Matplotlib for Storytellers

By: Alexander Clark

This version: November 12, 2021

Contents

Code

All code and data files are (not yet) available on the book's GitHub repository. Note I exclude imports from all Python files. These imports below should cover the entire text. All of these should be included if you installed Anaconda, except for the ternary library. When saving figures, I also run `fig.tight_layout()`, which is not always included in the Python files.
To the early reader: I will name and add all code blocks to this list later.

```
import numpy as np
import pandas as pd
import math
import matplotlib as mpl
import matplotlib.pyplot as plt
# from matplotlib import colors
import matplotlib.gridspec as gridspec
from matplotlib.ticker import MultipleLocator
from matplotlib.colors import colorConverter

# For Special Topics
import ternary # requires installation
from sklearn.manifold import MDS
from sklearn.decomposition import PCA
from scipy import stats
```

Preface

Technical Notes and Prerequisites

I use Python 3.7 and assume all code is to be run in a Jupyter Notebook. I assume familiarity with basic Python programming, NumPy, pandas, and even matplotlib. In Part I, the premise is that you can make a plot, but now you want to polish it. Other parts assume less background knowledge. For those needing to review some Python before approaching this text, I recommend *A Whirlwind Tour of Python* and *Python Data Science Handbook*, both by Jake VanderPlas. There is also a good Data Visualization section in *Coding for Economists* by Arthur Turrell.

Why Matplotlib?

Though a bit aged, matplotlib is the standard in Python. matplotlib is integrated with pandas and Seaborn is based off matplotlib. You might prefer Plotnine if you already know R's ggplot2. You might prefer to leave Python and use D3 if you know javascript. You might prefer Microsoft Excel if you want consultants in your audience to feel at home.

I recommend matplotlib to anyone who is already committed to working in Python (and with the Python community) and values reproducibility and customizability. By the time we get to Part III, we'll be drawing more than plotting. This allows for more creativity than Excel allows and we'll maintain a reproducible Python-only workflow.

Good Visualization is Like Good Writing

This book isn't a guide to visualization design, but we must consider, at least briefly, what makes for good visualization and then why you might find matplotlib useful in that pursuit.

Data visualization is a form of communication not much different than writing. Cole Nussbaumer Knaflic's *Storytelling with Data* parallels writing style guides like Sir Ernest Gowers' *The Complete Plain Words*. They both emphasize clarity and stripping out what is not essential. Matplotlib doesn't offer any unique advantage in pursuing clarity. Instead, the advantage is a tactical one. Matplotlib will expand your options. Sometimes straightforward prose is appropriate and sometimes only poetry will be stirring enough to capture your audience's attention. There exist prosaic visualizations and poetic visualizations with all the same tradeoffs.

Prose is precise and direct. Poetry has a certain beauty that invites interest and mediates higher truths. The familiar bar chart is prose, plainly reporting the numbers that need to be reported. Your boss will appreciate prose in a routine meeting. But imagine the king must wrestle with a difficult truth. Prose won't do. Only a jester or a Shakespearean fool can deliver the message and only by rhyme and riddle. So it may be with your C-level audience. The small truths of your bar charts don't matter to a busy CEO. Easier said than done, but capture your CEO's attention with a poetic visualization that might sacrifice some precision for its larger message.

A hurdle to crafting good visualizations is being limited to a short menu of cookie cutter graphics, whatever is available in Excel, a dashboard tool, or from a limited knowledge of matplotlib. Ahead of us is the chance to break free from those cookie cutter, ready-made visuals. In writing, George Orwell made good note of the "invasion of one's mind by ready-made phrases," in his worthwhile essay *Politics and the English Language*:

> [Ready-made phrases] will construct your sentences for you—even think your thoughts for you, to a certain extent—and at need they will perform the important service of partially concealing your meaning even from yourself.

The important point here is that the unimaginative application of ready-made visualizations, just like phrases, can conceal your

meaning from yourself, not to mention your audience, and create a monotonous presentation of bar chart after bar chart.

The parallels between writing and making visuals go one level further. If you want to *become* a good writer, you will learn grammar, read good writers who came before you, write a lot, and skirt the rules a bit as you find your voice. In other words, you will do many things. Data visualization is no different. In what follows, you will begin to master just one thing, the technical grammar of matplotlib.

