Matplotlib for Storytellers

By: Alexander Clark

This version: July 7, 2023

BY NC SA

This text is released under the Creative Commons Attribute-NonCommercial-ShareAlike 4.0 International License.

The code is released under the MIT license.

The front cover image (on Leanpub) is from The First Book of Urizen, Plate 8, "In Living Creations Appear'd...." by William Blake (public domain).

Find more material on this book's GitHub page and on YouTube.
github.com/alexanderthclark/Matplotlib-for-Storytellers
youtube.com/@alexanderthclark

Contents

Preface **vii**
Technical Notes and Prerequisites vii
Why Matplotlib? vii
Good Visualization is like Good Writing vii
Resources and Inspiration ix
Text Organization x

I Prose **1**

1 The Object-oriented Interface **3**
1.1 Figure, Axes 4
1.2 Mixing the Interfaces 6

2 Axes Appearance, Ticks, and Grids **11**
2.1 Axis Aspect and Limits 11
2.2 Axis Lines and Spines 14
2.3 Ticks 18
2.4 Grids 22

3 Plot Elements and Coordinate Systems **27**
3.1 Primitives and Containers 27
3.1.1 Ordering with `zorder` 29
3.2 Coordinate Systems and Transformations 33
3.3 Use Window Extents 37

4 Text and Titles **41**
4.1 Simple Titles 41
4.2 Text and Placement 43
4.2.1 Text Formatting for Numbers 46
4.3 Legends 48

4.4 Annotations 53
4.4.1 Labeling and Arrows 54
4.5 Fancy Titles 58
4.5.1 Multi-colored Titles 58
4.6 Fonts 62
4.6.1 Importing Fonts with Font Manager 62

Code

1 imports.py v

All code and data files are (not yet) available on the book's GitHub repository. Note I exclude imports from the Python files in the main text. The imports below should cover the entire text. All of these should be included if you installed Anaconda, except for the ternary library, which I comment out below. When saving figures, I also sometimes run `fig.tight_layout()`, which is not always included in the Python files. See `Figure-Dev.ipynb` which contains the complete code for every figure.

```
import numpy as np
import pandas as pd
import math
from itertools import combinations
from itertools import product
from sklearn.datasets import load_iris

# matplotlib specific
import matplotlib as mpl
import matplotlib.pyplot as plt

# For Special Topics
# import ternary # requires install
# from ternary.helpers import simplex_iterator
from sklearn.manifold import MDS
from sklearn.decomposition import PCA
from scipy import stats

# Made redundant in the text
from matplotlib import colors
from matplotlib.patches import ConnectionPatch
from matplotlib.patches import Rectangle
import matplotlib.gridspec as gridspec
from matplotlib.ticker import MultipleLocator
from matplotlib.colors import colorConverter
from mpl_toolkits.mplot3d import Axes3D
```

```
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import matplotlib.dates as mdates
from matplotlib import font_manager
```

imports.py

Preface

Technical Notes and Prerequisites

I use Python 3.9 and matplotlib 3.5.1. 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.

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 **??**, 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. They are useful for their treatment of fundamentals like preattentive processing 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.

There is also a good Data Visualization section in *Coding for Economists* by Arthur Turrell. For a more advanced treatment of matplotlib, check out Scientific Visualization: Python + Matplotlib.

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 **??**, we reach poetry in Part **??**. 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 **??**, multi-dimensional scaling and ternary plots.

Part I

Prose

Still Life with Apples and Grapes by Claude Monet (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.

```python
import matplotlib.pyplot as plt
x = 1,0
y = 0,1

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

matlab-plot.py

The object-oriented interface looks like this.

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

oop-plot.py

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.

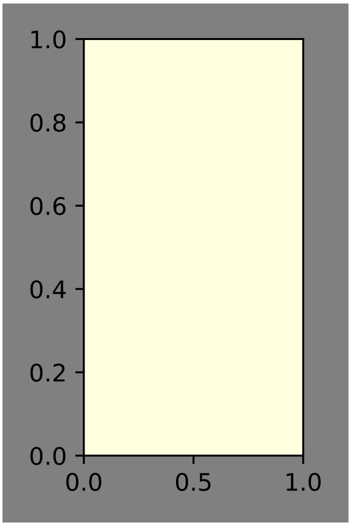
An object's data are called its *attributes* and the procedures or functions are called *methods*. In the previous code, we have figure and axes objects, making use of axes methods `plot()` and `set_title()`, both of which add data to the axes object in some sense, as we could extract the lines and title 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')
```

figparams.py



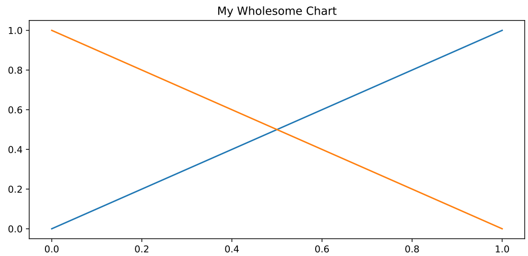
The axes object, named `ax` by convention, gets more use in most programs. In place of `plt.plot()`, you'll use `ax.plot()`. Similary, `plt.hist()` is replaced with `ax.hist()` to create a histogram. If you have experience with the MATLAB interface, you might get reasonably far with the object-oriented 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)
```

```python
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
```

gettersetter.py



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]

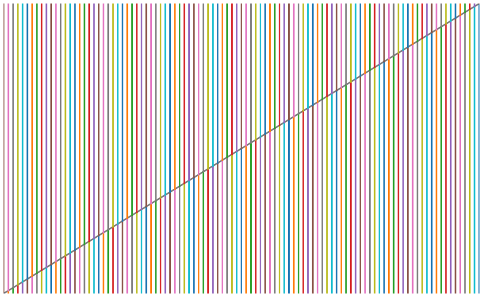
1.2 Mixing the Interfaces

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.

```python
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)
```

chart.py

[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.



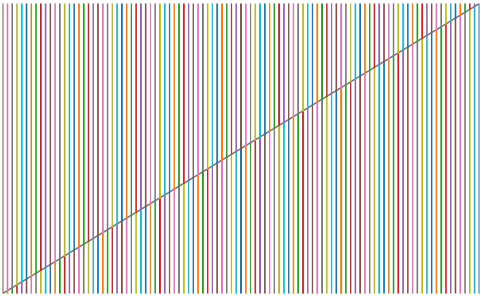
In the above, we started with MATLAB and then converted to object-oriented. We can also go in the opposite direction, though it's not always ideal, especially when working with subplots. Below, we start with our figure and axes objects, and then revert back to the MATLAB style with the `axvline()` functions (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')
```

colorful.py

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

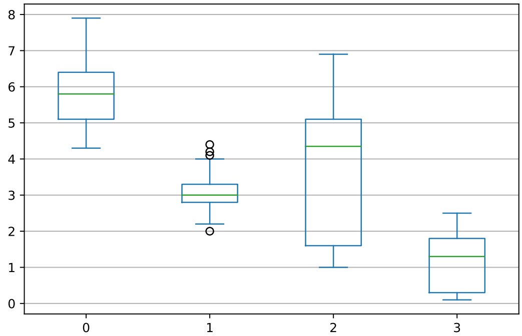
[2] https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html



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.

```python
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('irisbox.pdf')
```

irisbox.py



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

Chapter 2

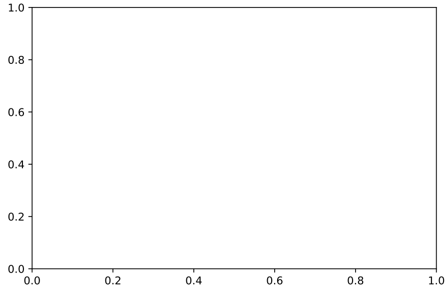
Axes Appearance, Ticks, and Grids

2.1 Axis Aspect and Limits

The most basic plot is the empty plot.

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

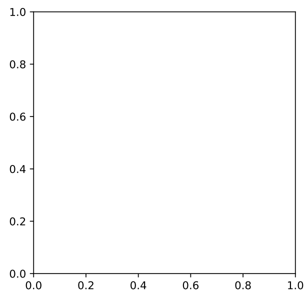
empty.py



You'll notice this defaults to plotting the square region between data points (0,0) and (1,1). However, the plot is not square by default. That is to say the *aspect* is not one, where the aspect is the ratio of height to width. This can be changed with the axes method `set_aspect()`. For equal scaling, use `ax.set_aspect('equal')` or `ax.set_aspect(1)`.

```
fig, ax = plt.figure(), plt.axes()
ax.set_aspect('equal')
```

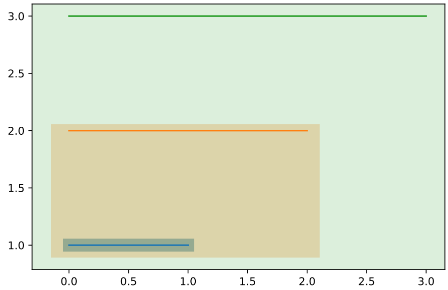
empty-square.py



As we already covered in Chapter 1, the x and y limits can be adjusted with axes methods `set_xlim()` and `set_ylim()`, taking a sequence for the minimum and maximum values. If you don't explicitly set the limits, matplotlib will set the limits automatically based on the data. You can retrieve those limits with the getter methods, `get_xlim()` and `get_ylim()`. The program below makes use of both methods. We plot a few lines, and after each plot call, matplotlib is quietly updating the axes limits. Using the `fill_between()` method, which creates a color fill in the defined region, the expanding limits are shown. The colors are chosen automatically by matplotlib because I haven't explicitly specified a color value.

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

for i in range(1,4):
    ax.plot([0,i], [i,i])
    bottom_y, top_y = ax.get_ylim()
    left_x, right_x = ax.get_xlim()
    ax.fill_between(x = [left_x,right_x],
                    y1 = bottom_y,
                    y2 = top_y,
                    alpha = 0.5/i)

# Prevent limits from automatically stretching further
# The last fill_between would stretch limits again
ax.set_ylim(bottom_y, top_y)
ax.set_xlim(left_x, right_x)
```

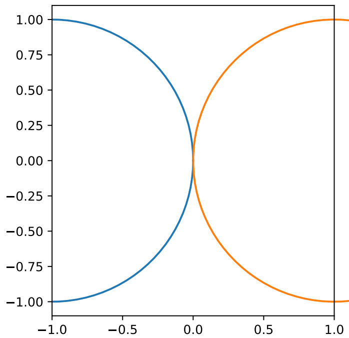
expanding-lims.py



If your axes limits are too restrictive, plot elements will be cut off. If you want your plot element to break past the end of the axes, spilling into the outer figure space, you can change this by setting `clip_on = False` in the appropriate method. Below, we create two circles with `ax.plot()` and set restrictive x-axis limits. The first circle, in blue, would extend further to the left if the limits were more generous. By default, it is clipped so we only see half of a circle. In the next call to `ax.plot()`, we create an orange circle and toggle `clip_on = False`. As a result, the circle extends to the right of the axes limits into the remaining figure space.

