Setting Up the Axes

In this lesson, we will learn to modify the axes to fit our needs.

Python provides the functionality to modify the axes to provide high-quality graphs.

Axes range

One of the most important things we need to configure is the axes of a graph. There are 3 possible ways:

1. Set range by default.
2. For tight-fitting, use `axis('tight')` on the axis object.
3. Use `set_ylim([ymin, ymax])` and `set_xlim([xmin, xmax])` methods on the axis object to set a custom range.
   - `ymin` and `ymax` are the lower and upper bounds for the y-axis.
   - `xmin` and `xmax` are the lower and upper bounds for the x-axis.

The following code implements the above-mentioned ways:

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

```
x = np.linspace(-5, 5, 100)
fig, axes = plt.subplots(3, 1, figsize=(12, 8))
axes[0].plot(x, 2 * x + 2, x, 2 * x + 4)
axes[0].set_title("default axes")

axes[1].plot(x, 2 * x + 2, x, 2 * x + 4)
axes[1].axis('tight')
axes[1].set_title("tight axes")

axes[2].plot(x, 2 * x + 2, x, 2 * x + 4)
axes[2].set_ylim([-10, 10])     # setting x
axes[2].set_xlim([-2, 2])
axes[2].set_title("custom axes");

fig.tight_layout()
```

Logarithmic scale

Scientists and engineers like to work with linear relationships as they are easier to work with. When there are non-linear relationships with large values, it becomes imperative to use the logarithmic scale as it makes the data look cleaner and easy to work with.

Using `matplotlib`, it is possible to set one or both axes to a logarithmic scale using the following methods:

1. `set_yscale('log')`
2. `set_xscale('log')`

Let’s look at the implementation of the following equation:

$$y = e^{2x}$$

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

x = np.linspace(0, 10, 100)
y = np.exp(2 * x)

fig, axes = plt.subplots(1, 2, figsize=(12, 8))

axes[0].plot(x, y)
axes[0].set_title("Normal")

axes[1].plot(x, y)
axes[1].set_yscale('log') # setting yscale to log
axes[1].set_title("Logarithmic")
```

Grid

To improve the readability of our graphs, it is useful to have grids. With the `grid()` method, you can turn on the grid and even customize it using the same arguments we used for `plot()` in this lesson.

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

x = np.linspace(-10, 10, 100)
y = x**2 + (2 * x) + 6

fig, axes = plt.subplots(1, 2, figsize=(12, 8))

axes[0].plot(x, y)
axes[0].grid(True)
axes[0].set_title("Deafult Grid")

axes[1].plot(x, y)
axes[1].grid(color='g', linewidth=1, linestyle='dashed')
axes[1].set_title("Custom Grid")
```

We can also change the properties of axis spines:

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

x = np.linspace(-10, 10, 100)
y = x**2 + (2 * x) + 6

fig, axes = plt.subplots(figsize=(12, 8))
axes.spines['top'].set_color('r')   # setting color of top spine
axes.spines['bottom'].set_color('b')    # setting color of bottom spine
axes.spines['bottom'].set_linewidth(2)  # setting linewidth of bottom spine
axes.spines['left'].set_color('g')     # setting color of left spine
axes.plot(x, y)
```

Twin axis

Due to reactance in the wire, suppose there is a phase difference between the values of current and voltage in the wire and you want to examine this values of current and voltage in the wire and you want to examine this difference by plotting them on the same graph, but there is an issue: units for

current, denoted by (I), are *amperes* (A) and units for voltage, denoted by (V), are *volts* (V).

Matplotlib provides the functionality of plotting twin axes using the `twiny` and `twinx` methods to resolve the issue of plotting variables having different units on the same graph.

Let's plot the following equations:

$$I = 5\ sin(\omega t)$$

$$V = 24\ sin(\omega t + \theta)$$

where ω is the angular frequency, and θ is the phase difference.

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

t = np.linspace(0, np.pi/100, 100)    # an array of time
print(t)
W = 2 * np.pi * 50              # angular frequency
theta = np.pi/2    # phase difference

I = 5 * np.sin(W * t)    # values of current
V = 24 * np.sin(W * t + theta)    # values of voltage

fig, axesI = plt.subplots(figsize=(12, 8))

axesI.plot(t, I, 'r')
axesI.set_xlabel("time(s)")
axesI.set_ylabel("Current(A)", color='red')

axesV = axesI.twinx()  # instantiate a second axes that shares the same x-axis

axesV.plot(t, V, 'b')
axesV.set_ylabel("Voltage(V)", color='blue')
```

Ticks

With `set_xticks` and `set_yticks`, we can specify the exact ticks we want on the graph. We use the `set_xticklabels` and `set_yticklabels` methods to set custom ticks. Let's see the implementation below:

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

t = np.linspace(0, 2 * np.pi, 100)
I = 5 * np.sin(t)

fig, axes = plt.subplots(figsize=(12, 8))

axes.plot(t, I, 'r')
axes.set_xlabel("time(s)")
axes.set_ylabel("Current(A)")

axes.set_xticks(np.linspace(0, 2 * np.pi, 5)) # specifying ticks
axes.set_xticklabels(['0', r'$0.5\pi$', r'$\pi$', r'$1.5\pi$', r'$2\pi$' ]) # custom ticks
```

Next lesson discusses other commonly used graphs in Python.