Resources and Inspiration

Before you dive in, you ought to get excited about data visualization. While there is a glaring lack of major museum space devoted to data visualization (I just recall a disappointing exhibit at the Cooper Hewitt), you will find many wonderful displays if you only keep your eyes peeled.

If you like to listen to people talk about data visualization, I recommend the Data Stories podcast.

If you'd like to start by reading one of the pioneers, check out Edward Tufte, who continues to write new material. For more explicit or domain-specific guidance than Tufte might provide, see Storytelling with Data by Cole Nussbaumer Knaflic or Better Data Visualization by Jonathan Schwabish. Many of Schwabish's main themes are also communicated more briefly in Schwabish 2014. I have limited patience for how-to guides when they edge toward being overly prescriptive (I've never read any books on how to write well either), but I've profited from these titles nonetheless. They are useful in establishing fundamentals and surfacing more variety in visualizations, helping to inspire a richer repertoire. Knaflic's book is oriented toward business professionals and Schwabish adds his own public policy background. As a result, Knaflic concentrates on what I call prosaic visuals and Schwabish pushes further into the realm of poetry. Schwabish discusses the tradeoffs between standard and nonstandard graphs, noting that novelty can encourage more active processing, providing further justification for using a less accurate graph in select, exploratory cases.

Media outlets like the New York Times and Wall Street Journal make usually good use of data visualization. Take appropriate inspiration these sources and from the r/DataIsBeautiful and r/DataIsUgly subreddits.

Text Organization

Continuing the parallel to writing, I have built this text around two main parts: Prose and Poetry, though the distinction between prose and poetry is surely less exact than the division I've created. Prose, or Part I, focuses on the fundamentals of customizing plots through the object-oriented interface. This section attempts to be reasonably thorough in breadth while providing only a minimal effective dose in depth. Then, after a mathematical interlude in Part II, we reach poetry in Part III. There can be no comprehensiveness to this section. I provide a guide to drawing in matplotlib, mostly with various artist objects. The mathematical interlude is there for those who would like to review some trigonometry I use. Then, I introduce two special (for fun) topics in Part IV, multi-dimensional scaling and ternary plots.

Part I

Prose

Quince, Cabbage, Melon and Cucumber by Juan Sánchez Cotán
(Public Domain)

Chapter 1

The Object-oriented Interface

Matplotlib offers two interfaces: a MATLAB-style interface and the more cumbersome object-oriented interface. If you count yourself among the matplotlib-averse, you likely never had the stomach for object-oriented headaches. Still, we are using the object oriented interface because we can do more with this.

The MATLAB-style interface looks like the following.

```
plt.plot(x,y)
plt.title("My Chart")
```

The object-oriented interface looks like this.

```
fig, ax = plt.figure(), plt.axes()
ax.plot(x,y)
ax.set_title("My Chart")
```

There is no such thing as a free lunch, so you will observe this interface requires more code to do the same exact thing. Its virtues will be more apparent later. Object-oriented programming (OOP) also requires some new vocabulary. OOP might be contrasted with procedural programming as another common method of programming. In procedural programming, the MATLAB-style interface being an example, the data and code are separate and the programmer creates procedures that operate on the program's data. OOP instead focuses on the creation of *objects* which encapsulate both data and procedures.

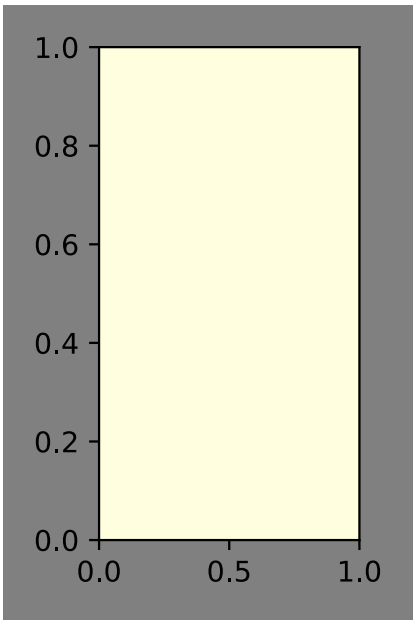
An object's data are called its *attributes* and the procedures or functions are called *methods*. In the previous code, we have

figure and axes objects, making use of axes methods `plot()` and `set_title()`, both of which add data to the axes object in some sense, as we could extract the lines and title from `ax` with more code. Objects themselves are instances of a *class*. So `ax` is an object and an instance of the Axes class. Classes can also branch into subclasses, meaning a particular kind of object might also belong to a more general class. A deeper knowledge is beyond our scope, but this establishes enough vocabulary for us to continue building an applied knowledge of matplotlib. Because `ax` contains its data, you can think of `set_title()` as changing `ax` and this helps make sense of the `get_title()` method, which simply returns the title belonging to `ax`. Having some understanding that these objects contain both procedures and data will be helpful in starting to make sense of intimidating programs or inscrutable documentation you might come across.

