Lecture#5

May 23, 2023

1 Python for Signal Processing

Danilo Greco, PhD - danilo.greco@uniparthenope.it - University of Naples Parthenope

1.0.1 Lecture 5

This lecture will provide an overview on Matplotlib and Seaborn graphical libraries:

1. installation,
2. documentation,
3. main functions and applications,
4. practical examples.

1.0.2 What is Matplotlib?

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

It simplifies creation and customization of data plots:

- creation of high quality plots with few lines of code
- interactive figures with pan, zoom, etc.
- full control over styles and colors
- easy export in multiple file formats

1.0.3 What is Seaborn?

Seaborn is a fundamental library for statistical data visualization in Python.

It is based on matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics:

- operate on dataframes and arrays containing whole datasets,
- internally perform the necessary semantic mapping and statistical aggregation to produce informative plots,
- supports data exploration and interpretation

2 Install

It can be installed by executing the following command:

```
pip install matplotlib (or seaborn)
```

or

python -m pip install matplotlib (or seaborn)

NOTES:

- when installing with python version 3.x, replace pip or python with pip3 or python3 in the commands above.
- seaborn requires Python3.6 or greater and implicitly downloads and installs numpy, scipy, pandas and matplotlib if not already present

3 Documentation

The official Matplotlib documentation is available on this website whilst for Seaborn you can access documentation at this website.

Beginner's tutorials can be found for matplotlib and seaborn.

There is also an interesting book explaining how to use pandas for data analysis: >Wes McKinney, Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media.

4 Main functions and applications

The first thing to do is inform the python interpreter that we are using the packages. In each section, the first command asks the python interpreter to import the library and use an alias for quicker references within our code.

Again, numpy is imported for data generation.

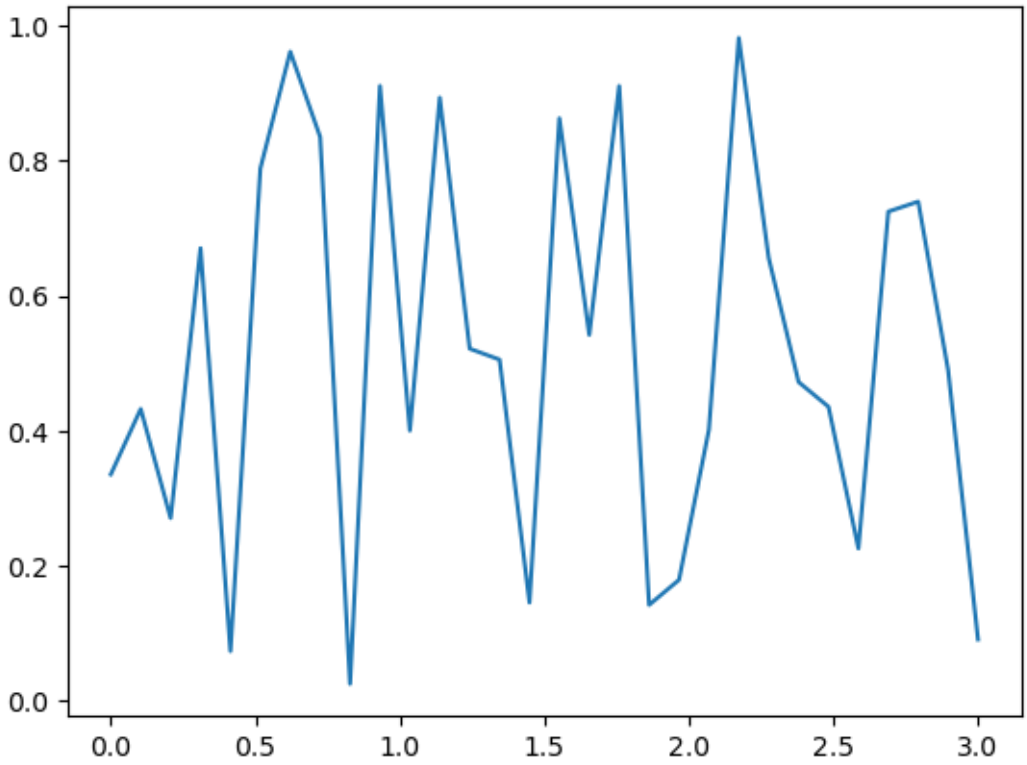
4.1 Matplotlib

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

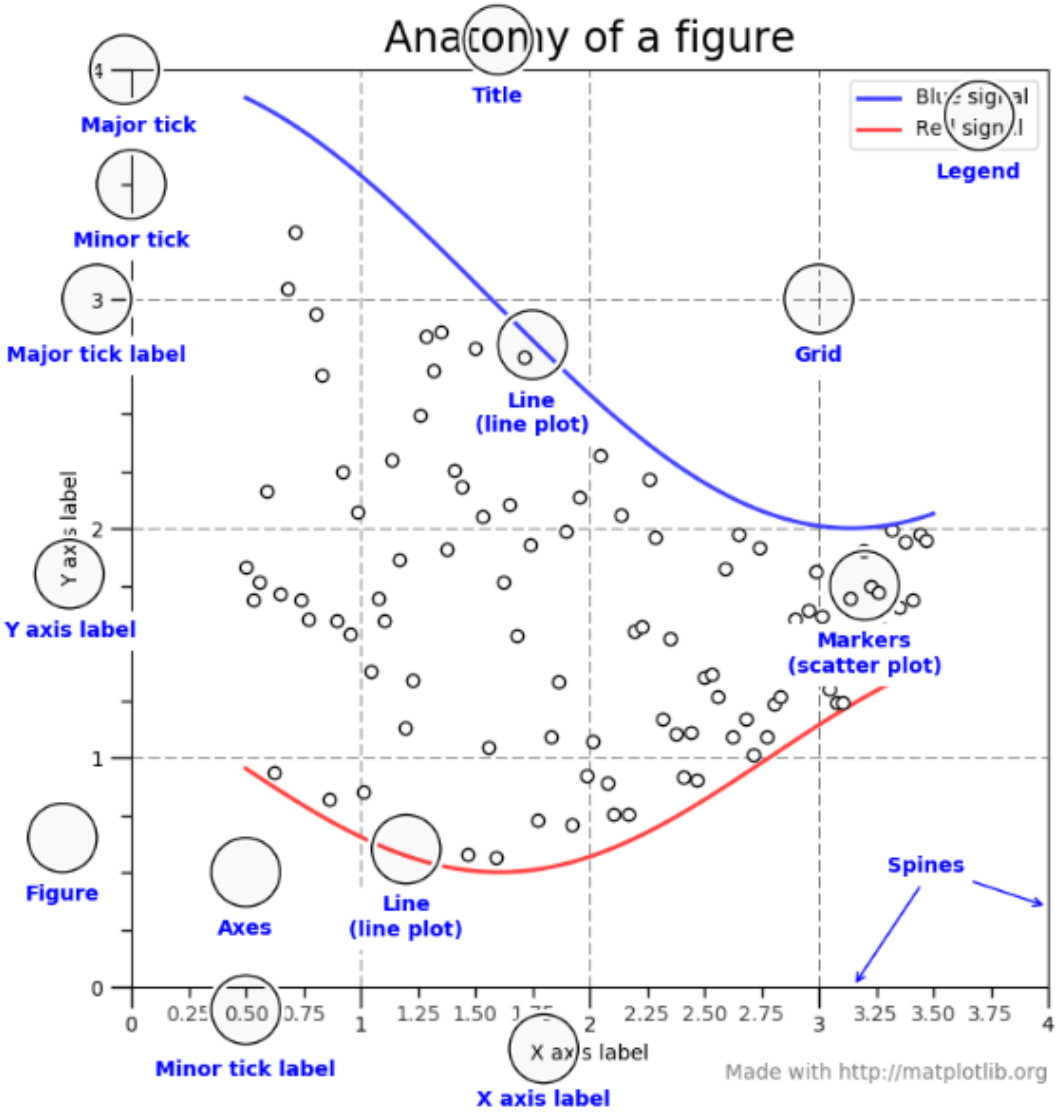
A first method to create a plot in matplotlib is:

[2]:
```
x = np.linspace(0,3,30)
y = np.random.random(30)

fig, ax = plt.subplots()
ax.plot(x, y)
```

[2]:
```
[<matplotlib.lines.Line2D at 0x2586e839610>]
```



Here we can see the detailed structure of a Matplotlib figure with its attribute names that can be customized:



Some notes on these objects: * **figure**: contains any number of axes and a canvas where object are plotted * **Axes**: its the region of figure that contains a data plot, therefore a figure can contain many axes. Each Axes has a title, its limits and can have labels for x and y respectively (or z for 3D plots). * **Axis**: linear graphical objects that contain ticks, located by the *Locator* object, and ticklabels customized via a *Formatter*.

Be aware of the difference between *Axes* and *Axis*.

