Mastering matplotlib

A practical guide that takes you beyond the basics of matplotlib and gives solutions to plot complex data

Duncan M. McGreggor
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 'Customization and Configuration'
- A synopsis of the book’s content
- More information on Mastering matplotlib
About the Author

Duncan M. McGregor, having programmed with GOTOs in the 1980s, has made up for that through community service by making open source contributions for more than 20 years. He has spent a major part of the past 10 years dealing with distributed and scientific computing (in languages ranging from Python, Common Lisp, and Julia to Clojure and Lisp Flavored Erlang). In the 1990s, after serving as a linguist in the US Army, he spent considerable time working on projects related to MATLAB and Mathematica, which was a part of his physics and maths studies at the university. Since the mid 2000s, matplotlib and NumPy have figured prominently in many of the interesting problems that he has solved for his customers. With the most recent addition of the IPython Notebook, matplotlib and the suite of the Python scientific computing libraries remain some of his most important professional tools.
Preface

In just over a decade, matplotlib has grown to offer the Python scientific computing community a world-class plotting and visualization library. When combined with related projects, such as Jupyter, NumPy, SciPy, and SymPy, matplotlib competes head-to-head with commercial software, which is far more established in the industry. Furthermore, the growth experienced by this open source software project is reflected again and again by individuals around the world, who make their way through the thorny wilds that face the newcomer and who develop into strong intermediate users with the potential to be very productive.

In essence, Mastering matplotlib is a very practical book. Yet every chapter was written considering this learning process, as well as a larger view of the same. It is not just the raw knowledge that defines how far developers progress in their goal. It is also the ability of motivated individuals to apply meta-levels of analysis to the problem and the obstacles that must be surmounted. Implicit in the examples that are provided in each chapter are multiple levels of analysis, which are integral to the mastery of the subject matter. These levels of analysis involve the processes of defining the problem, anticipating potential solutions, evaluating approaches without losing focus, and enriching your experience with a wider range of useful projects.

Finding resources that facilitate developers in their journey towards advanced knowledge and beyond can be difficult. This is not due to the lack of materials. Rather, it is because of the complex interaction of learning styles, continually improving codebases with strong legacies, and the very flexible nature of the Python programming language itself. The matplotlib developers who aspire to attain an advanced level, must tackle all of this and more. This book aims to be a guide for those in search of such mastery.
What this book covers

Chapter 1, Getting Up to Speed, covers some history and background of matplotlib, goes over some of the latest features of the library, provides a refresher on Python 3 and IPython Notebooks, and whets the reader's appetite with some advanced plotting examples.

Chapter 2, The matplotlib Architecture, reviews the original design goals of matplotlib and then proceeds to discuss its current architecture in detail, providing visualizations of the conceptual structure and relationships between the Python modules.

Chapter 3, matplotlib APIs and Integrations, walks the reader through the matplotlib APIs, adapting a single example accordingly, examines how third-party libraries are integrated with matplotlib, and gives migration advice to the advanced users of the deprecated pylab API.

Chapter 4, Event Handling and Interactive Plots, provides a review of the event-based systems, covers event loops in matplotlib and IPython, goes over a selection of matplotlib events, and shows how to take advantage of these to create interactive plots.

Chapter 5, High-level Plotting and Data Analysis, combines the interrelated topics, providing a historical background of plotting, a discussion on the grammar of graphics, and an overview of high-level plotting libraries. This is then put to use in a detailed analysis of weather-related data that spans 120 years.

Chapter 6, Customization and Configuration, covers the custom styles in matplotlib and the use of grid specs to create a dashboard effect with the combined plots. The lesser-known configuration options are also discussed with an eye to optimization.

Chapter 7, Deploying matplotlib in Cloud Environments, explores a use case for matplotlib in a remote deployment, which is followed by a detailed programmatic batch-job example using Docker and Amazon AWS.

