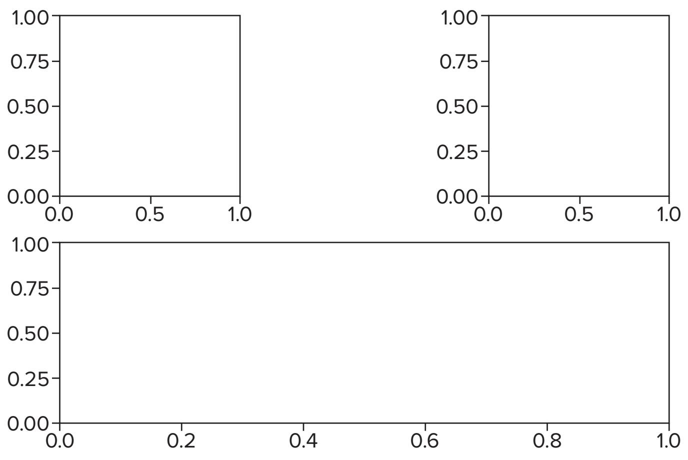
1. Examine the following code:

```python
from matplotlib import pyplot as plt
fig, ax = plt.subplots(2, 2)
print(type(ax))  # <class 'numpy.ndarray'>
print(type(ax[0][0]))  # <class 'matplotlib.axes._subplots.AxesSubplot'>
```

   The output of both the print statements is shown as comment. So type of ax is ndarray and type of ax[0][0] is AxesSubplot. Can you explain why? How many items does the ndarray ax contain? What is the shape of ax?
2. Now examine the following code:

```python
from matplotlib import pyplot as plt
fig1, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
print(type(ax1))  # <class 'matplotlib.axes._subplots.AxesSubplot'>
```

   The print output is again shown as comment. What is the type of ax1 AxesSubplot? Compare this method of creating Axes objects to the method in previous example. Can you think of a situation where it is better to create Axes objects as ax rather than as ((ax1, ax2), (ax3, ax4))? (***Hint:*** If you need to iterate over the Axes objects, then it is perhaps better to create Axes objects as an ndarray but since the created ndarray is 2D, you may have to “flatten” the array or write a loop which takes care of a 2D container).
3. Using subplot2grid() method, draw figure as follows:



Hint:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
ax1 = plt.subplot2grid(shape = (2, 3), loc = (0, 0))
ax2 = plt.subplot2grid((2, 3), (0, 2), colspan=1)
ax3 = plt.subplot2grid((2, 3), (1, 0), colspan=3)
plt.show()
```

Beyond Textbook/Assignment

1. **3D Plotting with Matplotlib**

Matplotlib has provision for doing 3D plotting. For this there is a class Axes3D. You can create an Axes3D object by using the projection='3D' keyword. So if you add the keyword-value pair projection = '3D' to either `add_axes` or `add_subplot` functions, you can do 3D plotting.

The following example creates two subplots, one in 3D and other in 2D. The 3D plot is a helix. Basically you can think of a helix as a circle in x-y plane and as you move around the circumference of the circle, you also rise in the z-direction. The second subplot is simply a movement in 2D and generates a circle.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure(figsize = (12, 6))
## 3-D Plot
ax1 = fig.add_subplot(1,2,1, projection = '3d')# 1,2,1 is 1st subplot in row of 2
N = 1000
# A holds 1000 points from 0 to 40*np.pi
# 40*np.pi means the helix does 20 complete circles. ie 2*np.pi for 1 circle
A = np.linspace(0, 40 * np.pi, N)
# R is the radius of the helix
R = 5
X = R * np.cos(A)
Y = R * np.sin(A)
#While A goes from 0 to 40*pi, Z goes from 0 to 5
# So in 20 circles, the helix rises by 5
Z = np.linspace(0, 5, N)
ax1.plot(X, Y, Z)
ax1.set_xlabel('X-axis')
ax1.set_ylabel('Y-axis')
ax1.set_zlabel('Z-axis')
##Plot z-axis in red in 3-D plot
Zax = np.linspace(0, 5, N)
Xax = np.zeros(N)
```

```
Yax = np.zeros(N)
ax1.plot(Xax, Yax, Zax, c = 'r')
## 2-D Plot
ax2 = fig.add_subplot(1,2,2)
A = np.linspace(0, 4 * np.pi, N)
X = R * np.cos(A)
Y = R * np.sin(A)
#Z = np.linspace(0, 5, N)
ax2.plot(X, Y)
ax2.set_xlabel('X-axis')
ax2.set_ylabel('Y-axis')
plt.show()
```

The output is shown in Figure 21.29.