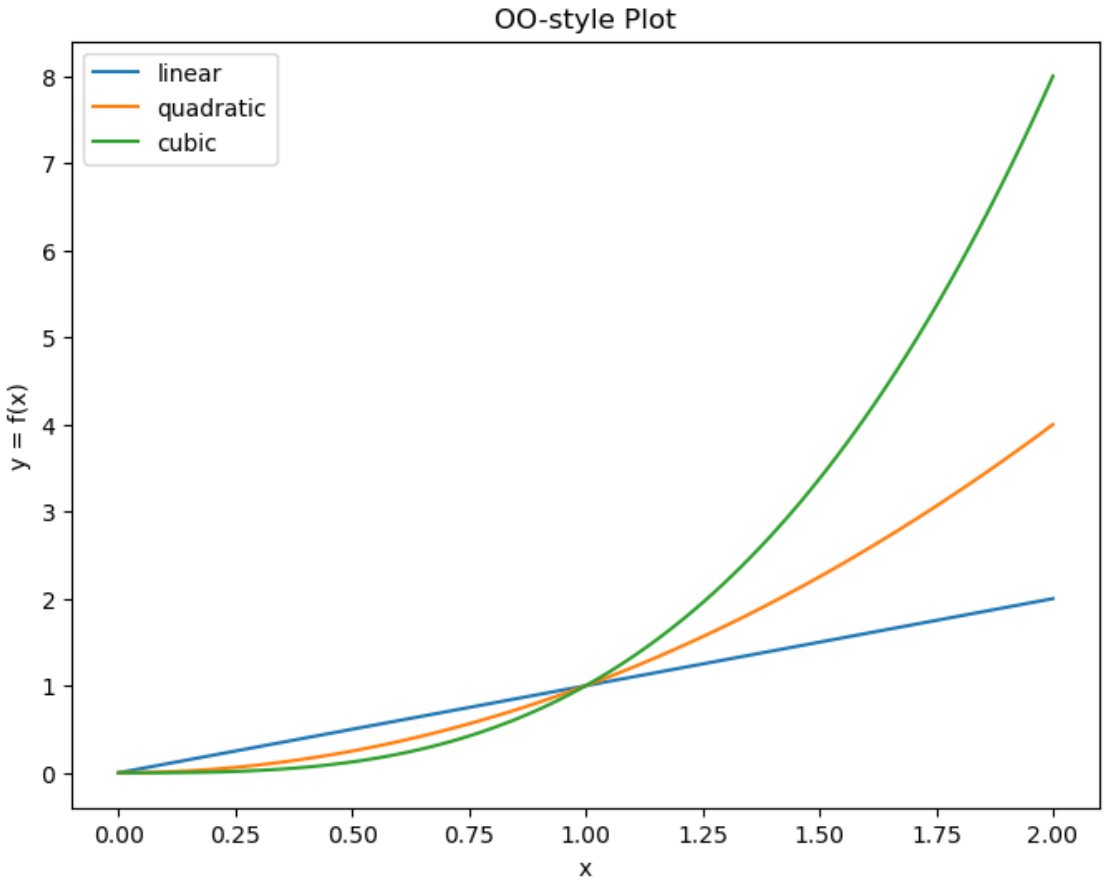
All plotting functions expect numpy.array objects as input therefore it's always better to convert our data before plotting, such as in the case of Pandas DataFrames.

A more complex plot example:

[3]:
```
# OO-style: preferred for use in functions
x = np.linspace(0, 2, 100)
```

```
fig, ax = plt.subplots(figsize=(8,6))  # Create a figure and an axes.
ax.plot(x, x, label='linear')          # Plot some data on the axes.
ax.plot(x, x**2, label='quadratic')    # Other plots are added to
ax.plot(x, x**3, label='cubic')        # the same figure
ax.set_xlabel('x')                     # Add an x-label to the axes.
ax.set_ylabel('y = f(x)')              # Add a y-label to the axes.
ax.set_title("OO-style Plot")          # Add a title to the axes.
ax.legend()                            # Add a legend.
```

[3]: <matplotlib.legend.Legend at 0x258708c5400>

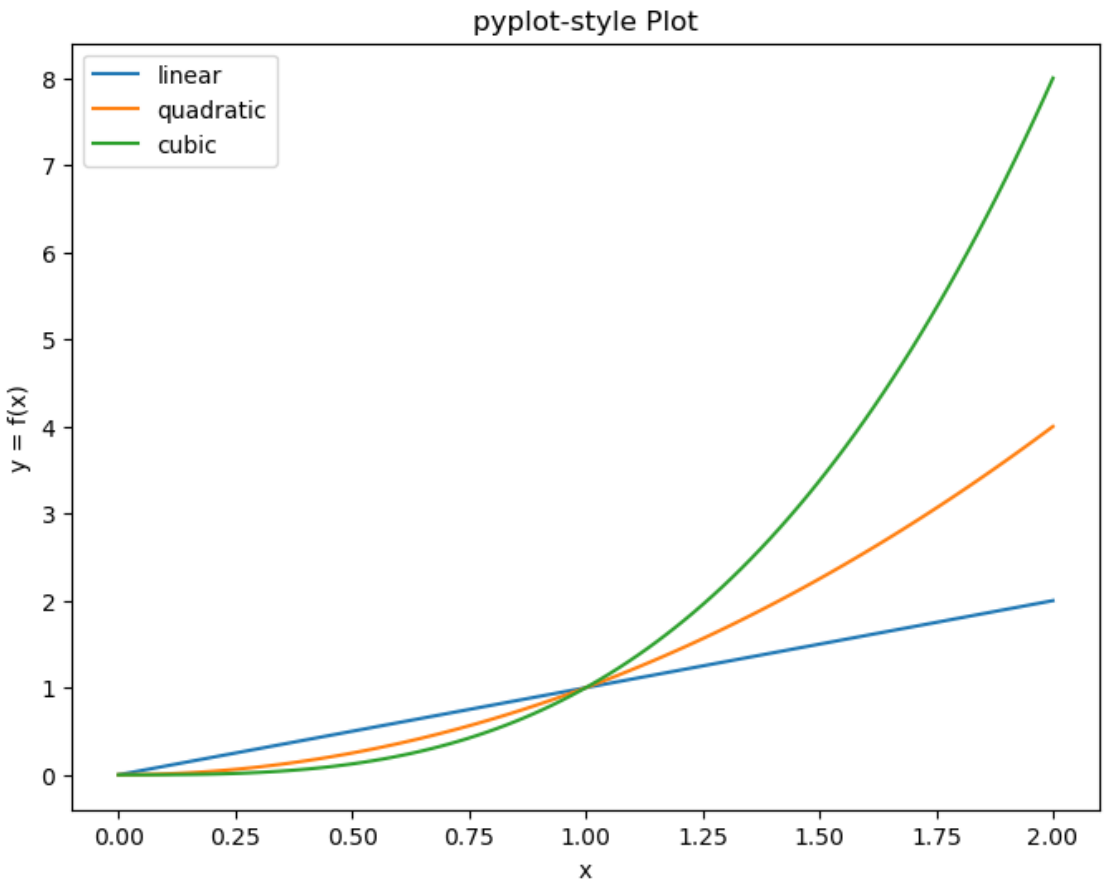


or alternatively:

[4]:
```
# pyplot-style: preferred for interactive plots
x = np.linspace(0, 2, 100)

plt.figure(figsize=(8,6))
plt.plot(x, x, label='linear')  # Plot some data on the (implicit) axes.
plt.plot(x, x**2, label='quadratic')  # etc.
```

```
plt.plot(x, x**3, label='cubic')
plt.xlabel('x')
plt.ylabel('y = f(x)')
plt.title("pyplot-style Plot")
plt.legend()
```

[4]: <matplotlib.legend.Legend at 0x258708c5610>

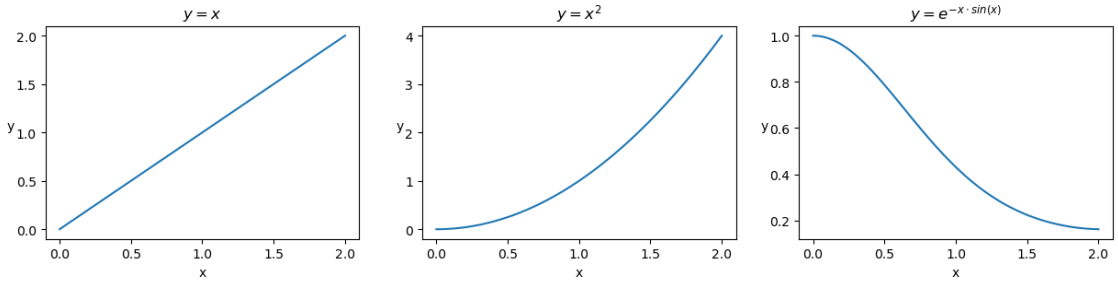


We can also use subplots to draw on different parts of the same figure and annotate the plots with LaTeX:

