Interactive Applications Using Matplotlib

Matplotlib makes it easy to generate plots, histograms, power spectra, bar charts, error charts, and other kinds of plots, with just a few lines of code. Interactive Applications Using Matplotlib will teach you how to turn your plots into fully interactive applications for data exploration and information synthesis. After being introduced to the plotting library, you'll learn how to create simple figures and come to grips with how they work. After these first steps, we will start work on a weather radar application.

Next, you will learn about Matplotlib's event handler to add not only keymaps and mouse actions but also custom events, enabling our radar application to transition from a simple visualization tool into a useful severe storm tracking application, complete with animations and widgets. The book will conclude with enhancements from the GUI toolkit of your choice.

Who this book is written for

This book is intended for Python programmers who want to do more than just see their data. Experience with GUI toolkits is not required, so this book can be an excellent complement to other GUI programming resources.

What you will learn from this book

- Add keymaps, mouse button actions, and custom events to your application
- Build and record animations of your plots
- Enhance your data display with buttons, sliders, and other widgets
- Insert Matplotlib figures into any GUI application
- Create a session recorder for your application
- Learn about Matplotlib's event handler to add custom events
- See Matplotlib as more than just a plotting library

Don't just see your data, experience it!
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introducing Interactive Plotting'
- A synopsis of the book’s content
- More information on Interactive Applications Using Matplotlib

About the Author

**Benjamin V. Root** has been a member of the Matplotlib development team since 2010. His main areas of development have been the documentation and the mplot3d toolkit, but now he focuses on code reviews and debugging. Ben is also an active member of mailing lists, using his expertise to help newcomers understand Matplotlib. He is a meteorology graduate student, working part-time on his PhD dissertation. He works full-time for Atmospheric and Environmental Research, Inc. as a scientific programmer.
Interactive Applications Using Matplotlib

Why Matplotlib? Why Python, for that matter? I picked up Python for scientific development because I needed a full-fledged programming language that made sense. Too often, I felt hemmed in by the traditional tools in the meteorology field. I needed a language that respected my time as a developer and didn't fight me every step of the way. "Don't you find Python constricting?" asked a colleague who was fond of bad puns. "No, quite the opposite," I replied, the joke going right over my head.

Matplotlib is the same in this respect. Switching from traditional graphing tools of the meteorology field to Matplotlib was a breath of fresh air. Not only were useful programs being written using the Matplotlib library, but it was also easy to write my own. Furthermore, I could write out modules and easily use them in both the hardcopy generating scripts for my publications and for my data exploration interactive applications. Most importantly, the Matplotlib library let me do what I needed it to do.

I have been an active developer for Matplotlib since 2010 and I am still discovering Matplotlib. It isn't that the library is insanely huge and unwieldy—it isn't. Instead, Matplotlib appeals to all levels of expertise and interests. One can simply care enough only to get a single plot displayed in three line of code and never think of the library again. Or, one could assume control over every single minute plotting detail, ensuring that everything is displayed "just right." And even when one does this and thinks they have seen every single nook and cranny of the library, they will discover some other feature that they have never seen before.

Matplotlib is 12 years old now. New plotting projects have cropped up—some supplementing Matplotlib's design, while others trying to replace Matplotlib entirely. However, there has been no slacking of interest in Matplotlib, not from the users and definitely not from the developers. The new projects are interesting, and as with all things open source, we try to learn from these projects. But I keep coming back to this project. Its design, developers, and community of users are some of the best and most devoted in the open source world.

The book you are reading right now is actually not the book I originally wanted to write. The interactive aspect of Matplotlib is not my area of expertise. After some nudging from fellow developers and users, I relented. I proceeded to rewrite the only interactive application I had ever finished and published. Working through the chapters, I tried to find better ways of doing the things I did originally, pointing out major pitfalls and easy mistakes as I encountered them. It was a significant learning experience for me, which was wholly unexpected.
I now invite you to discover Matplotlib for yourself. Whether it is the first time or not, it certainly won't be the last.

**What This Book Covers**

*Chapter 1, Introducing Interactive Plotting*, covers basic figure-axes-artist hierarchy and other Matplotlib essentials such as displaying the plot. It also introduces you to the interactive Matplotlib figure.

*Chapter 2, Using Events and Callbacks*, provides Matplotlib's events and a callback system to bring your figures to life. It also explains how you can extend it with custom events, making the application truly interactive.

