Visualization with Matplotlib

Matplotlib is a **multiplatform data visualization** library **built on NumPy** arrays and designed to work with the **broader SciPy stack.**

One of Matplotlib's most important **features** is its ability to play well with **many operating systems** and **graphics backends.**

It has led to a **large user base,** which in turn has led to an **active developer base** and Matplotlib's powerful tools and ubiquity within the scientific Python world.

In recent years, however, the **interface** and style of Matplotlib have begun to **show their age.**

Newer tools like `ggplot` and `ggvis` in the **R language,** along with web visualization toolkits based on D3js and HTML5 canvas, often make Matplotlib feel clunky and old-fashioned.

Recent Matplotlib versions make it relatively easy to set **new global plotting styles,**

and people have been developing **new packages** that build on its powerful internals to drive Matplotlib via cleaner, more modern APIs—for example, 'Seaborn', 'ggpy', 'HoloViews'.

Even with wrappers like these, **it is still often useful to dive into Matplotlib's syntax** to adjust the final plot output.

General Matplotlib Tips

Before we dive into the details of creating visualizations with Matplotlib, there are a few **useful things** you should know about using the package.

Importing Matplotlib

Just as we use the `np` shorthand for NumPy and the `pd` shorthand for Pandas, we will use some **standard shorthands** for Matplotlib imports:

In [ ]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

The `plt` **interface** is what we will use most often, as you shall see throughout this part of the book.

Setting Styles

We will use the `plt.style` directive to choose appropriate **aesthetic styles** for our figures.

Here we will set the `classic` style, which ensures that the plots we create use the **classic Matplotlib style:**

In [ ]:
```
plt.style.use('classic')
```

Throughout this chapter, we will adjust this style as needed.

show or No show? How to Display Your Plots

A visualization you can't see won't be of much use, but just how you **view** your Matplotlib plots depends on the **context.**

The best use of Matplotlib differs depending on how you are using it;

roughly, the t**hree applicable contexts** are using Matplotlib in a script, in an IPython terminal, or in a Jupyter notebook.

Plotting from a Script

If you are using Matplotlib from within a **script,** the function `plt.show` is your friend.

`plt.show` starts an event loop, looks for all currently active `Figure` objects, and opens one or more interactive windows that display your figure or figures.

So, for example, you may have a file called *myplot.py* containing the following:

```python
# file: myplot.py
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```

You can then **run** this script from the **command-line prompt**, which will result in a window opening with your figure displayed:

```
$ python myplot.py
```

The `plt.show` command does a lot under the hood, as it must interact with your **system's interactive graphical backend.**

The **details** of this operation can **vary** greatly from system to system and even installation to installation,

but Matplotlib does its best to **hide** all these details from you.

One thing to be aware of:

the `plt.show` command should be used **only once** per Python session, and is most often seen at the very end of the script.

Multiple `show` commands can lead to **unpredictable** backend-dependent **behavior**, and should mostly be avoided.

Plotting from an IPython Shell

Matplotlib also works seamlessly within an **IPython shell. bold text** IPython is built to work well with Matplotlib if you specify **Matplotlib mode.**

To enable this mode, you can use the `%matplotlib` **magic command** after starting `ipython`:

```
In [1]: %matplotlib
Using matplotlib backend: TkAgg

In [2]: import matplotlib.pyplot as plt
```

At this point, any `plt` plot command will cause a **figure window** to open, and **further commands** can be run to **update** the plot.

Some changes (such as modifying properties of lines that are already drawn) will not draw automatically: to **force an update,** use `plt.draw`.

Using `plt.show` in IPython's Matplotlib mode is not required.

Plotting from a Jupyter Notebook

The **Jupyter notebook** is a browser-based interactive data analysis tool that can combine narrative, code, graphics, HTML elements, and much more into a single executable document.

Plotting interactively within a Jupyter notebook can be done with the `%matplotlib` command, and works in a **similar** way to the **IPython shell.**

You also have the option of **embedding graphics** directly in the notebook, with two possible options:

- `%matplotlib inline` will lead to **static** images of your plot embedded in the notebook.

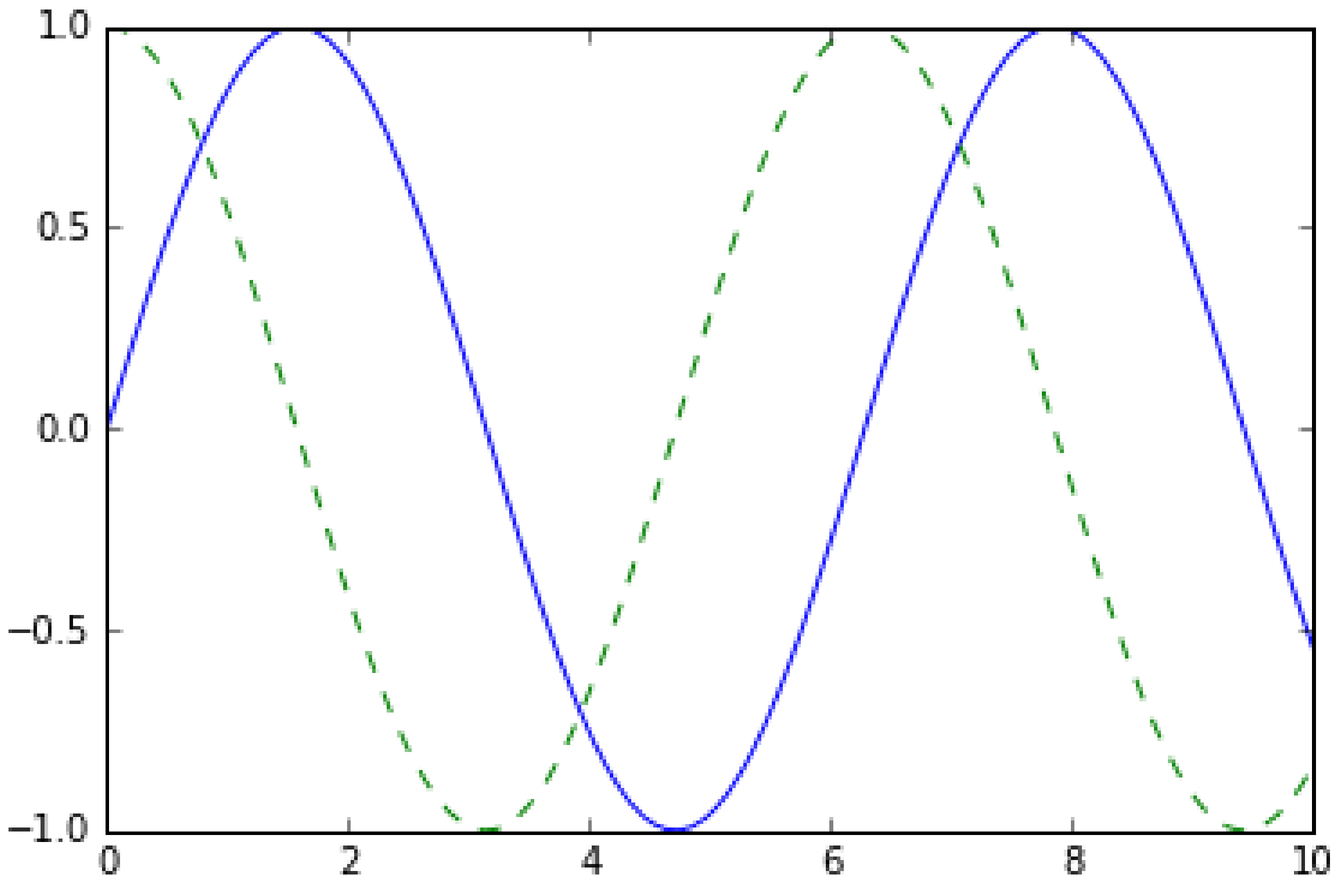
- `%matplotlib notebook` will lead to **interactive** plots embedded within the notebook.

For **this book,** we will generally stick with the **default,** with figures rendered as static images:

In [ ]:
```
%matplotlib inline
```

In [ ]:
```
import numpy as np
x = np.linspace(0, 10, 100)

fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--');
```



Saving Figures to File

One nice feature of Matplotlib is the ability to **save figures** in a wide variety of formats.

Saving a figure can be done using the `savefig` **command.**

For example, to save the previous figure as a PNG file, we can run this:

```
In [ ]: fig.savefig('my_figure.png')
```

We now have a file called *my_figure.png* in the current working directory:

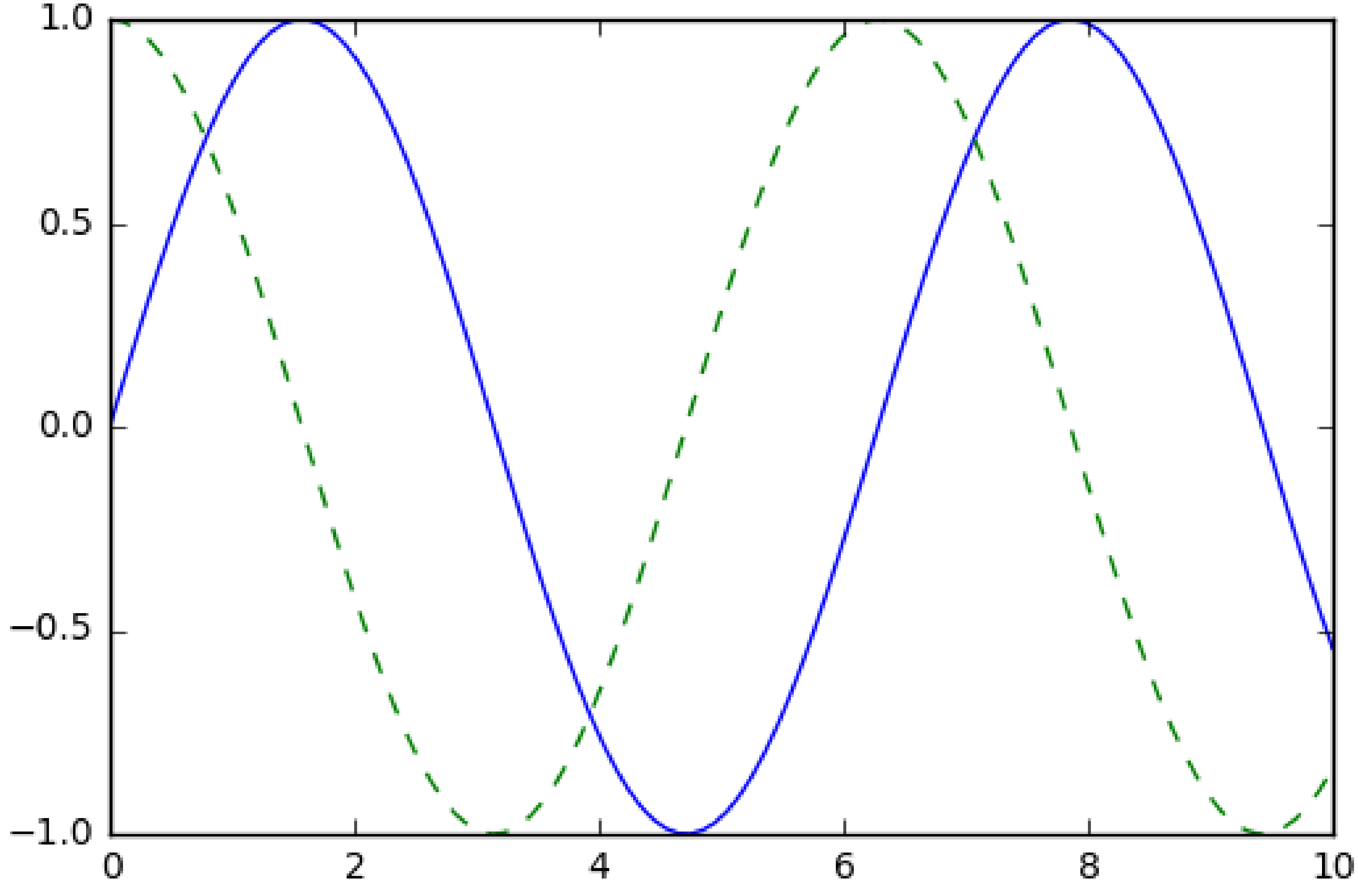
```
In [ ]: !ls -lh my_figure.png
```

```
-rw-r--r--  1 jakevdp  staff    26K Feb  1 06:15 my_figure.pn
g
```

To confirm that it contains what we think it contains, let's use the IPython `Image` object to **display the contents** of this **file** (see the following figure):