[5]:
```
x = np.linspace(0, 2, 100)

fig, (ax1,ax2,ax3) = plt.subplots(1,3,figsize=(15,3))
ax1.plot(x, x, label='linear')
ax2.plot(x, x**2, label='quadratic')
ax3.plot(x, np.exp(-x*np.sin(x)), label='exponential')
ax1.set_xlabel('x')
ax1.set_ylabel('y', rotation=0)
```

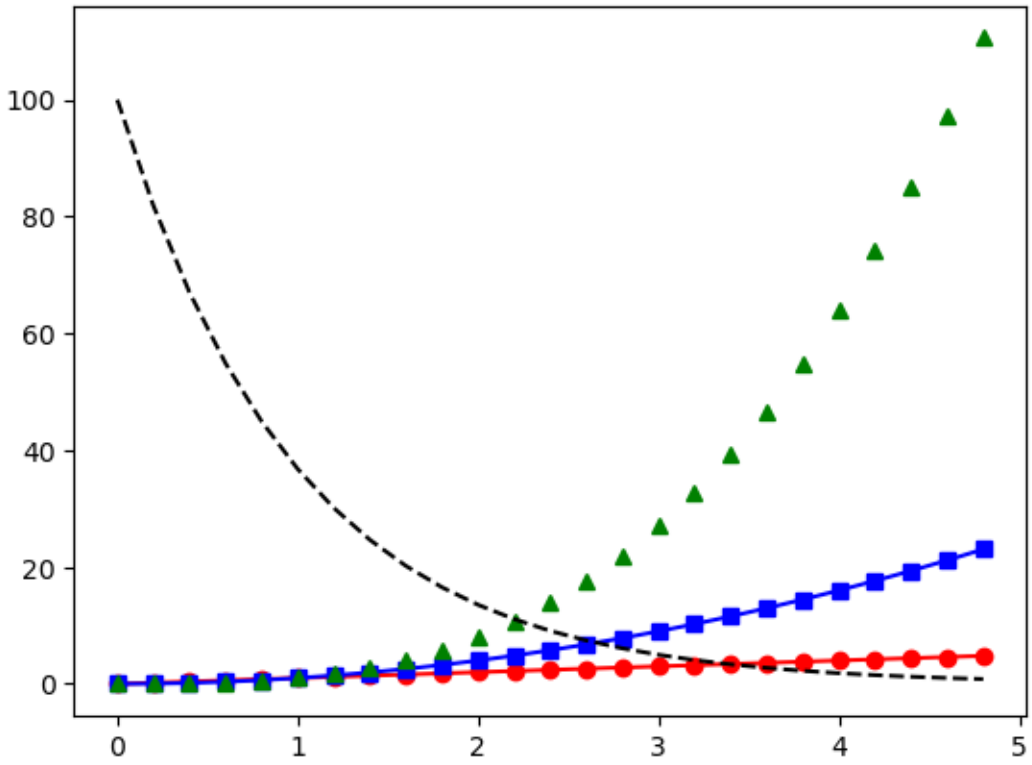
```
ax1.set_title('$y = x$')
ax2.set_xlabel('x')
ax2.set_ylabel('y', rotation=0)
ax2.set_title('$y = x^2$')
ax3.set_xlabel('x')
ax3.set_ylabel('y', rotation=0)
ax3.set_title('$y = e^{-x \cdot sin(x)}$')
```

[5]: `Text(0.5, 1.0, '$y = e^{-x \\cdot sin(x)}$')`



Colors and marker shapes can be customized by specifying additional attributes:

[6]:
```
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'ro-', t, t**2, 'bs-', t, t**3, 'g^', t, 100*np.exp(-t), 'k--')
plt.show()
```

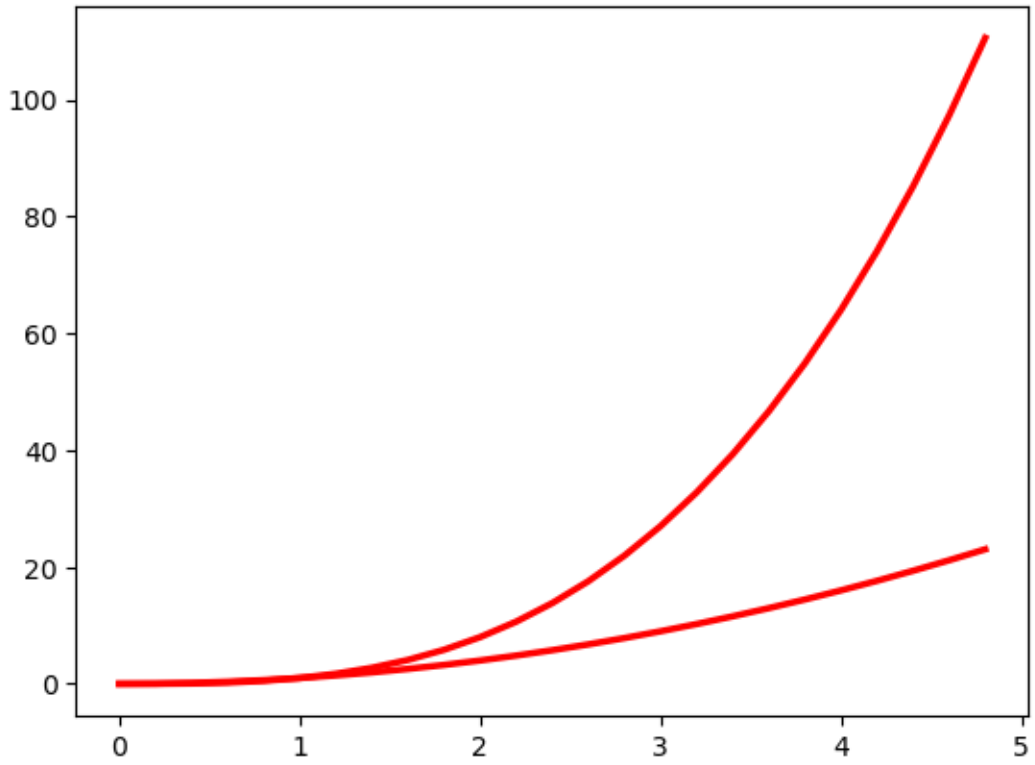


Many additional properties for 2D lines can be set for the plots by the setp function, whose reference can be found here.

An example of the setp usage is reported in the following cell:

[7]:
```
ln = plt.plot(t, t**3, t, t**2)
plt.setp(ln,color='r', linewidth=2.5)
```

[7]:
```
[None, None, None, None]
```



The list of properties available for the object can also be displayed by calling the setp function on the object:

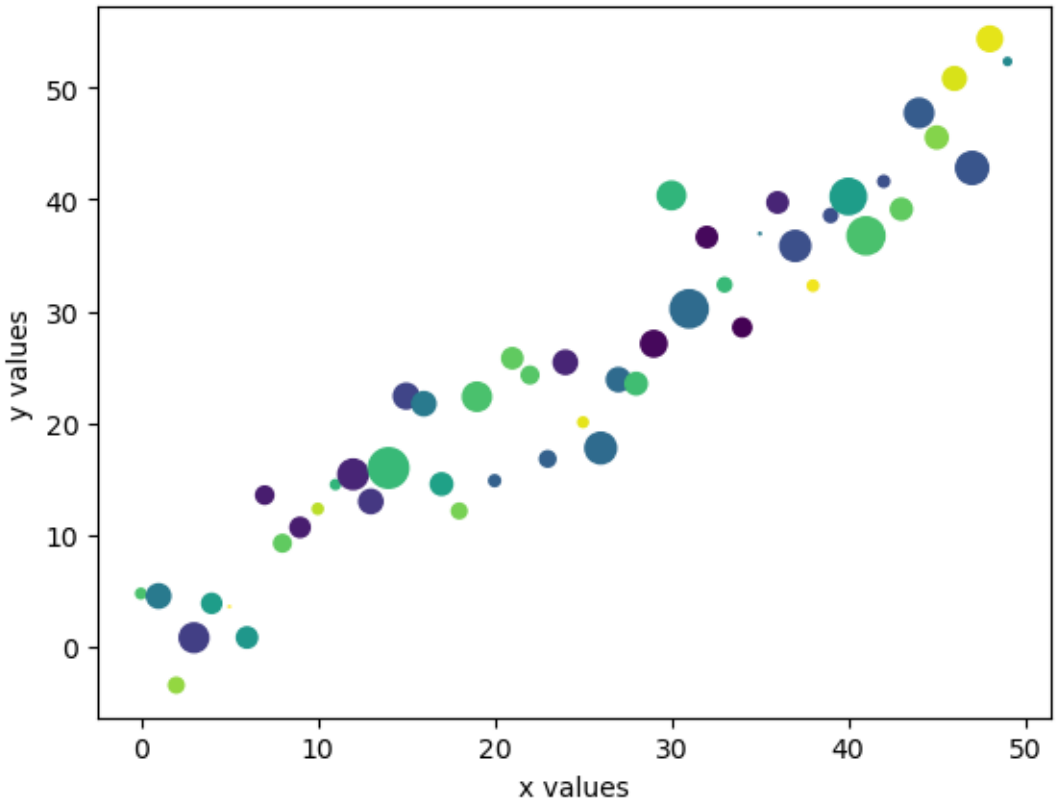
[8]:
```
plt.setp(ln)
```

```
  agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi
value, and returns a (m, n, 3) array
  alpha: scalar or None
  animated: bool
  antialiased or aa: bool
  clip_box: `.Bbox`
  clip_on: bool
  clip_path: Patch or (Path, Transform) or None
  color or c: color
  dash_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
  dash_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
  dashes: sequence of floats (on/off ink in points) or (None, None)
  data: (2, N) array or two 1D arrays
  drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'},
default: 'default'
  figure: `.Figure`
  fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
  gid: str
```

```
  in_layout: bool
  label: object
  linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), …}
  linewidth or lw: float
  marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
  markeredgecolor or mec: color
  markeredgewidth or mew: float
  markerfacecolor or mfc: color
  markerfacecoloralt or mfcalt: color
  markersize or ms: float
  markevery: None or int or (int, int) or slice or list[int] or float or (float,
float) or list[bool]
  path_effects: `.AbstractPathEffect`
  picker: float or callable[[Artist, Event], tuple[bool, dict]]
  pickradius: float
  rasterized: bool
  sketch_params: (scale: float, length: float, randomness: float)
  snap: bool or None
  solid_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
  solid_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
  transform: `.Transform`
  url: str
  visible: bool
  xdata: 1D array
  ydata: 1D array
  zorder: float
```