*Chapter 3, Animations*, deals with ArtistAnimation, FuncAnimation, and timers to make animations of all types. It also deals with animations that can be saved as movies.

*Chapter 4, Widgets*, covers built-in widgets such as buttons, checkboxes, selectors, lassos, and sliders, which are all explained and demonstrated. Here, you'll also learn about other useful third-party widgets and tools.

*Chapter 5, Embedding Matplotlib*, teaches you how to add GUI elements to an existing Matplotlib application. Here you'll also see how to add your interactive Matplotlib figure to an existing GUI application. Identical examples are presented using GTK, Tkinter, wxWidgets, and Qt.
Introducing Interactive Plotting

A picture is worth a thousand words

The goal of any interactive application is to provide as much information as possible while minimizing complexity. If it can’t provide the information the users need, then it is useless to them. However, if the application is too complex, then the information’s signal gets lost in the noise of the complexity. A graphical presentation often strikes the right balance.

The Matplotlib library can help you present your data as graphs in your application. Anybody can make a simple interactive application without knowing anything about draw buffers, event loops, or even what a GUI toolkit is. And yet, the Matplotlib library will cede as much control as desired to allow even the most savvy GUI developer to create a masterful application from scratch. Like much of the Python language, Matplotlib's philosophy is to give the developer full control, but without being stupidly unhelpful and tedious.

Installing Matplotlib

There are many ways to install Matplotlib on your system. While the library used to have a reputation for being difficult to install on non-Linux systems, it has come a long way since then, along with the rest of the Python ecosystem. Refer to the following command:

$ pip install matplotlib
Most likely, the preceding command would work just fine from the command line. Python Wheels (the next-generation Python package format that has replaced "eggs") for Matplotlib are now available from PyPi for Windows and Mac OS X systems. This method would also work for Linux users; however, it might be more favorable to install it via the system's built-in package manager.

While the core Matplotlib library can be installed with few dependencies, it is a part of a much larger scientific computing ecosystem known as SciPy. Displaying your data is often the easiest part of your application. Processing it is much more difficult, and the SciPy ecosystem most likely has the packages you need to do that. For basic numerical processing and N-dimensional data arrays, there is NumPy. For more advanced but general data processing tools, there is the SciPy package (the name was so catchy, it ended up being used to refer to many different things in the community). For more domain-specific needs, there are "Sci-Kits" such as scikit-learn for artificial intelligence, scikit-image for image processing, and statsmodels for statistical modeling. Another very useful library for data processing is pandas.

This was just a short summary of the packages available in the SciPy ecosystem. Manually managing all of their installations, updates, and dependencies would be difficult for many who just simply want to use the tools. Luckily, there are several distributions of the SciPy Stack available that can keep the menagerie under control. The following are Python distributions that include the SciPy Stack along with many other popular Python packages or make the packages easily available through package management software:

- Anaconda from Continuum Analytics
- Canopy from Enthought
- SciPy Superpack
- Python(x, y) (Windows only)
- WinPython (Windows only)
- Pyzo (Python 3 only)
- Algorete Loopy from Dartmouth College

For this book, we will assume at least Python 2.7 or 3.2. The requisite packages are numpy, matplotlib, basemap, and scipy. Just about any version of these packages released in the past 3 years should work for most examples in this book (exceptions are noted in this book). The version 0.14.0 of SciPy (released in May 2014) cannot be used in this book due to a (now fixed) regression in its NetCDF reader. Chapter 5, Embedding Matplotlib will have special notes with regards to GUI toolkit packages.
Show() your work
With Matplotlib installed, you are now ready to make your first simple plot. Matplotlib has multiple layers. `Pylab` is the topmost layer, often used for quick one-off plotting from within a live Python session. Start up your favorite Python interpreter and type the following:

```python
>>> from pylab import *
>>> plot([1, 2, 3, 2, 1])
```

Nothing happened! This is because Matplotlib, by default, will not display anything until you explicitly tell it to do so. The Matplotlib library is often used for automated image generation from within Python scripts, with no need for any interactivity. Also, most users would not be done with their plotting yet and would find it distracting to have a plot come up automatically. When you are ready to see your plot, use the following command:

```python
>>> show()
```