1.1 Figure, Axes

A plot requires a figure object and an axes object, typically defined as `fig` and `ax`. The figure object is the top level container. In many cases like in the above, you'll define it at the beginning of your code and never need to reference it again, as plotting is usually done with axes methods. A commonly used figure parameter is `figsize`, to which you can pass a sequence to alter the size of the figure. Both the figure and axes objects have a `facecolor` parameter which might help to illustrate the difference between the axes and figure.

```
fig = plt.figure(figsize = (2,3),
                 facecolor = 'gray')
ax = plt.axes(facecolor = 'lightyellow')
```

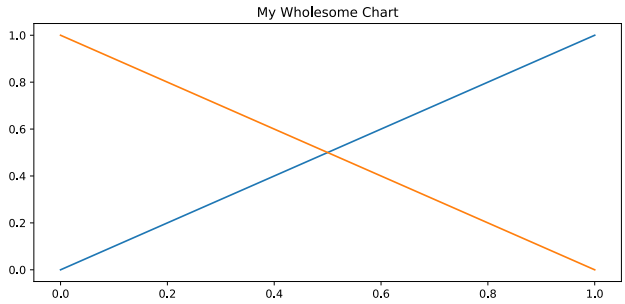


The axes object, named `ax` by convention, gets more use in most programs. In place of `plt.plot()`, you'll use `ax.plot()`. Similary, `plt.hist()` is replaced with `ax.hist()` to create a histogram. If you have experience with the MATLAB interface, you might get reasonably far with the object-oriented style just replacing the `plt` prefix on your pyplot functions with `ax` to see if you have an equivalent axes method.

This wishful coding won't take you everywhere though. For example, `plt.xlim()` is replaced by `ax.set_xlim()` to set the x-axis view limits. To modfiy the title, `plt.title()` is replaced with `ax.set_title()` and there is `ax.get_title()` simply to get the title. The axes object also happens to have a `title` attribute, which is only used to access the title, similar to the `get_title()` method. Many matplotlib methods can be classified as *getters* or *setters* like for these title methods. The plot method and its logic is different. Later calls of `ax.plot()` don't overwrite earlier calls and there is not the same getter and setter form. There's a `plot()` method but no single `plot` attribute being mutated. Whatever has been plotted can be retrieved, or gotten (getter'd?), but it's more complicated and rarely necessary. Use the code below to see what happens with two calls of `plot()` and two calls of `set_title()`. The second print statement demonstrates that the second call of `set_title()` overwrites the title attribute, but a second plot does not nullify the first.

```
x = np.linspace(0,1,2)
```

```
fig, ax = plt.figure(figsize = (8,4)), plt.axes()
ax.plot(x, x)
ax.plot(x, 1 - x)
ax.set_title("My Chart")
print(ax.title)
print(ax.get_title()) # Similar to above line
ax.set_title("My Wholesome Chart")
print(ax.get_title()) # long
```



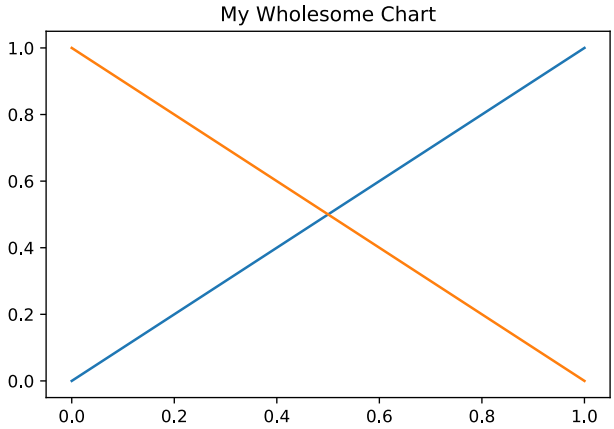
Axes methods `set_xlim()` and `get_xlim()` behave just like `set_title()` and `get_title()`, but note there is no attribute simply accessible with `ax.xlim`, so the existence of getters and setters is the more fundamental pattern.[1]

1.2 Mixing the Interfaces

You can also mix the interfaces. Use `plt.gca()` to *g*et the *c*urrent *a*xis. Use `plt.gcf()` to *g*et the *c*urrent *f*igure.

```
x = np.linspace(0,1,2)
plt.plot(x,x)
plt.title("My Chart")

ax = plt.gca()
print(ax.title)

ax.plot(x, 1 - x)
ax.set_title('My Wholesome Chart')
print(ax.title)

fig = plt.gcf()
fig.savefig('chart.pdf') # same as plt.savefig
```

[1] Getters and setters are thought of as old-fashioned. It's more Pythonic to access attributes directly, but matplotlib doesn't yet support this.

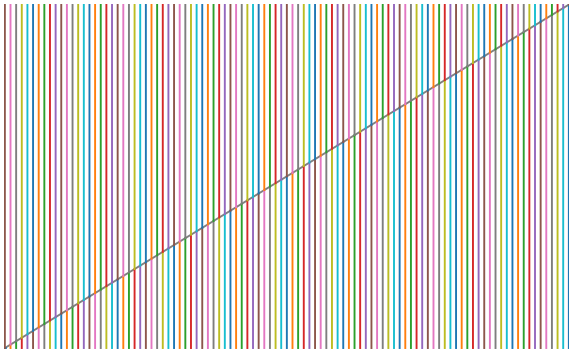


In the above, we started with MATLAB and then converted to object-oriented. We can also go in the opposite direction, though it's not always ideal, especially when working with subplots. Below, we start with our figure and axes objects, and then revert back to the MATLAB style with the `axvline()` functions (producing vertical lines across the axes), toggling off the axis lines and labels, and then saving the figure. This graph would appear unchanged if you replaced `plt.axvline()` with `ax.axvline()`, `plt.axis()` with `ax.axis()`, and `fig.savefig()` would do the same as `plt.savefig()`.

```
# OOP Start
fig, ax = plt.figure(figsize = (8,5)), plt.axes()

x = np.linspace(0,100,2)
ax.plot(x, x, color = 'gray')

ax.set_xlim([0,100])
ax.set_ylim([0,100])

# Back to pyplot functions
for i in range(101):
    plt.axvline(i,0, i / 100, color = 'C' + str(i))
    plt.axvline(i, i/100, 1, color = 'C' + str(i+5))

plt.axis('off')
plt.savefig('colorful.pdf')
```

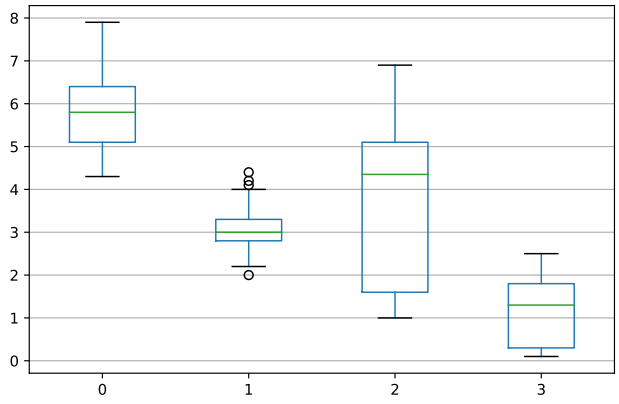
Matplotlib is also integrated into pandas, with a `plot()` method for both Series and DataFrame objects, among other functionalities. There is excellent documentation available.[2] These plots can be mixed with the object-oriented interface. You can use a plot

[2] https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html



method and specify the appropriate axes object as an argument. Below we import the iris dataset and make a boxplot with a mix of axes methods and then pyplot functions.

```
from sklearn.datasets import load_iris
data = load_iris()['data']
df = pd.DataFrame(data)

fig, ax = plt.figure(), plt.axes()

df.plot.box(ax = ax)
ax.yaxis.grid(True)
ax.xaxis.grid(False)

plt.tight_layout()
plt.savefig('iris_box.pdf')
```



The above capability is handy, especially with subplots, where every subplot will have its own axes object as we will see later.