Column names can also be used when data is stored in a Pandas dataframe, such as in the following example where bubble color and size are related to proper data columns. A scatter plot is drawn, with parameter **c** assigned to specify the color of markers and **s** for their size:

[9]:
```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 5 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('x values')
plt.ylabel('y values')
plt.show()
```

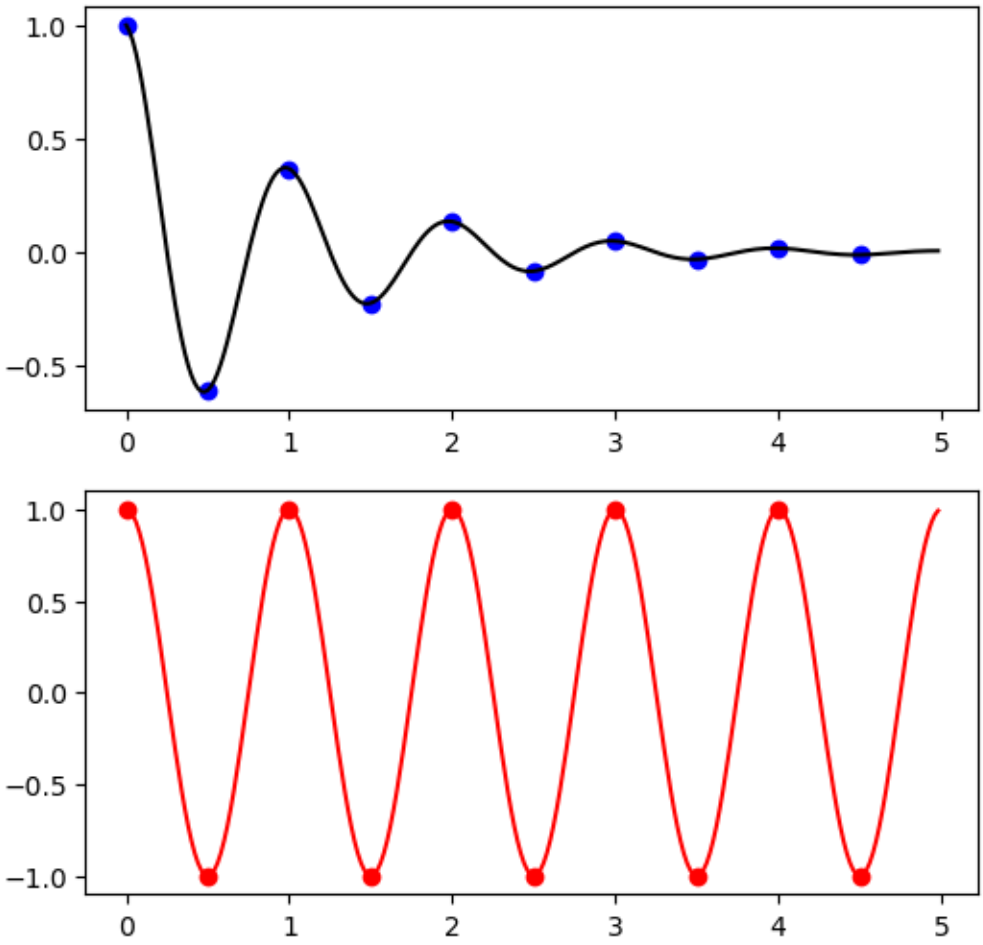


Quite often we need to compare different plots either horizontally or vertically: this is easily done with the help of subplots.

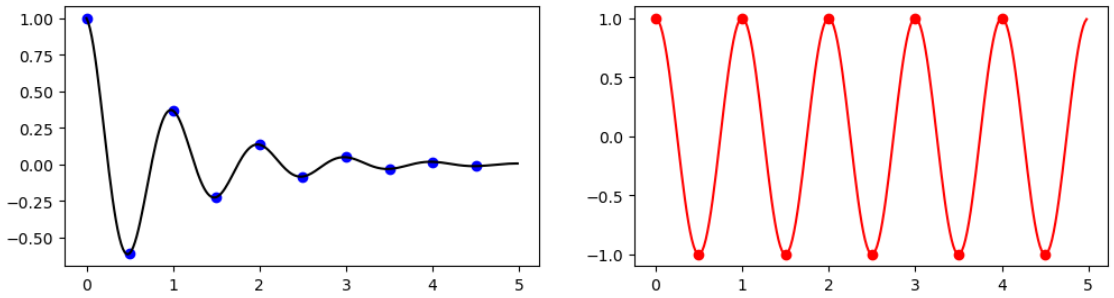
The subplot function requires us pass an integer coded value that declares the number of rows and columns we want to use and the position of current subplot. In the examples belows we first plot a vertical stack of subplots and then a horizontal span:

[10]:
```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

def g(t):
    return np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.5)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(figsize=(6,6))
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, g(t2), 'r-', t1, g(t1), 'ro')
```

```
plt.show()
```



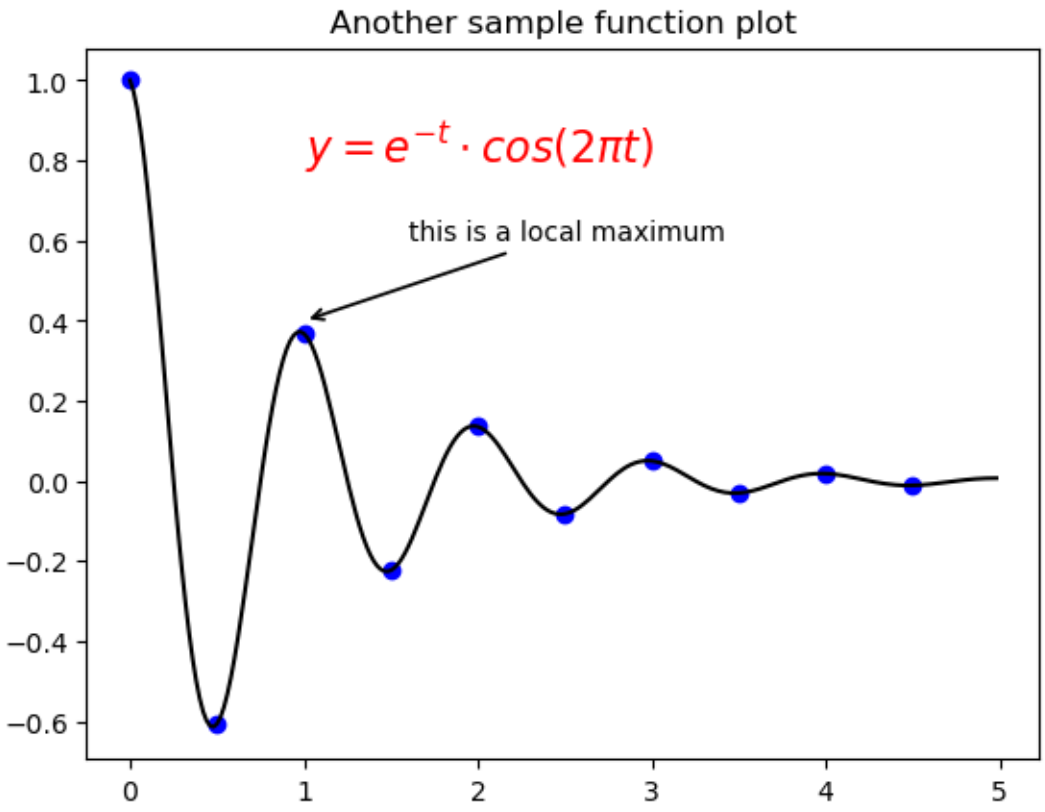
[11]:
```
plt.figure(figsize=(12,3))
plt.subplot(121)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(122)
plt.plot(t2, g(t2), 'r-', t1, g(t1), 'ro')
plt.show()
```



Further text can be inserted in a figure through the plt.text command as shown in the example below, where a desired font size and color are assigned.