Interactive navigation
A figure window should now appear, and the Python interpreter is not available for any additional commands. By default, showing a figure will block the execution of your scripts and interpreter. However, this does not mean that the figure is not interactive. As you mouse over the plot, you will see the plot coordinates in the lower right-hand corner. The figure window will also have a toolbar:

From left to right, the following are the tools:

- **Home, Back, and Forward**: These are similar to that of a web browser. These buttons help you navigate through the previous views of your plot. The "Home" button will take you back to the first view when the figure was opened. "Back" will take you to the previous view, while "Forward" will return you to the previous views.
Introducing Interactive Plotting

- **Pan (and zoom):** This button has two modes: pan and zoom. Press the left mouse button and hold it to pan the figure. If you press \( x \) or \( y \) while panning, the motion will be constrained to just the \( x \) or \( y \) axis, respectively. Press the right mouse button to zoom. The plot will be zoomed in or out proportionate to the right/left and up/down movements. Use the \( X \), \( Y \), or \( Ctrl \) key to constrain the zoom to the \( x \) axis or the \( y \) axis or preserve the aspect ratio, respectively.

- **Zoom-to-rectangle:** Press the left mouse button and drag the cursor to a new location and release. The axes view limits will be zoomed to the rectangle you just drew. Zoom out using your right mouse button, placing the current view into the region defined by the rectangle you just drew.

- **Subplot configuration:** This button brings up a tool to modify plot spacing.

- **Save:** This button brings up a dialog that allows you to save the current figure.

The figure window would also be responsive to the keyboard. The default keymap is fairly extensive (and will be covered fully later), but some of the basic hot keys are the **Home** key for resetting the plot view, the left and right keys for back and forward actions, **p** for pan/zoom mode, **o** for zoom-to-rectangle mode, and **Ctrl + s** to trigger a file save. When you are done viewing your figure, close the window as you would close any other application window, or use **Ctrl + w**.

**Interactive plotting**

When we did the previous example, no plots appeared until `show()` was called. Furthermore, no new commands could be entered into the Python interpreter until all the figures were closed. As you will soon learn, once a figure is closed, the plot it contains is lost, which means that you would have to repeat all the commands again in order to `show()` it again, perhaps with some modification or additional plot. Matplotlib ships with its **interactive plotting mode** off by default.

There are a couple of ways to turn the interactive plotting mode on. The main way is by calling the `ion()` function (for Interactive ON). Interactive plotting mode can be turned on at any time and turned off with `ioff()`. Once this mode is turned on, the next plotting command will automatically trigger an implicit `show()` command. Furthermore, you can continue typing commands into the Python interpreter. You can modify the current figure, create new figures, and close existing ones at any time, all from the current Python session.
Scripted plotting

Python is known for more than just its interactive interpreters; it is also a fully fledged programming language that allows its users to easily create programs. Having a script to display plots from daily reports can greatly improve your productivity. Alternatively, you perhaps need a tool that can produce some simple plots of the data from whatever mystery data file you have come across on the network share. Here is a simple example of how to use Matplotlib's `pyplot` API and the `argparse` Python standard library tool to create a simple CSV plotting script called `plotfile.py`.

**Code:** chp1/plotfile.py

```python
#!/usr/bin/env python

from argparse import ArgumentParser
import matplotlib.pyplot as plt

if __name__ == '__main__':
    parser = ArgumentParser(description="Plot a CSV file")
    parser.add_argument("datafile", help="The CSV File")
    # Require at least one column name
    parser.add_argument("columns", nargs='+',
                       help="Names of columns to plot")
    parser.add_argument("--save", help="Save the plot as...")
    parser.add_argument("--no-show", action="store_true",
                       help="Don't show the plot")
    args = parser.parse_args()

    plt.plotfile(args.datafile, args.columns)
    if args.save:
        plt.savefig(args.save)
    if not args.no_show:
        plt.show()
```

Note the two optional command-line arguments: `--save` and `--no-show`. With the `--save` option, the user can have the plot automatically saved (the graphics format is determined automatically from the filename extension). Also, the user can choose not to display the plot, which when coupled with the `--save` option might be desirable if the user is trying to plot several CSV files.

When calling this script to show a plot, the execution of the script will stop at the call to `plt.show()`. If the interactive plotting mode was on, then the execution of the script would continue past `show()`, terminating the script, thus automatically closing out any figures before the user has had a chance to view them. This is why the interactive plotting mode is turned off by default in Matplotlib.
Also note that the call to `plt.savefig()` is before the call to `plt.show()`. As mentioned before, when the figure window is closed, the plot is lost. You cannot save a plot after it has been closed.