```
fig, ax = plt.figure(), plt.axes()
ax.set_aspect(1)

# Create a unit circle
u = np.linspace(0,2*np.pi,100)
x = np.cos(u)
y = np.sin(u)

# Default, clip_on = True
ax.plot(x-1, y)

# Unclipped, extends beyond the axes
ax.plot(x+1, y, clip_on = False)

ax.set_xlim(-1,1)
```

circle-clip.py



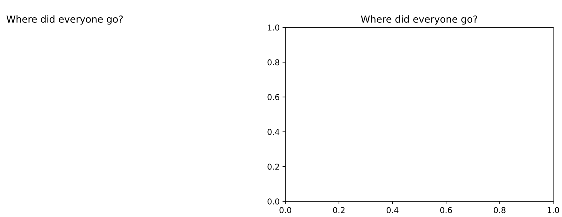
2.2 Axis Lines and Spines

You might be used to plots that aren't surrounded by a box. Those enclosing lines, included by default, are called the *spines*. The default might also be jarring if you're used to the typical x- and y-axis lines at $y = 0$ and $x = 0$, like in most math textbook plots. In this section we'll cover how to modify these.

First, you might just eliminate everything with `ax.axis('off')`. We saw `plt.axis('off)` used similarly in Chapter 1 with a program that alternated between pyplot functions and the object-oriented approach. Below is a simple plot, empty but for a title, that becomes even emptier by eliminating the axis lines and labels. For reference, on the right is the same plot if `ax.axis('off')` were excluded from the program.

```
fig, ax = plt.figure(), plt.axes()
ax.set_title("Where did everyone go?")
ax.axis('off')
```

no-axis.py

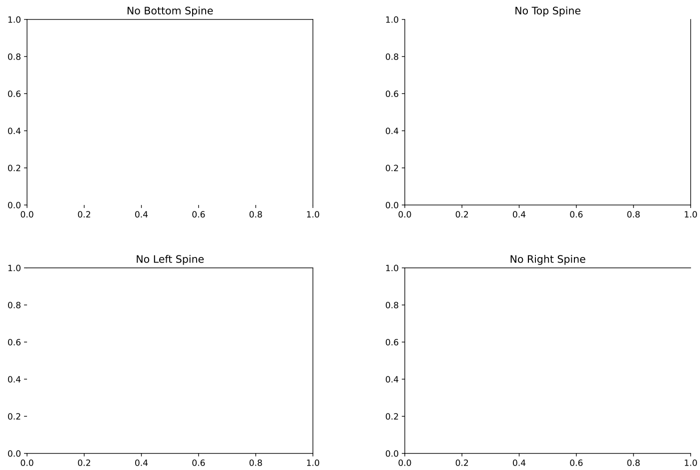


Next, we can access and modify specific spines through `ax.spines`
, which returns an `OrderedDict`. Access a specific spine using the

appropriate key: "left", "right", "top", or "bottom". A spine can be toggled on or off by passing the appropriate boolean value to set_visible().

```
for spine in 'bottom', 'top', 'left', 'right':
    fig, ax = plt.figure(), plt.axes()
    ax.set_title("No " + spine.title() + " Spine")
    ax.spines[spine].set_visible(False)
    plt.show()
```

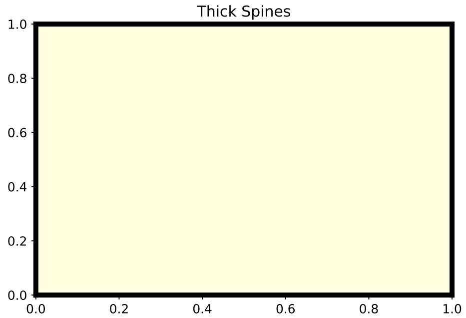
spine-vis.py



Other spine modifications might be their width and color. Again, we access a particular spine and then make use of setter methods, set_color and set_linewidth in particular.

```
fig, ax = plt.figure(), plt.axes(facecolor = '
    lightyellow')
ax.set_title("Thick Spines")
for spine in 'bottom', 'top', 'left', 'right':
    ax.spines[spine].set_color('black')
    ax.spines[spine].set_linewidth(4)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
```

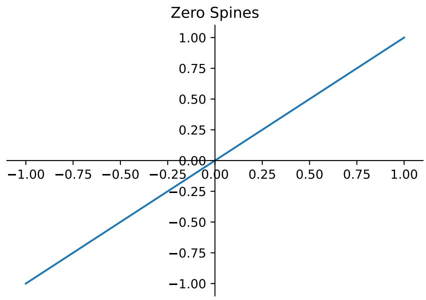
thick-spines.py



It's easy to get this far imagining that spines are simply the pieces of the box enclosing your plot. But they don't have to enclose the plot if we alter them with the `set_position` method. Below, we set the bottom spine to be along the usual x-axis and the left spine to be along the usual y-axis by passing `'zero'` to `set_position`. The right and top spines are removed.

```
fig, ax = plt.figure(), plt.axes()
ax.set_title("Zero Spines")
ax.plot([-1,1], [-1,1])
for spine in 'top', 'right':
    ax.spines[spine].set_visible(False)
for spine in 'bottom', 'left':
    ax.spines[spine].set_position('zero')
```

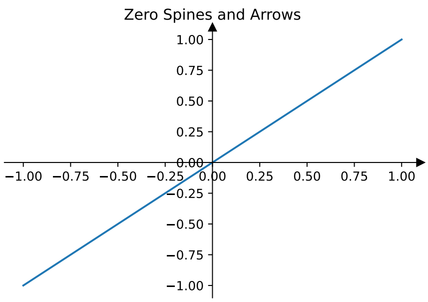
zero-spines.py



We can go a step further and add arrows at the ends of our axis lines with some clever plotting.

```
fig, ax = plt.figure(), plt.axes()
ax.set_title("Zero Spines and Arrows")
ax.plot([-1,1], [-1,1])
for spine in 'top', 'right':
    ax.spines[spine].set_visible(False)
for spine in 'bottom', 'left':
    ax.spines[spine].set_position('zero')

# get current limits
xlims = ax.get_xlim()
ylims = ax.get_ylim()

# Add arrows
ax.plot(xlims[1], 0, ">k", clip_on = False)
ax.plot(0, ylims[1], "^k", clip_on = False)

# revert limits to before the arrows
ax.set_xlim(xlims)
ax.set_ylim(ylims)
```

arrow-axes.py



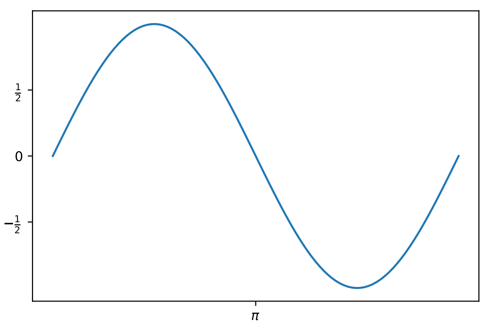
The tick labels do clutter the graph above. This can be solved after we cover Section 2.3. Knaflic 2015 recommends removing the top and right spines as part of the imperative to declutter and remove unnecessary chart border. I think it is arguable. I'm used to default spines enclosing the data. Removing them can seem untidy, like the plot guts might spill out onto the page, or as if the plot is now vulnerable to intruders without any fencing. Arrows on axis lines subtly prod the reader to imagine what happens outside of the plotted region. I don't like that if, for example, I don't want to create the impression that a linear trend in a time series graph will continue into the future.

2.3 Ticks

The important axes methods for ticks are `set_xticks`, `set_xticklabels`, and the natural y-axis counterparts. One may also use the general `set_ticks` and `set_ticklabels` with `ax.xaxis` or `ax.yaxis`—as axis (not axes) methods. These are demonstrated below, taking an array of tick locations and then the corresponding labels. I use LaTeXstrings to label the ticks. Here, that allows for a prettier y-axis, using fractions instead of decimals for tick labels. And on the x-axis, we can give a proper label of π at $x = \pi$.

```
x = np.linspace(0, np.pi * 2, 100)

fig, ax = plt.figure(), plt.axes()
ax.plot(x, np.sin(x))

# Y axis
ax.set_yticks( [-0.5, 0, 0.5] )
ax.set_yticklabels( [r"$-\frac{1}{2}$", 0,  r"$\frac{1}{2}$"] )

# X axis
ax.xaxis.set_ticks([np.pi])
ax.xaxis.set_ticklabels([r"$\pi$"])
```

ticks1.py



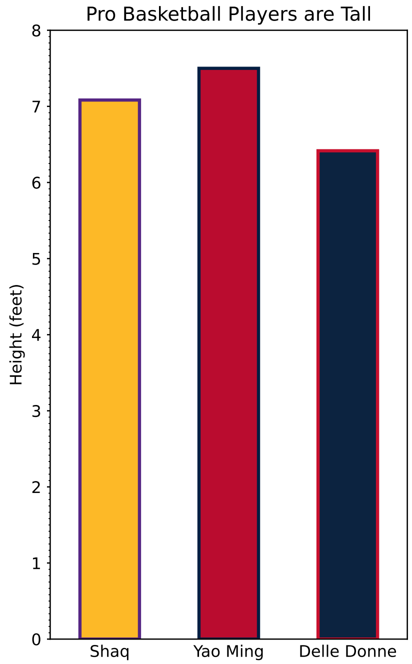
To remove the ticks entirely, simply pass an empty array to `set_ticks()`. To customize the appearance of your axis ticks and the labels, use the `set_tick_params` axis method. Parameters include `direction`, `width`, `length`, `color`, `pad`, `rotation`, `labelsize`, `labelcolor`

Imagine a measuring ruler, with ticks for every inch and smaller ticks at smaller intervals. So far our ticks have lacked that level of depth, but in fact we can work with two tick levels in matplotlib, major and minor ticks. Minor ticks are not shown by default.

To start exploring these further customizations, you'll need to import additional formatters and or locators. For the below, you must import `MultipleLocator`, running `from matplotlib.ticker import MultipleLocator`.