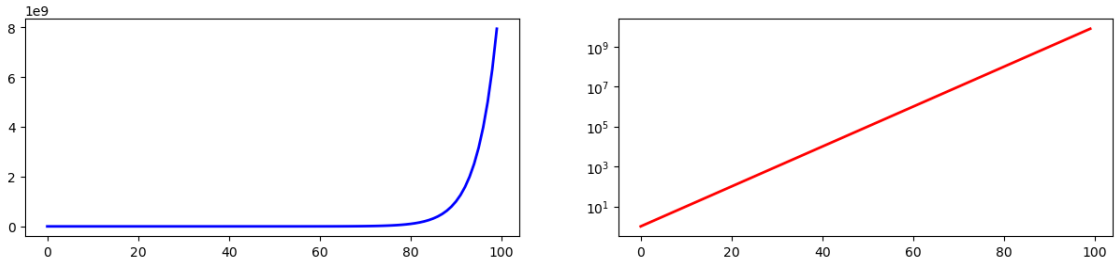
Figures can also be annotated everywhere in the canvas with the plt.annotate command that allows to add an arrow pointer to indicate special parts of the picture.

[12]:
```
plt.figure()
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.text(1,0.8,r'$y = e^{-t} \cdot cos(2 \pi t)$',fontsize=16, color='red')
plt.title('Another sample function plot')
plt.annotate('this is a local maximum', xy=(1, 0.4), xytext=(1.6,0.
↪6),arrowprops=dict(arrowstyle='->',linewidth=1.2))
plt.show()
```



Axes can be set to different scales to display linear or logarithmic plots by calling set_scale for the relevant object:

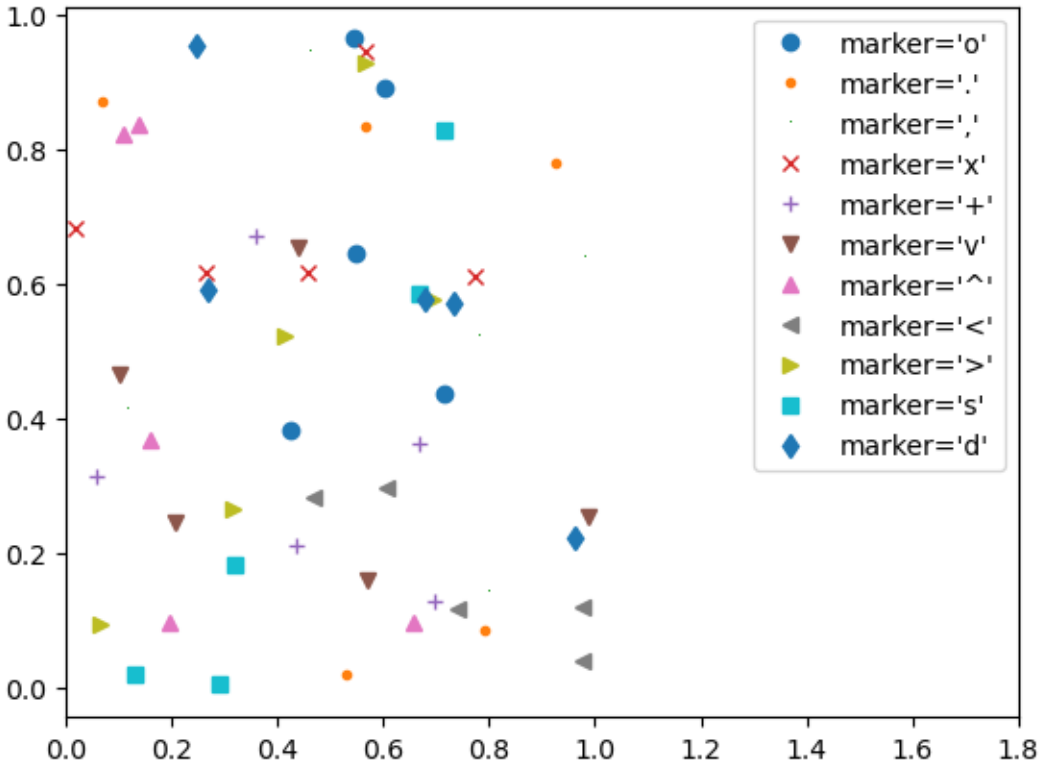
[13]:
```
a = [pow(10, i/10.0) for i in range(100)]

fig, (ax1,ax2) = plt.subplots(1,2,figsize=(15,3))
ax1.plot(a, color='blue', lw=2)
ax1.set_yscale('linear')
ax2.plot(a, color='red', lw=2)
ax2.set_yscale('log')
```



Another important representation is based on **scatter plots**, that are very useful to show how much a variable is affected by another. Additional feature variables can be represented by properly coding marker shape, color and size of markers in the scatter plot.

The first example shows all possible markers that can be usedin the plots:

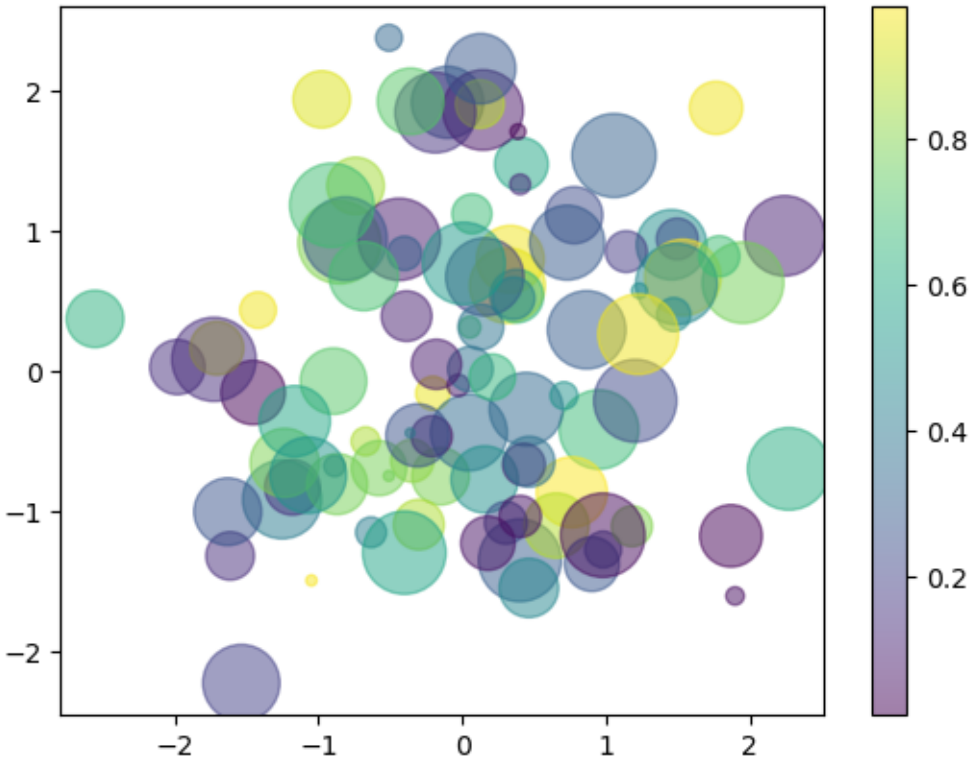
[14]:
```
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker,
             label="marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```



The following example shows how to associate colors and sizes to display additional numerical features related to x and y variables.

[15]:
```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='viridis')
```

```
plt.colorbar();
```



Bi-dimensional arrays can be plot in different ways: we've already seen how to plot as images but we can also treat them as meshes. The following example shows how to plot using pcolormesh:

[37]:
```
delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
XX, YY = np.meshgrid(x, y)
Z1 = np.exp(-XX**2 - YY**2)
Z2 = np.exp(-(XX - 1)**2 - (YY - 1)**2)
ZZ = (Z1 - Z2) * 2

fig, ax = plt.subplots()
c = ax.pcolormesh(XX, YY, ZZ, cmap='RdBu', vmin=-np.abs(ZZ).max(), vmax=np.
  ↪abs(ZZ).max(),shading='auto')
ax.set_title('pcolormesh')
fig.colorbar(c, ax=ax)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[37]:
```
<matplotlib.colorbar.Colorbar at 0x25879ff8bb0>
```

or to display only the level lines:

[38]:
```
fig, ax = plt.subplots()
c = ax.contour(XX, YY, ZZ)
ax.clabel(c, inline=1, fontsize=8)
ax.set_title('contour plot')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[38]:
```
Text(0.5, 1.0, 'contour plot')
```

Let's get into the details of contour plotting by generating and plotting $Z = X^2 + Y^2$:

[39]:
```
xlist = np.linspace(-3.0, 3.0, 50)
ylist = np.linspace(-3.0, 3.0, 50)
X, Y = np.meshgrid(xlist, ylist)

Z = np.sqrt(X**2 + Y**2)
```

we can plot the contours with default settings:

[40]:
```
fig, ax = plt.subplots()
cp = ax.contour(X, Y, Z)
ax.clabel(cp, inline=True, fontsize=10)
ax.set_title('Contour Plot')
ax.set_xlabel('x (cm)')
ax.set_ylabel('y (cm)')
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

or fill the contour and apply some customization on colors and the number of levels that will be displayed in our picture:

[41]:
```
levels = [0.5, 1.0, 1.5, 2.5, 3.0]
fig, ax = plt.subplots()
cp = ax.contour(X, Y, Z, levels, colors='w')
ax.clabel(cp, colors='black',inline=True, fontsize=10)

cp = plt.contourf(X, Y, Z, levels, cmap='Reds')
plt.colorbar(cp)

ax.set_title('Contour Plot')
ax.set_xlabel('x (cm)')
ax.set_ylabel('y (cm)')
```

```
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Pictures can be saved with the savefig command that supports many possible file formats, such as png, pdf, svg, etc.:

[42]:
```
fig.savefig('contours.svg')
```

[43]:
```
plt.gcf().canvas.get_supported_filetypes()
```

[43]:
```
{'eps': 'Encapsulated Postscript',
 'jpg': 'Joint Photographic Experts Group',
 'jpeg': 'Joint Photographic Experts Group',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'ps': 'Postscript',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format'}
```

Finally, with matplotlib it is also possible to create animated plots, such as a scrolling ECG graph, but **this feature is not easily implemented in jupyter**. The following code simulates the continuous plotting of data from a temperature sensor at a 2Hz rate:

[44]:
```
%matplotlib notebook
import datetime as dt
import matplotlib.pyplot as plt
import numpy as np
from time import sleep

# Create figure for plotting
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1, 1, 1)
#plt.ion()

fig.show()
fig.canvas.draw()

xs = []
ys = []
```

```
# simulate temperature read
def read_temp():
    temp = 25.0 + np.random.randn()
    return temp

for i in range(100):
    # Read temperature (Celsius)
    temp_c = round(read_temp(), 2)

    # Add x and y to lists
    xs.append(dt.datetime.now().strftime('%H:%M:%S.%f'))
    ys.append(temp_c)

    # Limit x and y lists to 20 items
    xs = xs[-20:]
    ys = ys[-20:]

    # Draw x and y lists
    ax.clear()
    ax.set_ylim([0,40])
    ax.plot(xs, ys)

    # Format plot
    plt.grid(linestyle=':')
    plt.xticks(rotation=45, ha='right')
    plt.subplots_adjust(bottom=0.30)
    plt.title('Temperature Monitor - Current value {:.2f}'.format(ys[-1]))
    plt.ylabel('Temperature (deg C)')
    fig.canvas.draw()
    sleep(0.5)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Matplotlib also supports the creation of complex 3D plots in a pretty simple way. This is the 3D version of the meshplot proposed before:

[45]:
```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(4,4))
ax = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(ax)

ax.plot_surface(XX, YY, ZZ, rstride=1, cstride=1, cmap=plt.cm.hot)
ax.contourf(XX, YY, ZZ, zdir='z', offset=-2, cmap=plt.cm.hot)
ax.set_zlim(-2, 2)
ax.view_init(elev=20., azim=-30)
```

```
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Using Axes3D, it's also possible to display tri-dimensional scatter plots. These are randomly generated data whose color and size can be associated to significant data features:

[36]:
```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
z = 1+2*rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

fig = plt.figure(figsize=(6,6))
ax = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(ax)
ax.scatter(x, y, z, c=colors, s=sizes, alpha=0.5, cmap='viridis')
ax.view_init(elev=20., azim=-30)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

4.2 Seaborn

This graphical library allows to draw all the previously described plots, but it has some additional interesting features that may become handy in data analytics. The following part focuses only on those extra plots.

Basically, seaborn provides a collection of functions through different modules: relational, distributional, categorical. For a detailed list of those functions please refer to the official documentation.

We'll go through the most interesting functions that were not previously presented in matplotlib.

[26]:
```
import seaborn as sns
```

[27]:
```
iris = sns.load_dataset('iris')
print(iris['species'].unique())
iris.head()
```

```
['setosa' 'versicolor' 'virginica']
```

[27]:

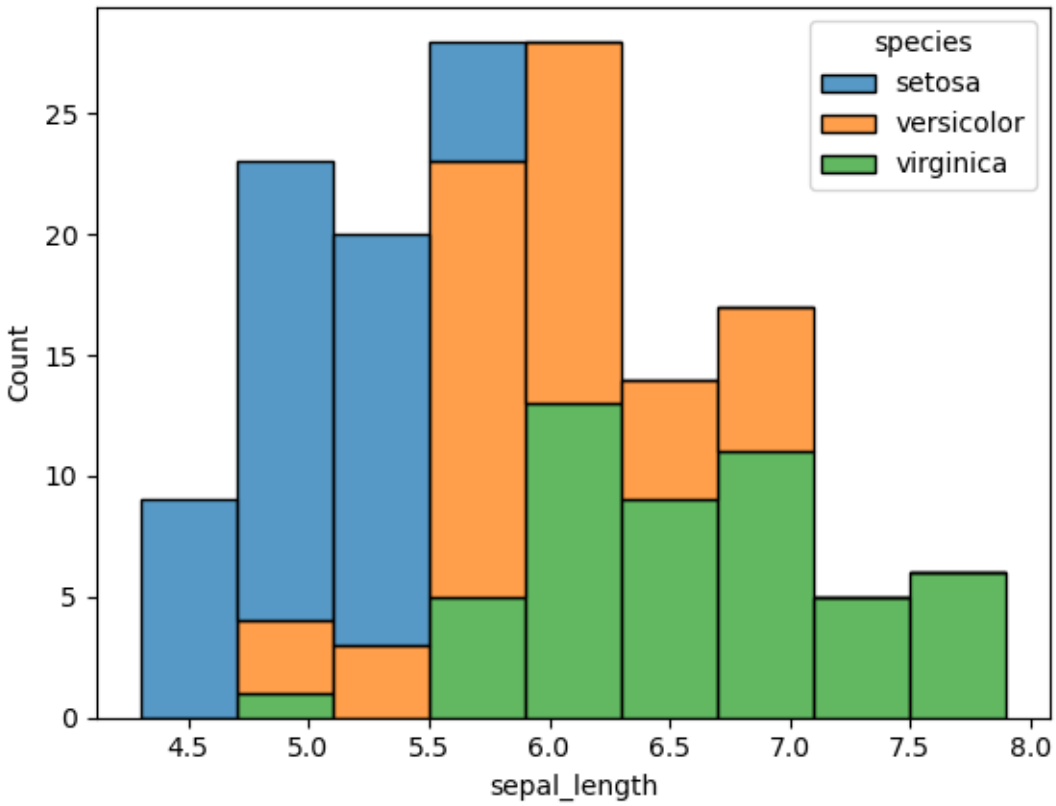
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa

```
4            5.0           3.6           1.4           0.2  setosa
```

The histogram can be used to represent the distribution of datapoints for a given feature:

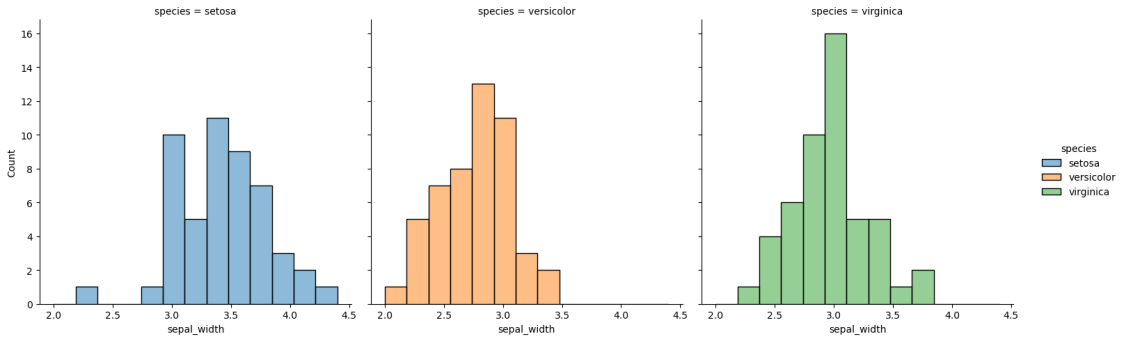
[28]:
```
%matplotlib inline
plt.figure()
sns.histplot(data=iris, x="sepal_length", hue="species", multiple="stack")
```