**Getting help**

We have covered how to install Matplotlib and went over how to make very simple plots from a Python session or a Python script. Most likely, this went very smoothly for you. The rest of this book will focus on how to use Matplotlib to make an interactive application, rather than the many ways to display data. You may be very curious and want to learn more about the many kinds of plots this library has to offer, or maybe you want to learn how to make new kinds of plots.

Help comes in many forms. The Matplotlib website (http://matplotlib.org) is the primary online resource for Matplotlib. It contains examples, FAQs, API documentation, and, most importantly, the gallery.

**Gallery**

Many users of Matplotlib are often faced with the question, "I want to make a plot that has this data along with that data in the same figure, but it needs to look like this other plot I have seen." Text-based searches on graphing concepts are difficult, especially if you are unfamiliar with the terminology. The gallery showcases the variety of ways in which one can make plots, all using the Matplotlib library. Browse through the gallery, click on any figure that has pieces of what you want in your plot, and see the code that generated it. Soon enough, you will be like a chef, mixing and matching components to produce that perfect graph.

**Mailing lists and forums**

When you are just simply stuck and cannot figure out how to get something to work or just need some hints on how to get started, you will find much of the community at the Matplotlib-users mailing list. This mailing list is an excellent resource of information with many friendly members who just love to help out newcomers. Be persistent! While many questions do get answered fairly quickly, some will fall through the cracks. Try rephrasing your question or with a plot showing your attempts so far. The people at Matplotlib-users love plots, so an image that shows what is wrong often gets the quickest response. A newer community resource is StackOverflow, which has many very knowledgeable users who are able to answer difficult questions.
From front to backend

So far, we have shown you bits and pieces of two of Matplotlib's topmost abstraction layers: `pylab` and `pyplot`. The layer below them is the object-oriented layer (the OO layer). To develop any type of application, you will want to use this layer. Mixing the `pylab/pyplot` layers with the OO layer will lead to very confusing behaviors when dealing with multiple plots and figures.

Below the OO layer is the backend interface. Everything above this interface level in Matplotlib is completely platform-agnostic. It will work the same regardless of whether it is in an interactive GUI or comes from a driver script running on a headless server. The backend interface abstracts away all those considerations so that you can focus on what is most important: writing code to visualize your data.

There are several backend implementations that are shipped with Matplotlib. These backends are responsible for taking the figures represented by the OO layer and interpreting it for whichever "display device" they implement. The backends are chosen automatically but can be explicitly set, if needed (see Chapter 5, Embedding Matplotlib).

Interactive versus non-interactive

There are two main classes of backends: ones that provide interactive figures and ones that don't. Interactive backends are ones that support a particular GUI, such as Tcl/Tkinter, GTK, Qt, Cocoa/Mac OS X, wxWidgets, and Cairo. With the exception of the Cocoa/Mac OS X backend, all interactive backends can be used on Windows, Linux, and Mac OS X. Therefore, when you make an interactive Matplotlib application that you wish to distribute to users of any of those platforms, unless you are embedding Matplotlib (again, see Chapter 5, Embedding Matplotlib), you will not have to concern yourself with writing a single line of code for any of these toolkits—it has already been done for you!

Non-interactive backends are used to produce image files. There are backends to produce Postscript/EPS, Adobe PDF, and Scalable Vector Graphics (SVG) as well as rasterized image files such as PNG, BMP, and JPEGs.
Introducing Interactive Plotting

Anti-grain geometry
The open secret behind the high quality of Matplotlib's rasterized images is its use of the Anti-Grain Geometry (AGG) library (http://agg.sourceforge.net/antigrain.com/index.html). The quality of the graphics generated from AGG is far superior than most other toolkits available. Therefore, not only is AGG used to produce rasterized image files, but it is also utilized in most of the interactive backends as well. Matplotlib maintains and ships with its own fork of the library in order to ensure you have consistent, high quality image products across all platforms and toolkits. What you see on your screen in your interactive figure window will be the same as the PNG file that is produced when you call `savefig()`.