```
In [ ]: from IPython.display import Image
        Image('my_figure.png')
```

Out[ ]:



In `savefig`, the **file format is inferred** from the extension of the given filename.

Depending on what **backends** you have installed, many **different file formats** are available.

The **list of supported file types** can be found for your system by using the following method of the figure **canvas object:**

In [ ]:
```
fig.canvas.get_supported_filetypes()
```

Out[ ]:
```
{'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'}
```

Note that when saving your figure, it is not necessary to use `plt.show` or related commands discussed earlier.

Two Interfaces for the Price of One

A potentially **confusing feature** of Matplotlib is its **dual interfaces:**

a convenient MATLAB-style **state-based** interface,

and a more powerful **object-oriented** interface.

MATLAB-style Interface

Matplotlib was **originally** conceived as a Python alternative for **MATLAB users,** and much of its syntax reflects that fact.

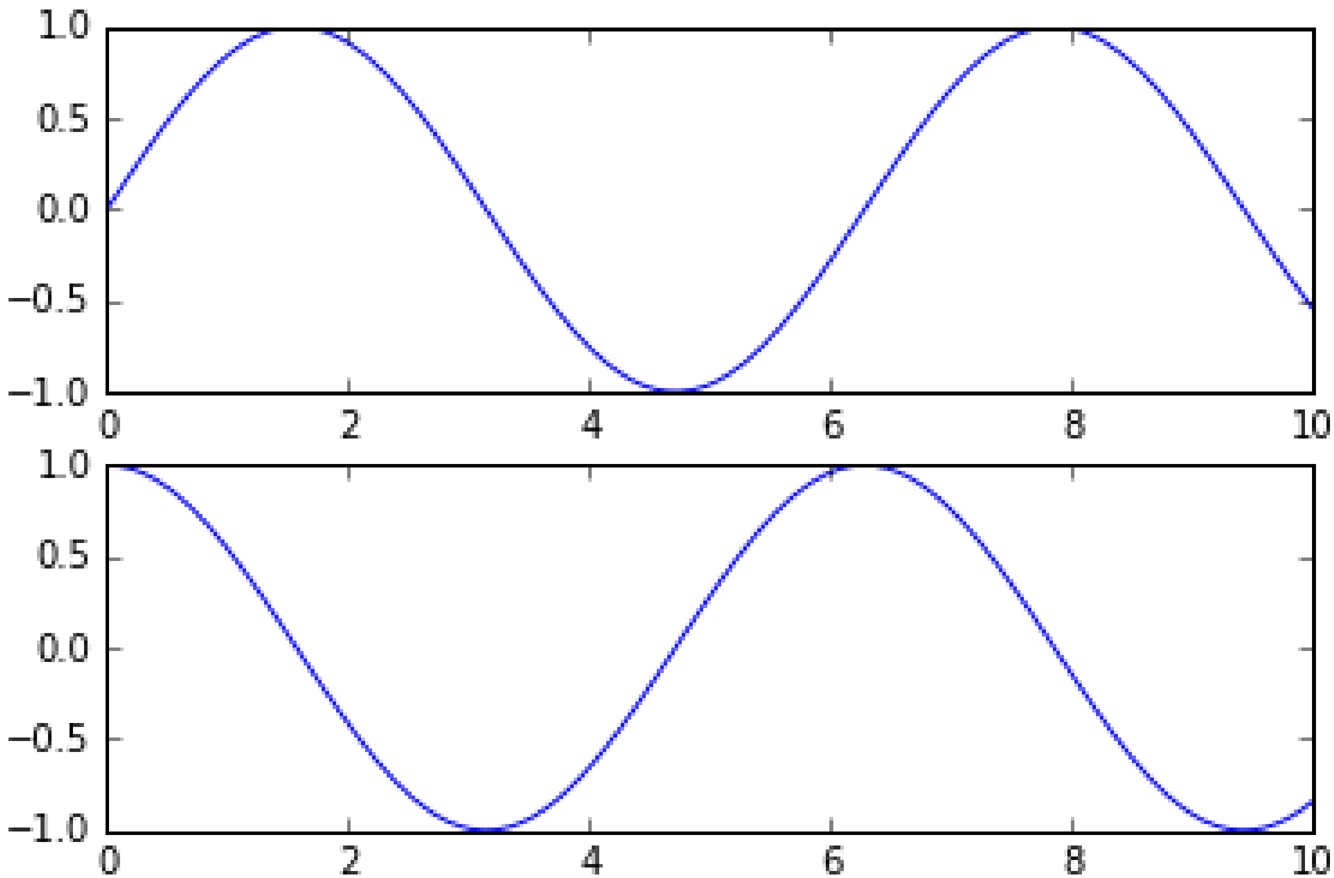
The **MATLAB-style tools** are contained in the `pyplot` (`plt`) interface.

For example, the following code will probably look quite familiar to MATLAB users (the following figure shows the result):

In [ ]:
```
plt.figure()  # create a plot figure

# create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```



It is important to recognize that this interface is **stateful**:

it **keeps track** of the "current" figure and axes, which are where all `plt` commands are applied.

You can **get a reference** to these using the `plt.gcf` (get current figure) and `plt.gca` (get current axes) routines.

While this stateful interface is fast and convenient for simple plots, it is easy to **run into problems.**

For example, once the second panel is created, how can we go back and add something to the first?