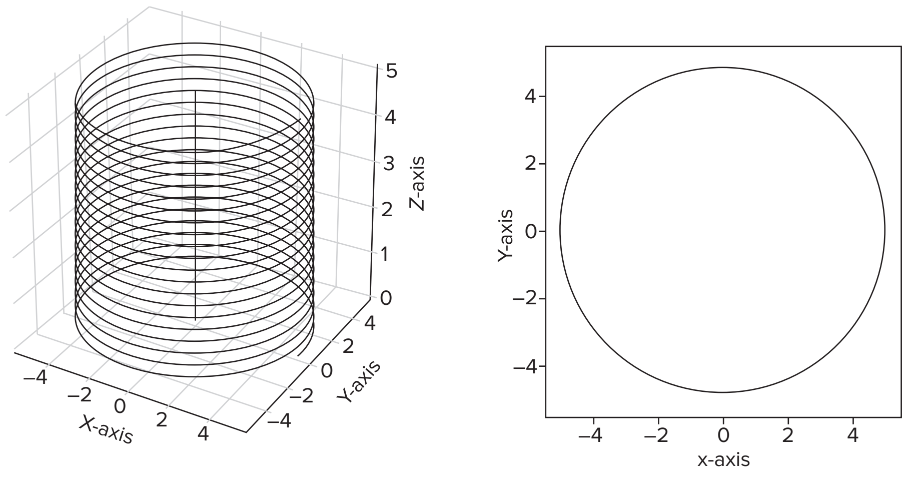


FIGURE 21.29 3D plotting by using projection as 3D. If you leave out the Z axis, then the 3D plot becomes a 2D plot.

2. **Using np.meshgrid() to plot in 3D**
   You may use the np.meshgrid() method to plot in 3D. Two methods namely (1) plot_surface() and (2) plot_wireframe() are used here. The signature of Axes3D.plot_surface() is as follows:

```
Signature: Axes3D.plot_surface(self, X, Y, Z, *args, **kwargs)
Docstring:Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports
color mapping by supplying the *cmap* argument.

The `rstride` and `cstride` kwargs set the stride used to sample the input data
to generate the graph. Defaults to 10. Raises a ValueError if both stride and
count kwargs are provided.

```

```
The `rcount` and `ccount` kwargs supersedes `rstride` and `cstride` for default
sampling method for surface plotting.  Will raise ValueError if both stride and
count are specified.
============= ================================================
Argument      Description
============= ================================================
*X*, *Y*, *Z* Data values as 2D arrays
*rstride*     Array row stride (step size)
*cstride*     Array column stride (step size)
*rcount*      Use at most this many rows, defaults to 50
*ccount*      Use at most this many columns, defaults to 50
*color*       Color of the surface patches
*cmap*        A colormap for the surface patches.
*facecolors*  Face colors for the individual patches
*norm*        An instance of Normalize to map values to colors
*vmin*        Minimum value to map
*vmax*        Maximum value to map
*shade*       Whether to shade the facecolors
```

The important points to note are:

- You must provide three arrays for X, Y and Z.
- You may provide rstride/cstride or rcount/ccount but not both.
- color and cmap are used for the surface patches

Similarly the signature of plot_wireframe is almost identical and is:

```
Signature: Axes3D.plot_wireframe(self, X, Y, Z, *args, **kwargs)
Docstring: Plot a 3D wireframe.
```

The following code shows how to use these two 3D methods to plot a 3D surface and a 3D wireframe. The meshgrid() function is also used. The script is as follows:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

fig = plt.figure(figsize = (12, 6))
ax = fig.add_subplot(111, projection = '3d')
X = np.linspace(-3, 3, 100)
Y = np.linspace(-3, 3, 100)
Xgrid, Ygrid = np.meshgrid(X, Y)

## Plot 1 is curved surface
Zval = Xgrid ** 2 + Ygrid ** 2
myP = ax.plot_surface(Xgrid, Ygrid, Zval, rstride = 2, cstride = 2, linewidth = 4,
cmap =cm.spring)