Selecting your backend
When you install Matplotlib, a default backend is chosen for you based upon your OS and the available GUI toolkits. For example, on Mac OS X systems, your installation of the library will most likely set the default interactive backend to MacOSX or CocoaAgg for older Macs. Meanwhile, Windows users will most likely have a default of TkAgg or Qt5Agg. In most situations, the choice of interactive backends will not matter. However, in certain situations, it may be necessary to force a particular backend to be used. For example, on a headless server without an active graphics session, you would most likely need to force the use of the non-interactive Agg backend:

```python
import matplotlib
matplotlib.use("Agg")
```

When done prior to any plotting commands, this will avoid loading any GUI toolkits, thereby bypassing problems that occur when a GUI fails on a headless server. Any call to `show()` effectively becomes a no-op (and the execution of the script is not blocked). Another purpose of setting your backend is for scenarios when you want to embed your plot in a native GUI application. Therefore, you will need to explicitly state which GUI toolkit you are using (see Chapter 5, Embedding Matplotlib). Finally, some users simply like the look and feel of some GUI toolkits better than others. They may wish to change the default backend via the `backend` parameter in the `matplotlibrc` configuration file. Most likely, your `rc` file can be found in the `.matplotlib` directory or the `.config/matplotlib` directory under your home folder. If you can't find it, then use the following set of commands:

```python
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
u'/home/ben/.config/matplotlib/matplotlibrc'
```
Here is an example of the relevant section in my matplotlibrc file:

```plaintext
#### CONFIGURATION BEGINS HERE

# the default backend; one of GTK GTKAgg GTKCairo GTK3Agg
# GTK3Cairo CocoaAgg MacOSX QtAgg Qt4Agg TkAgg WX WXAgg Agg Cairo
# PS PDF SVG
# You can also deploy your own backend outside of matplotlib by
# referring to the module name (which must be in the PYTHONPATH)
# as 'module://my_backend'
backend : GTKAgg
backend : QT4Agg
backend : TkAgg
# If you are using the Qt4Agg backend, you can choose here
# to use the PyQt4 bindings or the newer PySide bindings to
# the underlying Qt4 toolkit.
backend.qt4 : PyQt4         # PyQt4 | PySide
```

This is the global configuration file that is used if one isn’t found in the current working directory when Matplotlib is imported. The settings contained in this configuration serves as default values for many parts of Matplotlib. In particular, we see that the choice of backends can be easily set without having to use a single line of code.

### The Matplotlib figure-artist hierarchy

Everything that can be drawn in Matplotlib is called an artist. Any artist can have child artists that are also drawable. This forms the basis of a hierarchy of artist objects that Matplotlib sends to a backend for rendering. At the root of this artist tree is the figure.

In the examples so far, we have not explicitly created any figures. The `pylab` and `pyplot` interfaces will create the figures for us. However, when creating advanced interactive applications, it is highly recommended that you explicitly create your figures. You will especially want to do this if you have multiple figures being displayed at the same time. This is the entry into the OO layer of Matplotlib:

```python
fig = plt.figure()
```
Introducing Interactive Plotting

Canvassing the figure

The figure is, quite literally, your canvas. Its primary component is the FigureCanvas instance upon which all drawing occurs. Unless you are embedding your Matplotlib figures into a GUI application, it is very unlikely that you will need to interact with this object directly. Instead, as plotting commands are issued, artist objects are added to the canvas automatically.

While any artist can be added directly to the figure, usually only Axes objects are added. A figure can have many axes objects, typically called subplots. Much like the figure object, our examples so far have not explicitly created any axes objects to use. This is because the pylab and pyplot interfaces will also automatically create and manage axes objects for a figure if needed. For the same reason as for figures, you will want to explicitly create these objects when building your interactive applications. If an axes or a figure is not provided, then the pyplot layer will have to make assumptions about which axes or figure you mean to apply a plotting command to. While this might be fine for simple situations, these assumptions get hairy very quickly in non-trivial applications. Luckily, it is easy to create both your figure and its axes using a single command:

```python
fig, axes = plt.subplots(2, 1)  # 2x1 grid of subplots
```