Chapter 8, matplotlib and Big Data, provides detailed examples of working with large local data sets, as well as distributed ones, covering options such as numpy.memmap, HDF5, and Hadoop. Plots with millions of points will also be demonstrated.

Chapter 9, Clustering for matplotlib, introduces parallel programming and clusters that are designed for use with matplotlib, demonstrating how to distribute the parts of a problem and then assemble the results for analysis in matplotlib.
This chapter marks a conceptual dividing line for the book. We've focused on topics such as matplotlib internals and APIs, plot interaction, high-level plotting, and the use of third-party libraries. We will continue in that vein in the first part of this chapter as we discuss advanced customization techniques for matplotlib. We will finish the chapter by discussing the elements of the advanced and lesser-known matplotlib configuration. The configuration theme will continue into the next chapter and then go beyond that into the realm of deployment. As such, this chapter will mark a transition to our exploration of matplotlib in the real world and its usage in computationally intensive tasks.

This chapter will provide an overview of the following, giving you enough confidence to tackle these in more depth at your own pace:

- **Customization**
  - matplotlib styles
  - Subplots
  - Further exploration

- **Configuration**
  - The matplotlib run control
  - Options in IPython
To follow along with this chapter's code, clone the notebook's repository and start up IPython in the following way:

$ git clone https://github.com/masteringmatplotlib/custom-and-config.git
$ cd custom-and-config
$ make

**Customization**

On the journey through the lands of matplotlib, one of the signposts for intermediate territories is an increased need for fine-grained control over the libraries in the ecosystem. In our case, this means being able to tweak matplotlib for particular use cases such as specialty scales or projections, complex layouts, or a custom look and feel.

**Creating a custom style**

The first customization topic that we will cover is that of the new style support introduced in matplotlib 1.4. In the previous notebook, we saw how to get a list of the available styles:

```python
In [2]: print(plt.style.available)
['bmh', 'ggplot', 'fivethirtyeight', 'dark_background', 'grayscale']
```

Now, we're going to see how we can create and use one of our own custom styles.

You can create custom styles and use them by calling `style.use` with the path or URL to the style sheet. Alternatively, if you save the `<style-name>.mplstyle` file to the `~/.matplotlib/stylelib` directory (you may need to create it), you can reuse your custom style sheet with a call to `style.use('<style-name>')`. Note that a custom style sheet in `~/.matplotlib/stylelib` will override a style sheet defined by matplotlib if the styles have the same name.

There is a custom matplotlib style sheet included in this chapter's IPython Notebook git repository, but before we go further, let's create a function that will generate a demo plot for us. We'll then render it by using the default style in the following way, thus having a baseline to compare our work to:

```python
In [3]: def make_plot():
   x = np.random.randn(5000, 6)
   (figure, axes) = plt.subplots(figsize=(16,10))
```
(n, bins, patches) = axes.hist(
    x, 12, normed=1, histtype='bar',
    label=['Color 1', 'Color 2', 'Color 3',
           'Color 4', 'Color 5', 'Color 6'])

axes.set_title(
    "Histogram\nfor a\nNormal Distribution", fontsize=24)
axes.set_xlabel("Data Points", fontsize=16)
axes.set_ylabel("Counts", fontsize=16)
axes.legend()
plt.show()

In [4]: make_plot()

The following is the sample plot obtained as result of the preceding code:

![Graphical representation of a histogram for a normal distribution with colored bars.]