##Plot 2 is a plane wireframe
Z2 = 18 * np.ones((100, 100))
```

```
myP = ax.plot_wireframe(Xgrid, Ygrid, Z2, rstride = 15, cstride = 15, linewidth =
4, cmap =cm.flag)
plt.show()
```

The output is shown in Figure 21.30.

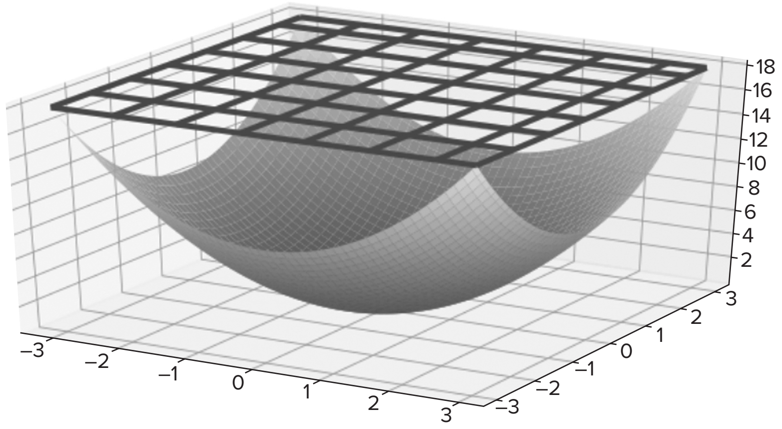


Figure 21.30 Output when two 3D methods are used to plot a 3D surface and a 3D wireframe. The meshgrid() function is also used (***Note:*** The screen shot may not show all the colors, but they will be visible in actual output)

3. Animation

In animation you may use a method provided by Matplotlib called FuncAnimation(). This function in turn calls a callback function. So before understanding the working of FuncAnimation(), you need to understand how a callback mechanism works. A callback function is a function which gets called from within another function.

Matplotlib has a module called animation. This module animation has a class FuncAnimation. The signature of this class is given in the following code (The class has many parameters, only the important ones are discussed here):

```
# Only some parameters are shown
Init signature: animation.FuncAnimation(fig, func, frames=None, init_func=None,
fargs=None, save_count=None, **kwargs)
Docstring: Makes an animation by repeatedly calling a function ``func``.
Parameters
(a)fig : matplotlib.figure.Figure-> The figure object that is used to get draw,
resize, and any    other needed events.
(b)func : callable. (The function to call at each frame.)
(c)frames : iterable, int, generator function, or None, (optional Source of data
to pass ``func`` and each frame of the animation)
(d)init_func : callable, optional. (A function used to draw a clear frame.)
(e)fargs : tuple or None, optional. Additional arguments to pass to each call to
*func*.
(f)interval : number, optional. Delay between frames in milliseconds.  Defaults
to 200.
```

The basic idea is to create your figure and then use a callback function that updates your figure. So the FuncAnimation works in the following manner:

- Create a Figure object (just like in any normal script using Matplotlib.)
- Create an optional init_func() function. This is optional. This is useful if you want a good initial picture to start the animation with.
- Create a callable function and use it to pass to parameter func. This function will be repeatedly called to do the animation.
- Now create an instance of FuncAnimation class. This class will repeatedly call the callback function which will update the various objects on the plot/subplot.

The following script creates a rotating line somewhat similar to what you see on the screen. Polar coordinates are used here. The script is as follows:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
p = np.pi
r = 50
angle_step = np.linspace(0,2*p,361)
#1-------------Set up initial line---------------
fig = plt.figure()
ax = fig.gca(projection = 'polar')
a_line, = ax.plot([0, 0], [0, r], color = 'k', linewidth = 1)
fig.canvas.draw()
#2-------call back function which updates the plot
def next_frame(theta):
    print(int(theta*180/p))# print angle as it is updated
    a_line.set_data([theta, theta],[0, r])
    return a_line,
#3----call FuncAnimation---------
my_anim = FuncAnimation(fig = fig, func = next_frame,
                    frames= angle_step, blit=True,
                    interval=20)
plt.show()
```

The output is shown in Figure 21.31.

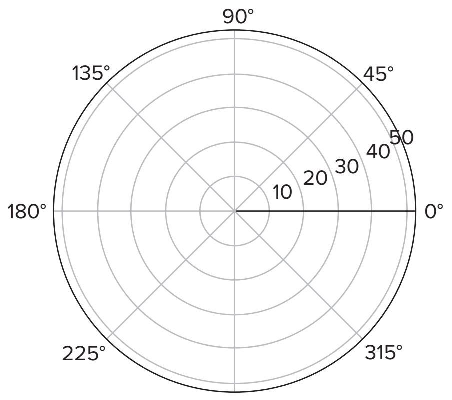


FIGURE 21.31 Output of a rotating rod on Jupyter. You may not see the animation on Jupyter, but you will see it on Spyder.