These objects are highly advanced complex units that most developers will utilize for their plotting needs. Once placed on the figure canvas, the axes object will provide the ticks, axis labels, axes title(s), and the plotting area. An axes is an artist that manages all of its scale and coordinate transformations (for example, log scaling and polar coordinates), automated tick labeling, and automated axis limits. In addition to these responsibilities, an axes object provides a wide assortment of plotting functions. A sampling of plotting functions is as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>Make a bar plot</td>
</tr>
<tr>
<td>barbs</td>
<td>Plot a two-dimensional field of barbs</td>
</tr>
<tr>
<td>boxplot</td>
<td>Make a box and whisker plot</td>
</tr>
<tr>
<td>cohere</td>
<td>Plot the coherence between x and y</td>
</tr>
<tr>
<td>contour</td>
<td>Plot contours</td>
</tr>
<tr>
<td>errorbar</td>
<td>Plot an errorbar graph</td>
</tr>
<tr>
<td>hexbin</td>
<td>Make a hexagonal binning plot</td>
</tr>
<tr>
<td>hist</td>
<td>Plot a histogram</td>
</tr>
<tr>
<td>imshow</td>
<td>Display an image on the axes</td>
</tr>
<tr>
<td>pcolor</td>
<td>Create a pseudocolor plot of a two-dimensional array</td>
</tr>
<tr>
<td>pcolormesh</td>
<td>Plot a quadrilateral mesh</td>
</tr>
</tbody>
</table>
Throughout the rest of this book, we will build a single interactive application piece by piece, demonstrating concepts and features that are available through Matplotlib. This application will be a storm track editing application. Given a series of radar images, the user can circle each storm cell they see in the radar image and link those storm cells across time. The application will need the ability to save and load track data and provide the user with mechanisms to edit the data. Along the way, we will learn about Matplotlib's structure, its artists, the callback system, doing animations, and finally, embedding this application within a larger GUI application.

So, to begin, we first need to be able to view a radar image. There are many ways to load data into a Python program but one particular favorite among meteorologists is the Network Common Data Form (NetCDF) file. The SciPy package has built-in support for NetCDF version 3, so we will be using an hour's worth of radar reflectivity data prepared using this format from a NEXRAD site near Oklahoma City, OK on the evening of May 10, 2010, which produced numerous tornadoes and severe storms.

The NetCDF binary file is particularly nice to work with because it can hold multiple data variables in a single file, with each variable having an arbitrary number of dimensions. Furthermore, metadata can be attached to each variable and to the dataset itself, allowing you to self-document data files. This particular data file has three variables, namely Reflectivity, lat, and lon to record the radar reflectivity values and the latitude and longitude coordinates of each pixel in the reflectivity data. The reflectivity data is three-dimensional, with the first dimension as time and the other two dimensions as latitude and longitude. The following code example shows how easy it is to load this data and display the first image frame using SciPy and Matplotlib.
Introducing Interactive Plotting

Code: chp1/simple_radar_viewer.py

```python
import matplotlib.pyplot as plt
from scipy.io import netcdf_file

cf = netcdf_file('KTLX_20100510_22Z.nc')
data = cf.variables['Reflectivity']
lats = ccf.variables['lat']
lons = ccf.variables['lon']
i = 0

cmap = plt.get_cmap('gist_ncar')
cmap.set_under('lightgrey')

fig, ax = plt.subplots(1, 1)
im = ax.imshow(data[i], origin='lower',
               extent=(lons[0], lons[-1], lats[0], lats[-1]),
               vmin=0.1, vmax=80, cmap='gist_ncar')

fig.colorbar(im)

ax.set_ylabel('Reflectivity (dBZ)')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
plt.show()
```

Running this script should result in a figure window that will display the first frame of our storms that we will become very familiar with over the next few chapters. The plot has a colorbar and the axes ticks label the latitudes and longitudes of our data. What is probably most important in this example is the `imshow()` call. Being an image, traditionally, the origin of the image data is shown in the upper-left corner and Matplotlib follows this tradition by default. However, this particular dataset was saved with its origin in the lower-left corner, so we need to state this with the `origin` parameter. The `extent` parameter is a tuple describing the data extent of the image. By default, it is assumed to be at $(0, 0)$ and $(N - 1, M - 1)$ for an $MxN$ shaped image. The `vmin` and `vmax` parameters are a good way to ensure consistency of your colormap regardless of your input data. If these two parameters are not supplied, then `imshow()` will use the minimum and maximum of the input data to determine the colormap. This would be undesirable as we move towards displaying arbitrary frames of radar data. Finally, one can explicitly specify the colormap to use for the image. The `gist_ncar` colormap is very similar to the official NEXRAD colormap for radar data, so we will use it here:
The `gist_ncar` colormap, along with some other colormaps packaged with Matplotlib such as the default `jet` colormap, are actually terrible for visualization. See the Choosing Colormaps page of the Matplotlib website for an explanation of why, and guidance on how to choose a better colormap.