The preceding plot is the default style for matplotlib plots. Let's do something fun by copying the style of Thomas Park's *Superhero* Bootstrap theme. It's a darker theme with muted blues and desaturated accent colors. There is a screenshot of a demo website in the IPython Notebook for this chapter.
Customization and Configuration

There are two styles provided, which differ only in the coloring of the text:

In [6]: ls -l ../styles
   total 16
   -rw-r--r-- 1 u g 473 Feb 4 14:54 superheroine-1.mplstyle
   -rw-r--r-- 1 u g 473 Feb 4 14:53 superheroine-2.mplstyle

Let's take a look at the second one's contents, which show the hexadecimal colors that we copied from the Bootstrap theme:

In [7]: cat ../styles/superheroine-2.mplstyle
   lines.color: 4e5d6c
   patch.edgecolor: 4e5d6c
   text.color: df691b
   axes.facecolor: 2b3e50
   axes.edgecolor: 4e5d6c
   axes.labelcolor: df691b
   axes.color_cycle: df691b, 5cb85c, 5bc0de, f0ad4e, d9534f, 4e5d6c
   axes.axisbelow: True
   xtick.color: 8c949d
   ytick.color: 8c949d
   grid.color: 4e5d6c
   figure.facecolor: 2b3e50
   figure.edgecolor: 2b3e50
   savefig.facecolor: 2b3e50
   savefig.edgecolor: 2b3e50
   legend.fancybox: True
   legend.shadow: True
   legend.frameon: True
   legend.framealpha: 0.6
The idea behind the matplotlib styles is wonderfully simple—don't reinvent anything, just offer an option for easy organization of data. If the preceding code looks familiar, it's because it is also available in the matplotlib run control configuration file, `matplotlibrc`, which will be discussed at the end of the chapter. Let's see how our custom style overrides the default color definitions:

In [8]: plt.style.use("../styles/superheroine-2.mplstyle")
In [9]: make_plot()

The following is the plot obtained as result of the preceding code:

For a tiny bit of an effort, we have a significantly different visual impact. We'll continue using this style for the remainder of the chapter. In particular, we'll see what it looks like in the following section, when we assemble a collection of subplots.

**Subplots**

In this section, we'll create a sophisticated subplot to give you a sense of matplotlib's plot layout capabilities. The system is flexible enough to accommodate everything from simple adjustments to the creation of dashboards in a single plot.
For this section, we have chosen to ingest data from the well-known **UCI Machine Learning Repository**. In particular, we’ll use the 1985 **Automobile Data Set**. It serves as an example of data that can be used to assess the insurance risks for different vehicles. We will use it in an effort to compare 21 automobile manufacturers (using the 1985 data) along the following dimensions:

- Mean price
- Mean city MPG
- Mean highway MPG
- Mean horsepower
- Mean curb weight
- Mean relative average loss payment
- Mean insurance riskiness

We will limit ourselves to automobile manufacturers that have data for losses, as well as six or more rows of data. Our subplot will comprise of the following sections:

- An overall title
- Line plots for maximum, mean, and minimum prices
- A stacked bar chart for combined riskiness or losses
- A stacked bar chart for riskiness
- A stacked bar chart for losses
- Radar charts for each automobile manufacturer
- A combined scatterplot for the city and highway MPG

These will be composed as subplots in the following manner:

```
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>overall title</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>price ranges</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>combined loss/risk</td>
<td>radar plots</td>
</tr>
<tr>
<td>risk</td>
<td>loss</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>mpg</td>
<td></td>
</tr>
</tbody>
</table>
```
Revisiting Pandas

We've going to use a set of demonstration libraries that we included with this notebook to extract and manipulate the automobile maker data. Like we did before, we will take advantage of the power provided by the Pandas statistical analysis library. Let's load our modules by using the following code:

```python
In [10]: import sys
    sys.path.append("../lib")
    import demodata, demoplot
```

As you can see in the IPython Notebook, there's more data there than what we need for the subplotting tasks. Let's created a limited set by using the following code:

```python
In [11]: limited_data = demodata.get_limited_data()
    limited_data.head()
```

Out[11]:

The following table is obtained as a result of the preceding command:

<table>
<thead>
<tr>
<th>make</th>
<th>price</th>
<th>city mpg</th>
<th>highway mpg</th>
<th>horsepower</th>
<th>weight</th>
<th>riskiness</th>
<th>losses</th>
</tr>
</thead>
<tbody>
<tr>
<td>audi</td>
<td>13950</td>
<td>24</td>
<td>30</td>
<td>102</td>
<td>2337</td>
<td>2</td>
<td>164</td>
</tr>
<tr>
<td>audi</td>
<td>17450</td>
<td>18</td>
<td>22</td>
<td>115</td>
<td>2824</td>
<td>2</td>
<td>164</td>
</tr>
<tr>
<td>audi</td>
<td>17710</td>
<td>19</td>
<td>25</td>
<td>110</td>
<td>2844</td>
<td>1</td>
<td>158</td>
</tr>
<tr>
<td>audi</td>
<td>23875</td>
<td>17</td>
<td>20</td>
<td>140</td>
<td>3086</td>
<td>1</td>
<td>158</td>
</tr>
<tr>
<td>bmw</td>
<td>16430</td>
<td>23</td>
<td>29</td>
<td>101</td>
<td>2395</td>
<td>2</td>
<td>192</td>
</tr>
</tbody>
</table>

This has provided us with the full set of data minus the columns that we don't care about right now. However, we want to apply an additional constraint—we want to exclude auto manufacturers that have fewer than six rows in our dataset. We will do so with the help of the following command:

```python
In [16]: data = demodata.get_limited_data(lower_bound=6)
```
We've got the data that we want, but we still have some preparations left to do. In particular, how are we going to compare data of different scales and relationships? Normalization seems like the obvious answer, but we want to make sure that the normalized values compare appropriately. High losses and a high riskiness factor are less favorable, while a higher number of miles per gallon is more favorable. All this is taken care of by the following code:

```python
In [19]: normed_data = data.copy()
    normed_data.rename(
        columns={"horsepower": "power"}, inplace=True)
In [20]: demodata.norm_columns(
    ["city mpg", "highway mpg", "power"], normed_data)
In [21]: demodata.invert_norm_columns(
    ["price", "weight", "riskiness", "losses"],
    normed_data)
```

What we did in the preceding code was make a copy of the limited data that we've established as our starting point, and then we updated the copied set by calling two functions—the first function normalized the given columns whose values are more favorable when higher, and the other function inverted the normalized values to match the first normalization (as their pre-inverted values are more favorable when lower). We now have a normalized dataset in which all the values are more favorable when higher.

If you would like to have more exposure to Pandas in action, be sure to view the functions in the `demodata` module. There are several useful tricks that are employed there to manipulate data.

**Individual plots**

Before jumping into subplots, let's take a look at a few individual plots for our dataset that will be included as subplots. The first one that we will generate is for the automobile price ranges:

```python
In [22]: figure = plt.figure(figsize=(15, 5))
    prices_gs = mpl.gridspec.GridSpec(1, 1)
    prices_axes = demoplot.make_autos_price_plot(
        figure, prices_gs, data)
plt.show()
```
Note that we didn't use the usual approach that we had taken, in which we get the figure and axes objects from a call to `plt.subplots`. Instead, we opted to use the `GridSpec` class to generate our axes (in the `make_autos_price_plot` function). We've done this because later, we wish to use `GridSpec` to create our subplots.

Here is the output that is generated from the call to `plt.show()`:

![Auto Price Changes](image)

Keep in mind that the preceding plot is a bit contrived (there's no inherent meaning in connecting manufacturer maximum, mean, and minimum values). Its sole purpose is to simply provide some eye candy for the subplot that we will be creating. As you can see from the instantiation of `GridSpec`, this plot has one set of axes that takes up the entire plot. Most of our individual plots will have the same geometry. The one exception to this is the radar plot that we will be creating.