4. **Saving the animation**

In previous example you created an object of class FuncAnimation. FuncAnimation class provides a save() method to save the animation in form of "movie". The signature of the save() method on Jupyter is as follows:

```
from matplotlib.animation import FuncAnimation
?FuncAnimation.save
```

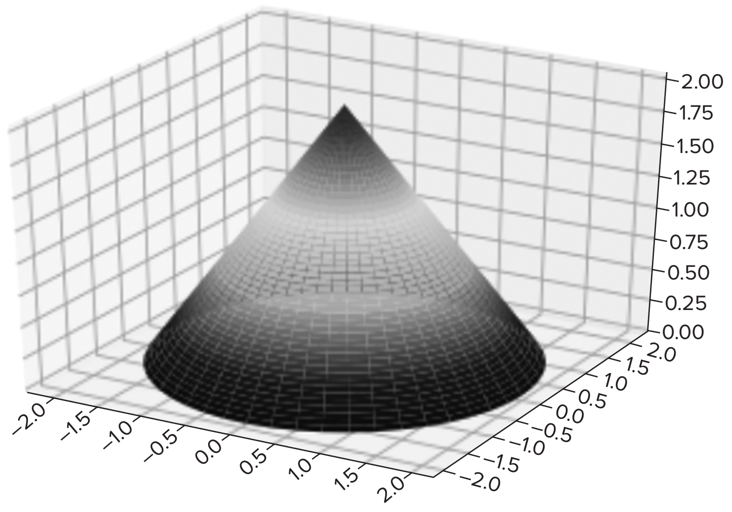
```
Signature: FuncAnimation.save(self, filename, writer=None, fps=None, dpi=None,
codec=None, bitrate=None, extra_args=None, metadata=None, extra_anim=None,
savefig_kwargs=None)
Docstring: Saves a movie file by drawing every frame.
Parameters #(Only some important parameters are listed)
(a)filename : str-> The output filename, e.g., :file:`mymovie.mp4`.
(b) writer : :class:`MovieWriter` or str, optional. -> A `MovieWriter` instance
to use or a key that identifies a class to use, such as 'ffmpeg' or 'mencoder'.
(c) fps : number, optional.-> Frames per second in the movie.
(d) dpi : number, optional.-> Controls the dots per inch for the movie frames.
(e) codec : str, optional.-> The video codec to be used. Not all codecs are
supported.
```

The following script saves the previous example code in form of a mp4 movie.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
#from matplotlib.animation import FuncAnimation, FFMpegWriter
p = np.pi
r = 50
angle_step = np.linspace(0,2*p,361)
#1-------------Set up initial line---------------
fig = plt.figure()
ax = fig.gca(projection = 'polar')
a_line, = ax.plot([0, 0], [0, r], color = 'k', linewidth = 1)
fig.canvas.draw()
#2-------call back function which updates the plot
def next_frame(theta):
    #print(int(theta*180/p))
    a_line.set_data([theta, theta],[0, r])
    return a_line,
#3----call FuncAnimation
my_anim = animation.FuncAnimation(fig = fig, func = next_frame,
                    frames= angle_step, blit=True,
                    interval=20)
print(type(my_anim))
plt.show()
#4--------save movie to file in mp4 format-------------
Writer = animation.writers['ffmpeg']
writer = Writer(fps=15, metadata=dict(artist='Me'), bitrate=1800)
path_to_save = r'C:\Temp\radar.mp4'# Specify file name and path on your machine
my_anim.save(path_to_save, writer=writer)
```

After the above code is run, a file radar.mp4 will be created in the C:\Temp folder(Or whatever path you have specified).

5. **Drawing a cone**
Draw a cone in 3D which should look similar to the following figure:



Hint: The following code draws a cone.

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

fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
two_pi = 2 * np.pi
# Cone is symmetric in x-y plane,
# So it is better to use cylindrical coordinates
base_radius = 2
r = np.linspace(0,base_radius,50)
t = np.linspace(0,two_pi,50)
R , T = np.meshgrid(r, t)
# Convert R and T to X and Y.
# X = Rcos(T), Y = Rsin(T)
X = R * np.cos(T)
Y = R * np.sin(T)
Z = 2 - (np.sqrt(X**2 + Y**2))

ax.plot_surface(X, Y, Z, cmap = cm.jet)

plt.show()
```