**The menagerie of artists**

Whenever a plotting function is called, the input data and parameters are processed to produce new artists to represent the data. These artists are either **primitives** or collections thereof. They are called primitives because they represent basic drawing components such as lines, images, polygons, and text. It is with these primitives that your data can be represented as bar charts, line plots, errorbars, or any other kinds of plots.
Primitives

There are four drawing primitives in Matplotlib: Line2D, AxesImage, Patch, and Text. It is through these primitive artists that all other artist objects are derived from, and they comprise everything that can be drawn in a figure.

A Line2D object uses a list of coordinates to draw line segments in between. Typically, the individual line segments are straight, and curves can be approximated with many vertices; however, curves can be specified to draw arcs, circles, or any other Bezier-approximated curves.

An AxesImage class will take two-dimensional data and coordinates and display an image of that data with a colormap applied to it. There are actually other kinds of basic image artists available besides AxesImage, but they are typically for very special uses. AxesImage objects can be very tricky to deal with, so it is often best to use the imshow() plotting method to create and return these objects.

A Patch object is an arbitrary two-dimensional object that has a single color for its "face." A polygon object is a specific instance of the slightly more general patch. These objects have a "path" (much like a Line2D object) that specifies segments that would enclose a face with a single color. The path is known as an "edge," and can have its own color as well. Besides the Polygons that one sees for bar plots and pie charts, Patch objects are also used to create arrows, legend boxes, and the markers used in scatter plots and elsewhere.

Finally, the Text object takes a Python string, a point coordinate, and various font parameters to form the text that annotates plots. Matplotlib primarily uses TrueType fonts. It will search for fonts available on your system as well as ship with a few FreeType2 fonts, and it uses Bitstream Vera by default. Additionally, a Text object can defer to LaTeX to render its text, if desired.

While specific artist classes will have their own set of properties that make sense for the particular art object they represent, there are several common properties that can be set. The following table is a listing of some of these properties.
<table>
<thead>
<tr>
<th>Property</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>0 represents transparent and 1 represents opaque</td>
</tr>
<tr>
<td>color</td>
<td>Color name or other color specification</td>
</tr>
<tr>
<td>visible</td>
<td>boolean to flag whether to draw the artist or not</td>
</tr>
<tr>
<td>zorder</td>
<td>value of the draw order in the layering engine</td>
</tr>
</tbody>
</table>

Let’s extend the radar image example by loading up already saved polygons of storm cells in the tutorial.py file.

**Code:** chp1/simple_storm_cell_viewer.py

```python
import matplotlib.pyplot as plt
from scipy.io import netcdf_file
from matplotlib.patches import Polygon
from tutorial import polygon_loader

ncf = netcdf_file('KTLX_20100510_22Z.nc')
data = ncf.variables['Reflectivity']
lats = ncf.variables['lat']
lons = ncf.variables['lon']
i = 0

cmap = plt.get_cmap('gist_ncar')
cmap.set_under('lightgrey')

fig, ax = plt.subplots(1, 1)
im = ax.imshow(data[i], origin='lower',
              extent=(lons[0], lons[-1], lats[0], lats[-1]),
              vmin=0.1, vmax=80, cmap='gist_ncar')
cb = fig.colorbar(im)
polygons = polygon_loader('polygons.shp')
for poly in polygons[i]:
    p = Polygon(poly, lw=3, fc='k', ec='w', alpha=0.45)
    ax.add_artist(p)

ax.set_xlabel("Longitude")
ax.set_ylabel("Latitude")
plt.show()
```
The polygon data returned from `polygon_loader()` is a dictionary of lists keyed by a frame index. The list contains Nx2 numpy arrays of vertex coordinates in longitude and latitude. The vertices form the outline of a storm cell. The Polygon constructor, like all other artist objects, takes many optional keyword arguments. First, `lw` is short for `linewidth`, (referring to the outline of the polygon), which we specify to be three points wide. Next is `fc`, which is short for `facecolor`, and is set to black (`'k'`). This is the color of the filled-in region of the polygon. Then `edgecolor` (`ec`) is set to white (`'w'`) to help the polygons stand out against a dark background. Finally, we set the `alpha` argument to be slightly less than half to make the polygon fairly transparent so that one can still see the reflectivity data beneath the polygons.
Note a particular difference between how we plotted the image using `imshow()` and how we plotted the polygons using polygon artists. For polygons, we called a constructor and then explicitly called `ax.add_artist()` to add each polygon instance as a child of the axes. Meanwhile, `imshow()` is a plotting function that will do all of the hard work in validating the inputs, building the AxesImage instance, making all necessary modifications to the axes instance (such as setting the limits and aspect ratio), and most importantly, adding the artist object to the axes. Finally, all plotting functions in Matplotlib return artists or a list of artist objects that it creates. In most cases, you will not need to save this return value in a variable because there is nothing else to do with them. In this case, we only needed the returned AxesImage so that we could pass it to the `fig.colorbar()` method. This is so that it would know what to base the colorbar upon.