Radar plots are useful when you wish to compare normalized data to multiple variables and populations. Radar plots are capable of providing visual cues that reveal insights instantly. For example, consider the following figure:
The preceding figure shows the data that was consolidated from several 1985 Volvo models across the dimensions of price, inverse losses to insurers, inverse riskiness, weight, horsepower, and the highway and city miles per gallon. Since the data has been normalized for the highest values as the most positive, the best scenario would be for a manufacturer to have colored polygons at the limits of the axes. The conclusions that we can draw from this is this—relative to the other manufacturers in the dataset, the 1985 Volvos are heavy, expensive, and have a pretty good horsepower. However, where they really shine is in the safety for insurance companies—low losses and a very low risk (again, the values that are larger are better). Even Volvo's minimum values are high in these categories. That's one manufacturer. Let's look at the whole group:

```
In [27]: figure = plt.figure(figsize=(15, 5))
   radar_gs = mpl.gridspec.GridSpec(
       3, 7, height_ratios=[1, 10, 10], wspace=0.50,
       hspace=0.60, top=0.95, bottom=0.25)
   radar_axes = demoplot.make_autos_radar_plot(
       figure, radar_gs, normed_data)
   plt.show()
```

The following table is obtained as a result of the preceding code:

There are interesting conclusions to the graph from this view of the data, but we will focus on the code that generated it. In particular, note the geometry of the grid—three by seven. What does this mean and how are we going to use it? We have two rows of six manufacturers. However, we added an extra row for an empty (and hidden) axis. This is used at the top for the overall title. We then added an extra column for the legend, which spans two rows. This brings us from a grid of two by six to a grid of three by seven. The remaining 12 axes in the grid are populated with a highly customized polar plot, giving us the radar plots for each of the manufacturers.
This example was included not only because it's visually compelling, but also because it will show how flexible the grid specification system for matplotlib is when we put them together. We have the ability to place plots within plots.

**Bringing everything together**

We've seen a small aspect of the GridSpec usage. This has been a tiny warm-up exercise compared to what's coming! Let's refresh with the ASCII sketch of the subplots that we wanted to create. Flip back to that page and look at the layout. We have three axes that will be stretching all the way across the title, price ranges, and the MPG data at the bottom. The three riskiness or losses plots will then be placed on the left-hand side in the middle of the page, and the radar plots will take the other half of that part of the plot on the right-hand side.

We can plot what this will look like before adding any of the data, just by creating the grid and subplot specification objects. The following may look a bit hairy, but keep in mind that when splicing the subplot specs, you're using the same technique that was used when splicing the NumPy array data:

```python
In [28]: figure = plt.figure(figsize=(10, 8))
    gs_master = mpl.gridspec.GridSpec(4, 2, height_ratios=[1, 2, 8, 2])
    # Layer 1 - Title
    gs_1 = mpl.gridspec.GridSpecFromSubplotSpec(1, 1, subplot_spec=gs_master[0, :])
    title_axes = figure.add_subplot(gs_1[0])
    # Layer 2 - Price
    gs_2 = mpl.gridspec.GridSpecFromSubplotSpec(1, 1, subplot_spec=gs_master[1, :])
    price_axes = figure.add_subplot(gs_2[0])
    # Layer 3 - Risks & Radar
    gs_31 = mpl.gridspec.GridSpecFromSubplotSpec(2, 2, height_ratios=[2, 1],
                                              subplot_spec=gs_master[2, :1])
    risk_and_loss_axes = figure.add_subplot(gs_31[0, :])
    risk_axes = figure.add_subplot(gs_31[1, :1])
    loss_axes = figure.add_subplot(gs_31[1:, 1])
    gs_32 = mpl.gridspec.GridSpecFromSubplotSpec(1, 1, subplot_spec=gs_master[2, 1])
```
In the preceding code, when we instantiated `GridSpec`, we provided a geometry of four rows and two columns. We then passed the data for the height ratios so that each row will have an appropriate size that is relative to the others. In the section at the middle, for the `risk` and `radar` plots, we gave a geometry of two rows and two columns, and again passed the height ratios that provide the proportions we desire. This code results in the following plot:
That's exactly what we were aiming for. Now, we're ready to start adding individual plots. The code that generated the preceding skeleton plot differs from the final result in the following three key ways:

- The axes that are created will now get passed to the plot functions
- The plot functions will update the axes with their results (and thus no longer be empty)
- The skeleton radar plot had a one-by-one geometry; the real version will instead have a five-by-three geometry in the same area

Here is the code that inserts all the individual plots into their own subplots:

```python
In [29]: figure = plt.figure(figsize=(15, 15))
gs_master = mpl.gridspec.GridSpec(  
   4, 2, height_ratios=[1, 24, 128, 32], hspace=0,
   wspace=0)

# Layer 1 - Title
gs_1 = mpl.gridspec.GridSpecFromSubplotSpec(  
   1, 1, subplot_spec=gs_master[0, :])
title_axes = figure.add_subplot(gs_1[0])
title_axes.set_title(  
   "Demo Plots for 1985 Auto Maker Data",
   fontsize=30, color="#cdced1")
demoplot.hide_axes(title_axes)

# Layer 2 - Price
gs_2 = mpl.gridspec.GridSpecFromSubplotSpec(  
   1, 1, subplot_spec=gs_master[1, :])
price_axes = figure.add_subplot(gs_2[0])
demoplot.make_autos_price_plot(
```

...
Customization and Configuration

    figure, pddata=data, axes=price_axes)

# Layer 3, Part I - Risks
    gs_31 = mpl.gridspec.GridSpecFromSubplotSpec(2, 2, height_ratios=[2, 1], hspace=0.4, subplot_spec=gs_master[2, :1])
    risk_and_loss_axes = figure.add_subplot(gs_31[0, :])
    demoplot.make_autos_loss_and_risk_plot(figure, pddata=normed_data, axes=risk_and_loss_axes, x_label=False, rotate_ticks=True)
    risk_axes = figure.add_subplot(gs_31[1, :1])
    demoplot.make_autos_riskiness_plot(figure, pddata=normed_data, axes=risk_axes, legend=False, labels=False)
    loss_axes = figure.add_subplot(gs_31[1:, 1])
    demoplot.make_autos_losses_plot(figure, pddata=normed_data, axes=loss_axes, legend=False, labels=False)

# Layer 3, Part II - Radar
    gs_32 = mpl.gridspec.GridSpecFromSubplotSpec(5, 3, height_ratios=[1, 20, 20, 20, 20], hspace=0.6, wspace=0, subplot_spec=gs_master[2, 1])
    (rows, cols) = geometry = gs_32.get_geometry()
    title_axes = figure.add_subplot(gs_32[0, :])
    inner_axes = []
    projection = radar.RadarAxes(spoke_count=len(normed_data.groupby("make").mean().columns))

[154]
[inner_axes.append(figure.add_subplot(
    m, projection=projection))
    for m in [n for n in gs_32][cols:]]
demoplot.make_autos_radar_plot(
    figure, pddata=normed_data,
    title_axes=title_axes, inner_axes=inner_axes,
    legend_axes=False, geometry=geometry)

# Layer 4 - MPG
gs_4 = mpl.gridspec.GridSpecFromSubplotSpec(1, 1, subplot_spec=gs_master[3, :])
mpg_axes = figure.add_subplot(gs_4[0])
demoplot.make_autos_mpg_plot(
    figure, pddata=data, axes=mpg_axes)

# Tidy up
gs_master.tight_layout(figure)
plt.show()

Though there is a lot of code here, keep in mind that it's essentially the same as the skeleton of subplots that we created. For most of the plots, all we had to do was make a call to the function that creates the desired plot, passing the axes that we created by splicing a part of the spec and adding a subplot for that splice to the figure. The one that wasn't so straightforward was the radar plot collection. This is due to the fact that we not only needed to define the projection for each radar plot, but also needed to create the 12 axes needed for each manufacturer. Despite this complication, the use of GridSpec and GridSpecFromSubplotSpec clearly demonstrates the ease with which complicated visual data can be assembled to provide all the power and convenience of a typical dashboard view.
The creation of complex subplots in matplotlib can be perceived as a daunting task. However, the following basic practices can help you make it a painless process of creating visual goodness:

1. Write down an explicit plan for what you want to present, which data you want to combine, where you will use the stacked data and means, and so on.
2. Sketch out on paper or in an ASCII diagram the desired layout. This will often reveal something that you hadn’t considered.
3. With the layout decided upon, create a `GridSpec` and `GridSpecFromSubplotSpec`-based collection of subplots with empty axes. Don't add any plot data. Your grid-tweaking should happen at this point.
4. With your girds ironed out, update your axes with the desired plots.
Further explorations in customization

We have covered two areas of customization that come up frequently in various online forums. The other topics in advanced matplotlib customization include the creation of axes, scales, projections, and backends for some particular data or project requirements. Each of these have tutorials or examples that are provided by the matplotlib project, and given your newly attained comfort level with reading the matplotlib sources directly, these are now within your reach.

Several of these are worth mentioning specifically:

- The API example code for `custom_projection_example.py` provides a highly detailed look into the means by which you can create custom projections. Another example of this is the radar plot that we created earlier in this chapter. If you view the library files for this chapter, you will see that we based the work on the polar projection that comes with matplotlib.

- The API example code for `custom_scale_example.py` shows how to create a new scale for the y axis, which uses the same system as that of the Mercator map projection. This is a smaller amount of code, which is more easily digestible than the preceding projection example.

- The matplotlib Transformations Tutorial will teach you how to create data transforms between coordinate systems, use axes transforms to keep the text bubbles in fixed positions while zooming, and blend transformations for the highlighting portions of the plotted data.

Finally, Joe Kington, a geophysicist, created an open source project for equal-angle Stereonets in matplotlib. Stereonets, or Wulff net are used in geological studies and research, and Dr. Kington's code provides excellent examples of custom transforms and projections. All of this has been documented very well. This is an excellent project to examine in detail after working on the matplotlib.org tutorials and examples on creating custom projections, scales, and transformations.

Configuration

We've just covered some examples of matplotlib customization. Hand in hand with this topic is that of configuration—the tweaking of predefined values to override default behaviors. The matplotlib module offers two ways to override the default values for the configuration settings—you can either run the control files, or run the control parameters that are stored in-memory to make changes to a running instance.
Customization and Configuration

The run control for matplotlib

While commonly expanded to the run control, the .rc extension and -rc suffix trace their origins to 1965 and the Multics (short for Multiplexed Information and Computing Service) operating system, where rc stood for the run command. Like many software systems that were developed on UNIX- or BSD-based machines, matplotlib has an rc file where the control of matplotlib may be configured. This control is not limited to configuration files; one may also access an rc object via the matplotlib API. Each of these is covered in the following few sections.

File and directory locations

The configuration of matplotlib is possible through the creation and editing of the matplotlibrc file. The matplotlib module will search for this file in the following locations:

- The current working directory
- The $HOME/.matplotlib directory
- $HOME/.matplotlib/mpl-data/, where INSTALL is the Python site-packages directory where matplotlib was installed
- A temporary directory created by Python, in case $HOME/.matplotlib is not writable
- The directory defined by the MPLCONFIGDIR environment variable (if defined, this directory will override the use of $HOME/.matplotlib)

You can use matplotlib to find the location of your configuration directory by using the following code:

In [30]: mpl.get_configdir()
Out[30]: '/Users/yourusername/.matplotlib'

Similarly, you can display the currently active matplotlibrc file with the help of the following code:

In [31]: mpl.matplotlib_fname()

Using the matplotlibrc file

There are hundreds of configuration options that are available to you via the matplotlibrc file:

In [32]: len(mpl.rcParams.keys())
Out[32]: 200
You can have a look at some of these with the following code:

```python
In [33]: dict(list(mpl.rcParams.items())[:10])
```

```
Out[33]: {'axes.grid': False,
         'mathtext.fontset': 'cm',
         'mathtext.cal': 'cursive',
         'docstring.hardcopy': False,
         'animation.writer': 'ffmpeg',
         'animation.mencoder_path': 'mencoder',
         'backend.qt5': 'PyQt5',
         'keymap.fullscreen': ['f', 'ctrl+f'],
         'image.resample': False,
         'animation.ffmpeg_path': 'ffmpeg'}
```

The configuration options that you need depend entirely upon your use cases, and thanks to matplotlib's ability to search multiple locations, you can have a global configuration file as well as per-project configurations.

We've already run into a special case of matplotlib configuration—the contents of the style files that we saw at the beginning of this chapter. If you were so inclined, all of those values could be entered into a `matplotlibrc` file, thus setting the default global look and feel for matplotlib.

A complete template for the `matplotlibrc` file is available in the matplotlib repository on GitHub. This is the canonical reference for all your matplotlib configuration needs. However, we will point out a few that may be helpful if you keep them in mind, including some that may be used to decrease the render times:

- `agg.path.chunksize: 20000`: This improves the speed of operations slightly and prevents an `Agg` rendering failure
- `path.simplify: true`: This removes the invisible points to reduce the file size and increase the rendering speed
- `savefig.jpeg_quality: xx`: This lowers the default `.jpg` quality of the saved files
- `axes.formatter.limits`: This indicates when you use scientific notations for exponents
- `webagg.port`: This is the port that you should use for the web server in the `WebAgg` backend
- `webagg.port_retries`: With this, the number of other random ports will be tried until the one that is available is found
Updating the settings dynamically

In addition to setting the options in the `matplotlibrc` file, you have the ability to change the configuration values on the fly by directly accessing the `rcParams` dictionary that we saw earlier:

```
In [34]: mpl.rcParams['savefig.jpeg_quality'] = 72
Out[34]: mpl.rcParams['axes.formatter.limits'] = [-5, 5]
```

If you either find out that your changes have caused some problems, or you want to revert to the default values for any reason, you can do so with `mpl.rcdefaults()`, which is demonstrated in the following code:

```
In [35]: mpl.rcParams['axes.formatter.limits']
Out[35]: [-5, 5]
In [36]: mpl.rcdefaults()
In [37]: mpl.rcParams['axes.formatter.limits']
Out[37]: [-7, 7]
```

Options in IPython

If you are using matplotlib via IPython, as many do, there are IPython matplotlib configuration options that you should be aware of, especially if you regularly use different backends or integrate with different event loops. When you start up IPython, you have the ability to configure matplotlib for interactive use by setting a default matplotlib backend in the following way:

```
--matplotlib=XXX
```

In the preceding code, `XXX` is one of `auto`, `gtk`, `gtk3`, `inline`, `nbagg`, `osx`, `qt`, `qt4`, `qt5`, `tk`, or `wx`. Similarly, you can enable a GUI event loop integration with the following option:

```
--gui=XXX
```

In the preceding code, `XXX` is one of `glut`, `gtk`, `gtk3`, `none`, `osx`, `pyglet`, `qt`, `qt4`, `tk`, or `wx`.

While you may see the `--pylab` or `%pylab` option being referred to in older books and various online resources (including some of matplotlib's own official documentation), its use has been discouraged since IPython version 1.0. It is better to import the modules that you will be using explicitly and not use the deprecated `pylab` interface at all.
Summary
In this chapter, we covered two areas of detailed customization—the creation of custom styles, as well as complex subplots. In the previous chapters, you have been exposed to the means by which you can discover more of matplotlib's functionality through its sources. It was in this context that the additional topics in customization were mentioned. With this, we transitioned into the topic of matplotlib configuration via files as well as rcParams. This is a transitional topic that will be picked up again at the beginning of the next chapter, where we will cover matplotlib deployments.
Where to buy this book

You can buy Mastering 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.