[28]:
```
<AxesSubplot:xlabel='sepal_length', ylabel='Count'>
```



The distribution of datapoints can also be displayed as separate classes, such as:

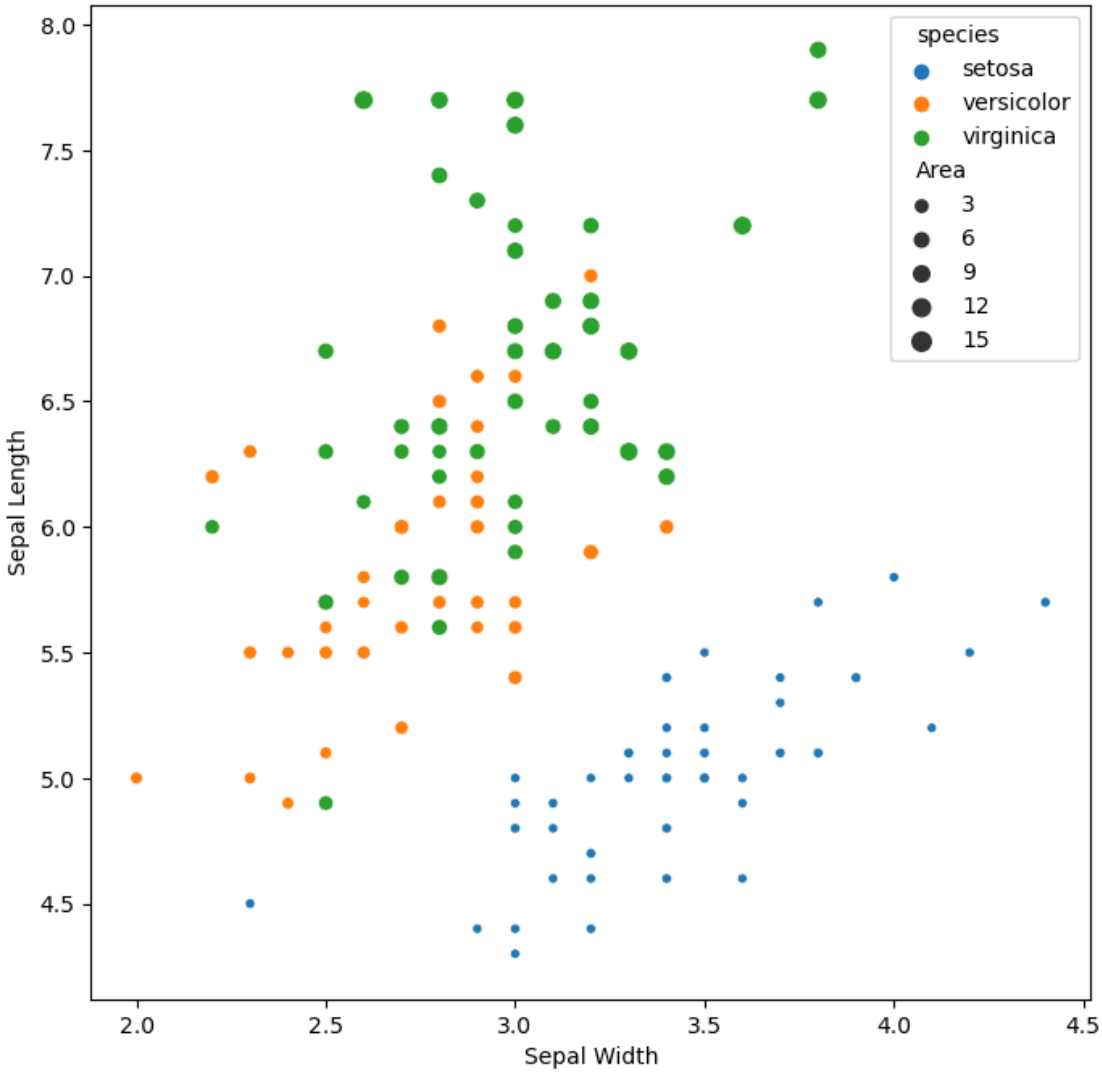
[29]:
```
sns.displot(data=iris, x="sepal_width", hue="species", col="species");
```



In order to assess the relationship between two variables a scatter plot may be used, eventually characterizing data by hue and/or size of the markers.

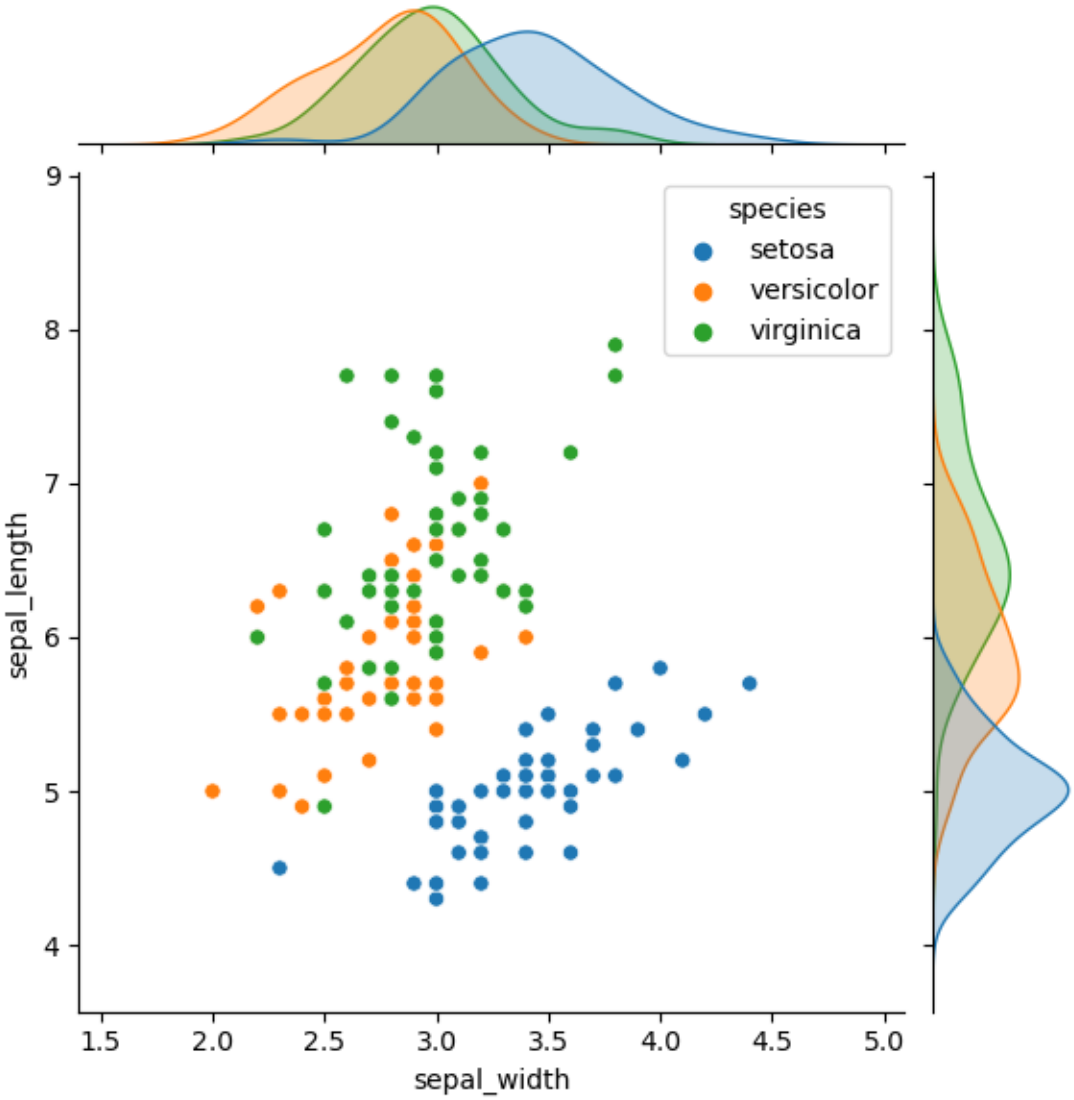
[30]:
```
fig, axs = plt.subplots(1, 1, figsize=(8, 8))
petal_area = iris.petal_length * iris.petal_width
petal_area.rename('Area',inplace=True)

g = sns.scatterplot(data=iris, x="sepal_width", y='sepal_length',hue="species",␣
↪size=petal_area, ax=axs)
axs.set_xlabel("Sepal Width")
axs.set_ylabel("Sepal Length")
```