This is **possible** within the MATLAB-style interface, but a bit **clunky.**

Fortunately, there is **a better way.**

Object-oriented interface

The **object-oriented interface** is available for these more **complicated situations,**

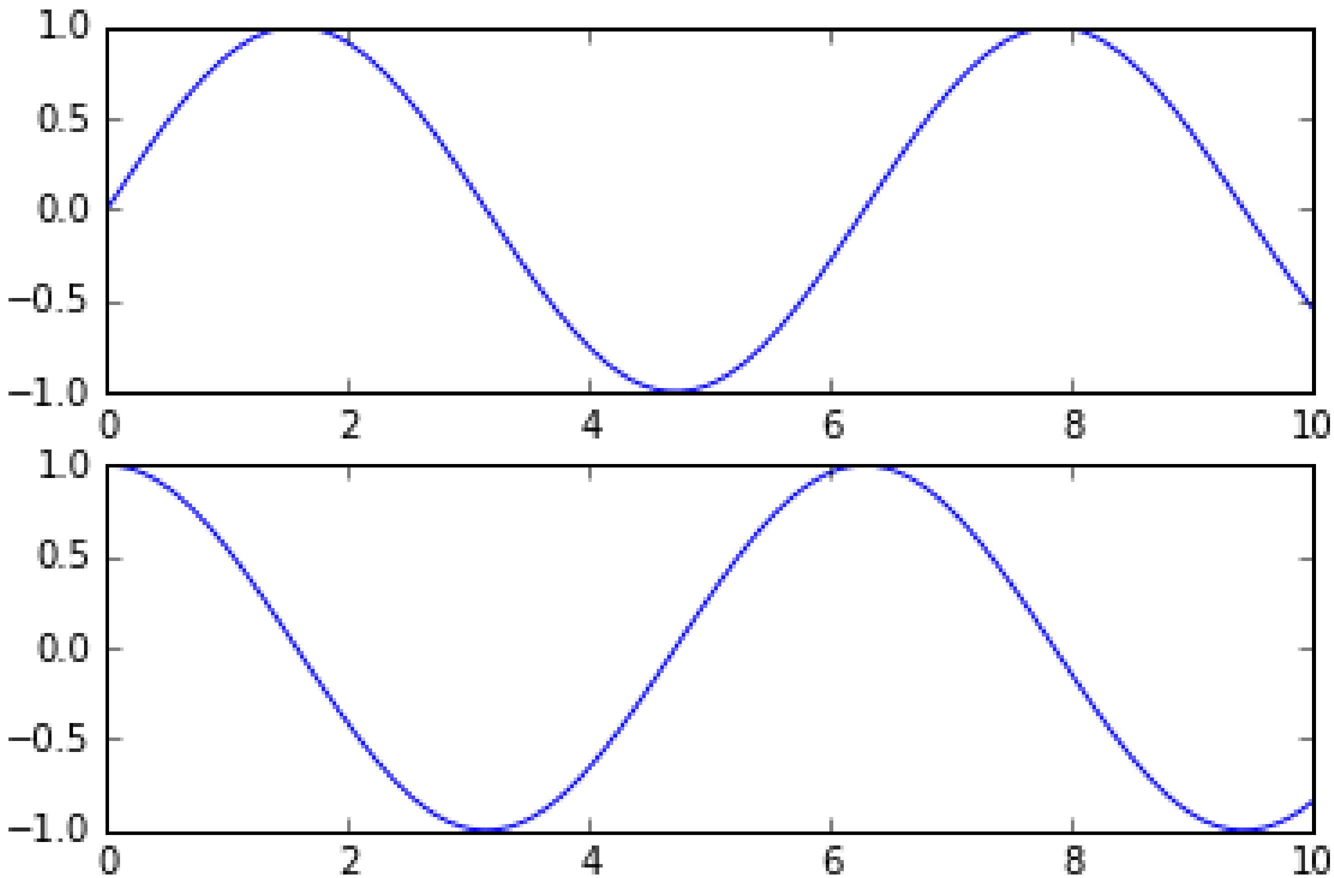
and for when you want **more control** over your figure.

Rather than depending on some notion of an "active" figure or axes, in the object-oriented interface the **plotting functions** are **methods** of explicit `Figure` and `Axes` **objects.**

To re-create the previous plot using this style of plotting, as shown in the following figure, you might do the following:

In [ ]:
```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```



For **simpler plots,** the choice of which style to use is largely **a matter of preference,**

but the **object-oriented approach** can become a **necessity** as plots become more **complicated.**

Throughout the **following chapters,** we will **switch** between the MATLAB-style and object-oriented interfaces, depending on what is most convenient . **In most cases,** the difference is as small as switching `plt.plot` to `ax.plot`.