The plotting functions in Matplotlib exist to provide convenience and simplicity to what can often be very tricky to get right by yourself. They are not magic! They use the same OO interface that is accessible to application developers. Therefore, anyone can write their own plotting functions to make complicated plots easier to perform.

**Collections**

Any artist that has child artists (such as a figure or an axes) is called a **container**. A special kind of container in Matplotlib is called a **Collection**. A collection usually contains a list of primitives of the same kind that should all be treated similarly. For example, a CircleCollection would have a list of Circle objects, all with the same color, size, and edge width. Individual values for artists in the collection can also be set. A collection makes management of many artists easier. This becomes especially important when considering the number of artist objects that may be needed for scatter plots, bar charts, or any other kind of plot or diagram.

Some collections are not just simply a list of primitives, but are artists in their own right. These special kinds of collections take advantage of various optimizations that can be assumed when rendering similar or identical things. `RegularPolyCollection`, for example, just needs to know the points of a single polygon relative to its center (such as a star or box) and then just needs a list of all the center coordinates, avoiding the need to store all the vertices of every polygon in its collection in memory.
Introducing Interactive Plotting

In the following example, we will display storm tracks as LineCollection. Note that instead of using `ax.add_artist()` (which would work), we will use `ax.add_collection()` instead. This has the added benefit of performing special handling on the object to determine its bounding box so that the axes object can incorporate the limits of this collection with any other plotted objects to automatically set its own limits which we trigger with the `ax.autoscale(True)` call.

**Code:** chp1/linecoll_track_viewer.py

```python
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from tutorial import track_loader

tracks = track_loader('polygons.shp')
# Filter out non-tracks (unassociated polygons given trackID of -9)
tracks = {tid: t for tid, t in tracks.items() if tid != -9}

fig, ax = plt.subplots(1, 1)
lc = LineCollection(tracks.values(), color='b')
ax.add_collection(lc)
ax.autoscale(True)
ax.set_xlabel("Longitude")
ax.set_ylabel("Latitude")
plt.show()
```

![Plot of storm tracks]
Much easier than the radar images, Matplotlib took care of all the limit setting automatically. Such features are extremely useful for writing generic applications that do not wish to concern themselves with such details. We will come back to the handling of LineCollections later in the book as we develop this application.

Summary

In this chapter, we introduced you to the foundational concepts of Matplotlib. Using `show()`, you showed your first plot with only three lines of Python. With this plot up on your screen, you learned some of the basic interactive features built into Matplotlib, such as panning, zooming, and the myriad of key bindings that are available. Then we discussed the difference between interactive and non-interactive plotting modes and the difference between scripted and interactive plotting. You now know where to go online for more information, examples, and forum discussions of Matplotlib when it comes time for you to work on your next Matplotlib project. Next, we discussed the architectural concepts of Matplotlib: backends, figures, axes, and artists.

Then we started our construction project for this book, an interactive storm cell tracking application. We saw how to plot a radar image using a pre-existing plotting function, as well as how to display polygons and lines as artists and collections. While creating these objects, we had a glimpse of how to customize the properties of these objects for our display needs, learning some of the property and styling names. We also learned some of the steps one needs to consider when creating their own plotting functions, such as autoscaling.

In the next chapter, we will learn how to extend the basic interactivity of Matplotlib, adding our own features and controls in order to make a truly interactive application.
Where to buy this book

You can buy Interactive Applications Using Matplotlib from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.