```
heights = pd.Series( {'Shaq': 7 + (1/12),
                      'Yao Ming': 7.5,
                      'Delle Donne': 6 + (5/12)})

fig, ax = plt.figure(figsize = (4,7)), plt.axes()

heights.plot.bar(ax = ax,
        color = ['#FDB927', '#BA0C2F', '#0C2340'],
        edgecolor = ['#552583', '#041E42', '#C8102E'],
        linewidth = 2)
# https://teamcolorcodes.com/
# LA Lakers and Houston Rockets and DC Mystics

# Get rid of ticks on x-axis, rotate text
ax.xaxis.set_tick_params(length = 0, which = 'major',
                         rotation = 0)

ylim0, ylim1 = 0,8
ax.set_ylim([ylim0, ylim1])

ax.set_yticks(range(ylim0, ylim1+1))
#ax.yaxis.set_major_locator(MultipleLocator(1))

ax.yaxis.set_minor_locator(MultipleLocator(1/12))
ax.yaxis.set_tick_params(length = 1, which = 'minor')
ax.yaxis.set_tick_params(length = 2, which = 'major')

ax.set_ylabel("Height (feet)")
ax.set_title("Pro Basketball Players are Tall")
```

tall-ballers.py

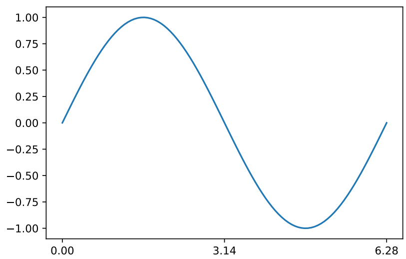


Major ticks can easily be set with `set_ticks` and its variants. Still, `MultipleLocator` and other locators are useful for setting major ticks without fooling with the details of the axes limits.

With a function like $\sin x$, ticks might most naturally be placed at multiples of π. This can be accomplished by the below.

```
x = np.linspace(0, np.pi * 2, 100)

fig, ax = plt.figure(), plt.axes()
ax.plot(x, np.sin(x))

ax.xaxis.set_major_locator(MultipleLocator(np.pi))
```

mult-locator.py

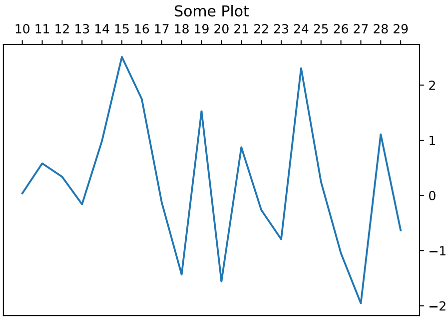


It's true you could avoid the complication of locator classes by just using `ax.set_xticks([0, np.pi, 2*np.pi])`. For a plot this simple, do that. Suppose, you put ticks up to 3π though. Then you've extended the x-axis limit of the plot past your data. So you need to know your data to make the right tick adjustments by hand. If you'll be using the same code with different datasets, it'll be easier to use the details-free `MultipleLocator` and you can still rely on limit defaults or adjust them independently.

Next, you might want to change the positioning of the ticks. By default x-axis ticks are on the bottom and y-axis ticks are on the left. You can modify these positions with axis methods. In time series data, for example, you might prefer to have the y-axis ticks on the right. Time marches on to the right and placing your ticks on the right can help emphasize that movement. This can be done with `set_ticks_position('right')` or the more concise `tick_right()`. The latter also accepts arguments of `'left'`, `'bottom'`, and `'top'`. Each has an abbreviated method like `tick_left()`.

```
fig, ax = plt.figure(), plt.axes()
x = np.arange(10, 30, 1)
y = np.random.normal(size = len(x))
ax.plot(x,y)

# set what ticks are shown
ax.xaxis.set_ticks(x)

# move the ticks
ax.yaxis.tick_right()
ax.xaxis.set_ticks_position('top')

ax.set_title("Some Plot")
```

tick-right.py

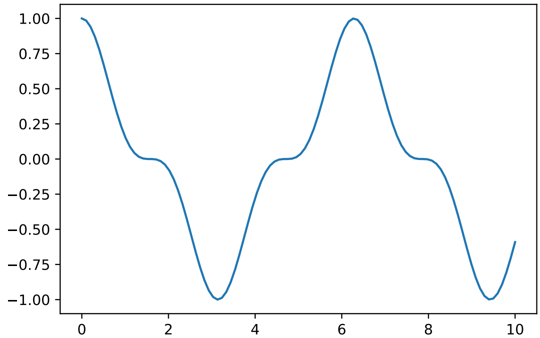


2.4 Grids

Including gridlines in a plot is generally discouraged (Knaflic 2015, Schwabish 2021). It's clutter that won't spark joy. Perhaps we could stop here, with the instruction to run `ax.grid(False)` as in the code below (or rely on a style, like the default, that does this automatically).

```
fig, ax = plt.figure(), plt.axes()
x = np.linspace(0,10,100)
ax.plot(x, np.cos(x)**3)
ax.grid(False)
```

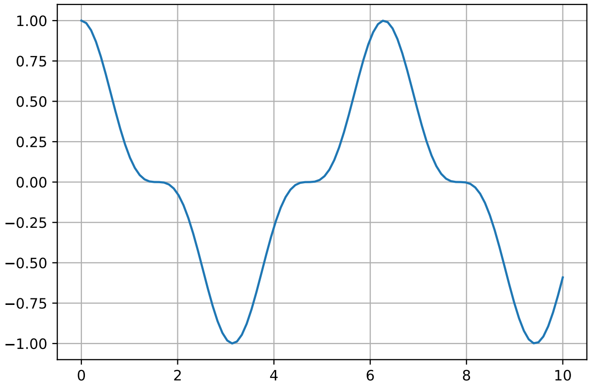
grid-false.py



This does seem preferable to the following, but it's hardly an abomination.

```
fig, ax = plt.figure(), plt.axes()
x = np.linspace(0,10,100)
ax.plot(x, np.cos(x)**3)
ax.grid(True)
```

grid-true.py



As a compromise, you might include gridlines for a single axis. If you want to emphasize that there is a slight trend in the data, then y-axis gridlines can help bring that pattern to the eye. Below we plot plots with and without a line of best fit and gridlines. Axis gridlines can be toggled independently by using `ax.xaxis.grid()` and `ax.yaxis.grid()`.

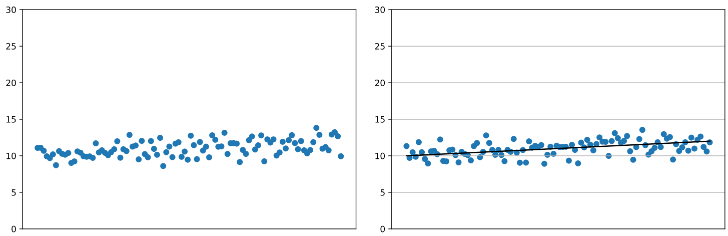
```
fig, ax = plt.figure(), plt.axes()

x = np.linspace(0, 10, 100)
y = 10 + .2*x
points = y + np.random.normal(size = len(x))
ax.scatter(x,points)

ax.set_ylim(0,30)
ax.set_xticks([])
```

y-grid-false.py

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

x = np.linspace(0,10, 100)
y = 10 + .2*x
points = y + np.random.normal(size = len(x))
ax.scatter(x,points)

```

```
ax.set_ylim(0,30)
ax.set_xticks([])

# Add grid and line of best fit
ax.yaxis.grid(True)
ax.plot(x, y, color = 'black')
```

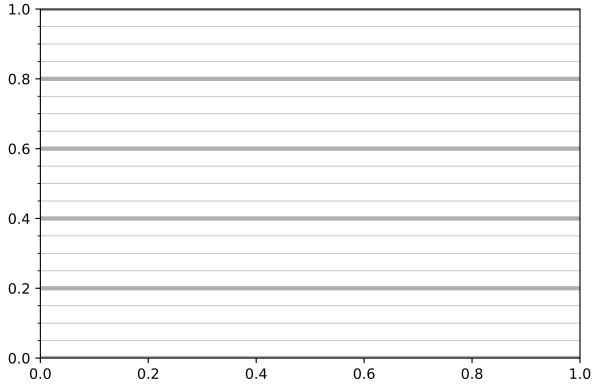
y-grid-true.py



What we learned previously about locating ticks in Section 2.3 can be reapplied here, as seen in the examples further below. The location of gridlines and ticks can be set by the `set_major_locator()` and `set_minor_locator()` methods. `ax.grid()` is used to display the gridlines, but note it features a parameter `which`. The default value of `which` is `'major'`. To include minor gridlines, those minor ticks must be explicitly created (at least in the default style) and then the gridlines must be toggled on with `ax.grid(True, which = 'minor')` or for a single axis with `ax.xaxis.grid(True, which = 'minor')` for example.

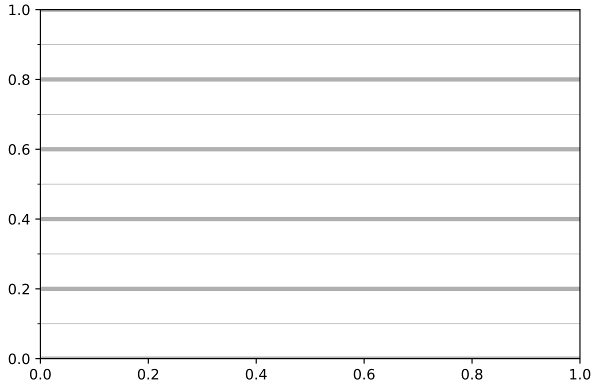
```
fig, ax = plt.figure(), plt.axes()
ax.xaxis.grid(False)
ax.yaxis.grid(True, linewidth = 3)
ax.yaxis.grid(True, which = 'minor', linewidth = 0.5)
ax.yaxis.set_minor_locator(mpl.ticker.AutoMinorLocator()
    )
```

grids-auto.py




```
fig, ax = plt.figure(), plt.axes()
ax.xaxis.grid(False)
ax.yaxis.grid(True, linewidth = 3)
ax.yaxis.grid(True, which = 'minor', linewidth = 0.5)
ax.yaxis.set_minor_locator(mpl.ticker.MultipleLocator
    (.1))
```

grids-multi.py



Chapter 3

Plot Elements and Coordinate Systems

This chapter can be skipped by the reader in a hurry. I include it to establish some vocabulary about the basic plot elements and then discuss the different coordinate systems that can be used within a single plot—not polar vs. Cartesian coordinates but data coordinates vs. figure coordinates, for example. Coordinate systems do come up repeatedly in future chapters.

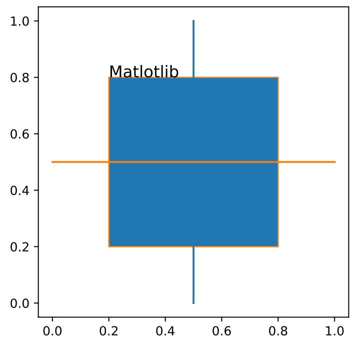
3.1 Primitives and Containers

Once you have a your figure and axis objects, you'll want to add actual plot elements to them, lines for a line chart, bars for a bar chart, annotations, etc. We already did that in Chapter 1, creating line plots. In matplotlib, these elements belong to the Artist class, it being a very general base class. Artists objects are basically the water you've been swimming in this whole time—you just might not have noticed it. Artist objects can be either primitives or containers. Containers include background items like the figure and axes objects. Primitives are the meat of the plot, like the line created by a call to `ax.plot()`. Important primitive Artist objects include Line2D, Patches, and Text.

```
fig, ax = plt.figure(), plt.axes()
ax.set_aspect(1)

# Patches
rect = plt.Rectangle(xy = (0.2, 0.2),
```

```
                          width = 0.6,
                          height = .6,
                          facecolor = 'C0',
                          edgecolor = 'C1')
patch = ax.add_artist(rect)

# Lines
x, y = [0.5, 0.5], [0, 1]
line, = ax.plot(x, y)
lines = ax.plot(y,x)

# Text
text = ax.text(0.2, 0.8, 'Matlotlib', size = 13)
```

artists.py



What might be unusual in the above is that we don't simply run `ax.plot(x, y)`. Instead we actually assign the plot call to a variable, `line, = ax.plot(x,y)`. Usually, this isn't necessary, but this allows us to reference the same object later in the program. The plot method creates a tuple of Line2D objects. In this case, that tuple contains only one item and it is assigned to the variable `line`.

Now that we have the object as `line`, we can get properties or make changes. You can obtain the color with the `get_color()` method or change it with `set_color()`. You can even remove the plot element with `line.remove()`. These are all niche uses. However, we will later make use of `remove()` when iteratively centering text. We'll also use the `get_window_extent()` artist method frequently to help space objects in the plot.

3.1.1 Ordering with `zorder`

Default Ordering

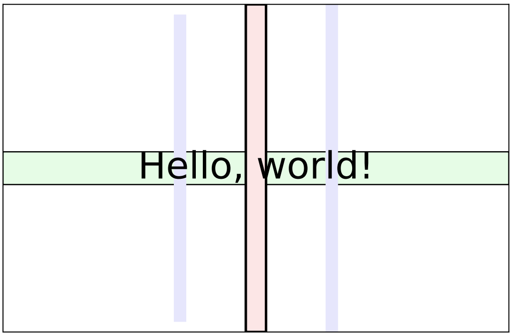
By default, text is plotted over lines and lines are plotted over patches, like the fill created by `fill_between()`. Within each of these three categories, objects created later in the program are plotted over previously created objects. The `zorder` parameter can be used to create a different ordering. Objects with a greater `zorder` value are ordered further to the front.

First, we create and plot without specifying the `zorder` for any object to observe default behavior. We also print the zorder for each object using `get_zorder()`. Text has a `zorder` of 3, lines have a `zorder` of 2, and each patch object will have `zorder = 1`. Note `patch1` and `patch2` have the same `zorder`, but the red `patch2` is added later in the program so it is plotted over the green `patch1`, being as if `patch1` has a lower `zorder`.

```
fig, ax = plt.figure(), plt.axes()
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xticks([])
ax.set_yticks([])

# make colors
green = (.9, .99, .9)
blue = (.9, .9, .99)
red = (.99, .9, .9)

# Text with default zorder of 3
text = ax.text(0.5, 0.5, "Hello, world!",
               size = 30,
               ha = 'center',
               va = 'center')

# Lines with default zorder of 2
line1 = ax.axvline(0.65,
                   linewidth = 10,
                   color = blue)
line2 = ax.plot([0.35, 0.35], [.05, .95],
                linewidth = 10,
                color = blue)

# Patches with default zorder of 1
patch1 = ax.fill_between([0,1], 0.45, .55,
                         facecolor = green,
                         edgecolor = 'black')
patch2 = ax.fill_between([.48,.52], 0, 1,
                         facecolor = red,
                         edgecolor = 'black',
```

```
                                linewidth = 2)

# Check zorders
print(text.get_zorder())
print(line1.get_zorder())
print(line2[0].get_zorder())
print(patch1.get_zorder())
print(patch2.get_zorder())
```

default-z.py

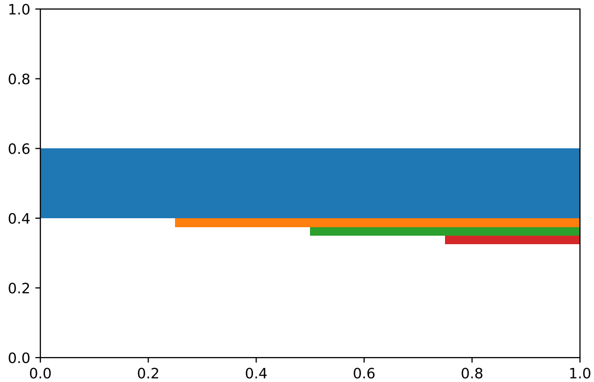


Custom Ordering

Then, we reverse the ordering.

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

print(fig.get_zorder())
print(ax.get_zorder())

for i in [0, 0.25, .5, .75]:

    t = ax.fill_between([i, 1], 0.4 - i/10, .6 - i/20,
                              zorder = 1 - i)
    print(t.get_zorder())

ax.set_xlim(0,1)
ax.set_ylim(0,1)
```

reverse-z.py

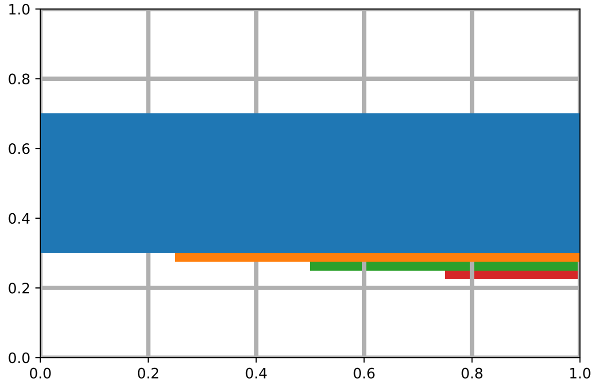


Axes and Tick Ordering

Notice that by default, gridlines are ordered below artists added to a plot regardless of where the call to show the gridlines is placed. This can be changed using `ax.set_axisbelow()`, which also reorders the ticks. The `XAxis` and YAxis can be ordered independently using the `set_order()` axis method.

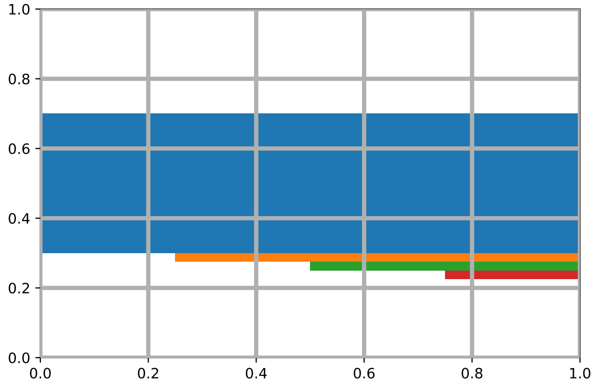
```
fig, ax = plt.figure(), plt.axes()
for i in [0, 0.25, .5, .75]:
    ax.fill_between([i,1], 0.3 - i/10, .7 - i/20,
                    zorder = 2-i)
ax.grid(True, linewidth = 3)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
print(ax.get_zorder())
```

default-axes.py




```python
fig, ax = plt.figure(), plt.axes()
for i in [0, 0.25, .5, .75]:
    ax.fill_between([i,1], 0.3 - i/10, .7 - i/20,
                         zorder = 2-i)
ax.grid(True, linewidth = 3)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_axisbelow(False)
print(ax.get_zorder())
```

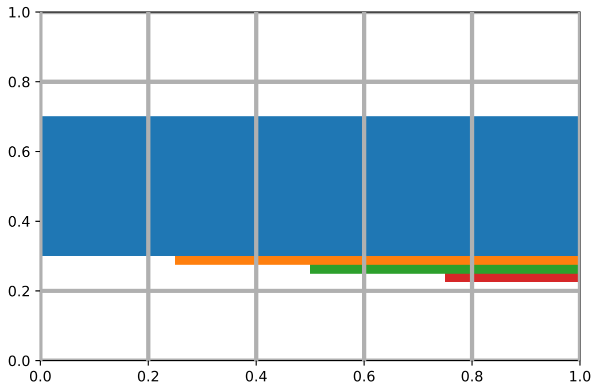
front-axes.py




```python
fig, ax = plt.figure(), plt.axes()
for i in [0, 0.25, .5, .75]:
    ax.fill_between([i,1], 0.3 - i/10, .7 - i/20,
                         zorder = 2-i)
```

```
ax.grid(True, linewidth = 3)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.xaxis.set_zorder(3)
```

front-xaxis.py



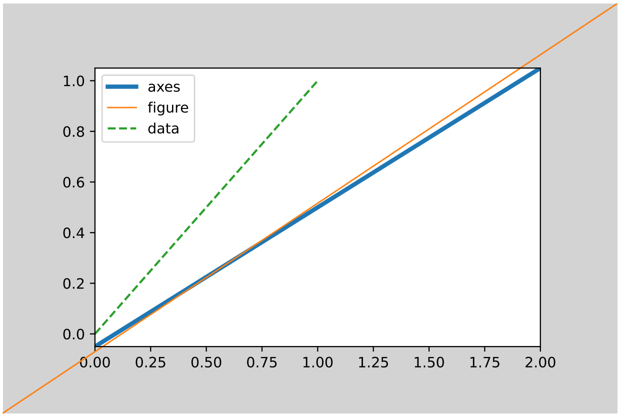
3.2 Coordinate Systems and Transformations

So far we have worked with data coordinates and you might not even realize there could be anything else. When we plotted a line between the points $(0,0)$ and $(1,1)$, we meant those as values in the usual xy-plane. But with use of transformations, we might also plot according to axes, figure, and display coordinates. In axes coordinates, $(0,0)$ is the bottom left of the axes and $(1,1)$ is the top right. Similarly, in figure coordinates, $(0,0)$ is the bottom left of the figure and $(1,1)$ is the top right. We won't cover the fourth type, display coordinates, which is the pixel coordinate system (for certain backends). The matplotlib documentation cautions that you should rarely work with display coordinates. However, display coordinates are a necessary evil when converting from one system to another. Note, it is important not to manipulate the figure or axes dimensions after referencing the display coordinate system or you might encounter unexpected behavior.

The plot below features a group of plot calls using axes coordinates, then a group using figure coordinates, and then a single call using data coordinates.

```
fig, ax = plt.figure(facecolor = 'lightgray'), plt.axes()

ax.plot([0, 1], [0, 1],
        linewidth = 3,
        transform = ax.transAxes,
        label = 'axes')

ax.plot([0, 1], [0, 1],
        color = 'C1',
        linewidth = 1,
        transform = fig.transFigure,
        clip_on = False,
        label = 'figure')

ax.plot([0, 1], [0, 1],
        color = 'C2',
        linestyle = 'dashed',
        clip_on = False,
        label = 'data')

ax.set_xlim(0,2)
ax.legend()
```

coords.py

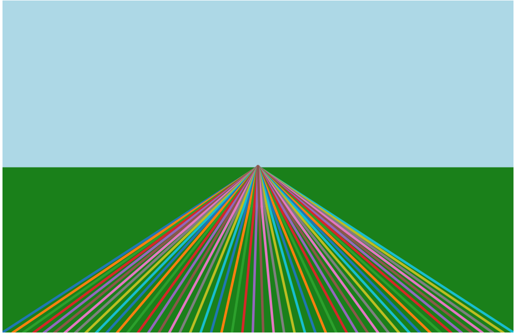


Axes and figure coordinates are often useful when you would like placement to be independent of the data, perhaps to enforce that something remain in the center of the plot by using an axes coordinate of 0.5. Below, we make use of that to set a vanishing point at the vertical halfway point.

```
fig, ax = plt.figure(), plt.axes()
ax.axis('off')
```

```
# lines to horizon
for i in np.linspace(0,1,50):
    ax.plot([i,.5], [0.00, .5],
            transform = ax.transAxes,
            linewidth = 2,
            zorder = 10-(i-0.5)**2)

# fill bottom half
green = (.1, .5, .1)
ax.fill_between(x = (0,1),
                y1 = 0,
                y2 = 0.5,
                transform = ax.transAxes,
                color = green)

# fill top half
ax.fill_between(x = (0,1),
                y1 = 0.5,
                y2 = 1,
                transform = ax.transAxes,
                color = 'lightblue')
```

coord-horizon.py

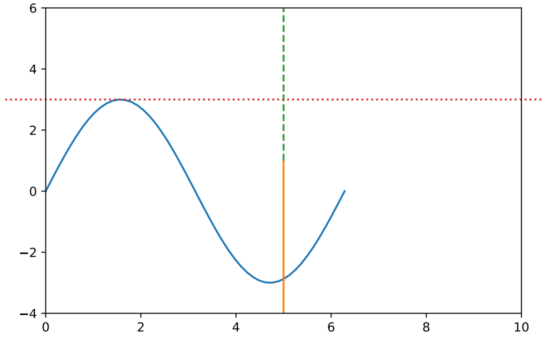


We can convert a point or sequence of points from one coordinate system to another using the appropriate transform object. `ax.transData.transform([x,y])` converts `x,y` from data coordinates to display coordinates. Simply replacing `ax.transData` with `ax.transAxes` or `fig.transFigure` converts from the corresponding coordinate system to display coordinates. The opposite direction is achieved by inverting the transformation—`ax.transData.inverted().transform([x,y])`. To go from data coordinates to figure or axes coordinates, you can make a pit stop in display coordinates. For example, `ax.transData.inverted().transform(ax.transAxes.transform([0.5, 0.5]))` returns the middle of the axes window in data coordinates.

The example below breaks this up into two steps. Again, take note that all plotting is done after setting a tight layout and after setting the axes limits to avoid resizing the figure and endangering the reliability of our coordinate transformations.

```python
# Plot setup
fig, ax = plt.figure(), plt.axes()
x = np.linspace(0, 2*np.pi)
sin, = ax.plot(x, 3*np.sin(x))
ax.set_xlim(0, 10)
ax.set_ylim(-4, 6)
fig.tight_layout()

# Vertical line with axes coordinates
middle = [0.5, 0.5]
bottom_half = [0, 0.5]
ax.plot(middle, bottom_half,
        transform = ax.transAxes)

# Continue vertical line with data coordinates
mid_in_display = ax.transAxes.transform([0.5, 0.5])
mid_in_data = ax.transData.inverted().transform(
    mid_in_display)
top_mid_in_display = ax.transAxes.transform([0.5, 1])
top_mid_in_data = ax.transData.inverted()\
                        .transform(top_mid_in_display)
x = mid_in_data[0], top_mid_in_data[0]
y = mid_in_data[1], top_mid_in_data[1]
ax.plot(x, y, linestyle = 'dashed')

# Horizontal lines in figure coordinates
top_wave_display = ax.transData.transform([np.pi/2, 3])
top_wave_figure = fig.transFigure.inverted()\
                        .transform(top_wave_display)

y = top_wave_figure[1],  top_wave_figure[1]
ax.plot([0,1], y,
        transform = fig.transFigure,
        linestyle = 'dotted',
        clip_on = False)
```

coord-trans.py



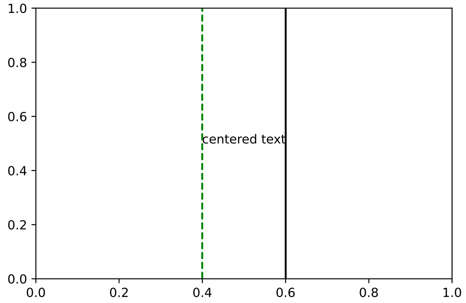
3.3 Use Window Extents

Another useful method is `get_window_extent()`, which allows you to find the bounding box (the coordinates for the corners of the enclosing rectangle) for something added to a plot. This can be used to find the display coordinates for where an annotation begins or ends, for example. Like in the previous section, note that the results will not update and be inaccurate if changes are made to the figure size, axes limits, or the canvas used. The method also requires a renderer. The technicalities for why can be put aside. Either include `fig.canvas.draw()` first, so the rendered is already cached, or include the argument `renderer = fig.canvas.get_renderer()` in the call to `get_window_extent()`. Below is a simple example. We create a text object with the axes method `ax.text()` in the normal way, but we take the atypical step of assigning the object to a variable. Below, that variable is named `center_text` and then we call `get_window_extent()` as a Text method, or an Artist method more abstractly.

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

center_text = ax.text(0.5, 0.5,
                      'centered text',
                      ha = 'center')

fig.canvas.draw()
box = center_text.get_window_extent()
data_box = ax.transData.inverted().transform(box)

# left limit
ax.axvline(data_box[0][0],
           color = 'green',
```

```
            linestyle = 'dashed')

# right limit
ax.axvline(data_box[1][0],
            color = 'black')
```

window-extent.py

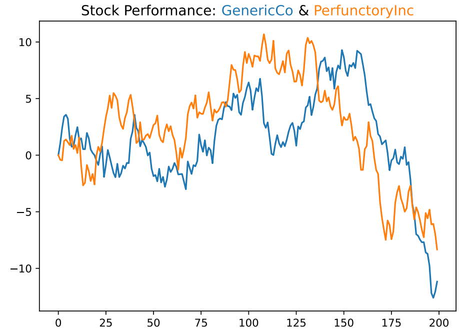


So what? A formatted title can stand in for a legend, helping reduce clutter. This helps us heed the call from Schwabish 2021 to label data directly and avoid legends when possible. In the line chart below, a legend is unnecessary given the color-coding in the title. We create a title not with the typical `ax.set_title()` but with a series of `ax.text()` calls. There are several because a single Text object can't have multiple colors. The `ha` parameter is for horizontal alignment, and this is covered in more detail in a later chapter. By using `ha = 'left'`, the text will begin at the given x and y coordinates.

```
x_len = 200
x = range(0, x_len)

# Create a Gaussian random walk starting at 0
start = np.zeros(1)
y1 = np.concatenate([start,np.random.normal(0, 1, x_len
    -1)]).cumsum()
y2 = np.concatenate([start,np.random.normal(0, 1, x_len
    -1)]).cumsum()

# Start plot
fig, ax = plt.figure(figsize = (7,5)), plt.axes()
fig.canvas.draw()

# Color arguments added to make defaults explicit
ax.plot(x, y1, color = 'C0')
```

```
ax.plot(x, y2, color = 'C1')

# Tuned by hand
shift = .099814 # Where titling starts on x-axis
y_level = 1.02
transform = ax.transAxes # use axes coords

t1 = ax.text(shift, y_level, 'Stock Performance:',
        transform = transform,
        ha = 'left',
        fontsize = 13,
        color = 'black')

# Get where text ended
x_pos = t1.get_window_extent()\
       .transformed(transform.inverted()).x1

t2 = ax.text(x_pos, y_level, ' GenericCo',
        transform = transform,
        ha = 'left',
        fontsize = 13,
        color = 'C0')

x_pos = t2.get_window_extent()\
       .transformed(transform.inverted()).x1

t3 = ax.text(x_pos, y_level, ' &',
        transform = transform,
        ha = 'left',
        fontsize = 13,
        color = 'black')

x_pos = t3.get_window_extent()\
       .transformed(transform.inverted()).x1

t4 = ax.text(x_pos, y_level, ' PerfunctoryInc',
        transform = transform,
        ha = 'left',
        fontsize = 13,
        color = 'C1')

x_pos = t4.get_window_extent()\
       .transformed(transform.inverted()).x1

# compare distances to the edge
# equal means perfect centering
print(shift, 1-x_pos)
```

multicolor-title.py


Stock Performance: GenericCo & PerfunctoryInc
10
5
0
−5
−10
0
25
50
75
100
125
150
175
200

Chapter 4

Text and Titles

4.1 Simple Titles

As we learned in Chapter 1, we can add a title with the axes method `set_title()`. Simply pass the string of your choice as the argument. For multi-line titles, recall `\n` can be used in a string to start a new line. Common optional arguments include `color`, `fontsize`, `weight`, and `loc`.

Colors will be addressed in Chapter ??, but to start you can simply use the name of any not-too-exotic color as a string.

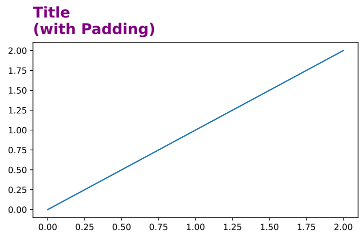
`fontsize` (or `size`) can be a number or chosen from `'small'`, `'medium'`, or `'large'`, and `'small'` and `'large'` may be intensified with a `'x-'` or `'xx-'` prefix. Similarly, `weight` (or `fontweight`) can be a number or chosen from options like `'bold'` or `'light'`.

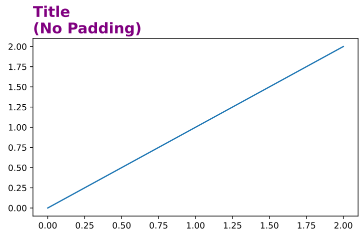
`loc` determines the location of the title, either `'left'`, `'center'`, or `'right'`. In the default style, the default value will be `'center'`. You might prefer using `'left'` to match the Google Sheets default (thus matching the vast majority of plots I've seen in industry). `pad` controls the space between the title and the top of the axes.

```
x = np.linspace(0,2,2)
fig, ax = plt.figure(), plt.axes()

ax.plot(x,x)
ax.set_title("Title\n(with Padding)",
             fontsize = 'xx-large',
             weight = 'bold',
             color = 'purple',
             loc = 'left',
```

```
                pad = 10)
```

title-pad.py

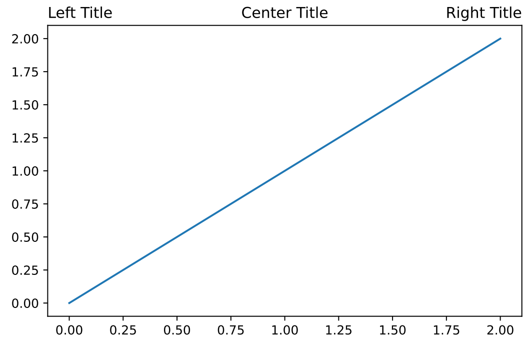




A plot can actually have one title for every `loc` value as well.

```
x = np.linspace(0,2,2)
fig, ax = plt.figure(), plt.axes()

ax.plot(x,x)
ax.set_title("Left Title",
                loc = 'left')
ax.set_title("Right Title",
                loc = 'right')
ax.set_title("I won't be long for this world.",
                loc = 'center')

# This only overwrites the center title above
ax.set_title("Center Title")
```

title-loc.py

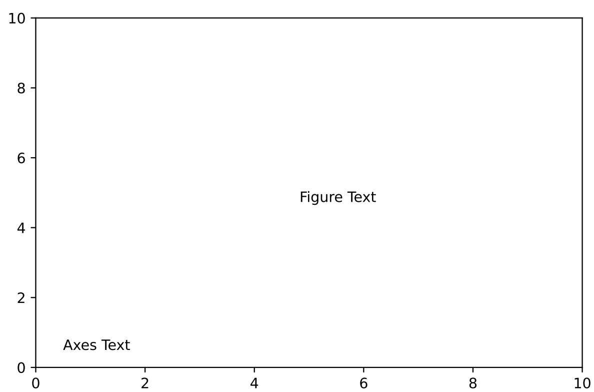


4.2 Text and Placement

Matplotlib offers `text` as both a figure and an axes method. Let's start with some code to understand what they do. Both take x and y positions as the first two arguments and then a string. The figure method method is the same as using the axes method with a transformation to figure coordinates.

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

ax.set_xlim([0,10])
ax.set_ylim([0,10])

fig.text(0.5, 0.5, 'Figure Text')
ax.text(0.5, 0.5, 'Axes Text')
```

text-methods.py



Immediately, we see that despite passing the same x and y position values, the figure and axes methods place the text differently. By default, the figure method uses "figure" coordinates, where (0,0) is the bottom left and (1,1) is the top right. The axes method uses x and y data coordinates by default. We will modify this shortly.

A more common concern is the alignment of the text. Both figure and axes text methods include parameters `verticalalignment` and `horizontalalignment`, which can be abbreviated as `va` and `ha`. By default, the text is placed so that the given coordinate is at the bottom-left corner of the text.

```
fig, ax = plt.figure(), plt.axes()
x, y = 0.5, 0.5
ax.scatter([x], [y])
ax.text(x,y, 'text', fontsize = 20)
ax.axis('off')
```

text-default-align.p

For vertical alignment, the options are `'top'`, `'bottom'`, or `'center'`. For horizontal alignment, the options are `'left'`, `'right'`, or `'center'`. The default demonstrated above was `'bottom'` and `'left'`. It does result in the text being above and to the right of the coordinate point, perhaps confusingly, but the interpretation is that the coordinate point is at the bottom-left of the text. The possible alignments are illustrated below.

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

x1, x2, y = 0.49, 0.51, 0.5
ax.scatter([x1,x2], [y,y])

va_options = ['top', 'bottom', 'center']
ha_options = ['left', 'right', 'center']

counter = 0 # for color cycling
for va in va_options:
```

```
    for ha in ha_options:
        # first letter of each option
        label = va[0] + "-" + ha[0]

        # assign label to point
        x = x1
        if 'c' in label:
            x = x2

        ax.text(x, y,
                label,
                va = va,
                ha = ha,
                fontsize = 20,
                color = 'C'+str(counter))
        counter += 1

ax.axis('off')
```

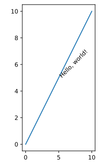
text-align.py

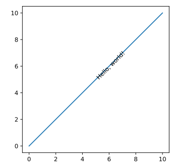
Text can be rotated with the `rotation` parameter. By default, a plot isn't square—the aspect ratio (the ratio of y-unit to x-unit) is not one. That means that the 45 degree line created by $y = x$ is not actually plotted at 45 degrees. Yet according to the `rotation` parameter, text rotated at 45 degrees is plotted at 45 degrees—that angle is not converted based on the aspect ratio. Later in Section **??**, I go into further detail in how to use some trigonometry to get the exact angle if you'd like to slope text at some angle, accounting for the aspect ratio.

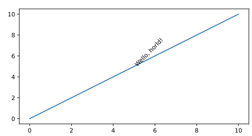
```
x = np.linspace(0,10)
fig, ax = plt.figure(), plt.axes()
ax.plot(x,x)
ax.set_aspect(2)
```

```
ax.text(5,5,
        'Hello, world!',
        rotation = 45)
```

text-rotation2.py







4.2.1 Text Formatting for Numbers

Here I've tucked away a subsection on formatting numbers in Python. This has nothing to do with matplotlib, formally speaking. Still, sometimes you want your text annotations or titles to contain numbers formatted just so and you'll want Python to figure that out instead of doing it by hand. You might want commas as the thousands separator (the more readable 1,000,000 instead of 1000000), you might want leading zeros (01 instead of 1), or you might want a currency symbol ($2 instead of 2). The table below demonstrates by example how to do this with `str.format`.

Code	Output
`'{:,}'.format(10**6)`	`'1,000,000'`
`'${:,.2f}'.format(10**6)`	`'$1,000,000.00'`
`'{:0>3.0f}'.format(1)`	`'001'`
`'{:>3.0f}'.format(1)`	`'  1'`
`'${:0>4.0f}'.format(1)`	`'$0001'`
`'{:+,.1f}'.format(1000)`	`'+1,000.0'`
`'{:0<+4,.1f}'.format(-1)`	`'-1.0'`
`'{:0<5.0f}'.format(1)`	`'10000'`
`'{:0<5,.0f}'.format(1)`	`'10000'`
`'{:0<8,.0f}'.format(1000)`	`'1,000000'`
`'{:.0e}'.format(10.1**6)`	`'1e+06'`
`'{:.1f} and {:.1f}'.format(9, 1)`	`'9.0 and 1.0'`
`'{1:.1f} and {0:.1f}'.format(9, 1)`	`'1.0 and 9.0'`
`'{0:} and {0}'.format(1)`	`'1 and 1'`
`'{:} and {:}'.format(1)`	`IndexError`

Understanding everything above requires some knowledge of format specifications. A format specifier is a string that can specify fill, align, sign, width, grouping option, precision, and type (`[[fill]align][sign][#][0][width][grouping_option][.precision][type]`. These must be properly ordered but anything can be omitted to accept the default. These arguments go inside curly braces and to the right of a colon, `{:}`. The curly braces tell Python where to place the argument you pass to the `format()` method. You can also pass multiple arguments inside `format()`. By default, they are placed in order (the first argument replaces the first `{}` and so on), but to the left of the colon, you can also specify the index value for the argument to use.

The *fill* is a character that can be used to pad the number. Used with a *align* and *width*, we can add leading zeros. The default is a space if no fill character is provided. Using `'0>4'`, this will create leading zeros (right-aligned) up to a width of 4. So `1` becomes `'0001'` and `10000` is not padded, being simply `'10000'`.

The *grouping option* would come next, allowing for a thousands separator of a comma or an underscore. `'{:,}'.format(10000)` produces `'10,000'`. Note that when used with padded numerals on the right, the padding is ignored in finding the thousands separators, so `'{:0<8,.0f}'.format(1000)` produces the confusing `'1,000000'`.

Precision is next with a decimal and then how many digits to display past the decimal place or before and after, depending on the lastly specified *type*. Observe `'{:.2}'.format(np.pi)` produces `'`

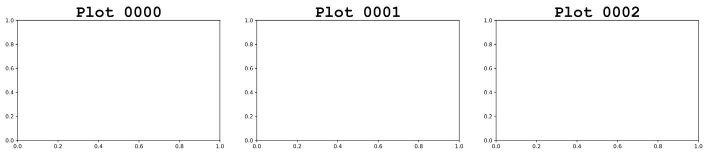
3.1' and '{:.2f}'.format(np.pi) produces '3.14'. You'll want type 'f' for a float. Use 'e' for scientific notation. You may read up on the many other types, including locale aware types, in the Python documentation.[1]

Whatever we put outside the curly braces is simply concatenated to the text on the left or right. So '${}'.format(123) turns 123 into the dollar figure '$1231'. And '{} lbs.'.format(123) would produce '1231 lbs'.

Perhaps this will come in handy when you'd like figure text or the filename in a certain format. I often use leading zeros in some filenames so that alphabetically ordering the files will be coherent (your file system will likely maintain '1' < '10' < '2'). If you are creating many plots that will be frames in an animation, and you'll have some number ticking up as the frames progress, the padding might help the eye.

```
for i in range(3):
    fig, ax = plt.figure(), plt.axes()
    label = '{:0>4}'.format(i)
    ax.set_title("Plot " + label,
                 fontname = 'Courier New',
                 weight = 'bold',
                 fontsize = 30)
    fig.tight_layout()
    fig.savefig(label + ".pdf")
    plt.show()
```

text-formatting.py



4.3 Legends

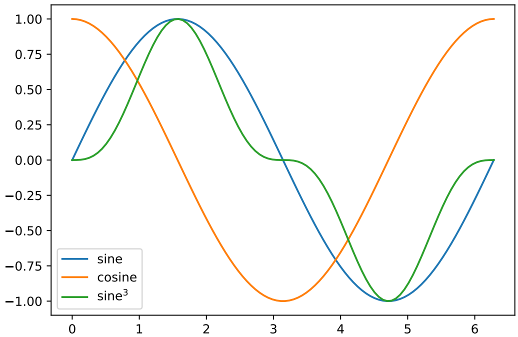
As you should know, legends provide a key to the colors and symbols used in a plot. You can create a legend with legend(), as either a figure or axes method. Without any extra customization this is done with ax.legend() or fig.legend(). Here, we will only cover axes legends. We'll return to figure legends when they are more naturally useful in Chapter ?? on multiple axes and multiple plots.

[1] https://docs.python.org/3/library/string.html#grammar-token-type

But first, you need labels for your plot elements (called *artist* objects) before you can create a legend. This can be done with the `label` parameter in methods like `plot()`. Or you can use `set_label()` on the plot element object. Using `set_label()` adds some complication to the code, as seen below in an otherwise simple example. Note the legend needs to be added after the labeled plot elements you want included in the legend.

```
fig, ax = plt.figure(), plt.axes()
x = np.linspace(0,2*np.pi,100)

# Label in one go
ax.plot(x, np.sin(x), label = 'sine')

# Label as Artist method
cos, = ax.plot(x, np.cos(x))
cos.set_label('cosine')

# Label as Artist method
sine3 = ax.plot(x, np.sin(x)**3)
sine3[0].set_label(r'sine$^3$')

# Construct legend
ax.legend()
```

legend-labels.py

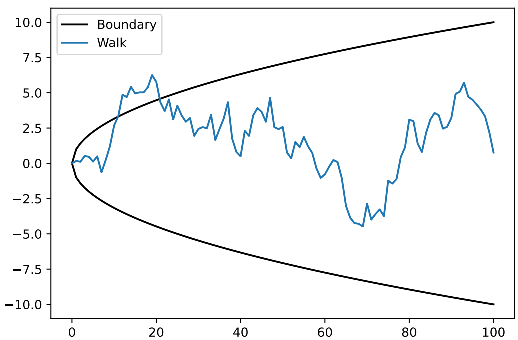


If you are using a pandas plot method, the labels will be set automatically according to the column or series names. For such instances where an element is automatically included in a legend and you want to exclude it, you can exclude that element by specifying `label = '_nolegend_'` in the plot call.

```
# Construct DataFrame
n = 100
```

```
sqrts = np.concatenate([np.zeros(1),np.ones(n).cumsum()**0.5] )
ser1 = pd.Series(data = -sqrts, name = 'Lower Bound')
ser2 = pd.Series(data = sqrts, name = 'Upper Bound')
df = pd.DataFrame([ser1,ser2]).T

# Add random walk
df['Walk'] = np.concatenate([np.zeros(1),np.random.normal(size = n).cumsum()])

# Plot
fig, ax = plt.subplots()
df['Lower Bound'].plot(color = 'black', label = 'Boundary')
df['Upper Bound'].plot(color = 'black', label = '_nolegend_')
df['Walk'].plot()

ax.legend()
```

pd-legend.py



A more common concern might be how to customize the placement of the legend and its actual appearance.

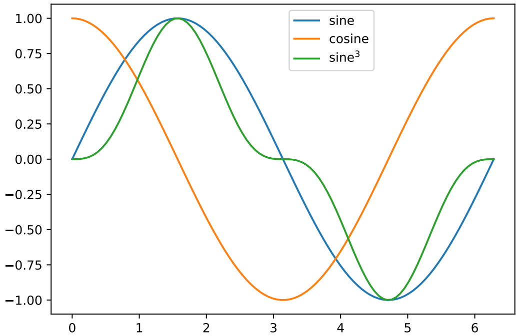
To change the placement of the legend, you may use the `loc` parameter. The default value is `'best'`, where best is determined by matplotlib. Other valid values are `'center'` and `'right'` (but not `'left'`) and then modifications like `'upper center'`, `'center right'`, and `'lower left'`.

For further customization of the placement, use the `bbox_to_anchor` parameter. This accepts 2-tuple or 4-tuple, giving the x location, the y location, and the width and height optionally.

By default, x and y are in axes coordinates. So the program below places a legend in the top and center of the axes. The alignment is done accodring to `loc`. If, for example, `loc = 'lower right'`, then the lower right corner of the legend is placed at the specified x and y.

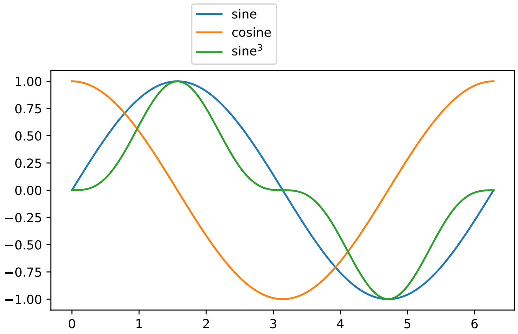
```
fig, ax = plt.figure(), plt.axes()
x = np.linspace(0,2*np.pi,100)
ax.plot(x, np.sin(x), label = 'sine')
ax.plot(x, np.cos(x), label = 'cosine')
ax.plot(x, np.sin(x)**3, label = r'sine$^3$')

# Construct legend
ax.legend(bbox_to_anchor = (0.5,1))
```

legend-bb.py




```
fig, ax = plt.figure(), plt.axes()
x = np.linspace(0,2*np.pi,100)
ax.plot(x, np.sin(x), label = 'sine')
ax.plot(x, np.cos(x), label = 'cosine')
ax.plot(x, np.sin(x)**3, label = r'sine$^3$')

# Construct legend
ax.legend(bbox_to_anchor = (0.5,1),
          loc = 'lower right')
```

legend-bb-loc.py

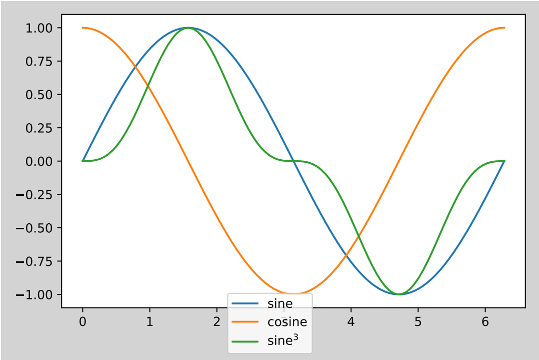


If using a 4-tuple, the tuple is interpreted as the plot region in which to put the legend, according to `loc`.

Use `bbox_transform` to use a coordinate system other than the default axes coordinates.

```
fig, ax = plt.figure(facecolor = 'lightgray'), plt.axes()
x = np.linspace(0,2*np.pi,100)
ax.plot(x, np.sin(x), label = 'sine')
ax.plot(x, np.cos(x), label = 'cosine')
ax.plot(x, np.sin(x)**3, label = r'sine$^3$')

# Construct legend
ax.legend(bbox_to_anchor = (0.5,0),
          loc = 'lower center',
          bbox_transform = fig.transFigure)
```

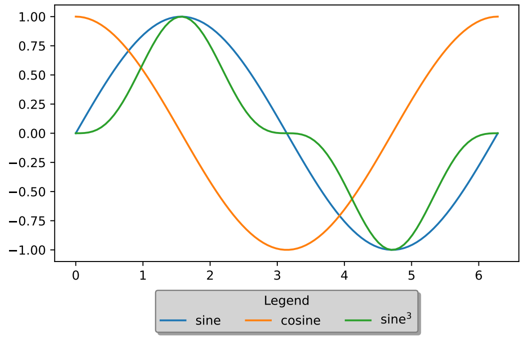
legend-transform.py



There are many parameters to change the appearance of the legend. We won't cover all of them. Two useful parameters are `facecolor` and `ncol`. The former changes the background color of the legend and the latter sets the number of columns, changing the default shape of the legend. I use these and a few other self-explanatory parameters in the program below.

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

x = np.linspace(0,2*np.pi,100)

ax.plot(x, np.sin(x), label = 'sine')
ax.plot(x, np.cos(x), label = 'cosine')
ax.plot(x, np.sin(x)**3, label = r'sine$^3$')

# Construct legend
ax.legend(bbox_to_anchor = (0.5,-0.3),
          loc = 'lower center',
          ncol = 3,
          facecolor = 'lightgray',
          edgecolor = 'gray',
          shadow = True,
          title = 'Legend')
```

legend-shape.py



4.4 Annotations

Knaflic 2015 and Schwabish 2021 both advise to label data directly and to annotate graphs with explanatory notes when helpful, as this helps convey the meaning of the graph more simply and directly.

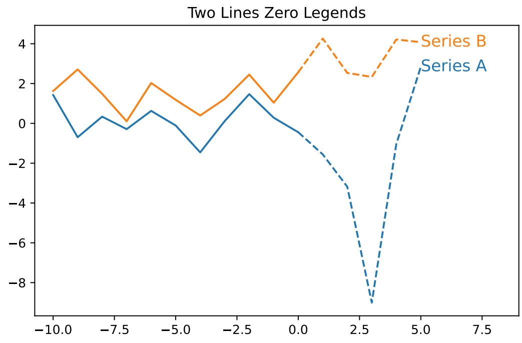
You can annotate a chart with `text()` method calls, or you can use the `annotate()` method, for which you specify the text placement and a line segment to the part of the graph the text references.

4.4.1 Labeling and Arrows

The following graph is nothing special, but we avoid having to create a legend by labeling the data with the text color matching the line color.

```
fig, ax = plt.figure(), plt.axes()
x = np.arange(-10,6,1)
past = x[x<=0]
future = x[x>=0]

y_historical = np.random.normal(0,1,size = len(past))
y_projected = np.concatenate([y_historical[-1:],
                    np.random.normal(0,3, size = len(future)-1)])

z_historical = np.random.normal(1,1,size = len(past))
z_projected = np.concatenate([z_historical[-1:],
                    np.random.normal(3,1, size = len(future)-1)])

ax.plot(past, y_historical)
ax.plot(future, y_projected, linestyle = 'dashed', color = 'C0')

ax.plot(past, z_historical, color = 'C1')
ax.plot(future, z_projected, linestyle = 'dashed', color = 'C1')

# Label Data
ax.text(future[-1], y_projected[-1],
        s = 'Series A',
        va = 'center',
        color = 'C0',
        size = 13)
ax.text(future[-1], z_projected[-1],
        s = 'Series B',
        va = 'center',
        color = 'C1',
        size = 13)

ax.set_xlim(ax.get_xlim()[0], 9)
ax.set_title("Two Lines Zero Legends")
```

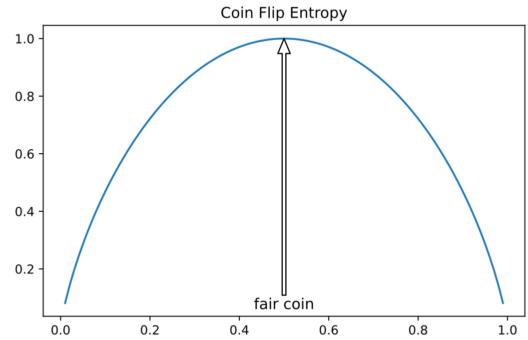
label-data.py



Next, we use the `annotate()` method. This method comes with the option to include an arrow pointing from `xytext` to the point `xy`.

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

x = np.linspace(0,1,100) # Pr(heads)
x = x[(x!=0) & (x!=1)]
entropy = -x*np.log2(x) - (1-x)*np.log2(1-x)
ax.plot(x,entropy)
ax.annotate('fair coin',
            xy = (0.5,1),
            xytext = (0.5, 0.1),
            arrowprops=dict(facecolor='white',
                            edgecolor = 'black',
                            width = 3,
                            headwidth = 10,
                            linewidth = 1),
           ha = 'center',
           va= 'top', # text alignment around xytext
           size = 12)

ax.set_title("Coin Flip Entropy")
```

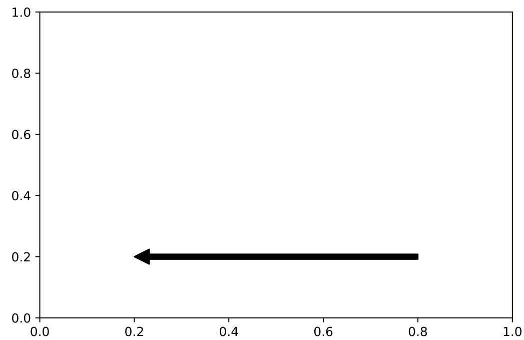
annotate-arrow.py



If you would like an arrow and no text, simply use the empty string ''. It is necessary to pass a dictionary to the `arrowprops` property.

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

# no arrow, no text
# this does nothing
ax.annotate('',
            xy = (0.1, 0.8),
            xytext = (0.9, 0.9))

# arrow
ax.annotate('', xy = (0.2, 0.2),
            xytext = (0.8, 0.2),
            arrowprops = dict(color = 'black'))
```

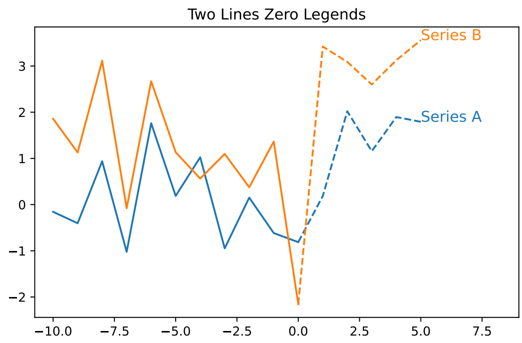
arrow-only.py



Lastly, one can also reference specific artist objects in the annotation instead of coordinates. In the below we place the annotations at the end of `a_line` and `b_line`.

```
fig, ax = plt.figure(), plt.axes()
x = np.arange(-10,6,1)
past = x[x<=0]
future = x[x>=0]
y_historical = np.random.normal(0,1,size = len(past))
y_projected = np.concatenate([y_historical[-1:],
                    np.random.normal(0,3, size = len(
    future)-1)])
z_historical = np.random.normal(1,1,size = len(past))
z_projected = np.concatenate([z_historical[-1:],
                    np.random.normal(3,1, size = len(
    future)-1)])
ax.plot(past, y_historical)
a_line, = ax.plot(future, y_projected, linestyle = '
    dashed', color = 'C0')
ax.plot(past, z_historical, color = 'C1')
b_line, = ax.plot(future, z_projected, linestyle = '
    dashed', color = 'C1')

# Label Data
ax.annotate('Series A',
            xy = (1, y_projected[-1]),
            xycoords = (a_line, 'data'),
            color = 'C0',
            size = 12)

ax.annotate('Series B',
            xy = (1, z_projected[-1]),
            xycoords = (b_line, 'data'),
            color = 'C1',
            size = 12)

ax.set_xlim(ax.get_xlim()[0], 9)
ax.set_title("Two Lines Zero Legends")
```

direct-annotation.py

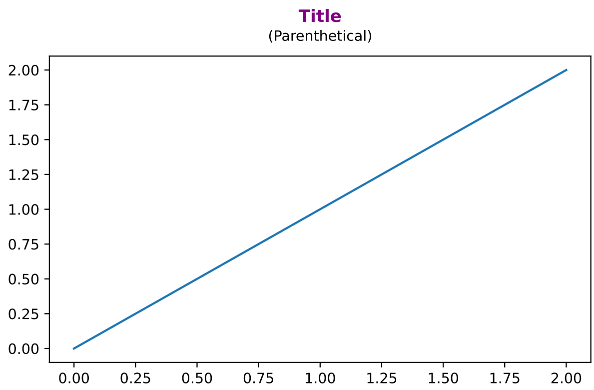


4.5 Fancy Titles

If you'd like to format different parts of the title different, you'll have to move beyond simply using `set_title`. The New York Times, for example, routinely includes a title and a subtitle in a plot. This requires using `text()` and `set_title()` separately, as there can only be one format style applied to a title. A simple example is below.

```
x = np.linspace(0,2,2)
fig, ax = plt.figure(), plt.axes()
ax.plot(x,x)

ax.set_title("Title",
             weight = 'bold',
             color = 'purple',
             pad = 24)

ax.text(0.5, 1.05,
        s = '(Parenthetical)',
        transform = ax.transAxes,
        ha = 'center')
```

subtitle.py

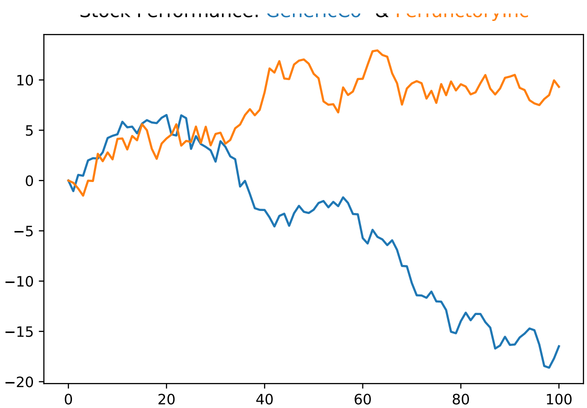


4.5.1 Multi-colored Titles

In Chapter 3, we created a multi-colored title using the Artist method `get_window_extent()`. The advantage of a multi-colored title is that we can do without a legend. For someone who doesn't want to get into the complications of `get_window_extent()`, the x and y placement of the text could be done by sight.

```
x = range(101)

# Create a Gaussian random walk starting at 0
start = np.zeros(1)
y1 = np.concatenate( [start,np.random.normal(0,1,100)] ).cumsum()
y2 = np.concatenate( [start,np.random.normal(0,1,100)] ).cumsum()

fig, ax = plt.figure(), plt.axes()
plt.tight_layout()
# Color arguments added to make defaults explicit
ax.plot(x,y1, color = 'C0')
ax.plot(x,y2, color = 'C1')

ax.text(0.4, 1.05, ' GenericCo',
        transform = ax.transAxes,
        ha = 'left',
        fontsize = 13,
        color = 'C0')

ax.text(0.4, 1.05, 'Stock Performance:',
        transform = ax.transAxes,
        ha = 'right',
        fontsize = 13,
        color = 'black')

ax.text(0.64, 1.05, '&',
        transform = ax.transAxes,
        ha = 'right',
        fontsize = 13,
        color = 'black')

ax.text(0.64, 1.05, ' PerfunctoryInc',
        transform = ax.transAxes,
        ha = 'left',
        fontsize = 13,
        color = 'C1')
```

multicolor-inexact.p



Greater elegance requires greater complication. If you are (understandably) dissatisfied with the above, invest in the topics covered in Chapter 3. Below, we build on the solution from Chapter `chapter:elements` by creating a function that creates a multi-colored title. Note we remove text options with the `remove()` method and work all in a single figure. This replaces the work of tuning the centering by hand that was done previously.

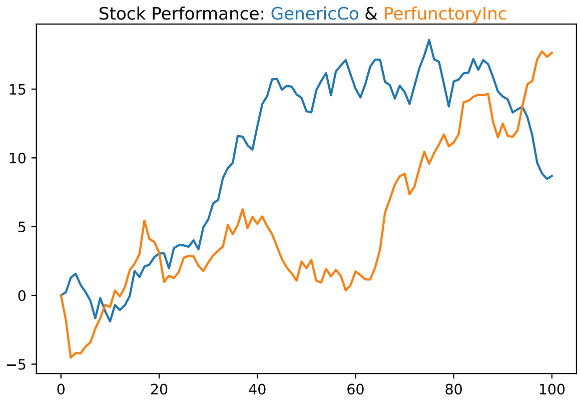
```
def color_title(labels, colors, textprops ={'size':'large'}, ax = None, y = 1.013,
                precision = 10**-2):

    "Creates a centered title with multiple colors. "

    if ax == None:
        ax = plt.gca()

    plt.gcf().canvas.draw()
    transform = ax.transAxes # use axes coords

    # initial params
    xT = 0 # where the text ends in x-axis coords
    shift = 0 # where the text starts

    # for text objects
    text = dict()

    while (np.abs(shift - (1-xT)) > precision) and (shift <= xT) :
        x_pos = shift

        for label, col in zip(labels, colors):

            try:
                text[label].remove()
```

```
            except KeyError:
                pass

            text[label] = ax.text(x_pos, y, label,
                        transform = transform,
                        ha = 'left',
                        color = col,
                        **textprops)

            x_pos = text[label].get_window_extent()\
                    .transformed(transform.inverted()).x1

        xT = x_pos # where all text ends

        shift += precision/2 # increase for next
    iteration

        if x_pos > 1: # guardrail
            break
```

color-title.py

```
x = range(101)
# Create a Gaussian random walk starting at 0
start = np.zeros(1)
y1 = np.concatenate( [start,np.random.normal(0,1,100)] )
    .cumsum()
y2 = np.concatenate( [start,np.random.normal(0,1,100)] )
    .cumsum()

fig, ax = plt.figure(), plt.axes()
plt.tight_layout()
# Color arguments added to make defaults explicit
ax.plot(x,y1, color = 'C0')
ax.plot(x,y2, color = 'C1')

labels = ['Stock Performance: ', 'GenericCo', ' & ', '
    PerfunctoryInc']
colors = ['black', 'C0', 'black', 'C1']
color_title(labels, colors)
```

color-title-ex.py



4.6 Fonts

Finally, you might want to customize the fonts. First, we'll consider the customizations available in matplotlib by default.

4.6.1 Importing Fonts with Font Manager

If you're unsatisfied with the basic fonts available in matplotlib, just add your own. You can find fonts available for download from theleagueofmoveabletype.com or fonts.google.com.

After you've downloaded a font family, you should have folder for that font with otf or ttf files. Matplotlib has a font manager and you just need to tell matplotlib to look for a font in that folder. This is done below using `fintSystemFonts()` and `addfont()`. Once the font files are added, you can simply specify the font in the `text()` call like any other in-built font.

```
# font download
# https://fonts.google.com/specimen/Pacifico
# access font and add to font manager
font_dirs = ['Downloads/Pacifico'] # change depending on where you downloaded it
font_files = font_manager.findSystemFonts(fontpaths=font_dirs)
for font_file in font_files:
    font_manager.fontManager.addfont(font_file)

# Make Figure
fig, ax  = plt.figure(), plt.axes()
t = fig.text(0.5,0.5,
             'Live Laugh Love',
```

```
                ha = 'center',
                va = 'center')
ax.axis('off')
t.set_size(50)
t.set_name("Pacifico")
t.set_color('yellow')
fig.set_facecolor('brown')
```

font.py

Bibliography

Borg, I. (2018). *Applied multidimensional scaling and unfolding*. Springer.

Harper, M. (2020). Python-ternary: Ternary plots in python. *Zenodo 10.5281/zenodo.594435*. https://doi.org/10.5281/zenodo.594435

Knaflic, C. N. (2015). *Storytelling with data: A data visualization guide for business professionals*. John Wiley & Sons.

Orwell, G. (2013). *Politics and the english language*. Penguin UK.

Schwabish, J. (2014). An economist's guide to visualizing data. *Journal of Economic Perspectives*, *28*(1), 209–34.

Schwabish, J. (2021). *Better data visualizations: A guide for scholars, researchers, and wonks*. Columbia University Press.

Turrell, A., & contributors. (2021). *Coding for economists*. Online. https://aeturrell.github.io/coding-for-economists

VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc."

About the Author

His name is Alexander. He is a data scientist at LinkedIn and is an adjunct assistant professor at Columbia University. He holds a Ph.D. in Economics from the University of Wisconsin–Madison and is an alumnus of the Insight Health Data Science Fellows Program. ✌