[30]:
```
Text(0, 0.5, 'Sepal Length')
```



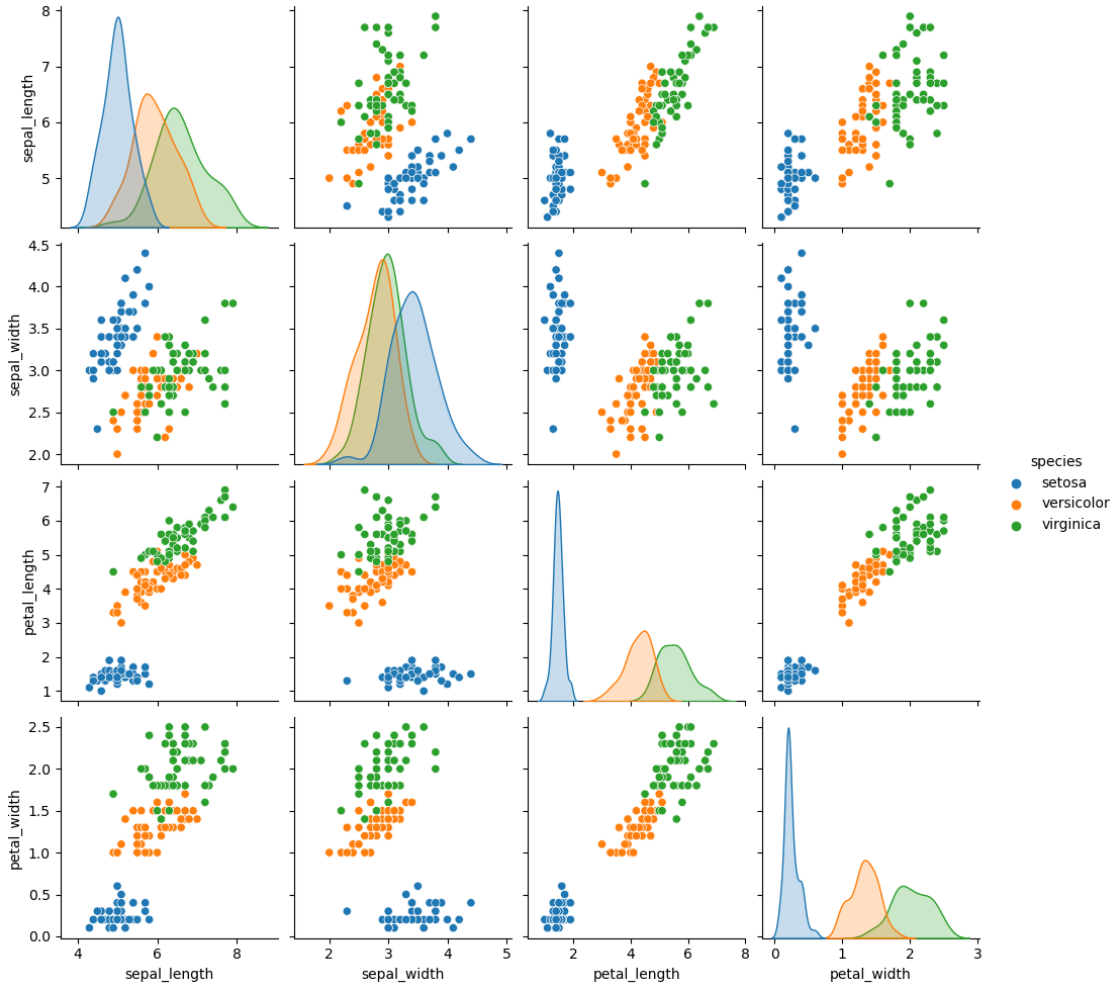
Another interesting plot available in seaborn is jointplot which combines the scatter plot with features distribution by class and produces the following output:

[31]:
```
sns.jointplot(data=iris, x="sepal_width", y='sepal_length', hue="species");
```



And the best way to visualize relationships among the pairs of features is through the pairplot. The diagonal represents the distribution of values for the selected feature and the off-diagonal plots display the pairwise scatter plot against the other features:

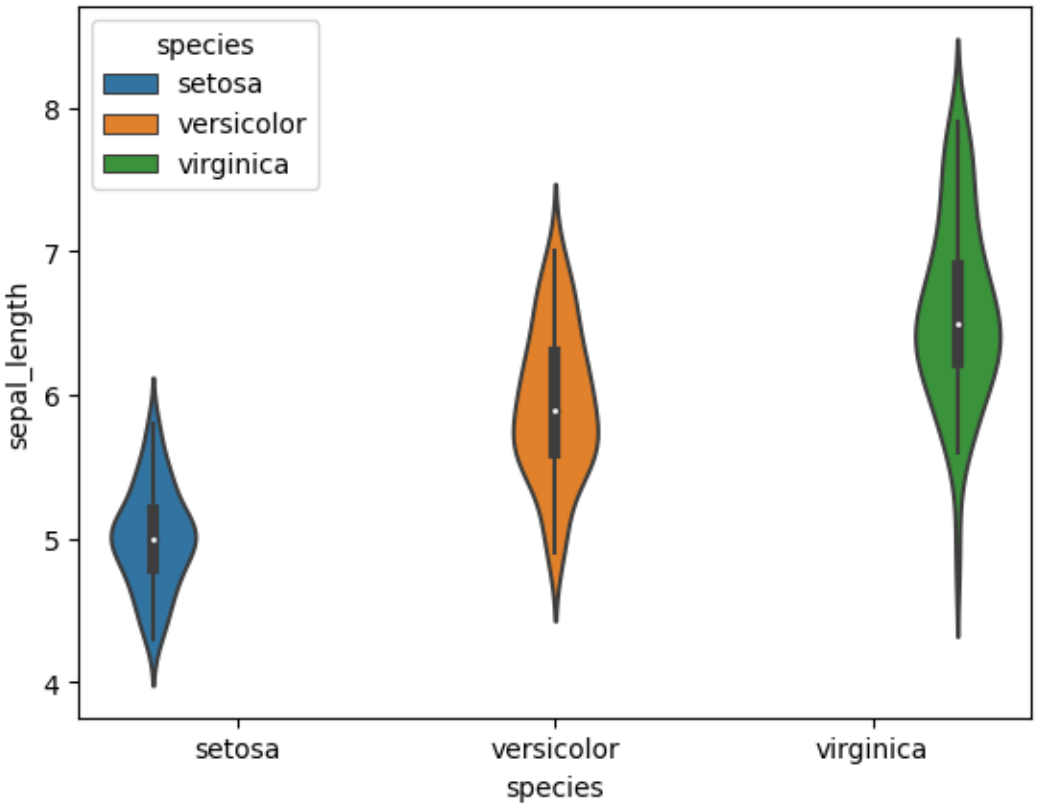
[32]:
```
sns.pairplot(data=iris, hue="species");
```



A violin plot plays a similar role as a box but adds the distribution of quantitative data across categorical variables such that those distributions can be compared.

Unlike a box plot, in which all of the plot components correspond to actual datapoints, the violin plot features a kernel density estimation of the underlying distribution.

[33]:
```
sns.violinplot(data=iris, x='species', y='sepal_length', hue='species');
```



The present lecture doesn't claim to be exhaustive about the two graphical libraries but tries to give you an overview of the most frequently used plots for data analytics and computational intelligence. Many other options are available on the library websites linked in the introduction.

4.2.1 Additional material

This is another live plotting example that works in a normal python script (outside jupyter): the following cell **must be saved in a python script** and executed by invoking python3 script.py.

[46]:
```
# to be removed for normal scripting
%matplotlib notebook
# to be removed for normal scripting
import datetime as dt
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np

# Create figure for plotting
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1, 1, 1)
xs = []
ys = []
```

```
# simulate temperature read
def read_temp():
    temp = 25.0 + np.random.randn()
    return temp

# This function is called periodically from FuncAnimation
def animate(i, xs, ys):

    # Read temperature (Celsius)
    temp_c = round(read_temp(), 2)

    # Add x and y to lists
    xs.append(dt.datetime.now().strftime('%H:%M:%S.%f'))
    ys.append(temp_c)

    # Limit x and y lists to 20 items
    xs = xs[-20:]
    ys = ys[-20:]

    # Draw x and y lists
    ax.clear()
    ax.set_ylim([0,40])
    ax.plot(xs, ys)

    # Format plot
    plt.xticks(rotation=45, ha='right')
    plt.subplots_adjust(bottom=0.30)
    plt.title('Temperature Monitor')
    plt.ylabel('Temperature (deg C)')

# Set up plot to call animate() function periodically
ani = animation.FuncAnimation(fig, animate, fargs=(xs, ys), interval=1000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[ ]: