Matplotlib for Storytellers

By: Alexander Clark

This version: July 21, 2023

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

All code and data files are 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 `plt.tight_layout()`, which is not always included in the Python files. See `Prose-Figure-Dev.ipynb`, `Math-Interlude-Figure-Dev.iypnb`, `Special-Figure-Dev.iypnb`, and `Poetry-Figure-Dev.iypnb` which contain the complete code for every figure.

```python
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.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 3.7.1. I also make use of Jupyter notebook cell magics. 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 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. 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.

The official matplotlib documentation continues to be a great resource, especially with new tutorials and galleries being added since I began work on this book. 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 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

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.

```
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.

```
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.

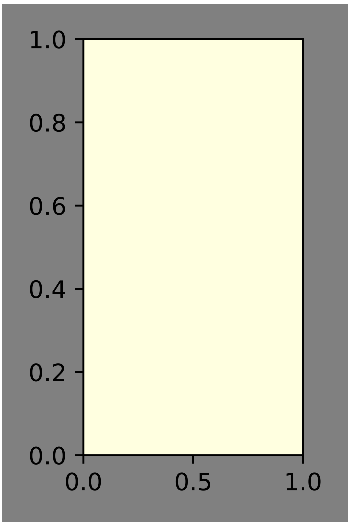
An object's data are called its *attributes* and the procedures or functions are called *methods*. In the previous code, we have figure and axes objects, making use of axes methods `plot()` and `set_title()`, both of which add data to the axes object in some sense, as we could extract the lines and title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



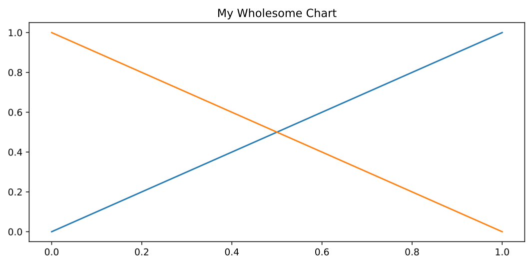
The axes object, named `ax` by convention, gets more use in most programs. In place of `plt.plot()`, you'll use `ax.plot()`. Similary, `plt.hist()` is replaced with `ax.hist()` to create a histogram. If you have experience with the MATLAB interface, you might get reasonably far with the object-oriented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

```
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]

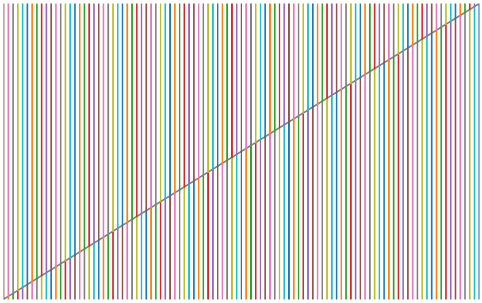
1.2 Mixing the Interfaces

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.

```
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.



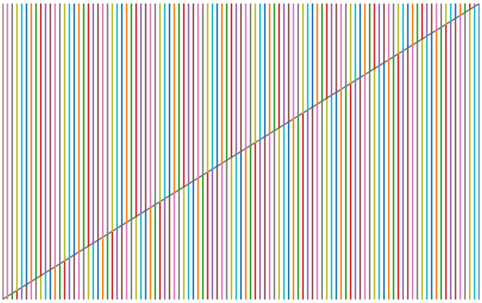
In the above, we started with MATLAB and then converted to object-oriented. We can also go in the opposite direction, though it's not always ideal, especially when working with subplots. Below, we start with our figure and axes objects, and then revert back to the MATLAB style with the `axvline()` functions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

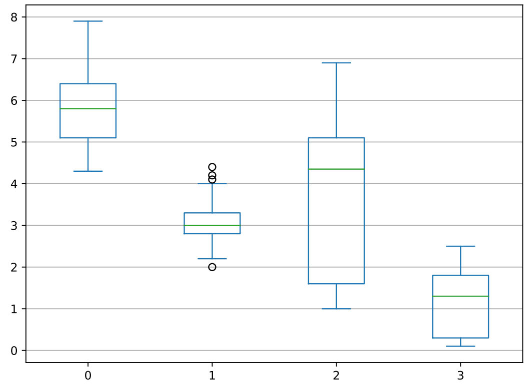
[2] https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html



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

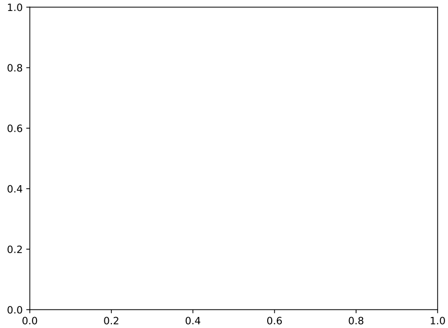
Axes Appearance, Ticks, and Grids

2.1 Axis Aspect and Limits

The most basic plot is the empty plot.

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

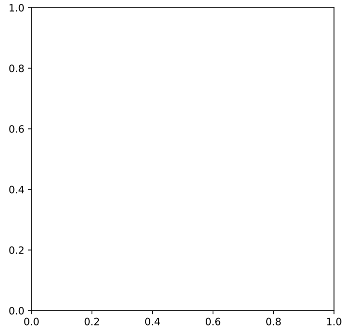
empty.py



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)`.

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

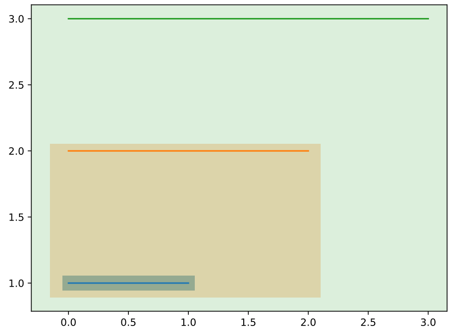
empty-square.py



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.

```python
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)
```

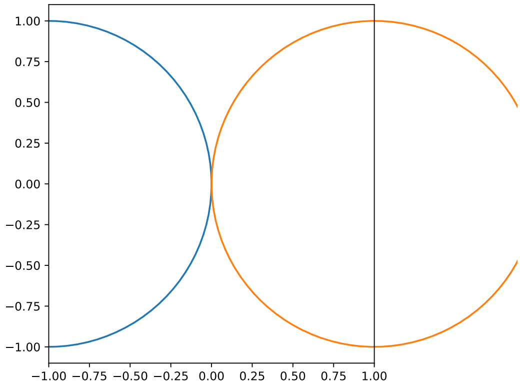
expanding-lims.py



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



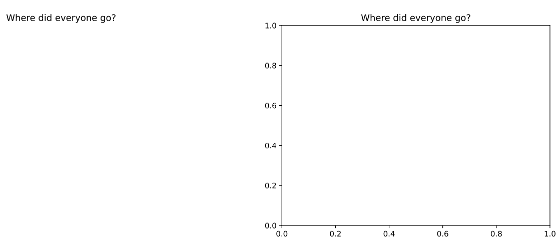
2.2 Axis Lines and Spines

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')
```

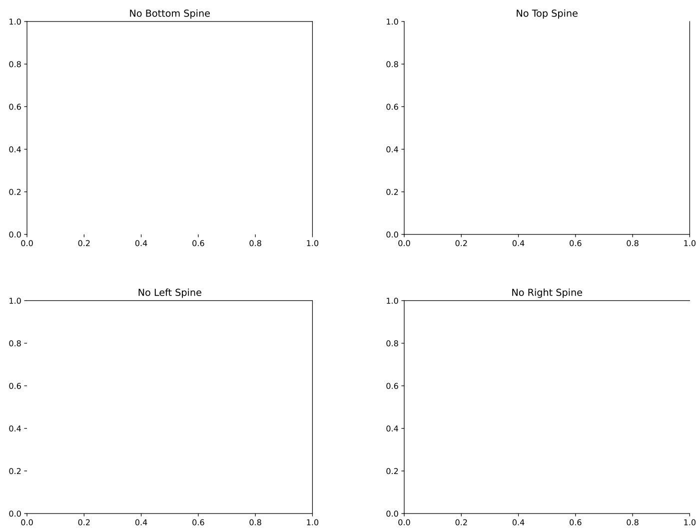
no-axis.py



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()
```

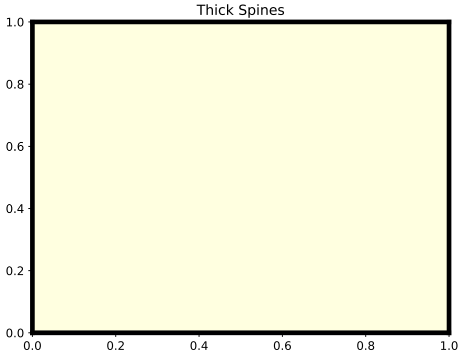
spine-vis.py



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)
```

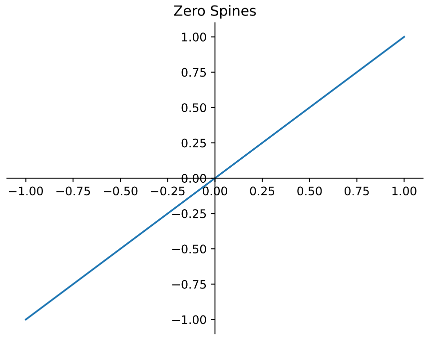
thick-spines.py



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')
```

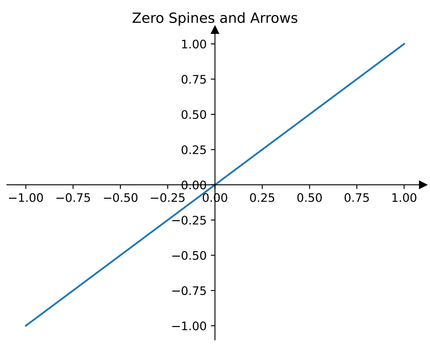
zero-spines.py



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



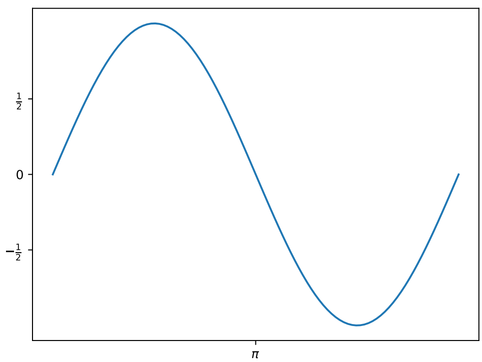
The tick labels do clutter the graph above. This can be solved after we cover Section 2.3. Knaflic 2015 recommends removing the top and right spines as part of the imperative to declutter and remove unnecessary chart border. I think it is arguable. I'm used to default spines enclosing the data. Removing 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



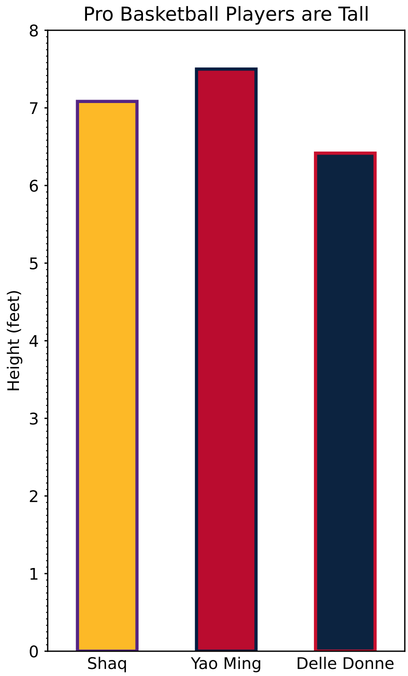
To remove the ticks entirely, simply pass an empty array to `set_ticks()`. To customize the appearance of your axis ticks and the labels, use the `set_tick_params` axis method. Parameters include `direction`, `width`, `length`, `color`, `pad`, `rotation`, `labelsize`, `labelcolor`

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

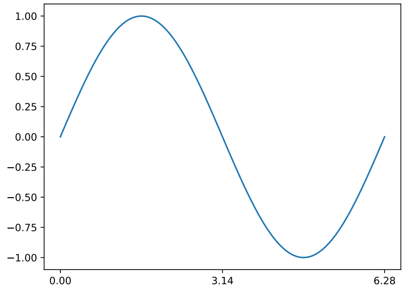


Major ticks can easily be set with `set_ticks` and its variants. Still, `MultipleLocator` and other locators are useful for setting major ticks without fooling with the details of the axes limits.

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

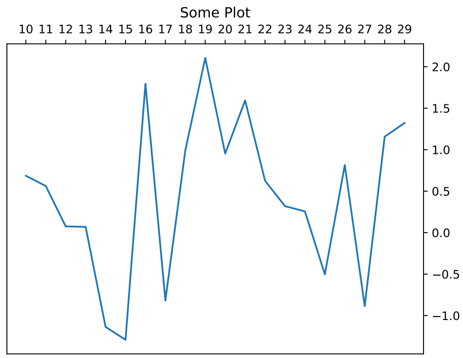


It's true you could avoid the complication of locator classes by just using `ax.set_xticks([0, np.pi, 2*np.pi])`. For a plot this simple, do that. Suppose, you put ticks up to 3π though. Then you've extended the x-axis limit of the plot past your data. So you need to know your data to make the right 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

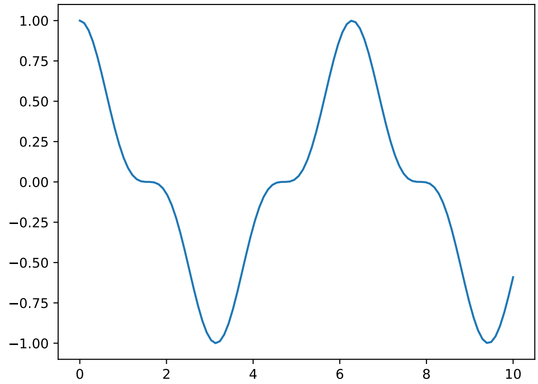


2.4 Grids

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)
```

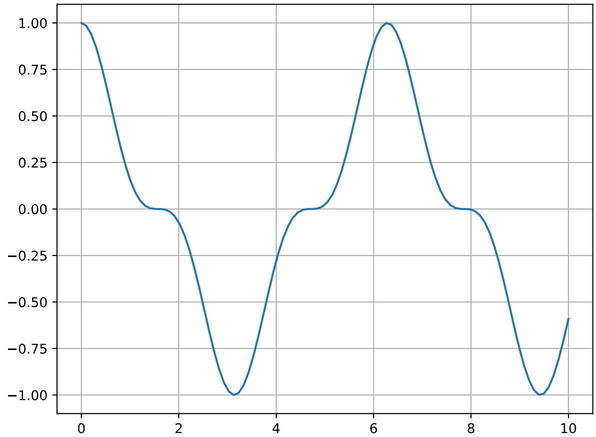
grid-false.py



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()`.

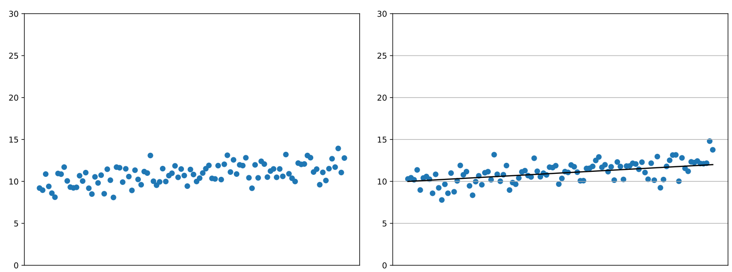
```
fig, ax = plt.figure(), plt.axes()

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')
```

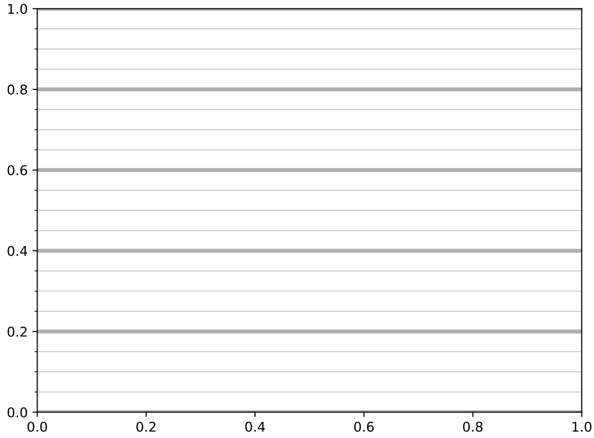
y-grid-true.py



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.

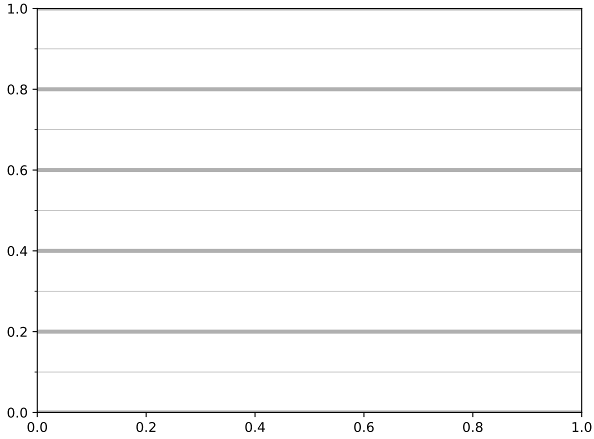
```
fig, ax = plt.figure(), plt.axes()
ax.xaxis.grid(False)
ax.yaxis.grid(True, linewidth = 3)
ax.yaxis.grid(True, which = 'minor', linewidth = 0.5)
ax.yaxis.set_minor_locator(mpl.ticker.AutoMinorLocator()
    )
```

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.

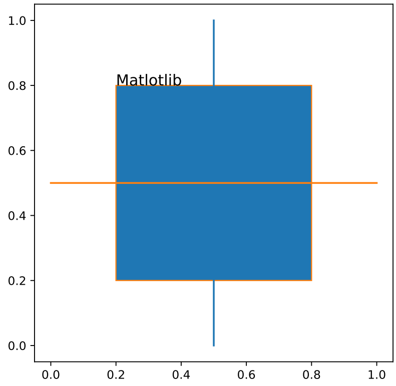
3.1 Primitives and Containers

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

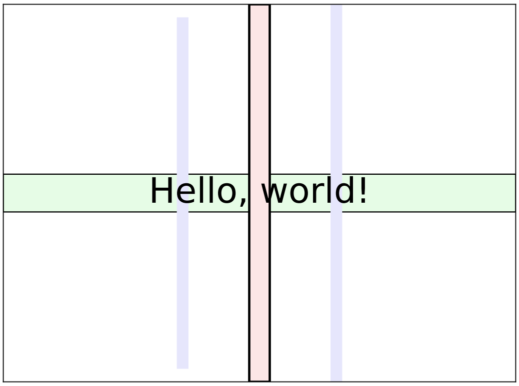
By default, text is plotted over lines and lines are plotted over patches, like the fill created by `fill_between()`. Within each of these three categories, objects created later in the program are plotted over previously created objects. The `zorder` parameter can be used to create a different ordering.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

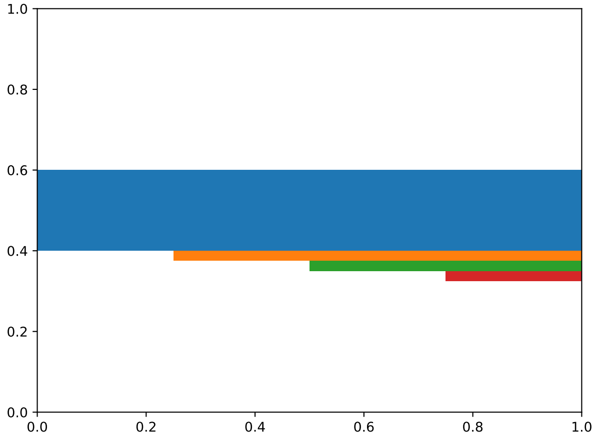


Custom Ordering

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

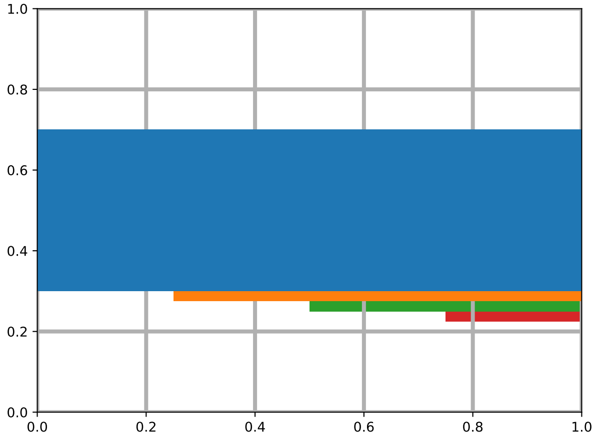


Axes and Tick Ordering

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.

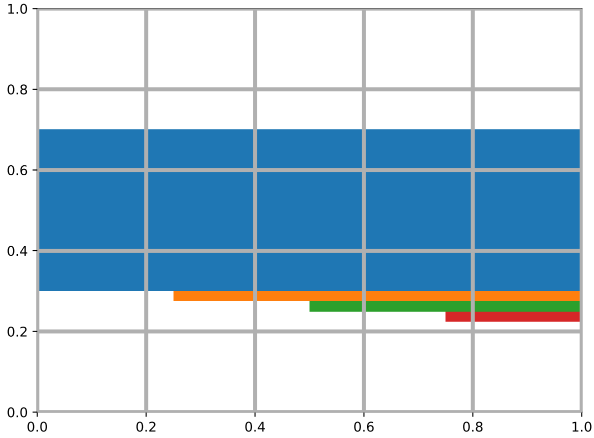
```
fig, ax = plt.figure(), plt.axes()
for i in [0, 0.25, .5, .75]:
    ax.fill_between([i,1], 0.3 - i/10, .7 - i/20,
                    zorder = 2-i)
ax.grid(True, linewidth = 3)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
print(ax.get_zorder())
```

`default-axes.py`



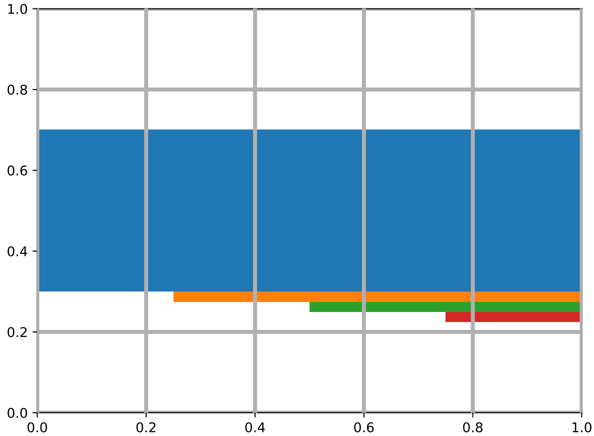

```
fig, ax = plt.figure(), plt.axes()
for i in [0, 0.25, .5, .75]:
    ax.fill_between([i,1], 0.3 - i/10, .7 - i/20,
                        zorder = 2-i)
ax.grid(True, linewidth = 3)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_axisbelow(False)
print(ax.get_zorder())
```

front-axes.py



```
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



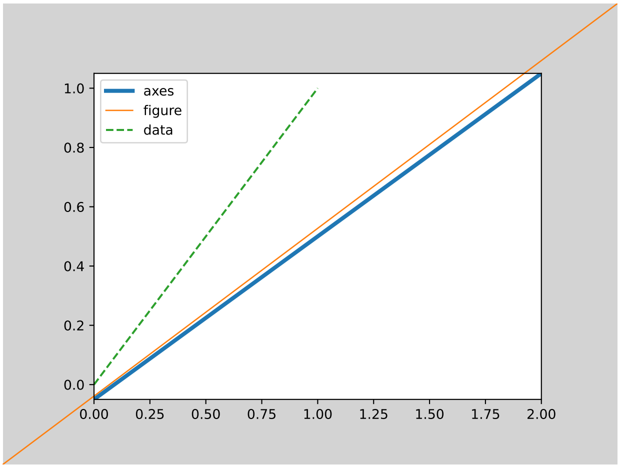
3.2 Coordinate Systems and Transformations

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()
```

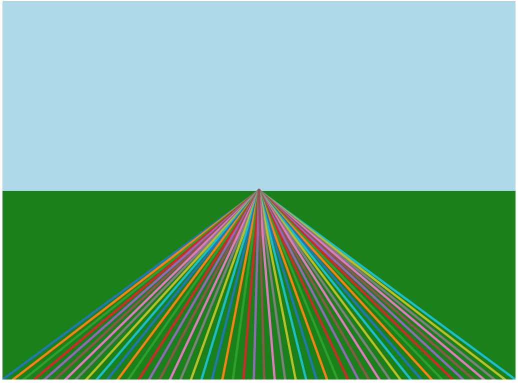
coords.py



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')
```

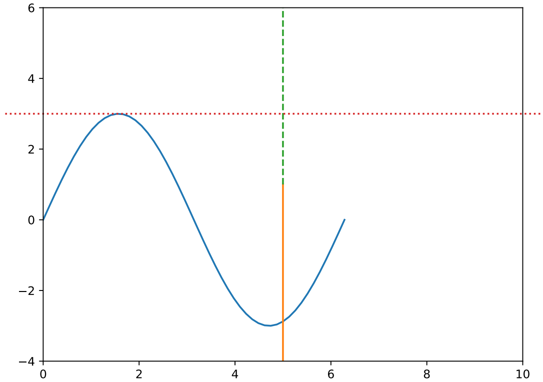
coord-horizon.py



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.

```
# 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



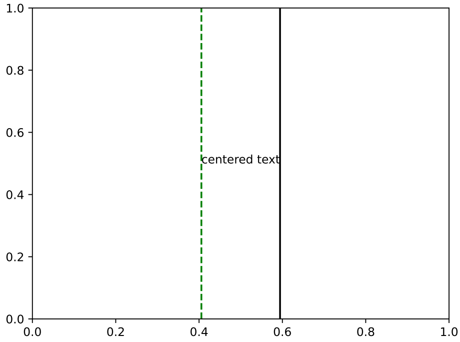
3.3 Use Window Extents

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

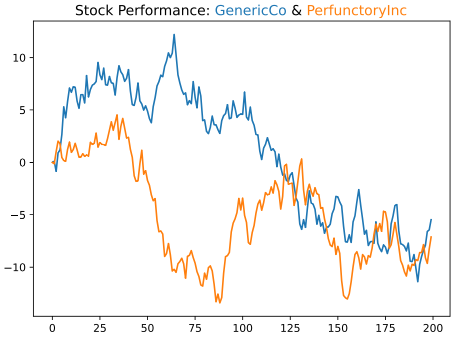


So what? A formatted title can stand in for a legend, helping reduce clutter. This helps us heed the call from Schwabish 2021 to label data directly and avoid legends when possible. In the line chart below, a legend is unnecessary given the color-coding in the title. We create a title not with the typical `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()
```

```python

# 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 6, but to start you can simply use the name of any not-too-exotic color as a string.

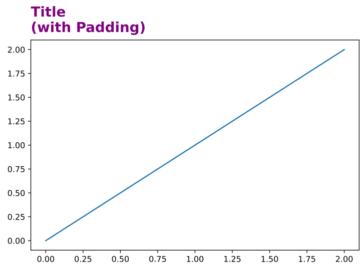
`fontsize` (or `size`) can be a number or chosen from `'small'`, `'medium'`, or `'large'`, and `'small'` and `'large'` may be intensified with a `'x-'` or `'xx-'` prefix. Similarly, `weight` (or `fontweight`) can be a number or chosen from options like `'bold'` or `'light'`.

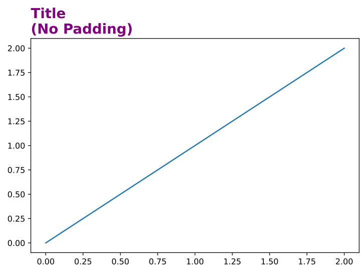
`loc` determines the location of the title, either `'left'`, `'center'`, or `'right'`. In the default style, the default value will be `'center'`. You might prefer using `'left'` to match the Google Sheets default (thus matching the vast majority of plots I've seen in industry). `pad` controls the space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

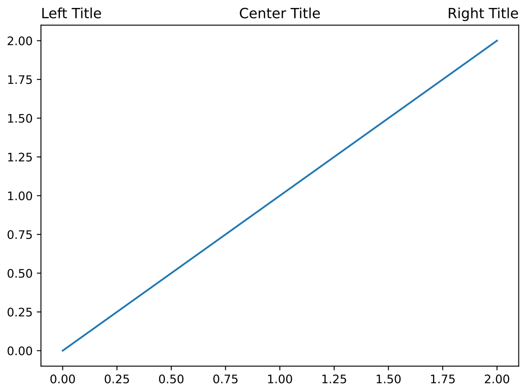




A plot can actually have one title for every `loc` value as well.

```
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

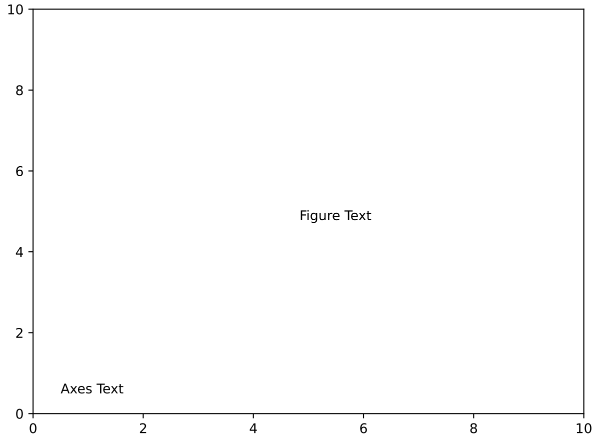


4.2 Text and Placement

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.

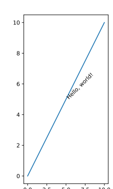
```
fig, ax = plt.figure(), plt.axes()

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')
```

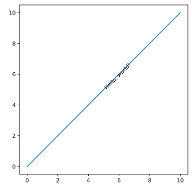
text-align.py

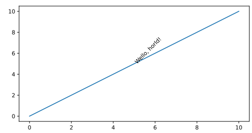
Text can be rotated with the `rotation` parameter. By default, a plot isn't square—the aspect ratio (the ratio of y-unit to x-unit) is not one. That means that the 45 degree line created by $y = x$ is not actually plotted at 45 degrees. Yet according to the `rotation` parameter, text rotated at 45 degrees is plotted at 45 degrees—that angle is not converted based on the aspect ratio. Later in Section 10.1, 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]

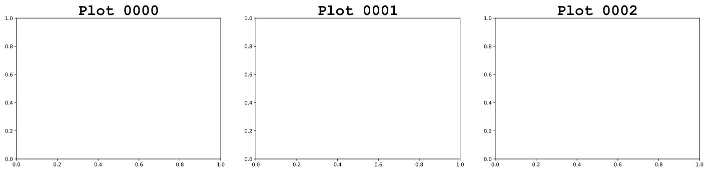
Whatever we put outside the curly braces is simply concatenated to the text on the left or right. So `'${}'.format(123)` turns 123 into the dollar figure `'$1231'`. And `'{} lbs.'.format(123)` would produce `'1231 lbs'`.

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.

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

```
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

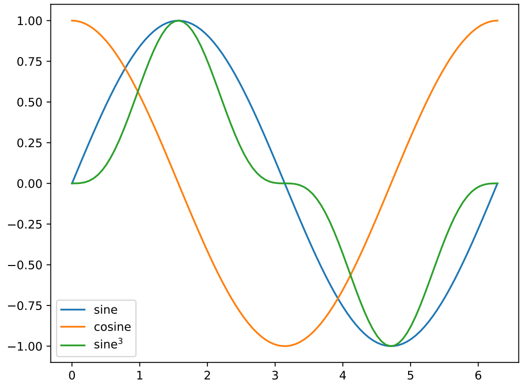
As you should know, legends provide a key to the colors and symbols used in a plot. You can create a legend with `legend()`, as either a figure or axes method. Without any extra customization this is done with `ax.legend()` or `fig.legend()`. Here, we will only cover axes legends. We'll return to figure legends when they are more naturally useful in Chapter 7 on multiple axes and multiple plots.

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

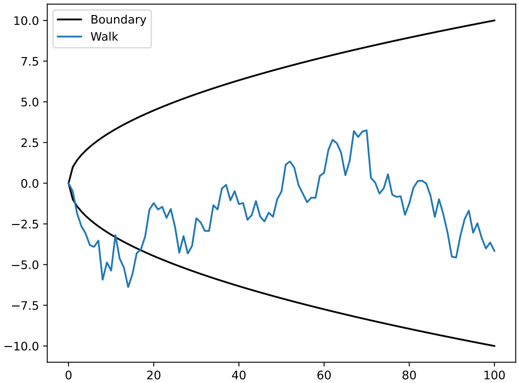


If you are using a pandas plot method, the labels will be set automatically according to the column or series names. For such instances where an element is automatically included in a legend and you want to exclude it, you can exclude that element by specifying `label = '_nolegend_'` in the plot call.

```
# 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.

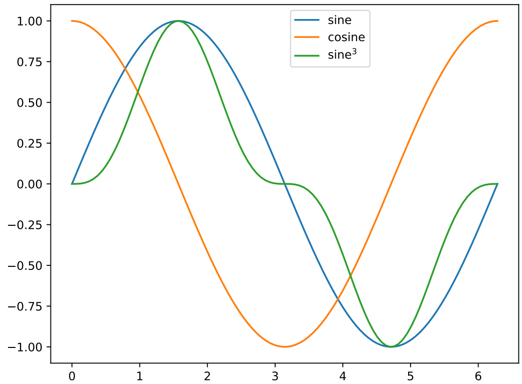
To change the placement of the legend, you may use the `loc` parameter. The default value is `'best'`, where best is determined by matplotlib. Other valid values are `'center'` and `'right'` (but not `'left'`) and then modifications like `'upper center'`, `'center right'`, and `'lower left'`.

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.

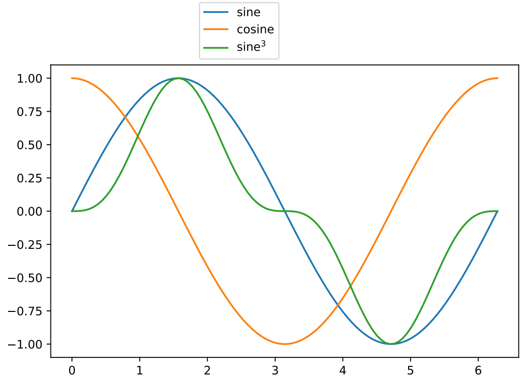
```
fig, ax = plt.figure(), plt.axes()
x = np.linspace(0,2*np.pi,100)
ax.plot(x, np.sin(x), label = 'sine')
ax.plot(x, np.cos(x), label = 'cosine')
ax.plot(x, np.sin(x)**3, label = r'sine$^3$')

# 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

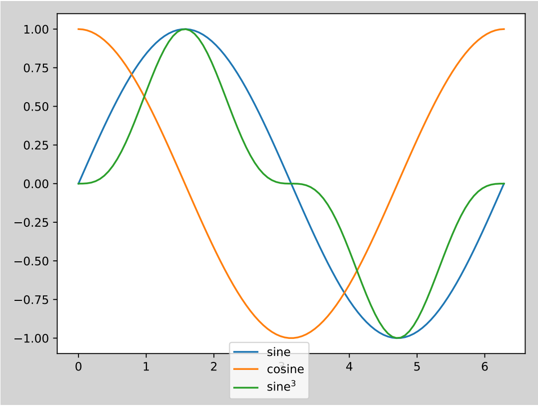


If using a 4-tuple, the tuple is interpreted as the plot region in which to put the legend, according to `loc`.

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

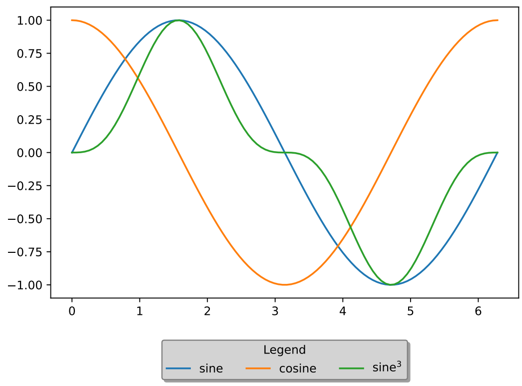


There are many parameters to change the appearance of the legend. We won't cover all of them. Two useful parameters are `facecolor` and `ncol`. The former changes the background color of the legend and the latter sets the number of columns, changing the default shape of the legend. I use these and a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

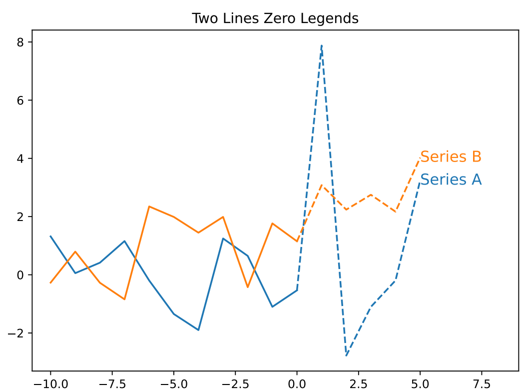
4.4.1 Labeling and Arrows

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

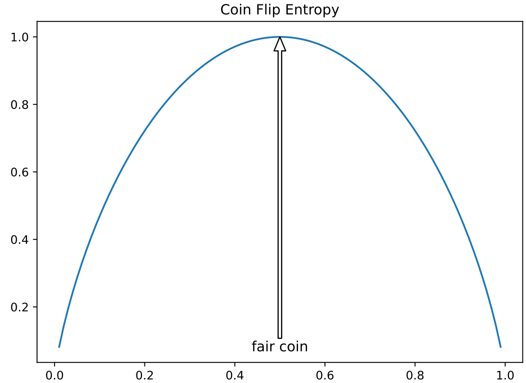


Next, we use the `annotate()` method. This method comes with the option to include an arrow pointing from `xytext` to the point `xy`.

```
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

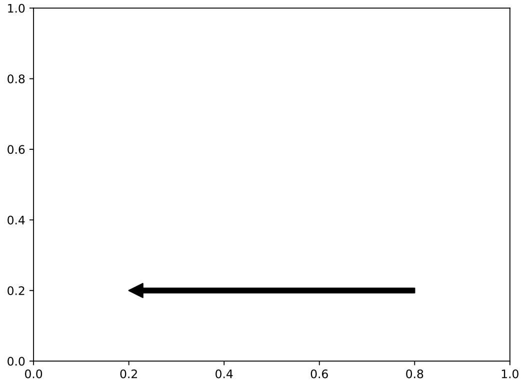


If you would like an arrow and no text, simply use the empty string ''. It is necessary to pass a dictionary to the `arrowprops` property.

```
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

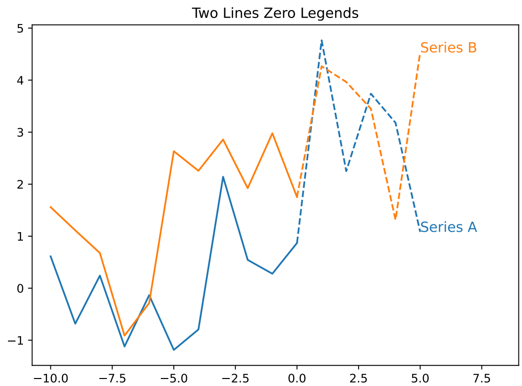


Lastly, one can also reference specific artist objects in the annotation instead of coordinates. In the below we place the annotations at the end of `a_line` and `b_line`.

```
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

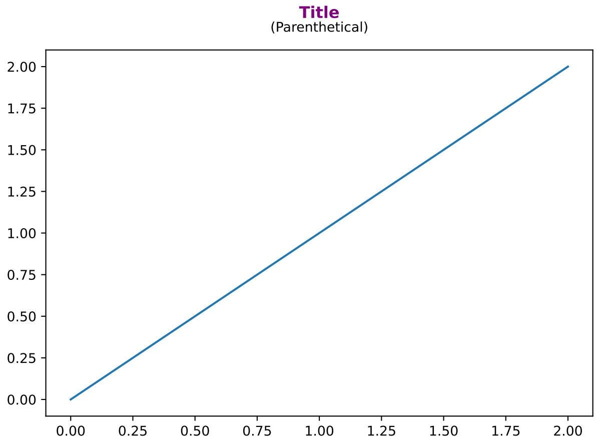


4.5 Fancy Titles

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



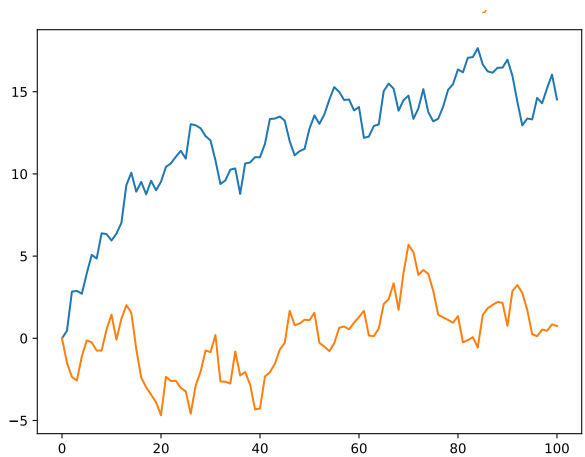
4.5.1 Multi-colored Titles

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. "

```

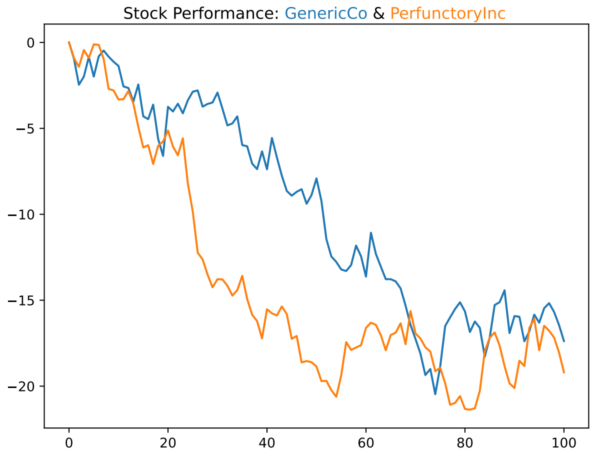
```
    if ax == None:
        ax = plt.gca()

    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. In matplotlib 3.6 and newer, there is a `get_font_names()` method that can be used to display available font names. The code below creates a figure for each font. I get several warnings with messages like "Glyph 105 (i) missing from current font."

```
font_list = font_manager.get_font_names()
for f in font_list:
    plt.plot()
    plt.gcf().text(0.5, 0.5,
                   s = f,
                   size = 20,
                   ha = 'center',
                   fontname = f)
    plt.axis('off')
    plt.show()
```

get-fonts.py

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


Live Laugh Love

Chapter 5

Dates

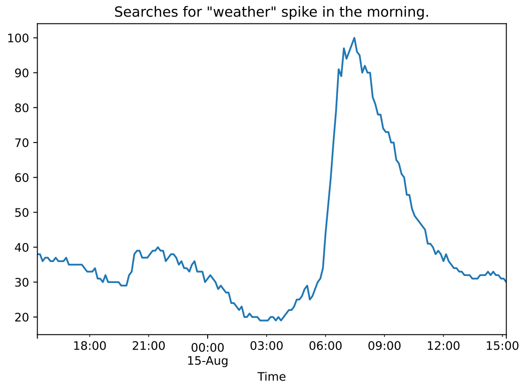
Matplotlib can handle dates, helping you to create better axis ticks and label formatting. Matplotlib's capabilities are built on the datetime and dateutil modules.

5.1 Plotting

Let's import some time series data. Below we use pandas integration and plot from a DataFrame with an index of pandas Timestamp values. Matplotlib recognizes these as dates and handles this reasonably well automatically, though the exact formatting could be improved.

```
url = 'https://github.com/alexanderthclark/MPL-Data/raw/
    main/WeatherAug1415Trends.csv'
df = pd.read_csv(url, parse_dates = ['Time'])

fig, ax = plt.figure(), plt.axes()
df.set_index("Time").plot(ax = ax)
ax.set_title("Searches for \"weather\" spike in the
    morning.")
ax.legend().set_visible(False)
```

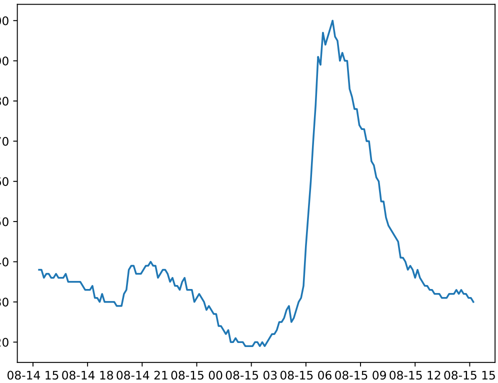
pd-dates.py



Before we try to improve the formatting, see what happens if we try to use the axes plot method.

```
fig, ax = plt.figure(), plt.axes()
ax.plot(df.Time, df.weather)
```

ax-dates.py



You might find code using `plot_date()`, which used to be used in place of `plot()`. This is no longer necessary.

5.1.1 Time Zone Handling

For a deeper knowledge, see the `datetime.tzinfo` class and the `pytz` library. TK

5.2 Ticks and Formatting

5.2.1 Date Formats

The specific format of the displayed dates and times can be modified with `mdates.DateFormatter()`. This takes a format string and creates a formatter that can be passed to an axis method `set_major_formatter()` or `set_minor_formatter()`.

Here are some common format codes, applied to Sunday January 30, 2000, 11:59PM, local to Louisville, Kentucky. These can all be verified with `pd.Timestamp(year = 2000, month = 1, day = 30, hour = 23, minute = 59, tz = 'America/Kentucky/Louisville').strftime()`.

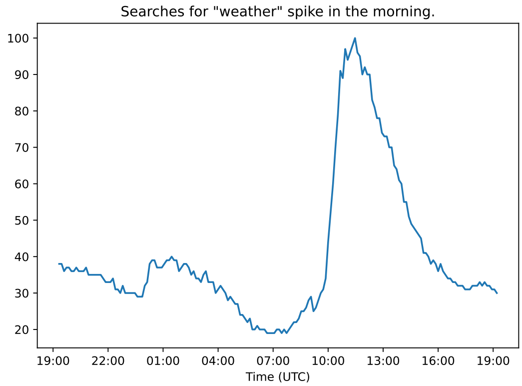
Code	Output/Example
`'%Y'`	4-Digit Year
`'%m'`	Month Number
`'%d'`	Day of Month
`'%B'`	Month Name
`'%H'`	24-Hour Clock Hour
`'%M'`	Minute
`'%H'`	12-Hour Clock Hour
`'%p'`	AM or PM
`'%A'`	Day of Week
`'%Z'`	Timezone Name
`'%Y-%m'`	`'2000-01'`
`'%Y/%m/%d'`	`'2000/01/30'`
`'%B %y'`	`'January 00'`
`'%H:%M %Z'`	`'23:59 EST'`
`'%A %I%p'`	`'Sunday 11PM'`

A more complete list of format codes can be found at strftime.org. Codes that generate actual names, like `'%A'` or `'%B'`, can be made lowercase to produce an abbreviated name. Notice that these formats create zero-padded numbers like `'07'` instead of `'7'`. On Mac or Linux, padding can be eliminated with the `'-'` modifier, using `'%-H'` or `'%-m'` instead of `'%H'` or `'%m'` for example. On Windows, use `'#'`.

```
fig, ax = plt.figure(), plt.axes()
xformatter = mdates.DateFormatter('%H:%M')
ax.plot(df.Time, df.weather)
ax.set_title("Searches for \"weather\" spike in the
    morning.")
ax.set_xlabel("Time (UTC)")
```

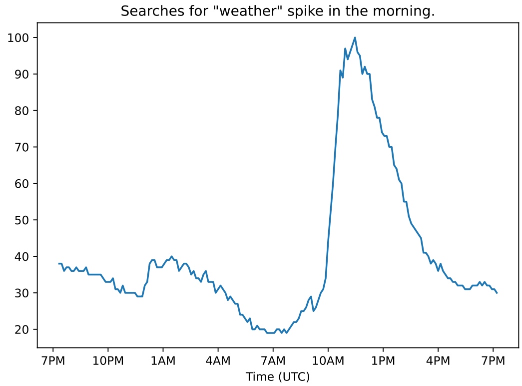
```
ax.xaxis.set_major_formatter(xformatter)
```

date-fmt.py




```
fig, ax = plt.figure(), plt.axes()
xformatter = mdates.DateFormatter('%-I%p')
ax.plot(df.Time, df.weather)
ax.set_title("Searches for \"weather\" spike in the
    morning.")
ax.set_xlabel("Time (UTC)")
ax.xaxis.set_major_formatter(xformatter)
```

date-fmt2.py



Chapter 6

Colors

Methods like `plot` and `text` include a color parameter, which we've already made use of. While you can get pretty far simply using `color = 'blue'`, you might also make use of colormaps or set your own colors using hex strings or RGB(A) tuples.

6.1 Colormaps

According to the style sheet you are using, there will be some colormap and you will cycle through those colors by default when plotting (but not for text). The colors can be identified by the strings `'C0'`, `'C1'`, ... If, as in the default, your color map has only 10 distinct colors, then the eleventh color `'C10'` is valid, but simply refers to `'C0'` and the colors cycle from there. You'll notice that with successive plot calls on the same axes, the colors will automatically move through the colormap. This is not the case with text, as is demonstrated in the program below.

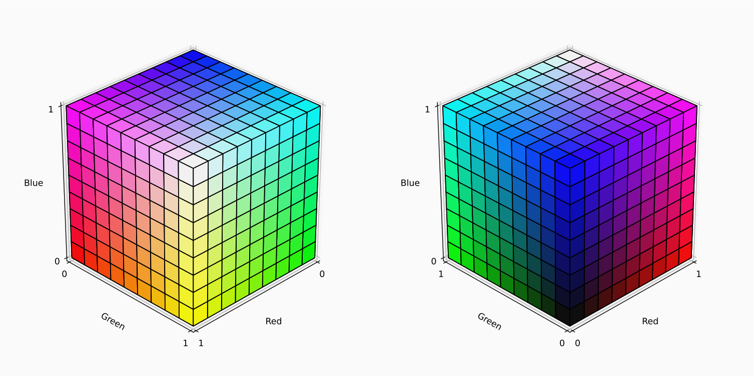
```
fig, ax = plt.figure(), plt.axes()
for i in range(12):
    # Plot color automatically cycles through color map
    ax.plot([0,1], np.ones(2)*i)

    # Text with default color on the left
    ax.text(0, i, 'C' + str(i),
    va = 'center', ha = 'right')

    # Text with variable color on the right
    ax.text(1, i, 'C' + str(i),
    va = 'center', ha = 'left',
    color = 'C'+str(i))
```

```
ax.axis('off')
```

colors.py

6.2 Red, Green, Blue, Alpha

An RGB color is given by three values, specifying the amount of red, green, and blue. In matplotlib, these values are between zero and one (you might also see RGB values between zero and 255 elsewhere). These colors live inside a cube, as a particular color is a triple $(r, g, b) \in [0, 1]^3$.



I like working with RGB tuples because they can be manipulated with mathematical operations. Two colors can easily be averaged or we can create a gradient between two.

```
# Set Colors
green = 76, 217, 100
green = np.array(green)/255
blue = 90, 200, 250
blue = np.array(blue)/255

# How many color changes
segments = 100
interval_starts = np.linspace(0, 1, segments)

fig, ax = plt.subplots(figsize = (8,8))

colors = dict()
for i in range(3):
    colors[i] = np.linspace(blue[i], green[i], segments)

for i in range(segments-1):
    rgb = colors[0][i], colors[1][i], colors[2][i]
    x = interval_starts[i], interval_starts[i+1]
    y = (0.5, 0.5)
    ax.plot(x, y, color = rgb,
            linewidth = 20,
            solid_capstyle = 'round')

ax.set_aspect('equal')
ax.axis('off')
```

gradient.py

Any color can be made lighter by averaging it with white, $(1,1,1)$, or darker by averaging it with black $(0,0,0)$. We can also find the inverse of an RGB color by simply subtracting that triple from $(1,1,1)$. RGBA tuples are very similar, adding a fourth $a$lpha value for the opacity.

With RGB and RGBA colors being so handy, you might want to convert strings like `'C0'` into RGB. `ColorConverter()` lets us do this, with the `to_rgb()` and `to_rgba()` methods. Below, we create another color gradient between the default `'C0'` blue, to `'C1'` orange, and on to light blue `'C9'`.

```
# Set Colors
blue = mpl.colors.ColorConverter().to_rgb('C0')
orange = mpl.colors.ColorConverter().to_rgb('C1')

n_colors = 10
color_strings = dict()
for i in range(n_colors):
    color_strings[i] = 'C'+str(i)
segments = 1000 # How many color changes

fig, ax = plt.subplots(figsize = (14,8))

for c in range(n_colors - 1):
    color1 = mpl.colors.ColorConverter().to_rgb(
    color_strings[c])
```

```python
    color2 = mpl.colors.ColorConverter().to_rgb(color_strings[c+1])

    interval_starts = np.linspace(c, c+1, segments)
    colors = dict()
    for i in range(3):
        colors[i] = np.linspace(color1[i], color2[i], segments)

    for i in range(segments-1):

        rgb = colors[0][i], colors[1][i], colors[2][i]

        x = interval_starts[i], interval_starts[i+1]
        y = [0.3,0.5]

        ax.plot(x, y,
                color = rgb,
                linewidth = 20,
                solid_capstyle = 'round')

    ax.text(c, .51,
            s = 'C'+str(c),
            va = 'bottom',
            size = 12,
            ha = 'center')

ax.text(9, .51,
        s = 'C9',
        va = 'bottom',
        size = 12,
        ha = 'center')

ax.set_aspect('equal')
ax.axis('off')
```

color-map.py

Color Cube Code

Here is the code for one of the RGB color cubes.

```
light_gray = [.98]*3
fig = plt.figure(figsize = (6,6),
                 facecolor = light_gray)
ax = plt.axes(projection='3d',
              facecolor = light_gray)

# control how many cubes/color changes
pieces = 10
grid = np.linspace(0, 1, pieces)[:-1]
width = grid[1] - grid[0]

# Make smaller cube units
for x in grid:
    for y in grid:
        for z in grid:
            vertices = list()
            for prod in product([x,x+width],[y,y+width],[z,z+width]):
                vertices.append(list(prod))

            faces = list()
            for key, face in enumerate([x,y,z]):
                # face is 0
                helper0 = [x for x in vertices if x[key] == face]
                helper1 = [x for x in vertices if x[key] == face + width]
                helper0.sort()
                helper0 = helper0[0:2] + helper0[::-1][0:2]
                helper1.sort()
                helper1 = helper1[0:2] + helper1[::-1][0:2]
                faces.append((helper0))
                faces.append(helper1)

            facecolor = (x + width / 2,
                         y + width / 2,
                         z + width / 2)
            pc = Poly3DCollection(faces,
                                  facecolor = facecolor,
                                  edgecolor = 'black')
            ax.add_collection3d(pc)

# Label Axes
ax.set_xlabel("Red")
ax.set_ylabel('Green')
ax.set_zlabel("Blue")
```

```
# Set Ticks
ax.set_xticks([0,1])
ax.set_yticks([0,1])
ax.set_zticks([0,1])
# Change padding
ax.xaxis.set_tick_params(pad = 0.1)
ax.yaxis.set_tick_params(pad = 0.1)
ax.zaxis.set_tick_params(pad = 0.1)
# Change azimuth
angle = 45 # + 180 # for second cube
ax.view_init(elev = None, azim = angle)
# Zoom out so labels are not cut off
ax.set_box_aspect([1,1,1], zoom = 0.86)
```

color-cube.py

Chapter 7

Multiple Axes and Plots

7.1 Multiple Axes

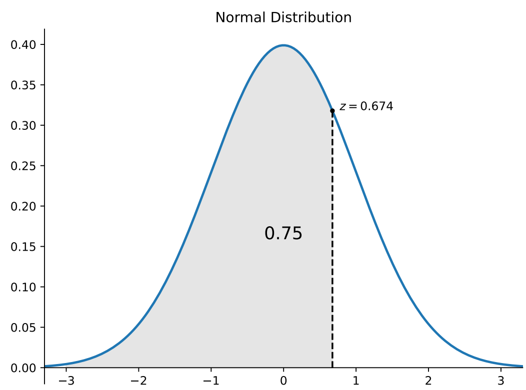
Let's start with a concrete goal to help illustrate possible uses of multiple axes. We want to plot a standard normal distribution. This is the familiar bell curve with a range of possible draws from the normal distribution on the x-axis and y values are the value of the probability density function (PDF) evaluated at each x value. Furthermore, we have a z-score and we want the visual to help us see how often we should get smaller z-scores if we are sampling from this distribution. In particular, let's say our z-score is 0.674.

To answer the question narrowly, the following plot does the job well and without reaching for multiple x- or y-axes.

```
fig, ax = plt.figure(), plt.axes()
ax.set_xlim(-3.3, 3.3)
x = np.linspace(-4, 4, 200)
ax.plot(x, stats.norm.pdf(x),
        linewidth = 2)
ax.set_title("Normal Distribution")

# Choose a point
z = 0.674
pdfz = stats.norm.pdf(z)
cdfz = stats.norm.cdf(z)

# Indicate point on plot
ax.plot([z, z],
        [0, pdfz],
        color = 'black',
        linestyle = 'dashed')
ax.plot([z], [pdfz],
```

```
        color = 'black',
        marker = '.')
ax.text(z + .1, pdfz,
        s = '$z = {:}$'.format(z) )

# Create fill to left
x_vals = np.linspace(-3,z,100)
ax.fill_between(x_vals,
                np.zeros(100),
                stats.norm.pdf(x_vals),
                color = 'gray',
                alpha = .2)

# Area/cumulative density text
ax.text(0, pdfz/2,
        s = "{:.2f}".format(cdfz),
        size = 15,
        ha = 'center')

# Change axes styling
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
```

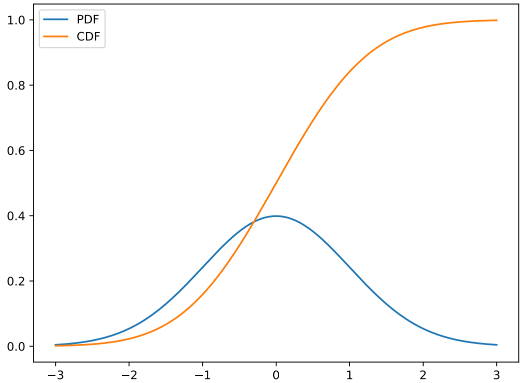
norm-pdf.py



Still, it leaves the reader to rely on their eyeballing abilities to imagine how that area might change if the z-score changed. The graph itself lacks information from the cumulative density function (CDF), used to calculate that our z-score at the 75%ile of values drawn from the standard normal distribution. If your reader might be interested in this kind of thought exercise, you should include more of this information in the plot. First, we might add this information by simply plotting both the PDF and CDF together. Eyeballing is still necessary to imagine how much rarer a z-score of 0.7 is, but at least with the CDF included, we can be a little more precise.

```
fig, ax = plt.figure(), plt.axes()
x = np.linspace(-3,3,200)
pdf_y = stats.norm.pdf(x)
cdf_y = stats.norm.cdf(x)
ax.plot(x,pdf_y, label = 'PDF')
ax.plot(x,cdf_y, label = 'CDF')
ax.legend()
```

cdf-pdf.py



Still, the plot above isn't very good. Here, more ticks or a grid would be helpful for tracing out what the CDF value is for a particular z-score. But apart from that, you might also see that the orange CDF dwarfs the blue PDF. While not terribly extreme, these functions cover different enough y values that having a shared y-axis is questionable, because the point isn't to draw attention to this difference. One fix for this is to create a second y-axis on the right. Knaflic 2015 advises against a secondary y-axis. Dual axis charts aren't as immediately readable, so do be judicious and take extra care to make it clear which plot corresponds to which y-axis.

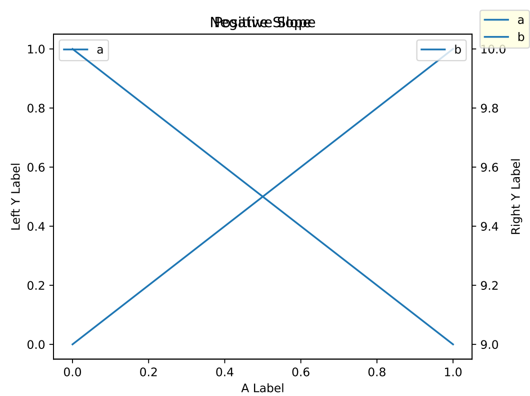
7.1.1 Using `twinx()` and `twiny()`

If we want a second y-axis, or a dual y-axis chart, we can start by creating a plot as usual, creating figure and axes objects `fig, ax`, and then create one more axes object with `ax.twinx()`. Give that a name, `ax2` is what I use below, and the basics are all the same from there. A dual y-axis chart is created with `twinx()` because it is the x-axis that is shared and the y-axes are independent.

Let's take a brief detour from our normal distribution plots to illustrate some of the basics. You'll notice a few problems with the following plot.

```
fig, ax = plt.figure(), plt.axes()
ax2 = ax.twinx()

x = np.linspace(0,1,2)
ax.plot(x, x, label = 'a')
ax2.plot(x, 10-x, label = 'b')

ax.set_xlabel("A Label")

# This does nothing
ax2.set_xlabel("Label Attempt")

ax.set_ylabel("Left Y Label")
ax2.set_ylabel("Right Y Label")

ax.set_title("Positive Slope")
ax2.set_title("Negative Slope")

ax.legend()
ax2.legend()
fig.legend(facecolor = 'lightyellow')
```

dual-bad.py



From the second `plot()` call, everything starts to go downhill.

1. The plotted lines are the same color.

2. `set_xlabel()` does nothing for the x-axis-sharing twin axes.

3. The titles overlap.

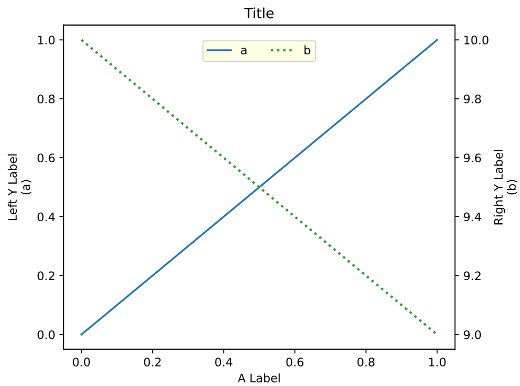
4. `legend()` fails as an *axes* method. The figure legend isn't placed well.

5. It's not clear what line plot corresponds to what axis.

To fix the color issue, we must explicitly pass color values. The fixes for the second and third items are simple. Just use the original axes object for titling and labeling the shared axis. For the fourth, legend issue, we must use `legend()` as a figure method and explicitly pass a `loc` value. To clarify what line plot corresponds to what y-axis, we can tell the reader with our y-axis labels. This isn't a great solution, but it's where we'll start for the most basic fix. To match a line to its axis, we have too many steps to follow: match the plot to its label with the legend and then match the label to its axis.

```
fig, ax = plt.figure(), plt.axes()
ax2 = ax.twinx()

x = np.linspace(0,1,2)
ax.plot(x, x, label = 'a')
ax2.plot(x, 10-x,
         label = 'b',
         color = 'C2',
         linestyle = 'dotted',
         linewidth = 2)

ax.set_title("Title")

ax.set_xlabel("A Label")
ax.set_ylabel("Left Y Label\n(a)")
ax2.set_ylabel("Right Y Label\n(b)")

fig.legend(facecolor = 'lightyellow',
           bbox_to_anchor = (.5,.92),
           ncol = 2,
           loc = 'center',
           bbox_transform = ax.transAxes)
```

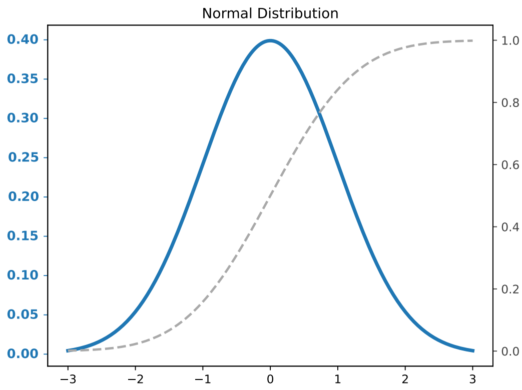
dual-ok.py



Returning to the normal distribution, we'll try to do a better job of making it more visually apparent what pieces of the plot belong to what y-axis.

```
fig, ax = plt.figure(), plt.axes()
ax2 = ax.twinx()
x = np.linspace(-3,3,200)
pdf_y = stats.norm.pdf(x)
cdf_y = stats.norm.cdf(x)

# Plot Curves
ax2.plot(x, cdf_y,
         color = 'darkgray',
         linestyle = 'dashed',
         linewidth =2)
ax.plot(x, pdf_y,
        linewidth = 3)

ax.set_title("Normal Distribution")

# Change Ticks
# Use LaTeX \mathbf to make ticks bold
bolded_ticks = [r'$\mathbf{'+"{:.2f}".format(x)+r"}$"
    for x in ax.get_yticks()]
ax.set_yticklabels(bolded_ticks)
ax.tick_params(axis ='y',
               colors = 'C0',
               labelsize = 11)
ax2.tick_params(axis ='y',
                colors = (.25,.25,.25)) # darker gray
```

dual-norm.py



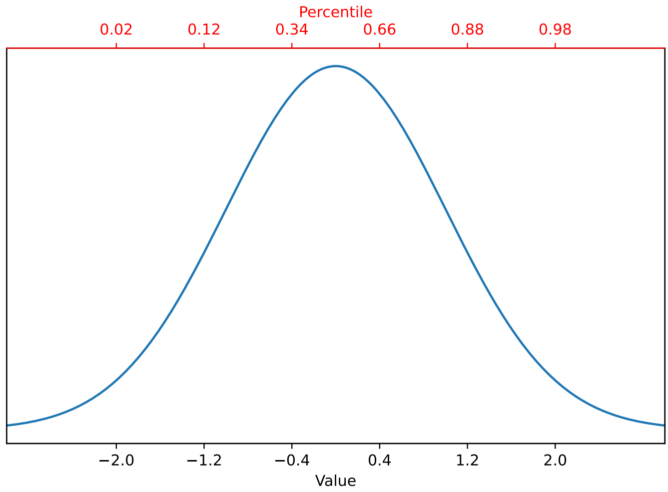
Here, the CDF plot and the secondary axis serve as a kind of footnote to the main point in the CDF.

Adding the cumulative distribution function helps, but that S-curve adds visual noise someone familiar with PDFs and CDFs might be better off without. One solution might be to add a second x-axis which annotates the chart with the CDF value at each point on the first x-axis.

```
fig, ax = plt.figure(), plt.axes()
ax2 = ax.twiny()

# Plot PDF
x = np.linspace(-3,3,200)
y = stats.norm.pdf(x)
ax.plot(x,y)

# Set x ticks for bottom x-axis
xticks = np.linspace(-2,2,6)
ax.set_xticks(xticks)

# Get corresponding CDF values for each tick
labels2 = list()
for tick in xticks:
    cumulative = stats.norm.cdf(tick)
    labels2.append(round(cumulative,2))

# Add ticks to top x-axis
ax2.set_xticks(xticks)
ax2.set_xticklabels(labels2, color = 'red')

# Clear y ticks
ax.set_yticks([])

# Set Limits
xlims = -3,3
```

```
ax2.set_xlim(xlims)
ax.set_xlim(xlims)

# Label and change color
ax.set_xlabel("Value")
ax2.set_xlabel("Percentile", color = 'red')
ax2.spines['top'].set_color('red')
ax2.tick_params(axis ='x', colors = 'red')
```

dual-norm-b.py



7.2 Multiple Plots

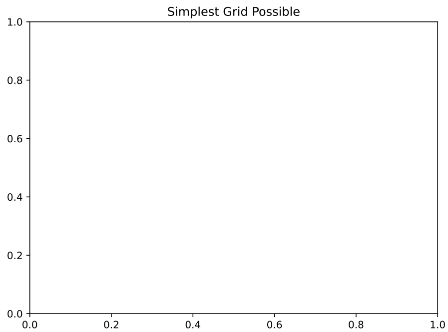
We can add several subplots to a figure in several different ways. We'll go over using `plt.subplots` and `fig.add_subplot`. `plt.subplots` is also useful as a shortcut, as `fig, ax = plt.figure(), plt.axes()` can be replaced with `fig, ax = plt.subplots()` for any figure with just one subplot (i.e. in every previous instance of `fig, ax` in this book.) as the default is a 1×1 grid of a single plot.

7.2.1 Using `subplots`

`plt.subplots` creates a figure *and* and axes object(s). The first two arguments are `nrows` and `ncols` for the number of rows and columns in the resulting plot grid. If the grid is not 1×1, then you will have multiple axes objects in an array. Let's have a look.

```python
fig, ax = plt.subplots()
ax.set_title("Simplest Grid Possible")
```

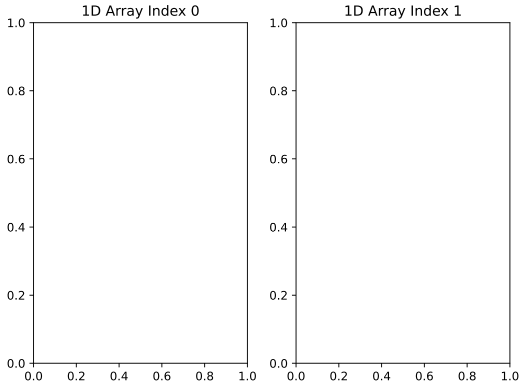
trivial-sub.py



Now, let's make non-trivial grids. Here, `ax` is a 1D array.

```python
fig, ax = plt.subplots(1,2)
ax[0].set_title("1D Array Index 0")
ax[1].set_title("1D Array Index 1")
plt.tight_layout()
```

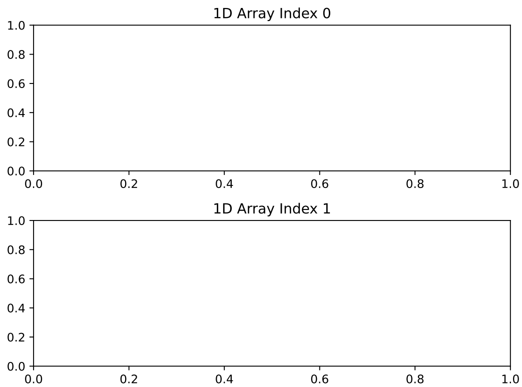
subplots-1d.py



Below, `ax` is again a 1D array.

```python
fig, ax = plt.subplots(2,1)
ax[0].set_title("1D Array Index 0")
ax[1].set_title("1D Array Index 1")
plt.tight_layout()
```

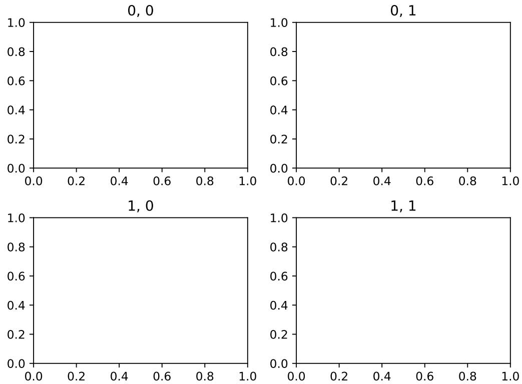
subplots-1d-vert.py



Next, with multiple rows and columns, `ax` is a 2D array.

```
fig, ax = plt.subplots(2,2)
ax[0][0].set_title("0, 0")
ax[0][1].set_title("0, 1")
ax[1][0].set_title("1, 0")
ax[1][1].set_title("1, 1")
plt.tight_layout()
```

subplots-2d.py



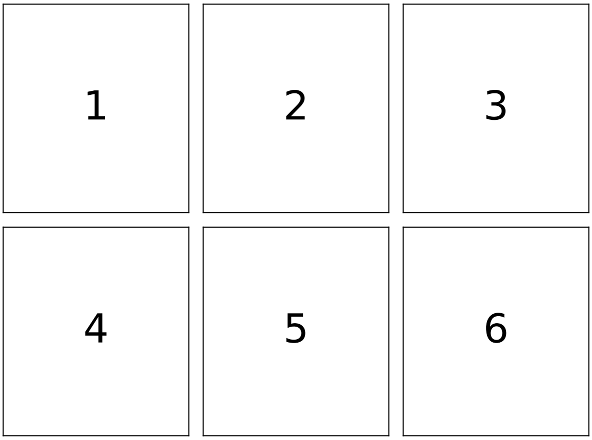
The `ax` object is made as simple as possible based on the `squeeze` parameter, where the default behavior is `squeeze = True` so that unnecessary dimensions are squeezed out of the array. By toggling `squeeze = False`, `ax` will always be made a 2D array. Setting this parameter to be false can be useful when you need to write more flexible code that can accommodate subplots of different dimensions.

7.2.2 Using `add_subplot`

You can avoid indexing an axes array by using the figure method `add_subplot`. The method creates an axes instance and requires specifying the subplot grid's dimensions and then the index or order within that grid. Subplots are not ordered by their row and column numbers, but by a single number. The numbering starts at 1 and increases moving to the right across the first row of graphs, and then proceeds to continue to the next row, again increases from left to right, and on and on. This is demonstrated below.

```
fig = plt.figure()
for i in range(1,7):
    ax = fig.add_subplot(2,3,i)
    ax.text(0.5, 0.5,
            s = str(i),
            ha = 'center',
            va = 'center',
            fontsize = 30)
    ax.set_yticks([])
    ax.set_xticks([])
fig.tight_layout()
```

add-subplot.py



The index value can also be a tuple.

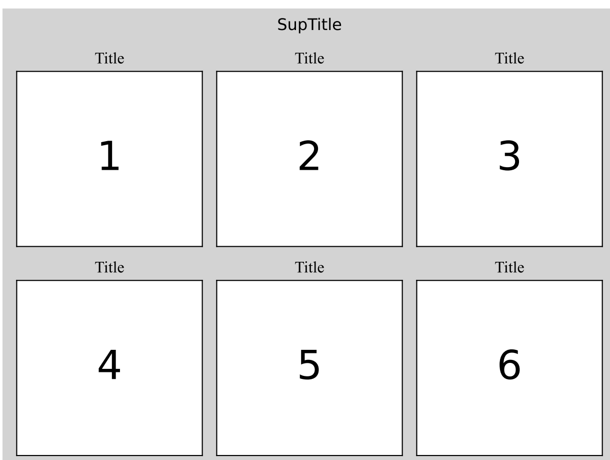
7.2.3 Figure Annotations and Legends

In this subsection, we concern ourselves with customizing the entire figure. Each subplot can be customized just as you might usually customize a single plot. For a figure object `fig`, the axes objects can be accessed by iterating over `fig.axes`, so that all axis limits can be changed in one loop. Figure customizations might include the spacing between plots, standardization of axes, and titling.

First, the figure method `suptitle()` is useful in creating a title that applies to the entire figure.

```
fig = plt.figure(facecolor = 'lightgray')

for i in range(1,7):
    ax = fig.add_subplot(2,3,i)
    ax.text(0.5, 0.5,
            s = str(i),
            ha = 'center',
            va = 'center',
            fontsize = 30)
    ax.set_yticks([])
    ax.set_xticks([])
    ax.set_title("Title",
                 fontsize = 12,
                 fontname = 'Times New Roman')

fig.suptitle('SupTitle')
fig.tight_layout(rect = (0,0,1,1)) # no change
```

suptitle.py



Sometimes a suptitle is cut off when saving the figure. This can be solved by changing the dimensions in `tight_layout()`. Set the `rect` argument to a 4-tuple, like `(0,0,1,.95)`. This modifies the space dedicated to the subplots, and the last value adjusts the vertical upper limit.

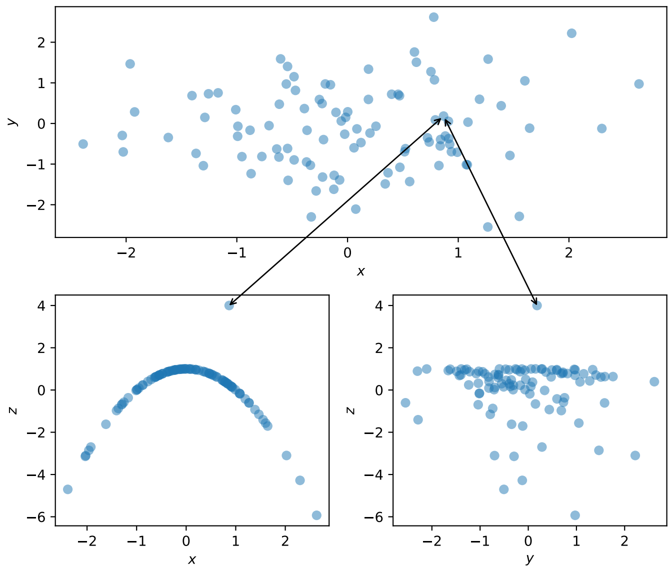
You can also draw lines between two different subplots with `ConnectionPatch`, a kind of *patch*. Patches will be covered more arise again in Part III, but for now it's simeply a tool for use to draw a line between points on two different axes. These points are specified by parameters `xyA` and `xyB`. We specify the coordinate systems using `coordsA` and `coordsB`, making use of what we learned about transforms in Section 4.2 to specify our given coordinates are data coordinates. Then we use the `arrowstyle` parameter to create a line with arrows on both ends and the `shrinkA` and `shrinkB` parameters control how much the line will fall short of, or shrink away from the referenced point.

This code also makes use of the `transform` parameter to specify that the passed coordinates are data coordinates. See Section 4.2 for a review of transformations and other coordinate systems.

```
fig = plt.figure(figsize = (7,6))

# Generate random data
n = 100
x = np.random.normal(size = n)
y = np.random.normal(size = n)
# z is determined by x except for one outlier
z = np.concatenate([np.array([4]), 1- x[1:]**2])

# Add x,y scatter plot
ax12 = fig.add_subplot(2,2,(1,2))
ax12.scatter(x,y, alpha = 0.5)

# Add x,z scatter plot
ax3 = fig.add_subplot(2,2,3)
ax3.scatter(x,z, alpha = 0.5)

# Add y,z scatter plot
ax4 = fig.add_subplot(2,2,4)
ax4.scatter(y,z, alpha = 0.5)

# Draw lines connecting the outlier as it appears in
    each scatter plot
con = ConnectionPatch(
        xyA = (x[0], y[0]),
        coordsA = ax12.transData,
        xyB = (x[0], z[0]),
        coordsB = ax3.transData,
        arrowstyle = "<->",
        shrinkA = 2,
        shrinkB = 0)
fig.add_artist(con)
```

```

con = ConnectionPatch(
        xyA = (x[0],y[0]),
        coordsA = ax12.transData,
        xyB = (y[0], z[0] ),
        coordsB = ax4.transData,
        arrowstyle = "<->",
        shrinkA = 2,
        shrinkB = 0)
fig.add_artist(con)

ax12.set_xlabel("$x$")
ax12.set_ylabel("$y$")
ax3.set_ylabel("$z$")
ax3.set_xlabel("$x$")
ax4.set_ylabel("$z$")
ax4.set_xlabel("$y$")
plt.tight_layout()
```

connect-path.py

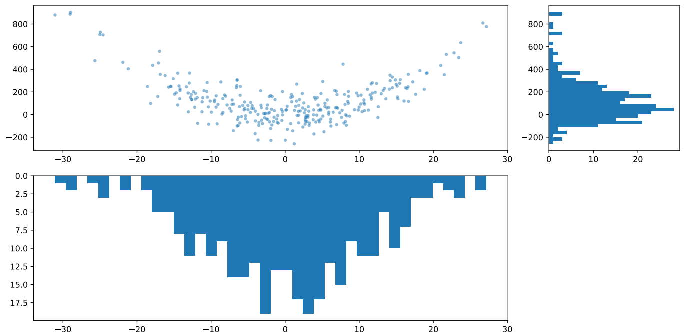


7.3 GridSpec

For irregular plot grids, `GridSpec` is your friend. You can specify a grid with some number of rows and columns and spacing between them. For example, `grid = plt.GridSpec(2, 3, wspace = 1, hspace = 0.3)`. Then, you can specify subplot locations using the typical slicing syntax. For example, `plt.subplot(grid[0,0])`. Or you can create an axis object for a subplot with `ax = fig.add_subplot(grid[0,0])`.

```
fig = plt.figure(figsize=(12,6))
spec = gridspec.GridSpec(ncols=4,
                         nrows=2,
                         figure=fig)
x = np.random.normal(0, 10, size = 300)
y = x**2 + np.random.normal(0, 100, size = 300)

ax1 = fig.add_subplot(spec[0, 0:3])
ax1.plot(x, y,
         linestyle='None',
         marker='.',
         alpha=0.5)

ax2 = fig.add_subplot(spec[0, 3:4], sharey = ax1)
ax2.hist(y, orientation='horizontal', bins=40)

ax3 = fig.add_subplot(spec[1, 0:3], sharex = ax1)
ax3.hist(x, bins = 40)
ax3.invert_yaxis()
plt.tight_layout()
```

gridspec.py



Chapter 8

Style Configuration

This is a brief chapter that might provide a sigh of relief. So many of the parameters we have tweaked so far, sometimes laboriously, can be altered in one go with `plt.style.use`. Try out the many style sheets already available.[1] You may also define your own or simply change certain parameters directly in your code to apply that styling for your entire session.

8.1 rcParams

Change the matplotlib parameters directly by updating the dictionary-like variable `mpl.rcParams`. A full list of the available parameters can be found in the documentation or you may simply print `mpl.rcParams` to inspect it directly. Note the above line is `mpl.rcParams` and not `plt.rcParams`, because we are not working within the pyplot submodule. Accordingly, you'll have to run `import matplotlib as mpl` first. Because this is like a Python dictionary, you can adjust the settings by simply updating the dictionary value, `mpl.rcParams['axes.grid'] = True` for example.

Working with `rcParams`, directly or through a custom style sheet, provides value that compounds as you add more and more plots to your code. Consider the two programs below. Without `rcParams`, we update each plot.

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

```

[1] https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html#sphx-glr-gallery-style-sheets-style-sheets-reference-py

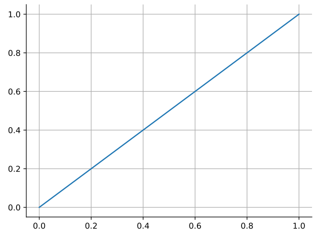
```
fig1, ax = plt.figure(), plt.axes()
ax.plot(x, x)
ax.grid(True)
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.show()

fig2, ax = plt.figure(), plt.axes()
ax.plot(x, 1 - x)
ax.grid(True)
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.show()
```

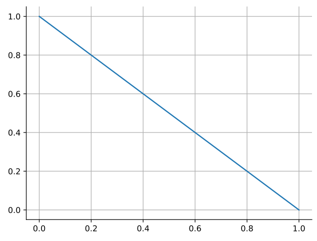
`style-manual.py`

Now let's update `rcParams` just once for a standard style.

```
# Use rcParams
mpl.rcParams['axes.grid'] = True
mpl.rcParams['axes.spines.top'] = False
mpl.rcParams['axes.spines.right'] = False

x = np.linspace(0,1,2)
plt.plot(x,x)
plt.savefig("style1.pdf")
plt.show()

plt.plot(x, 1-x)
plt.savefig("style2.pdf")
plt.show()
```

`style-rc.py`





This is a significant step, nearing us to a dramatic close of Part I. In the second program, we not only save on redundant code, we also revert back to pyplot functions. The object-oriented approach was useful in offering greater customization. But, at least in this case, that's a ladder we can kick away now that we've climbed it to the top.

8.2 Defining Your Own Style

Once you understand rcParams, you can define your own style for repeated use. Schwabish 2021 counsels organizations to adopt a data visualization style guide. Practically, that also means matplotlib-using organizations should choose or create a standard matplotlib style.

To define your own style, create a file with the `.mplstyle` extension, specifying a value for the various rcParams you wish to customize. Note that a colon separates the key and the default like in a dictionary, but that we do not separate key-value pairs by commas and each pair is on a separate line. Further, none of these values are formatted as strings, even though you would, for example, use a string `'Times'` when updating the font from rcParams dictionary. You only need to specify values you wish to change relative to the default.

```
axes.spines.left : True
axes.spines.right : False
axes.spines.bottom : True
axes.spines.top : False
xtick.labelsize : large
font.family : Times
```

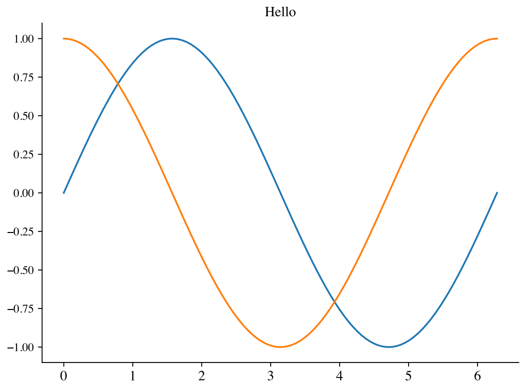
tiny-style.mplstyle

After saving the above as a file `tiny_style.mplstyle` and placing it in the working directory, we can use our custom style with the program below. Note the colormap, among many other things, has not changed relative to the default because we did not alter that in the style file.

```
plt.style.use('../../stylelib/tiny-style.mplstyle')

fig, ax = plt.figure(), plt.axes()
x = np.linspace(0,2*np.pi,100)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.title('Hello')
```

```

# Inspect the updated rcParams
#print(mpl.rcParams)
```

tiny-style-ex.py



You can also just save direct modifications to the rcParams dictionary and run that before plotting.

```
import matplotlib as mpl
mpl.rcParams['axes.spines.left'] = True
mpl.rcParams['axes.spines.right'] = False
mpl.rcParams['axes.spines.bottom'] = True
mpl.rcParams['axes.spines.top'] = False
mpl.rcParams['axes.titlesize'] = 25
mpl.rcParams['xtick.labelsize'] = 'large'
mpl.rcParams['font.family'] = 'Times'
```

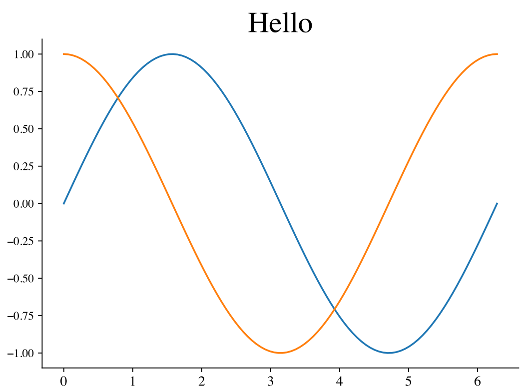
style-changes.py

Then add this code to ahead of creating your plot. I saved the above as `style_changes.py` and below I use the Jupyter `%run` magic command to run `style_changes.py` without having to copy and paste.

```
plt.style.use('default')
%run ../../python/style-changes.py

x = np.linspace(0,2*np.pi,100)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.title('Hello')
```

py-styled.py

The result is



8.2.1 Temporary Configurations

With some creativity, you can also avoid modifying rcParams by defining a function or writing a Python file that makes certain standardized plot modifications and then run that after you've created your figure and axes objects, using the IPython `%run` magic command.

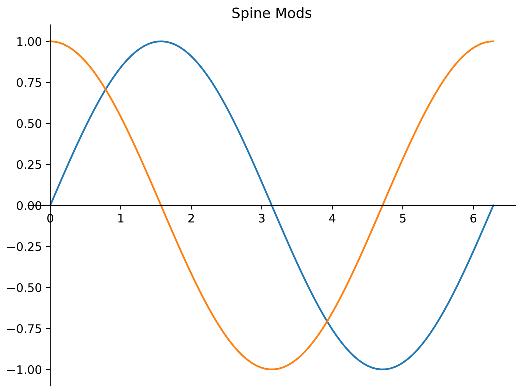
We save the following as `spine-mod.py` and then use it below to modify a plot.

```
import matplotlib.pyplot as plt

plt.gca().spines['left'].set_position('zero')
plt.gca().spines['bottom'].set_position('zero')

plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
```

spine-mod.py

```
x = np.linspace(0,2*np.pi,100)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.title('Spine Mods')

%run ../../python/spine-mod.py
```

spine-mod-ex.py



In the plot program, we use pyplot functions instead of using the OOP approach. Note that even if we created an `ax` variable, we must still use `plt.gca()` instead of `ax` in the file, because there is no `ax` to reference in `spine_mod.py`[2]. We can instead use `%run -i` to let the file access our global variables. Also, `%run -i` eliminates the need to import matplotlib again—we could delete the import statement from `spine-mod.py` and use `%run -i spine-mod.py`.

We save the following file as `spine-mod2.py` and can create the same plot with the program further below.

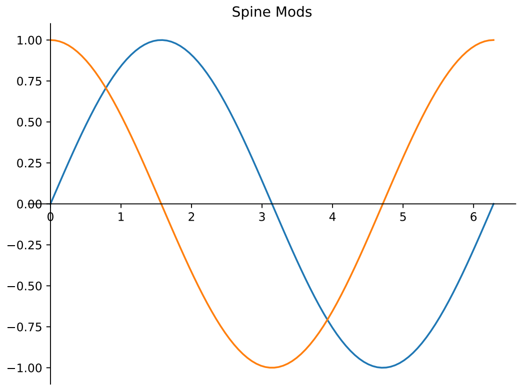
```
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
```

spine-mod2.py

```
x = np.linspace(0, 2*np.pi, 100)
fig, ax = plt.figure(), plt.axes()
ax.plot(x, np.sin(x))
ax.plot(x, np.cos(x))
ax.set_title('Spine Mods')
%run -i ../../python/spine-mod2.py
```

spine-mod2-ex.py

[2] Variables outside the file are not in a shared *scope* because `%run` creates a new *namespace* for variables inside the file)



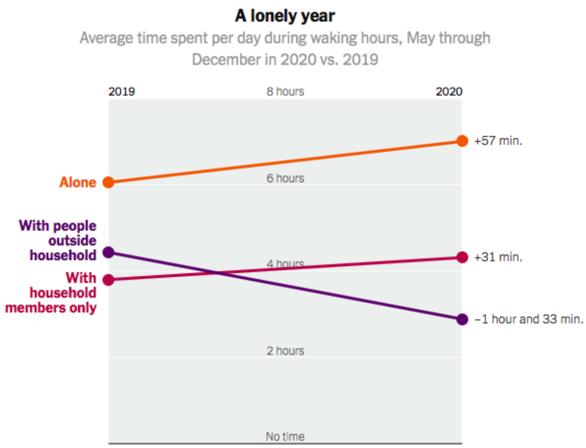
A further complication arises if our figure contains multiple subplots. In this case, we can access all the axes objects as an attribute of the figure object.

```
for ax in fig.axes:
    ax.spines['left'].set_position('zero')
    ax.spines['bottom'].set_position('zero')
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
```

spine-mod3.py

8.3 A Final Prose Example

In this section, we'll integrate much of what we've learned so far to create a line chart that, though simple, is far from the default. We'll imitate a chart from the New York Times article, *The Pandemic Changed How We Spent Our Time*, by Ben Casselman and Ella Koeze. Below is the original.



Imitating this graphic will take a lot of code, as demonstrated in Section 8.3.1. In Section 8.3.2, we invest in reconfiguring the style and creating functions that help reduce the tedium that would otherwise be required to make several plots of this style.

8.3.1 A First Go

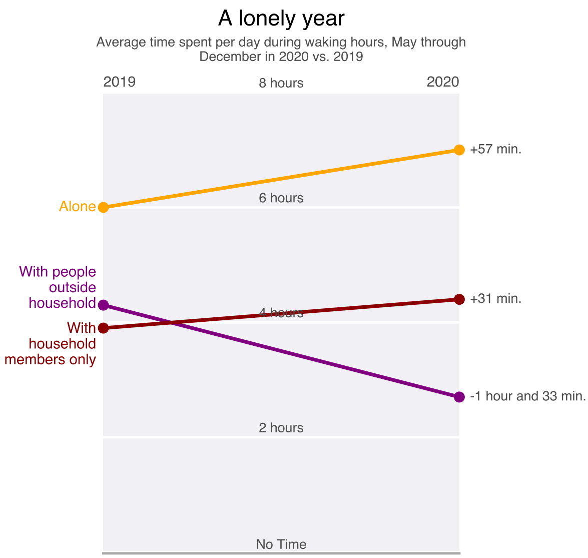
The program below would be even longer if not for the use of dictionaries like `plot_style`, which pairs pairs keyword arguments and specific values for the `plot()` method. This can be passed to `plot()` after being unpacked with the `**` operator.

```
ig = plt.figure(figsize = (7,6.5))
ax = plt.axes(facecolor = (.94, .94, .96))

## Add data and annotations
plot_style = dict(marker = 'o',
        clip_on = False, # don't clip markers at axes
    boundary
        linewidth = 3,
        markersize = 8)
right_text_style = dict(ha = 'left', fontsize = 11,
                      fontname = 'Helvetica',
                      color = (.3,.3,.3),
                      va = 'center')
left_text_style = dict(ha = 'right',
                      fontsize = 12,
                      fontweight = 'bold',
                      fontname = 'Helvetica')
x_vals = [2019,2020]
# alone
col = 'orange'
```

```
ax.plot(x_vals, [6,7],
        color = col,
        **plot_style)
ax.text(2019 - .02, 6, 'Alone',
        color = col,
        va = 'center',
        **left_text_style)
ax.text(2020.03, 7, '+57 min.',
        **right_text_style)

# within household
col = 'purple'
ax.plot(x_vals, [4.3,2.7],
        color = col,
        **plot_style)
ax.text(2019 - .02, 4.2, 'With people\noutside\nhousehold',
        color = col,
        va = 'bottom',
        **left_text_style)
ax.text(2020.03, 2.7, '-1 hour and 33 min.',
        **right_text_style)

# outside household
col = 'darkred'
ax.plot(x_vals, [3.9,4.4], color = col,
        **plot_style)
ax.text(2019 - .02, 4, 'With\nhousehold\nmembers only', color = col,
        va = 'top',
       **left_text_style)

ax.text(2020.03, 4.4, '+31 min.',
        **right_text_style)


# Label Gridlines
text_style = dict(color = (.3,.3,.3),
                  va = 'bottom',
                  ha = 'center',
                  fontname = 'Helvetica',
                  fontsize = 11)

ax.text(2019.5, 0.01, "No Time", **text_style)
for y in [2,4,6,8]:
    ax.text(2019.5, y+.04, "{} hours".format(y),
            **text_style)

# Label 2020/2021 for x-axis
year_text_style = dict(color = (.3,.3,.3),
                  va = 'bottom',
                  fontname = 'Helvetica',
                  fontsize = 12)
```

```
ax.text(2019, 8.05, '2019',
                ha = 'left', **year_text_style)
ax.text(2020, 8.05, '2020',
                ha = 'right', **year_text_style)

# set main title
ax.set_title("A lonely year", pad = 55, fontweight = '
    bold',
             fontname = 'Helvetica',
            fontsize = 18)

# set subtitle
s = 'Average time spent per day during waking hours, May
     through\nDecember in 2020 vs. 2019'
ax.text(2019.5, 8.5, s, **text_style)

# x axis
ax.set_xticks([])
ax.set_xlim(2019,2020)

# y axis
ax.yaxis.grid(True, color = 'white')
ax.set_yticks([0,2,4,6,8])
ax.set_yticklabels([])
ax.set_ylim([-.02,8])
ax.yaxis.set_tick_params(length = 0, grid_linewidth = 2)

# spines
for i in ['top','left','right']:
    ax.spines[i].set_visible(False)
ax.spines['bottom'].set_color('darkgray')
ax.spines['bottom'].set_linewidth(2)
```

nyt-rep1.py



8.3.2 Reconfigured, Refactored, and Reusable

```
axes.linewidth: 2
axes.facecolor: (.94, .94, .96)
axes.grid.axis: y
axes.grid: True
grid.color: white
grid.linewidth: 2
axes.spines.bottom: True
axes.spines.top: False
axes.spines.left: False
axes.spines.right: False
axes.edgecolor: darkgray
xtick.bottom: False
xtick.top: False
ytick.left: False
xtick.labeltop: True
xtick.labelbottom : False
ytick.labelleft: False
xtick.color: (.3,.3,.3)
text.color: (.3,.3,.3)
font.size: 12
lines.marker: o
lines.markersize: 8
lines.linewidth: 3
axes.titlesize: 18
axes.titleweight: bold
```

```
axes.formatter.useoffset: False
```

nyt-helper.mplstyle

```
def title_and_subtitle(title, subtitle = '', pad = 0.01,
     fig = None, ax = None):
    """Add a centered title and subtitle to a plot."""
    if ax == None:
        ax = plt.gca()
    if fig == None:
        fig = plt.gcf()
    fig.canvas.draw()

    top_of_figure = 1 # axes coords
    # update if there are xticks on the top
    tick0 = ax.get_xticklabels()[0]
    top_of_ticklabels = tick0.get_window_extent().
    transformed(ax.transAxes.inverted()).y1
    top_of_figure = max([top_of_ticklabels,top_of_figure
    ])

    # Add subtitle
    if subtitle:
        subt = ax.text(0.5, top_of_figure + pad,
                       s = subtitle,
                       ha = 'center',
                       va = 'bottom',
                       size = '11',
                       fontname = 'Helvetica',
                       transform = ax.transAxes)
        # update top of figure to top of the subtitle
        top_of_figure = subt.get_window_extent().
    transformed(ax.transAxes.inverted()).y1

    # add title
    ax.text(0.5, top_of_figure + pad,
            s = title,
            ha = 'center',
            va = 'bottom',
            size = '18',
            fontname = 'Helvetica',
            fontweight = 'bold',
            transform = ax.transAxes,
            color = 'black')
```

nyt-helper-function.

```
data = {2019: [6, 4.3, 3.9],
        2020: [7, 2.7, 4.4]}
df = pd.DataFrame(data = data,
                  index = ['Alone',
                           'With people outside household'
    ,
```

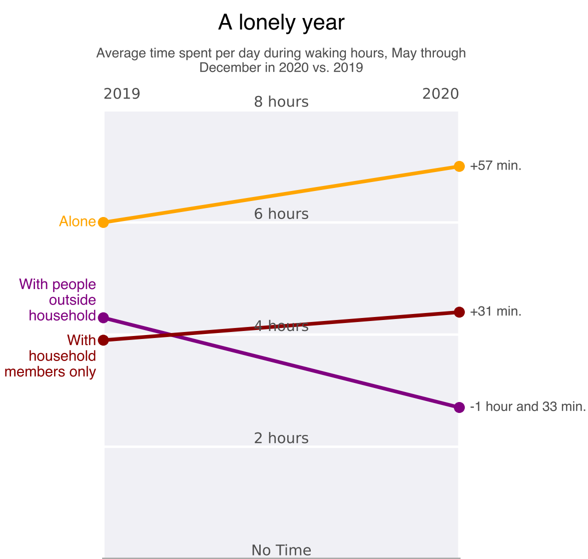
```
                                    'With hosehould members only'])
```

nyt-helper-data.py

```
with plt.style.context("../../stylelib/nyt-helper.mplstyle"):

    fig = plt.figure(figsize = (7,6.5))
    ax = plt.axes(facecolor = (.94, .94, .96))

    # plot the data
    colors = ['orange', 'purple', 'darkred']
    df.T.plot(ax = ax, clip_on = False, color = colors)

    # annotate lines
    right_text_style = dict(ha = 'left',
                            fontsize = 11,
                            fontname = 'Helvetica',
                            va = 'center')
    left_text_style = dict(ha = 'right',
                           fontsize = 12,
                           fontweight = 'bold',
                           fontname = 'Helvetica')

    # alone
    ax.text(2019 - .02, 6, 'Alone',
        color = colors[0],
        va = 'center',
        **left_text_style)
    ax.text(2020.03, 7, '+57 min.',
        **right_text_style)

    # Within household
    ax.text(2019 - .02, 4.2, 'With people\noutside\nhousehold',
            color = colors[1],
            va = 'bottom',
            **left_text_style)
    ax.text(2020.03, 2.7, '-1 hour and 33 min.',
            **right_text_style)

    # Outside household
    ax.text(2019 - .02, 4, 'With\nhousehold\nmembers only',
            color = colors[2],
            va = 'top',
            **left_text_style)
    ax.text(2020.03, 4.4, '+31 min.',
            **right_text_style)


    # Title and sub
    t = 'A lonely year'
```

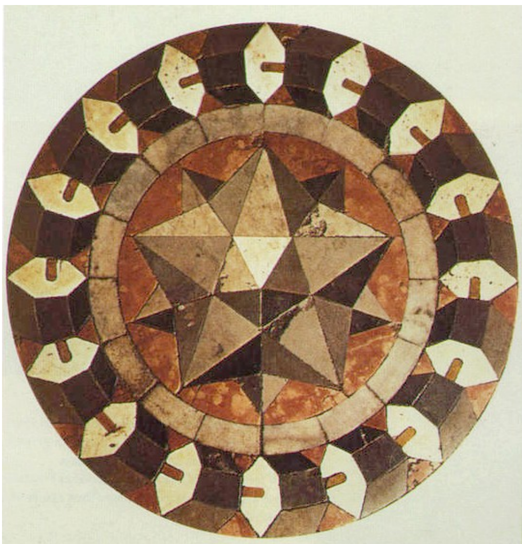
```python
    s = 'Average time spent per day during waking hours, May through\nDecember in 2020 vs. 2019'
    title_and_subtitle(t,s, fig = fig, ax = ax, pad = .03)

    # Clean ticks etc
    yticks = range(0,9,2)
    ax.set_yticks(yticks)
    ax.set_ylim(0,8)
    # add custom labels for y ticks
    for tick in yticks:
        label = "{} hours".format(tick)
        if tick == 0:
            label = 'No Time' # custom label
        ax.text(2019.5, tick + .01, label,
               va = 'bottom',
               ha = 'center')

    # move x-tick labels to horizontally align
    ax.legend().set_visible(False)
    x_vals = [2019,2020]
    ax.set_xticks(x_vals)
    ax.set_xlim(x_vals)
    xticks = ax.get_xticklabels()
    xticks[0].set_ha('left')
    xticks[-1].set_ha('right')
```

nyt-refactor.py



Part II

Mathematical Interlude



St Mark's Basilica, Floor mosaic by Paolo Uccello (Public Domain)

Chapter 9

Math

In Part III, we'll begin to treat Matplotlib more like a blank canvas. The complexity can evolve any number of ways, and one key complexity is the placement of items in a plot. Doing that well means understanding angles. So this math interlude guides us through trigonometry and some light linear algebra. Some pieces of this chapter are unnecessary. `plt.Circle()` can be used to create a circle without any knowledge of trigonometry. Instead, we plot circles the old-fashioned way. We create a lot of points that, when connected, form a circle.

Why bother? Indeed, your Python interpreter won't be impressed if you know trigonometry. We shouldn't bother in every case, but math can compensate for a lack of matplotlib knowledge. I'd rather know a lot of math and a little matplotlib than a little math and a lot of matplotlib. Math is durable knowledge, useful in non-plotting contexts. A deeper understanding is also what allows us to create the color gradient in Section 10.2, which can't be fashioned with a simple call to `plt.Circle()`.

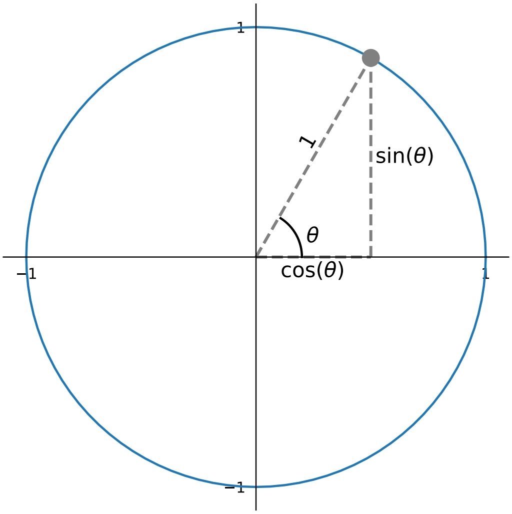
9.1 Circles

9.1.1 The Unit Circle

The unit circle is the gem of pre-calculus. Understanding it is useful for plotting circles or arcs by hand (though one can also use Circle or Arc objects). It tells us how to relate angles to a particular point in the xy-plane. For a point on the unit circle at angle θ from the origin, we can find its coordinates as

$$(x, y) = (\cos(\theta), \sin(\theta)) \, .$$

Tracing a line from the origin to the point on the circle, we can create a right triangle as shown below.




```
angles = np.linspace(0, 2*np.pi, 101)
x = np.cos(angles)
y = np.sin(angles)

fig, ax = plt.figure(figsize = (5,5)), plt.axes()
ax.set_aspect('equal')

# Make circle
ax.plot(x,y)

# Plot example right triangle
angle = np.pi/3

# make hypotenuse
ax.plot([0,np.cos(angle)], [0,np.sin(angle)],
        linestyle = 'dashed', color ='gray', linewidth =
    2)#

# mark point on circle
ax.plot([np.cos(angle)], [np.sin(angle)],
        marker = 'o', color ='gray', markersize = 11)

```

```
# dashed lines for opposite and adjacent
ax.plot([0,np.cos(angle)], [0,0],
        linestyle = 'dashed', color ='gray', linewidth = 2)
ax.plot([np.cos(angle),np.cos(angle)], [0,np.sin(angle)],
        linestyle = 'dashed', color ='gray', linewidth = 2)

# Triangle side lengths
fontsize = 14
ax.text(0.5*np.cos(angle) - .02, 0.5*np.sin(angle)+.02,
        '1', rotation = math.degrees(angle), ha = 'center', va = 'bottom', size = fontsize)
ax.text(0.5*np.cos(angle), -.02, r"$\cos(\theta)$",
        rotation = 0, ha = 'center', va = 'top', size = fontsize)
ax.text(np.cos(angle) + .02, 0.5*np.sin(angle), r"$\sin(\theta)$",
        rotation = 0, ha = 'left', va = 'center', size = fontsize)


# make small arc and mark angle
x = np.cos(angles[angles<= angle])
y = np.sin(angles[angles<= angle])
ax.plot(0.2*x,0.2*y, color = 'black')
ax.text(0.2*np.cos(np.pi/10), 0.2*np.sin(np.pi/10),
        r" $\theta$", size = 14)

# clean appearance
%run ../../python/spine-mod.py
ax.set_xticks([-1, 1])
ax.set_yticks([-1, 1])
```

unit-circle.py

We can plot a circle or an arc from θ_1 to θ_2, by connecting the points $(\cos(\theta_1), \sin(\theta_1)), \ldots, (\cos(\theta_2), \sin(\theta_2))$, where enough intermediate angles between θ_1 and θ_2 are included so the piecewise-linearity is smoothed out to give the appearance of a curve. In Subsection 9.1.2, we consider how to do the same, but for non-unit circles.

9.1.2 Non-unit Circles

The unit circle has a radius of one and it's centered at the origin. How do we obtain coordinates for other circles? There are two steps to change the radius and shift a circle off the origin.

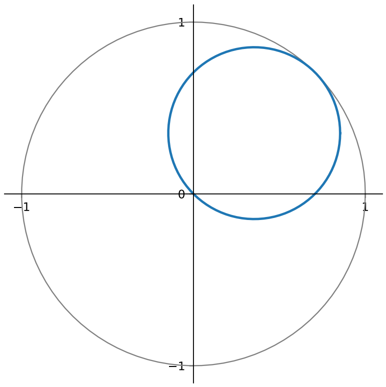
1. **Change the radius.** Multiply the coordinates by the desired radius r.

2. **Shift the circle.** Add the desired horizontal and vertical shifts to the x and y coordinates, respectively.

These are ordered because the radius multiplier should not be applied to the added shift term. Below, we shrink the unit circle and move it up and along the 45-degree line.

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

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

# Unit Circle
x = np.cos(angles)
y = np.sin(angles)
ax.plot(x, y, color = 'gray', linewidth = 1)

# Shifted
new_radius = 0.5
new_center = np.cos(np.pi/4)/2, np.sin(np.pi/4)/2
shift_x = new_radius*x + new_center[0]
shift_y = new_radius*y + new_center[1]
ax.plot(shift_x, shift_y, linewidth = 2)

%run ../../python/spine-mod.py

ax.set_xticks([-1, 1])
ax.set_yticks([-1, 0, 1])
```

unit-circle-shift.py



9.1.3 Rotations and Ellipses

Now we jump from trigonometry to linear algebra. Matrices can represent transformations, like rotations or stretching. Applied to each point in a circle, a rotation that stretches x and y coordinates differently creates an ellipse.

A rotation of angle θ can be represented as

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Stretching the x-dimension by a scalar r can be represented with

$$\begin{bmatrix} r & 0 \\ 0 & 1 \end{bmatrix},$$

and the y-dimension is stretched by

$$\begin{bmatrix} 1 & 0 \\ 0 & r \end{bmatrix}.$$

Each of these matrices is applied point by point by left multiplying that point (as a 2×1 column vector) by the transformation matrix,

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = T \begin{pmatrix} x \\ y \end{pmatrix}.$$

Below we take a circle and shrink it horizontally, stretch it vertically, and then rotate it. The x values are multiplied by $\frac{1}{2}$, the y values are multiplied by 2, and the angle of rotation is 45 degrees ($\frac{\pi}{4}$ radians). The transformation is constructed below.

```
theta = np.pi / 4
rotation_matrix = np.matrix([[np.cos(theta), -np.sin(
    theta)], [np.sin(theta), np.cos(theta)]])

x_scale = 0.5
x_stretch = np.matrix([[x_scale, 0], [0, 1]])

y_scale = 2
y_stretch = np.matrix([[1, 0], [0, y_scale]])

transformation = rotation_matrix * y_stretch * x_stretch
```

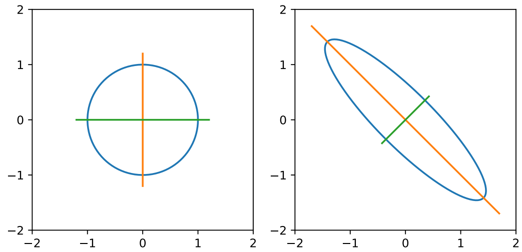
tform-matrix.py

Below we plot a unit circle and then apply the transformation to create an ellipse.

```python
# Create a circle of points
angles = np.linspace(0, 2*np.pi, 100)
x_vals = np.cos(angles)
y_vals = np.sin(angles)

# Begin plot
fig, ax = plt.subplots(1,2)

# simplify axes names
ax0, ax1 = ax[0], ax[1]

# Plot a circle
ax0.plot(x_vals, y_vals)

# Mark the y and x directions/axes
# vertical axis
height = 1.2
p1 = np.array([0,-height])
p2 = np.array([0,height])
points = [p1,p2]
x_vertical = [p[0] for p in points]
y_vertical = [p[1] for p in points]
ax0.plot(x_vertical, y_vertical)

# horizontal axis
width = height
p1 = np.array([height,0])
p2 = np.array([-height,0])
points = [p1,p2]
x_horiz = [p[0] for p in points]
y_horiz = [p[1] for p in points]
ax0.plot(x_horiz, y_horiz)

# Make Ellipse
new_points = [transformation * np.matrix(p).T for p in
    zip(x_vals,y_vals)]

new_x = [np.array(x).flatten()[0] for x in new_points]
new_y = [np.array(x).flatten()[1] for x in new_points]

# new vertical axis
new_vertical = [transformation * np.matrix(p).T for p in
     zip(x_vertical, y_vertical)]
new_x_vertical = [np.array(x).flatten()[0] for x in
    new_vertical]
new_y_vertical = [np.array(x).flatten()[1] for x in
    new_vertical]

# new horizontal axis
new_horiz = [transformation * np.matrix(p).T for p in
    zip(x_horiz, y_horiz)]
new_x_horiz = [np.array(x).flatten()[0] for x in
    new_horiz]
```

```
new_y_horiz = [np.array(x).flatten()[1] for x in
    new_horiz]

# Plot ellipse etc
ax1.plot(new_x, new_y)
ax1.plot(new_x_vertical, new_y_vertical)
ax1.plot(new_x_horiz, new_y_horiz)

# Change axes appearance
args = -2,2
for ax_ in ax0, ax1:
    ax_.set_xlim(args)
    ax_.set_ylim(args)
    ax_.set_xticks(np.linspace(*args,5))
    ax_.set_yticks(np.linspace(*args,5))
ax0.set_aspect('equal')
ax1.set_aspect('equal')
```

ellipse-tform.py



9.2 Right Triangles

Right triangles are important to understand not for plotting right triangles necessarily, but for understanding the angle between any two points. The line segment connecting two points forms the hypotenuse of a right triangle, just as was seen in the unit circle.

For any angle θ in a right triangle that is not the right angle itself, we can speak of the sides opposite or adjacent to the angle. The side opposite is the side directly across from the angle. The side opposite the right angle is the hypotenuse (of length c in Pythogoras' Theorem). The SOHCAHTOA mnemonic helps us understand how side lengths are related to the angles. More clearly written as SOH-CAH-TOA, as it stands for Sine Opposite Hypotenuse Cosine Adjacent Hypotenuse Tangent Opposite Adjacent and means

$$\sin\theta = \frac{\text{opposite}}{\text{hypotenuse}}$$
$$\cos\theta = \frac{\text{adjacent}}{\text{hypotenuse}}$$
$$\tan\theta = \frac{\text{opposite}}{\text{adjacent}},$$

where θ is some angle of a triangle in radians and opposite, adjacent, and hypotenuse refer to the lengths of these sides.

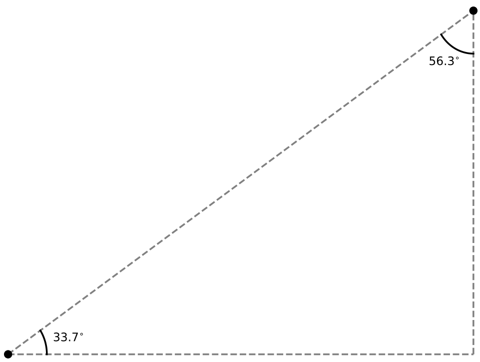
By understanding these functions and their inverses, we can recover the angles in a plot. These functions are available from the `math` module as `sin()`, `cos()`, and `tan()`. Their inverses are `asin()`, `acos()`, and `atan()` for arcsin, arccos, and arctan.

`math.atan()` is the most useful. Take two points and the slope m of the line connecting them. Then $\arctan(m) = \theta$ is angle between those points, in radians.

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

a = (1,2)
b = (7,6)

# rise over run
slope = (a[1] - b[1]) / (a[0] - b[0])
angle = math.atan(slope) # radians
degrees = math.degrees(angle)

top_angle = math

## add angle semi-circle
x = np.linspace(0, angle, 100)
ax.plot(0.5 * np.cos(x) + a[0],
        0.5 * np.sin(x) + a[1],
        color = 'black')
ax.text(0.5*np.cos(angle/2) + 1.1, 0.5*np.sin(angle/2) +
    2,
       s = r"${:.1f}".format(degrees) + r"^{\circ}$")

# top slope measured relative to a 90-deg rotation
top_slope = (b[0]-a[0])/(b[1]-a[1])
top_angle = math.atan(top_slope)
x = np.linspace(1.5*np.pi, 1.5*np.pi - top_angle, 100)
ax.plot(0.5*np.cos(x) + b[0],
        0.5*np.sin(x) + b[1],
```

```
          color = 'black')
label_angle = 1.5*np.pi - top_angle/2
ax.text(0.5*np.cos(label_angle) + b[0] - 0.13, 0.5*np.
    sin(label_angle) + b[1] - 0.2,
         s = r"${:.1f}".format(math.degrees(top_angle)) +
     r"^{\circ}$",
        ha = 'center')


# points on left and top
ax.plot([a[0], b[0]], [a[1], b[1]], linestyle = '',
    marker = 'o', color = 'black')

# make a right triangle
ax.plot([a[0], b[0]], [a[1], b[1]], linestyle = 'dashed'
    , marker = 'o', color = 'gray', zorder = -1)
ax.plot([a[0], b[0]], [a[1], a[1]], linestyle = 'dashed'
    , color = 'gray', zorder = -1)
ax.plot([b[0], b[0]], [a[1], b[1]], linestyle = 'dashed'
    , color = 'gray', zorder = -1)
ax.axis('off')
```

r-triangle.py



Chapter 10

Applications

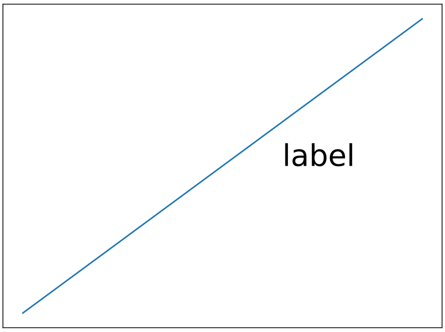
10.1 Sloping Text

Suppose you'd like to label a line.

For a sloped line, you might rather the text sit parallel to the line instead of suffering the below.

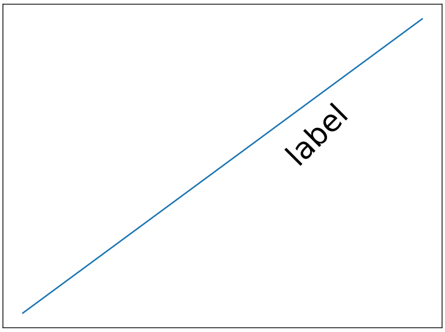
```
plt.plot([0,1], [0,1])
plt.text(0.65, 0.5,
         s = 'label',
         size = 30)

ax = plt.gca()
# Cosmetics
ax.grid(False)
ax.set_xticks([])
ax.set_yticks([])
```

no-slope.py



The `rotation` argument can help if you know the right angle in degrees. Here the angle is 45 degrees or $\frac{\pi}{4}$ radians. So we modify the second line to be `plt.text(0.65, 0.5, 'label', size = 30, rotation = 45)`.

But this doesn't do what we want! The plot coordinate system is stretched, because we didn't call `ax.set_aspect('equal')` and `text` doesn't recalculate the text angle to make it align.

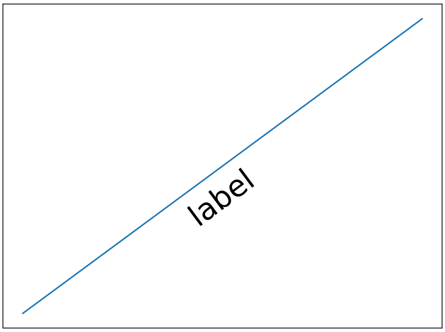


Now let's solve it for good in the general case, using trigonometry and then `transform_angles`. This is a method that we'll use with the transformation `ax.transData`. Try experimenting by replacing the `x2,y2` values to see this works for any angle.

```
x1, y1 = 0, 0
x2, y2 = 1, 1
x = (x1, x2)
y = (y1, y2)

# plot
fig, ax = plt.figure(), plt.axes()
ax.plot(x,y)

# Find angles and then insert text
slope = (y2 - y1) / (x2 - x1)
true_angle = math.degrees(math.atan(slope))

# dummy_array is the point where the angles are anchored
dummy_array = np.array([[0,0]]) # doesn't matter what
    pair you use
# matplotlib.org/stable/api/transformations.html#
    matplotlib.transforms.Transform.transform_angles

plot_angle = ax.transData.transform_angles(
                            np.array((true_angle,)),
                            dummy_array)[0]

```

```
ax.text(np.mean(x), np.mean(y),
        s = 'label',
        rotation = plot_angle,
        fontsize = 30,
        va = 'top',
        ha = 'center')

ax.grid(False)
ax.set_xticks([])
ax.set_yticks([])
print(true_angle, plot_angle)
```

slope-label.py



10.2 Circular Arrangements

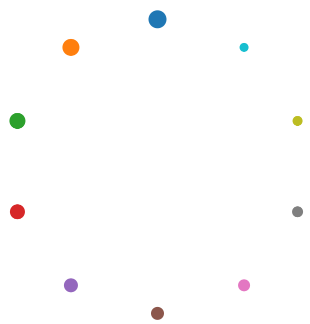
With our knowledge of the unit circle, we can arrange some points in a circle with no additional help beyond the math package. This might be useful if you want to avoid mixing polar and Cartesian axes.

```
n_points = 10
pie_angle = 360/n_points # angle of each slice
starting_angle = 90

fig, ax = plt.subplots()

for i in range(n_points):

    angle = starting_angle + i*pie_angle
    angle = math.radians(angle)
    x = math.cos(angle)
    y = math.sin(angle)

    ax.plot([x],[y], 'o', markersize = 17 - i)
```

```

ax.set_aspect('equal')
ax.axis('off')
```

circle.py

This code produces the following.

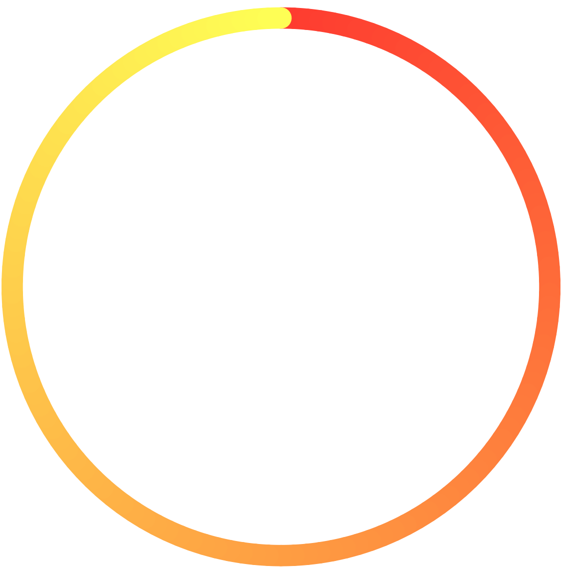


The below makes similar use of trigonometry to create a circle colored according to a gradient, like in Chapter 6. I make use of `solid_capstyle = 'round'` to round the endpoints of the plotted line, creating a cleaner look compared to the default.

```
# make a circle gradient
start_color = 255/256, 59/256, 48/256 # red
end_color = 255/256, 255/256, 85/256 # yellow

# How many color changes
segments = 130

# Create figure
fig, ax = plt.figure(figsize = (8,8)), plt.axes()

# Start at 90 degrees and return clockwise
angles = np.linspace(2.5*np.pi, np.pi/2, segments + 1)

# Create the intermediate colors
colors = dict()
for i in range(3):
    colors[i] = np.linspace(start_color[i], end_color[i
    ], segments)

# plot each arc
for i in range(segments):

    start_angle = angles[i]
    end_angle = angles[i+1]
```

```
    angle_slice = np.linspace(start_angle, end_angle, 100)

    x_values = np.cos(angle_slice)
    y_values = np.sin(angle_slice)

    rgb = colors[0][i], colors[1][i], colors[2][i]

    ax.plot(x_values, y_values,
            color = rgb,
            linewidth = 20,
            solid_capstyle = 'round')

ax.set_aspect('equal')
ax.axis('off')
```

circle-grad.py



10.3 Network Graphs

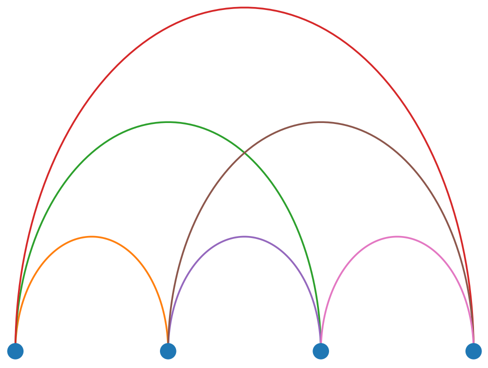
Networks are represented mathematically as graphs—a set of vertices and edges between them. In drawing a graph, there are many drawing algorithms available. For large networks or sophisticated algorithms, you should use something off the shelf in a package like nxviz. For a small network, you might avoid dealing with NetworkX and nxviz and do the drawing yourself. We will work through two simple layouts: arc diagrams and a circular layout for an undirected graph.

An arc diagram places all points on a straight line. The links are drawn as arcs from one point to another.

Let's consider the complete graph with four vertices, where every pair is connected.

```
fig, ax = plt.figure(), plt.axes()
x = np.linspace(0,1,4)
ax.plot(x, np.zeros(4),
        marker = 'o',
        linestyle = '',
        markersize = 13)

angles = np.linspace(0,np.pi,100)
for point in x:
    # connect other points
    other_x = x[x > point]
    # construct a half circle
    unit_x, unit_y = np.cos(angles), np.sin(angles)
    for other in other_x:
        # arc is centered between the two points
        shift = np.mean([point,other])
        r = (other - point)/2
        new_x = r*unit_x + shift
        new_y = r*unit_y
        ax.plot(new_x, new_y, zorder = -1)

ax.axis('off')
ax.set_aspect(1.5)
```

arc-graph.py

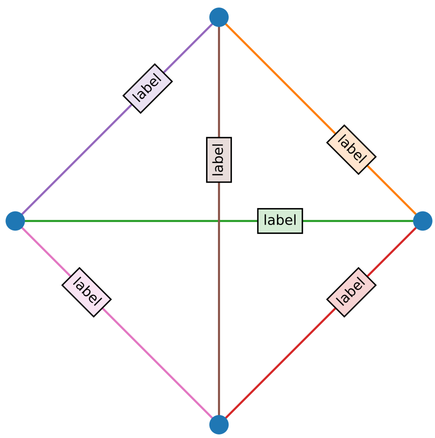


Next we move on to a circular layout. This layout places each vertex along a circle. Spaced evenly and with just four vertices in our graph, this will in fact produce a square. We also label each edge.

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

n_points = 4

# Draw vertices
angles = np.linspace(0, 2*np.pi, n_points + 1)[0:
    n_points]
x = np.cos(angles)
y = np.sin(angles)
ax.plot(x, y,
        marker = 'o',
        linestyle = '',
        markersize = 13)

# Draw Edges
points = [p for p in zip(x,y)]
counter = 1
for point, other in combinations(points,2):

    x = [p[0] for p in (point, other)]
    y = [p[1] for p in (point, other)]
    ax.plot(x, y, zorder = -1)

    # add a label
    label_point = .65*np.array(point) + .35*np.array(
    other)

    run = x[1]-x[0]
    rotation = 90
    ha = 'left'
    if run != 0:
        line_slope = (y[1]-y[0])/(x[1]-x[0])
        rotation = math.atan(line_slope)
        rotation = math.degrees(rotation)
        ha = 'center'
    else:
        print(point, other, rotation)

    # get rgb then blend with white
    line_color = mpl.colors.to_rgb("C"+str(counter))
    lighter = .8*np.ones(3) + .2*np.array(line_color)
    ax.text(label_point[0], label_point[1],
            'label', rotation = rotation,
            bbox = dict(facecolor = lighter),
            va = 'center',
            ha = 'center'
           )
```

```
    counter += 1

ax.axis('off')
ax.set_aspect('equal')
```

circle-graph.py



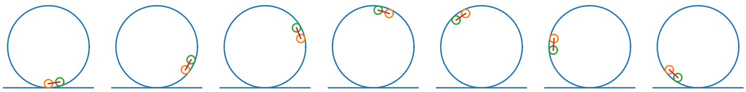
10.4 Tony Hawk's Vertical Loop

Tony Hawk became the first skateboarder to skate a vertical loop in 1998. We honor that accomplishment in two dimensions with the help of a rotation matrix. The unit circle is our vertical loop and we add two smaller circles to represent a skateboard. This is trigonometry. The small circles are placed along a ray from the origin of the unit circle to ensure they will lie tangent inside in the loop. In the first subplot, we place the skateboard at the bottom of the ramp. Though the same figure could be produced without using a rotation matrix, we use one so that the first subplot is essentially reused over and over by rotating the skateboard wheels up and around the loop.

```
thetas = np.linspace(0,2*np.pi,8)[0:-1]
fig = plt.figure(figsize = (12,3))

# Set radius for skateboard wheels
radius = 0.1

# Make individual subplots
```

```
for key, theta in enumerate(thetas):
    rotation_matrix = np.matrix([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])

    # Create panel for one frame
    ax = fig.add_subplot(1, len(thetas), key+1)
    ax.set_aspect('equal')

    # Plot the loop itself
    angles = np.linspace(0, 2*np.pi, 100)
    x = np.cos(angles)
    y = np.sin(angles)
    ax.plot(x,y)

    # Make skateboard wheels at bottom of the ramp
    # and then rotate them counter-clockwise according to theta
    centers = list()
    for ang in 1.5*np.pi, 1.6*np.pi:
        center = (1-radius)*np.cos(ang), (1-radius)*np.sin(ang)

        # rotate
        point = np.matrix(center).T
        rotated_point = rotation_matrix*point
        rotated_point = np.array(rotated_point).flatten()
        centers.append(rotated_point)

        # make wheel around new center
        wheel_x = radius*x + rotated_point[0]
        wheel_y = radius*y + rotated_point[1]

        ax.plot(wheel_x, wheel_y)

    # connect the two wheel centers
    c1, c2 = centers
    ax.plot([c1[0],c2[0]], [c1[1],c2[1]])

    ax.axis('off')

    xlim = ax.get_xlim()
    ax.plot(xlim, [-1,-1],
            color = 'C0',
            zorder = -1)
```

tony-hawk.py



Part III

Poetry

Newton by William Blake (Public Domain)

Chapter 11

Poetry

President George W. Bush is reported to have been fascinated by satellite pictures showing North and South Korea at night. South Korea is covered in bright lights and its communist neighbor, dark, could be mistaken for a collection of dead pixels. The pictures, like any data visualization, communicate and distill a lot of information. Karen Hughes, Bush's former counselor, said that to Bush, the pictures showed "the light and opportunity that comes with freedom, and the dark that comes with a regime that is oppressive." The pictures, like any data visualization, communicate and distill a lot of information. And like an effective novel data visualization, the pictures invite more active interest.

With prose behind us, we approach that end of the spectrum. Scruton 2015 argues that "poetry is concerned with the truth as a kind of revelation," standing apart from the "aboutness" of prose. Scruton adds, "When Keats writes his 'Ode to the Nightingale,' he does not describe the bird and its song only: he endows it with value." So here we are, trying to endow some data with value through its presentation.

This is a different kind of task you might take up once you've understood the important insights from your data and you have an editorial perspective. When the Bureau of Labor Statistics reports unemployment numbers, that should not be editorialized—prose line charts and tables will do. But sometimes your audience will benefit from receiving the data pre-chewed or more artfully presented.

To that end, this part aims to help you construct interesting charts, mostly with more considered use of Artist objects. This will also help you construct prosaic plots, as you might also use these objects to build plots from scratch if the easier way escapes you. This will be more laborious. Poets do write fewer words than prose writers over their careers.

Chapter 12

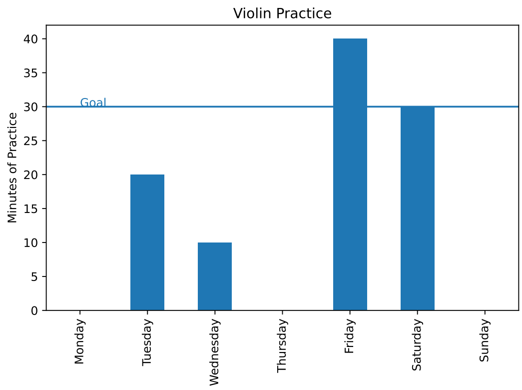
Applications

12.1 Activity Calendar

Let's start with a prose bar chart and turn it into poetry. Suppose you have a goal to practice the violin for at least 30 minutes every day and you've tracked your practice time for a whole week.

```
violin_practice = {'Monday': 0,
                'Tuesday': 20,
                'Wednesday': 10,
                'Thursday': 0,
                'Friday': 40,
                'Saturday': 30,
                'Sunday': 0}

pd.Series(violin_practice).plot.bar()
plt.axhline(30)
plt.ylabel('Minutes of Practice')
plt.text(0, 30, 'Goal', color = 'C0')
plt.title('Violin Practice')
```

violin-bar.py



This creates a chart that is perfectly fine. But a poet might say, why futz with all the numbers and obfuscate the essential meaning? We only need to know if we hit the goal of 30 minutes or not. This kind of simplification might be more motivating. There's probably some psychology research that supports this, but we'll proceed without deferring to any such authority because those studies so often fail to replicate. Let's create a simple activity calendar of sorts.

```
fig, ax = plt.figure(figsize = (8,5)), plt.axes()
spacing_scale = 2.5

for idx, day in enumerate(violin_practice):
    hit_target = violin_practice[day] >= 30

    facecolor = 'black' if hit_target else 'white'
    c = plt.Circle((idx*spacing_scale, 1),
                   radius = .4,
                   facecolor = facecolor,
                   edgecolor = 'black')
    ax.add_artist(c)
    ax.text(idx*spacing_scale, 0,
            s = day,
            ha = 'center',
            va = 'center')

# Eliminate Clutter
ax.set_aspect('equal')
ax.axis('off')

# Set plot window
ax.set_xlim([-1, 6*spacing_scale+1])
ax.set_ylim([0, 2])
```

violin-cal.py

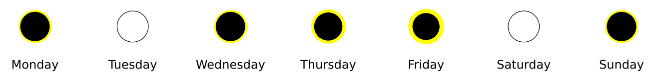
You might have in mind ways to spice up this plot even further. Maybe there should be partial credit given to days some with violin practice, but not at least 30 minutes. Perhaps on those days, the circles can be filled with a light gray. The key here is that we're creating a plot that we might think will be more motivating.

Apps like Duolingo or Peloton gamify the user experience, and part of that is rewarding activity streaks over consecutive days. Let's try to enhance the plot so that streaks stand out. We'll accomplish this by adding a yellow edge color and increasing its weight as a streak continues.

```
violin_practice = {'Monday': 30,
                   'Tuesday': 0,
                   'Wednesday': 30,
                   'Thursday': 30,
                   'Friday': 40,
                   'Saturday': 0,
                   'Sunday': 60}

fig, ax = plt.figure(figsize = (8,5)), plt.axes()
spacing_scale = 2.5

streak = 0
for idx, day in enumerate(violin_practice):

    hit_target = violin_practice[day] >= 30
    streak += 1*hit_target
    streak = streak if hit_target else 0

    facecolor = 'black' if hit_target else 'white'
    edgecolor = 'yellow' if hit_target else 'black'
    c = plt.Circle((idx*spacing_scale,1),
                   radius = .4,
```

```
                   facecolor = facecolor,
                   edgecolor = edgecolor,
                   linewidth = 0.5 + streak)
    ax.add_artist(c)
    ax.text(idx*spacing_scale, 0,
            s = day,
            ha = 'center',
            va = 'center')

# Eliminate Clutter
ax.set_aspect('equal')
ax.axis('off')

# Set plot window
ax.set_xlim([-1, 6*spacing_scale+1])
ax.set_ylim([0, 2])
```

violin-streak.py



Part IV

Special Topics

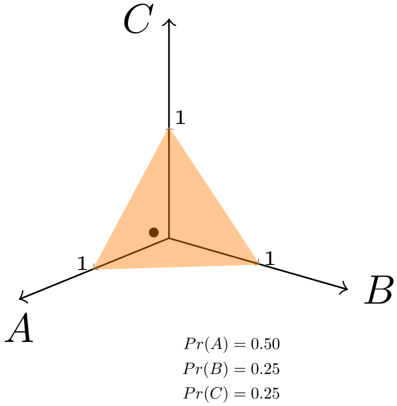
A Virgin with a Unicorn by Domenichino (Public Domain)

Chapter 13

Ternary Plots

13.1 Ternary

This section introduces the python-ternary package. You'll need to install this with pip or conda to follow along. You can use this to make various plots in the two-dimensional simplex. That is, you can make triangle plots where a point in the triangle represents a particular multinomial distribution over three possible outcomes. That is, the triangle is a two-dimensional projection of the space $\{(p_1, p_2, p_3) \in \mathbb{R}^3 : p_1 + p_2 + p_3 = 1 \text{ and } p_i \in [0,1] \text{ for } i = 1,2,3\}$.



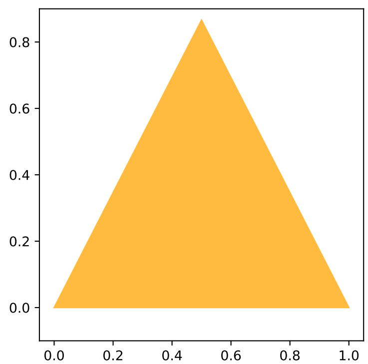
I encountered these diagrams in a few economics courses. Professor Bill Sandholm made particularly memorable use of these diagrams in his courses and research in evolutionary game theory.

These plots aren't the most natural selection for inclusion in this text. It's a personal indulgence and a favor to other game theorists.

After running `import ternary`, the construction of a plot isn't much different than what we covered in Chapter 1. Start with figure and *ternary* axes objects. However, `ternary.figure()` will create both objects. There is no analogue to `plt.axes()` as this more closely imitates `plt.subplots(1,1)` than `plt.figure()`. It's not a perfect replica though. For example, there is no `figsize` parameter, but this can be adjusted with the figure method `set_size_inches`.

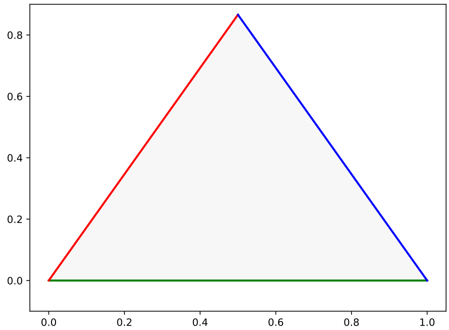
```
figure, tax = ternary.figure()# A tuple of plot objects
figure.set_size_inches(4,4)
tax.set_background_color('orange')
```

basic-ternary.py




```
# Create the Plot
scale = 1 # length of the sides
figure, tax = ternary.figure(scale=scale)

# Draw Boundary
tax.boundary(linewidth= 2.0,
             axes_colors = {'l':'red', 'r': 'blue', 'b':
    'green'})
```

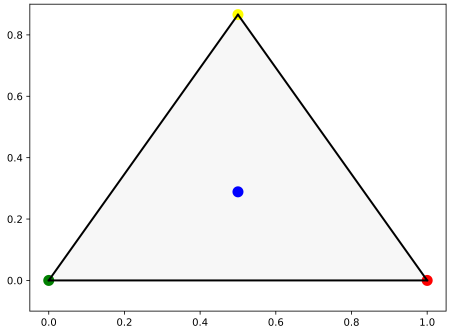
color-ternary.py



Next, let's add some points with the `scatter` method, which works as you might expect.

```
scale = 1
figure, tax = ternary.figure(scale=scale)

# Draw Boundary
tax.boundary(linewidth= 2.0)

# Scatter Points
points = [(1,0,0), (0,1,0), (0,0,1), (1/3, 1/3, 1/3)]
tax.scatter(points,
            marker = 'o',
            color = ['red', 'yellow', 'green', 'blue'],
            s = 100)
```

scatter-ternary.py

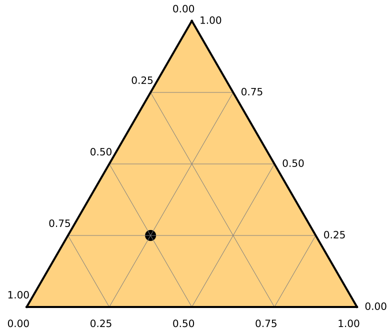


The x- and y-axis ticks above correspond to the bottom and right axes, but this can be confusing since the x, y point of $(0, 0)$ is the point $(0, 0, 1)$ in the ternary plot. Accordingly, you might prettify the plot by removing those ticks and these axes entirely.

This can be done with `tax.get_axes().axis('off')`. This works like `ax.axis` so we can also pass `'equal'` to equalize the axes. Adding gridlines with the `gridlines` method can also help the eye. This is demonstrated below. The horizontal gridlines correspond to the value along the right axis. The negatively sloped gridlines correspond to the left axis. The positively sloped gridlines correspond to the bottom axis. These gridlines are perpendicular to the direction of ascent along each axis and, just as for a typical plot, they show where the particular axis value is constant.

```
scale = 1
figure, tax = ternary.figure(scale=scale)
tax.set_background_color('orange',
                         alpha = 0.5)

# Add ticks along the triangle edges
tax.ticks(axis = 'lbr',
          multiple = .25,
          tick_formats = '%.2f',
          offset= 0.02,
          linewidth = 0)

tax.gridlines(multiple = .25,
              color = 'gray',
              linewidth = 0.5,
              linestyle = 'solid')

points = [(0.25, 0.25, 0.5)]
tax.scatter(points,
            marker = 'o',
            color = ['black'],
            s = 100)

tax.boundary(linewidth= 2.0)
tax.get_axes().axis('equal')
tax.get_axes().axis('off')
```

grid-ternary.py



13.2 Application: Rock, Paper, Scissors

Throughout this chapter, we'll analyze the game rock paper scissors. If you need a reminder, there are two players who simultaneously choose an action of either rock, paper, or scissors. Rock beats scissors beats paper beats rock. Choosing the same action results in a tie. Your job is to choose an action based on your expectation of what your opponent will choose.

First, we'll construct a heatmap to show the net winning percentage from choosing a particular action depending on the opponent's probability distribution over the three actions, so that a point in the simplex is that opponent's strategy and the color represents how often you win. The following is a function we'll use in making a heatmap, calculating the net winning percentage of an action against a particular distribution.

```
def winning_pct(p, action = 'rock'):

    """What is the net winning percentage given a choice
     of action and an opponent's strategy p, where p is
    a probability distribution over rock, paper, and
    scissors."""

    if action.lower() == 'rock':

        winning_pct = p[2] # pr win
        net = p[2] - p[1] # pr win - pr lose

    elif action.lower() == 'paper':

        winning_pct = p[0]
        net = p[0] - p[2]

    elif action.lower() == 'scissors':
```

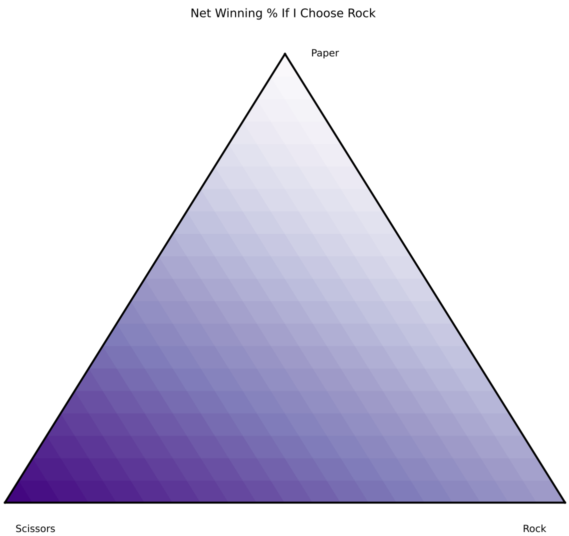
```python

        winning_pct = p[1]
        net = p[1] - p[0]
    else:
        raise ValueError("Input is not a valid action")

    return net
```

rps-helper.py

Now, we can create a heatmap using the function `winning_pct` and the `heatmapf` method.

```python
figure, tax = ternary.figure(scale = 20)
# vary scale above for higher resolution
figure.set_size_inches(9, 8)

# Add Heatmap
tax.heatmapf(winning_pct, boundary=True,
             style="triangular",
             cmap = 'Purples',
             colorbar = False)

tax.boundary(linewidth=2.0)

title = 'Net Winning % If I Choose Rock'
tax.set_title(title + " \n")

tax.right_corner_label('Rock',
        position = (.88,0.05,.09), fontsize=10)
tax.top_corner_label('Paper',
        position = (.01,1.11,.005),fontsize=10)
tax.left_corner_label('Scissors',
        position = (.07,0.05,1), fontsize=10)

tax.get_axes().axis('off')
# Workaround for a bug with labels
tax._redraw_labels()
```

heat-rps.py



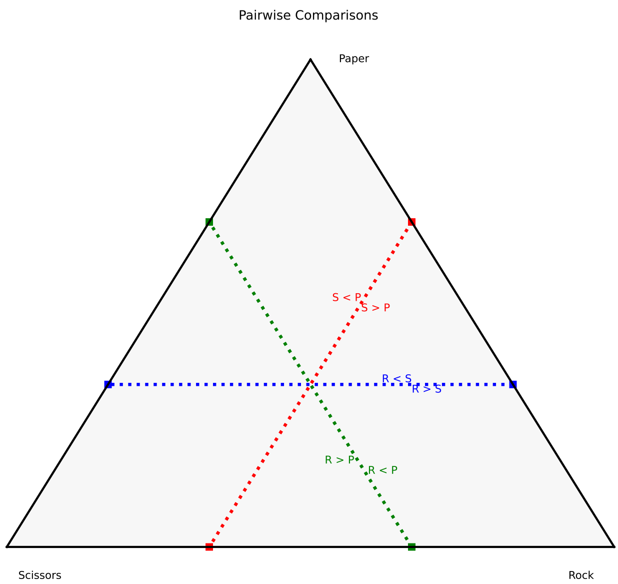
The natural extension of the above might be to repeat the above, which supposes you choose rock, for the actions paper and scissors. This can be done straightforwardly, by changing the default action in the `winning_pct` function. For the game theorist, the more interesting question is, given my opponent's distribution over actions, what is my best response? The code below plots the regions of pairwise indifference between two actions. Then, we divide up the simplex into three best response regions.

```
scale = 1
figure, tax = ternary.figure(scale = scale)
# vary scale above for higher resolution
figure.set_size_inches(9, 8)

# Rock > Paper
# s - p > r - s
# 2s > r + p
p1 = (2/3, 0, 1/3) # rps ordering
p1 = scale * np.array(p1)
p2 = (0, 2/3, 1/3)
p2 = scale * np.array(p2)

tax.line(p1, p2,
         linewidth=3.,
         marker='s',
         color='green',
         linestyle=":")
tax.annotate('  R < P', (.75*p1 + .25*p2),
```

```
                color = 'green',
                ha = 'left',
                va = 'top')
tax.annotate('R > P  ', (.75*p1 + .25*p2),
                color = 'green',
                ha = 'right',
                va= 'bottom')

# Rock > Scissors
# s - p > p - r
# s + r > 2p
p1 = (2/3, 1/3, 0) # rps ordering
p1 = scale * np.array(p1)
p2 = (0, 1/3, 2/3)
p2 = scale * np.array(p2)

tax.line(p1, p2,
           linewidth = 3.,
           marker = 's',
           color = 'blue',
           linestyle = ":",
           label = 'RockScissors')
tax.annotate('R > S', (.75*p1 + .25*p2),
                color = 'blue',
                ha = 'left',
                va = 'top')
tax.annotate('R < S', (.75*p1 + .25*p2),
                color = 'blue',
                ha = 'right',
                va= 'bottom')

# Paper > Scissors
# r - s > p - r
# 2r > p + s
p1 = (1/3, 2/3, 0) # rps ordering
p1 = scale * np.array(p1)
p2 = (1/3, 0, 2/3)
p2 = scale * np.array(p2)

tax.line(p1, p2,
           linewidth = 3,
           marker = 's',
           color = 'red',
           linestyle = ":")


tax.annotate('S > P ', (.75*p1 + .25*p2),
                color = 'red',
                ha = 'left',
                va = 'top')
tax.annotate(' S < P', (.75*p1 + .25*p2),
                color = 'red',
                ha = 'right',
```

```python
               va= 'bottom')

tax.boundary(linewidth=2.0)

# Make pretty as desired
title = 'Pairwise Comparisons'
tax.set_title(title + " \n")

tax.right_corner_label('Rock',
                       position = (.88,0.05,.09),
                       fontsize=10)
tax.top_corner_label('Paper',
                     position = (.01,1.11,.005),
                     fontsize=10)
tax.left_corner_label('Scissors',
                      position = (.07,0.05,1),
                      fontsize=10)

tax.get_axes().axis('off')
tax._redraw_labels()
```

rps-br-lines.py



We can color best responses zones by using the `heatmap` method, though it will take some work. First, we have some pure Python coding to do. Recall that `heatmap` requires a correspondence of points and the colors you want. We won't use a colormap, but we'll pass the colors in explicitly as RGBA values. This is a bit of a hack since the resulting plot and its colors won't have the ordered interpretation typical of heatmaps.

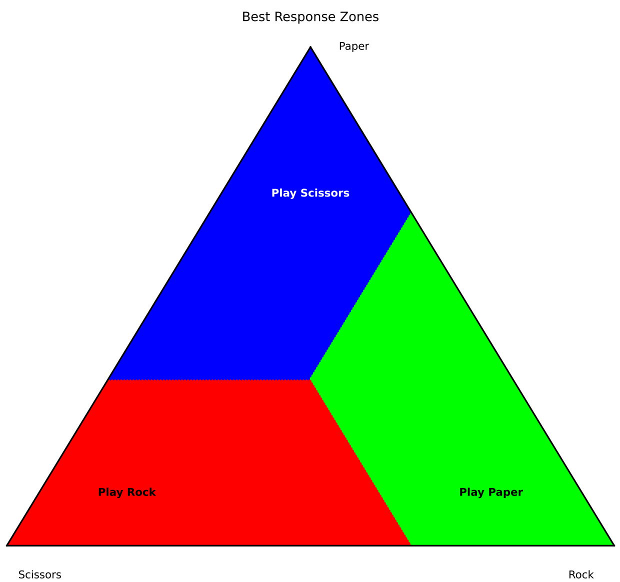
```python
def color_point(x, y, z):
    """Given an opponent plays rock at chance x, paper at y, and scissors at z, what is the best response?
    Best responses are mapped to RGB colors."""

    # winning pcts for possible responses
    rock_net = z - y
    paper_net = x - z
    scissors_net = y - x

    # get best response as highest net winning pct
    list_  = [rock_net, paper_net, scissors_net]
    best = list_.index(max(list_))

    # map into RGB color weights
    colors = [0, 0, 0]
    colors[best] = 1

    # return RGB tuple with fourth value for opacity (alpha)
    return (*tuple(colors), 1.)
```

rps-br-helper.py

```python
# Adapted from https://github.com/marcharper/python-ternary/blob/master/README.md RGBA section
def generate_heatmap_data(scale=10):
    from ternary.helpers import simplex_iterator
    d = dict()
    for (i, j, k) in simplex_iterator(scale):
        d[(i, j, k)] = color_point(i, j, k)
    return d

# Scale should be chosen high enough for sharp resolution
scale = 200
data = generate_heatmap_data(scale)
figure, tax = ternary.figure(scale=scale)
figure.set_size_inches(9, 8)

tax.heatmap(data, style="hexagonal",
    use_rgba=True, colorbar = False)
tax.boundary()
tax.set_title("Best Response Zones")


# Label the corners
```

```
labels = 'Rock', 'Paper', 'Scissors'
tax.right_corner_label(labels[0],
    position = (.88,0.05,.09), fontsize=10)
tax.top_corner_label(labels[1],
    position = (.01,1.11,.005),fontsize=10)
tax.left_corner_label(labels[2],
    position = (.07,0.05,1), fontsize=10)

# Label best response zones
tax.annotate("Play Rock",
    (.1* scale,.1 * scale,.8 * scale), weight = 'bold')
tax.annotate("Play Paper",
    (.8*scale, .1*scale, .1*scale), ha = 'right',
    weight = 'bold')
tax.annotate("Play Scissors",
    (.15*scale, .7*scale, .15*scale), ha = 'center',
    color = 'white',  weight = 'bold')

# Clear background and axes
tax.clear_matplotlib_ticks()
tax.get_axes().axis('off')
tax._redraw_labels()
```

rps-br-zones.py



Anyone inspecting what `data` looks like above will note that the points in the triangle aren't proper probability vectors as they add up to our `scale` value of 200 instead of one. That's immaterial for this application. A higher scale is chosen to create a sharper, exact border between between the best response regions.

Chapter 14

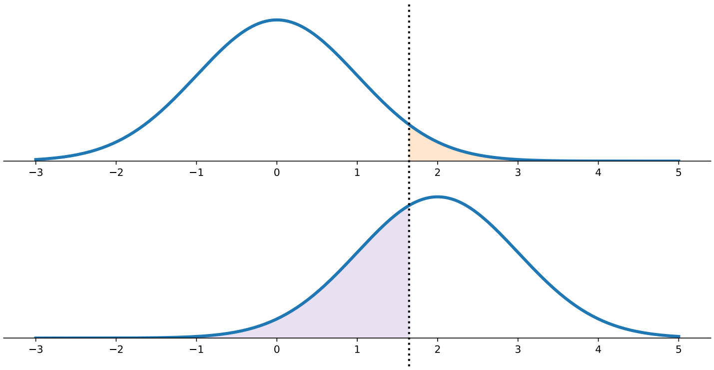
Intro Statistics

Since the fall of 2022, I've been teaching intro stats and I've been very particular about my graphs. According to my college ethics professor, the experience of beauty is a human good and statistics is not.[1] That doesn't provide a good prioritization for developing an intro stats syllabus, but as an instructor, I've at least tried to make my slides and notes a kind of beautiful. Watching a student pinch to zoom on her iPad without a particular graph pixelating or otherwise revealing any defect was a beautiful moment.

I make all of the figures for my class materials in Python. Working in Python offers the advantage of, well, working in Python. We are able to mix statistical functions from Python in our code, so we gain more precision than is available from anything hand-drawn (whether on pencil and paper or by dragging a cursor). We can also create high quality images in a vector format like PDF or SVG, meaning we can zoom without the image quality degrading.

It's true that you might achieve the same goals using languages like R or Ti*k*z. Indeed, for probability trees, I still use Ti*k*z. Otherwise, I offer no apology for choosing Python. In this chapter, I include a few common applications that arise in statistics.

[1]Shoutout to the late Alfonso Gómez-Lobo.



14.1 Probability Diagrams

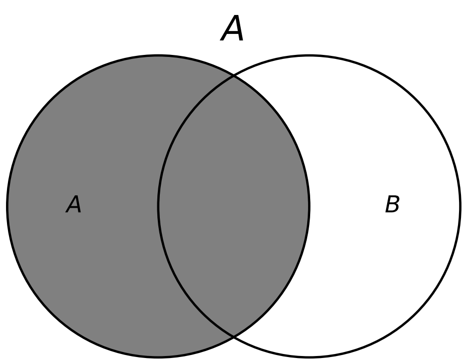
Below, we produce Venn diagrams to consider set operations for two sets, A and B. These sets are outlined as circles, either by plotting the outline or by using the Circle artist object. The result of the set operation will be filled with gray. This can be tricky because I insist that the circle outlines (the Artist object's edge) are always visible, meaning we have to be mindful of how one circle overlapping another could obscure the other's outline.

First, we plot A and B, filling only A. A and B are Circle objects and I plot B second so that it is in front of A. This means that the outline of B is not lost underneath A. We want the opposite from the fill color of B—we do not want it to overlap A. To prevent this, B is filled with `facecolor = (1, 1, 1, 0)`, which is the RGBA color for white (RGB 1,1,1) and an alpha or opacity of 0%, leaving it transparent. We could also have used an alpha parameter, but this would apply to both the `facecolor` and `edgecolor` arguments, which we want to treat separately.

```
fig, ax = plt.figure(), plt.axes()
ax.set_aspect(1)
ax.set_xlim(-1.6,1.6)
ax.set_ylim(-1.1,1.4)
ax.axis('off')

# Add circles with color fill
left_circle = plt.Circle((-0.5, 0), 1,
                        edgecolor = 'black',
                        facecolor = 'gray',
                        linewidth = 2)
right_circle = plt.Circle((0.5, 0), 1,
                        edgecolor = 'black',
```

```
                        facecolor = (1, 1, 1, 0), # transparent
                        linewidth = 2,
                        zorder = 5)
ax.add_artist(left_circle)
ax.add_artist(right_circle)

# Labels
ax.text(0, 1.05,
       s = r"$A$",
       va = 'bottom',
       ha = 'center',
       size = 30)
ax.text(-1, 0,
       s = r'$A$',
       ha = 'right',
       va = 'center',
       size = 20)
ax.text(1, 0,
       s = r'$B$',
       ha = 'left',
       va = 'center',
       size = 20)
```

event-A.py

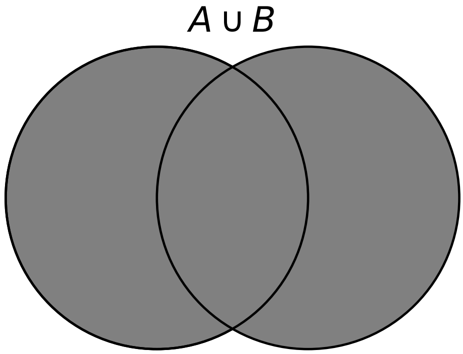


Next, we consider the union of A and B. This presents a similar challenge in not having the fill of one circle cover the outline of the other circle. First, we create two Circle objects, filled gray and with a black outline. If we stopped there, we would have a visible overlap. To recover the outline of A, we add a third circle with a black outline but a transparent fill.

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

```
ax.set_aspect(1)
ax.set_xlim(-1.6,1.6)
ax.set_ylim(-1.1,1.4)
ax.axis('off')

# Add circles with color fill
left_circle = plt.Circle((-0.5, 0), 1,
                        edgecolor = 'black',
                        facecolor = 'gray',
                        linewidth = 2)
left_circle_helper = plt.Circle((-0.5, 0), 1,
                        edgecolor = 'black',
                        facecolor = (1, 1, 1, 0),
                        linewidth = 2)
right_circle = plt.Circle((0.5, 0), 1,
                        edgecolor = 'black',
                        facecolor = 'gray',
                        linewidth = 2)
ax.add_artist(left_circle)
ax.add_artist(right_circle)
ax.add_artist(left_circle_helper)

# Label
ax.text(0, 1.05,
       s = r"$A \cup B$",
       va = 'bottom',
       ha = 'center',
       size = 30)
```

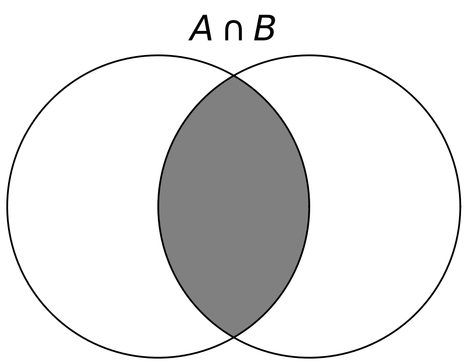
union.py



To plot the intersection of A and B, we abandon Circle objects for the `plot()` and `fill_between()` Axes methods. Filling the intersection is done in two pieces.

```python
fig, ax = plt.figure(), plt.axes()
ax.set_aspect(1)
ax.set_xlim(-1.6,1.6)
ax.set_ylim(-1.1,1.4)
ax.axis('off')

# Add Circles
k = 10_000
angles = np.linspace(0, 2*np.pi, k)
x = np.cos(angles)
y = np.sin(angles)
ax.plot(x - 0.5, y, color = 'black')
ax.plot(x + 0.5, y, color = 'black')

# fill circles in two pieces
x = np.cos(angles) - 0.5
t = 0
while x[t] > 0:
    t += 1
ax.fill_between(x[:t], -y[0:t], y[0:t],
                color = 'gray', zorder = -1)
ax.fill_between(-x[:t] , -y[0:t], y[0:t],
                color = 'gray', zorder = -1)

# Label
ax.text(0, 1.05,
        s = r"$A \cap B$",
        va = 'bottom',
        ha = 'center',
        size = 30)
```

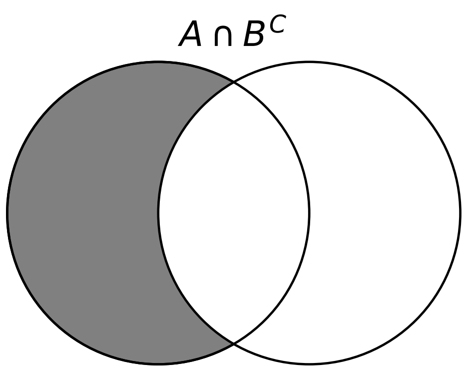
intersection.py



Finally, we plot $A \cap B^C$ or $A \setminus B$. For this, we again need a third transparent circle. We allow a white-filled B to cover $A \cap B$, essentially removing the fill from that region. Then, we use the Circle `left_circle_helper` to restore the outline of A.

```
fig, ax = plt.figure(edgecolor = 'black'), plt.axes()
ax.set_aspect(1)
ax.set_xlim(-1.6,1.6)
ax.set_ylim(-1.1,1.4)
ax.axis('off')

# Add circles with color fill
left_circle = plt.Circle((-0.5, 0), 1,
                        edgecolor = 'black',
                        facecolor = 'gray',
                        linewidth = 2)
left_circle_helper = plt.Circle((-0.5, 0), 1,
                        edgecolor = 'black',
                        facecolor = (1,1,1,0),
                        linewidth = 2)
right_circle = plt.Circle((0.5, 0), 1,
                        edgecolor = 'black',
                        facecolor = 'white',
                        linewidth = 2)
ax.add_artist(left_circle)
ax.add_artist(right_circle)
ax.add_artist(left_circle_helper)

# Label
ax.text(0, 1.05,
       s = r"$A \cap B^C$",
       va = 'bottom',
       ha = 'center',
       size = 30)
```

A-minus-B.py

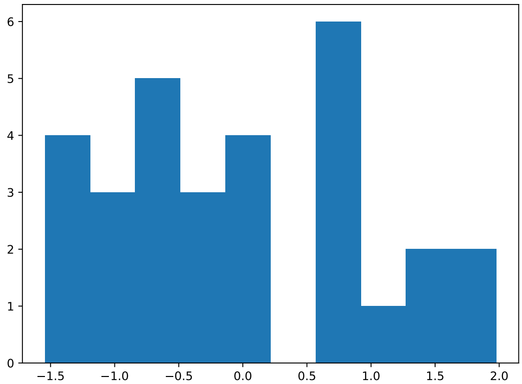


14.2 Distributions

We use the `hist()` Axes method to create a histogram. Beautifying the plot is only a matter of modifying the background elements, like the spines and labels, and then choosing the right number of bins and colors for the histogram. A histogram includes Patch objects, meaning we can specify both an `edgecolor` and `facecolor`. Below, we create a histogram using the defaults.

```
data = np.random.normal(size = 30)
fig, ax = plt.figure(), plt.axes()
ax.hist(data)
```

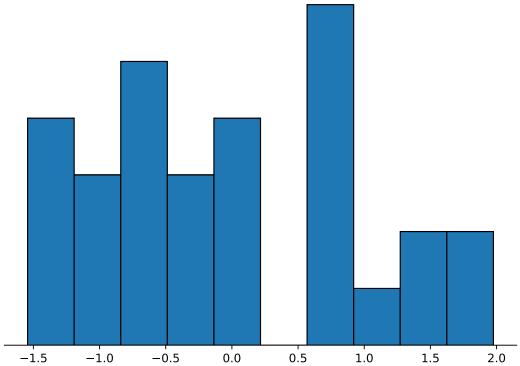
default-hist.py



When you are only interested in the shape of the data, you will find the y-axis information unimportant. Accordingly, we'll remove the y-axis to (repeat after me) reduce clutter.

```
#data = np.random.normal(size = 30)
fig, ax = plt.figure(), plt.axes()
ax.hist(data, edgecolor = 'black')
ax.set_yticks([])
for s in 'left', 'top', 'right':
    ax.spines[s].set_visible(False)
```

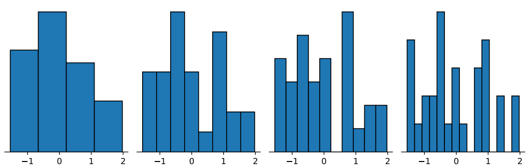
clean-hist.py



The last modification considered is the simple use of the `bins` parameter to adjust the number of bins in the histogram.

```
#data = np.random.normal(size = 30)
fig, ax = plt.subplots(1,4, figsize = (9,3))

for key, b in enumerate([4, 8, 10, 15]):
    ax[key].hist(data, edgecolor = 'black', bins = b)
    ax[key].set_yticks([])
    for s in 'left', 'top', 'right':
        ax[key].spines[s].set_visible(False)
```

bins-hist.py

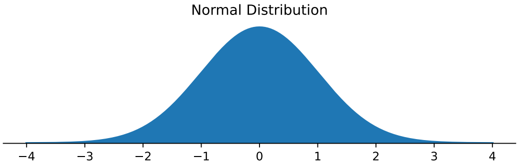


Next, density plots can be produced the SciPy package and its statistics module, which has been imported already with `import scipy.stats as stats`. Below is a simple plot for a normal distribution.

```
x = np.linspace(-4, 4, 100_000)
y = stats.norm.pdf(x)
fig, ax = plt.figure(), plt.axes()
ax.set_aspect(5)

ax.plot(x,y, linewidth = 2, color = 'C0')
ax.fill_between(x,y, color = 'C0')
```

```python

ax.set_yticks([])
for s in 'left', 'top', 'right':
    ax.spines[s].set_visible(False)

ylims = ax.get_ylim()
ax.set_ylim(0, ylims[1])
ax.set_title("Normal Distribution")
```

normal-pdf.py

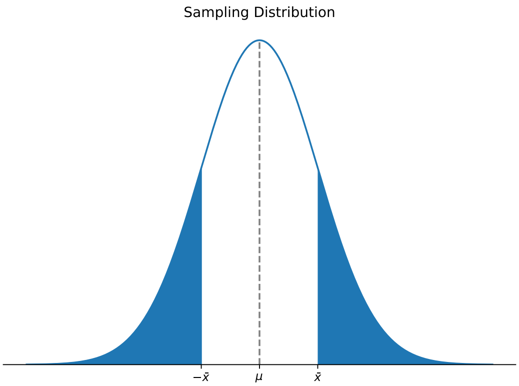


To motivate sampling distributions, it might be useful to label the x-axis not with numeric quantities but with μ and $\bar{x}$ for the true mean and the observed sample mean. This is done by with the `set_xticklabels()` Axes method, aligning the strings with ticks that are also manually set with `set_xticks()`.

```python
fig, ax = plt.figure(), plt.axes()
x = np.linspace(-4,4, 100_000)
norm_pdf = stats.norm.pdf(x)
ax.plot(x, norm_pdf)
ax.set_title("Sampling Distribution")

# Clean background elements
ax.set_yticks([])
for s in 'left', 'top', 'right':
    ax.spines[s].set_visible(False)

# Start y-axis at 0
ylims = ax.get_ylim()
ax.set_ylim(0, ylims[1])

# Set x-axis ticks and labels
ax.set_xticks([-1, 0,1])
```

```python
ax.set_xticklabels([r"$-\bar{x}$", r'$\mu$', r'$\bar{x}$
    '])

# Vertical dashed line
x0 = 0,0
y0 = [0,stats.norm.pdf(0)]
ax.plot(x0, y0, linestyle = 'dashed', color = 'gray')

# Fill area for a more extreme sample mean
x_right = x[x > 1]
y_right = norm_pdf[-len(x_right):]
ax.fill_between(x_right, y_right)

x_left = x[x < -1]
y_left = norm_pdf[:len(x_left)]
ax.fill_between(x_left, y_left, color = 'C0')
```

sampling-dist.py



Next, we conduct and illustrate a one-tailed z-test for a proportion,

$$H_0 : p = \frac{1}{2}$$
$$H_a : p > \frac{1}{2}.$$

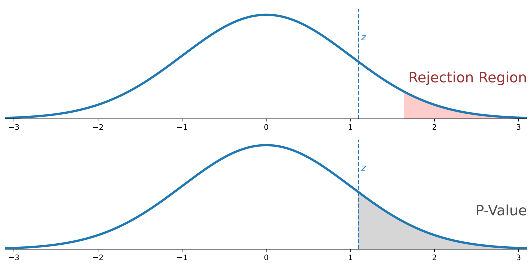
We set $\alpha = 0.05$ and suppose we observed 18 successes from 30 trials. The test statistic and p-value are calculated with the help of the statsmodels library and the `proportions_ztest()` function.

```python
# One-sided test
```

```
from statsmodels.stats.proportion import
    proportions_ztest
z_oneside, pval_oneside = proportions_ztest(
                                count = 18,
                                nobs = 30,
                                value = .5,
                                prop_var = 0.5,
                                alternative = 'larger')

# one tail norm
x = np.linspace(-4,4, 100_000)
norm_pdf = stats.norm.pdf(x)

# Make empty figure
fig, axx = plt.subplots(2, 1,
                        figsize = (10,5),
                        sharey = True)

for ax in axx:
      # add normal curve
    ax.plot(x, norm_pdf, lw = 3)

    # add z stat
    ax.axvline(z_oneside, linestyle = 'dashed')
    ax.text(z_oneside, 0.3, ' $z$', ha = 'left', size =
    12, color = 'C0')

    # Clean plot
    for s in 'left', 'right', 'top':
        ax.spines[s].set_visible(False)
    ax.yaxis.set_ticks([])
    ax.set_ylim(0,.42)
    ax.set_xlim(-3.1, 3.1)

# draw rejection region
ax = axx[0]
alpha = .05
critical_value = stats.norm.ppf(1 - alpha) # for one-
    sided
rejection_x = x[x>critical_value]
ax.fill_between(rejection_x, norm_pdf[-len(rejection_x)
    :],
                color = 'red',
                alpha = 0.2)
ax.text(3.1, .14,
        s = 'Rejection Region',
        fontsize = 19,
        color = (.6, .2, .2),
        ha = 'right')

# shade p-value
ax = axx[1]
more_extreme_x = x[x>z_oneside]
```

```
ax.fill_between(more_extreme_x, norm_pdf[-len(
    more_extreme_x):],
                color = (.2, .2, .2),
                alpha = 0.2)
ax.text(3.1, .13,
        s = 'P-Value',
        fontsize = 19,
        color = (.3, .3, .3),
        ha = 'right')
```

one-tail-norm.py

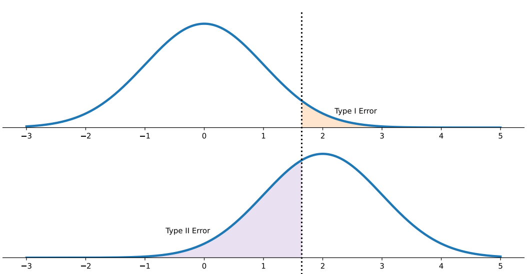


In the above, the two plots share the same x-axis scaling and I'd prefer the vertical line for the test statistic extend from one graph to the other, emphasizing that the two distributions are identical. We again face this challenge when using stacked plots to illustrate Type I and Type II error, and it is in that context that we solve the problem. This can be solved by using a Transform object. The x-axis coordinates are naturally expressed in data coordinates. Because we want the vertical line extending from one subplot Axes to the other, we should work in Figure coordinates. This requires just two special lines in the code below. The blended transformation is created with the `blended_transform_factory()` function and then we use the `transform` parameter in the Axes plotting method.

```
x = np.linspace(-3,5, 100_000)

fig = plt.figure(figsize = (10,5))
ax0 = fig.add_subplot(211, zorder = 99)
ax1 = fig.add_subplot(212)

for center, ax in zip([0, 2], [ax0, ax1]):
  # add normal curve
    norm_pdf = stats.norm.pdf(x, center, 1)
    ax.plot(x, norm_pdf, lw = 3)

    # Clean plot
    for s in 'left', 'right', 'top':
```

```
        ax.spines[s].set_visible(False)
    ax.yaxis.set_ticks([])
    ax.set_ylim(0, .42)

# draw rejection region
ax = ax0
alpha = 0.05
critical_value = stats.norm.ppf(1 - alpha) # for one-sided
rejection_x = x[x > critical_value]
ax.fill_between(rejection_x,
                stats.norm.pdf(x, 0, 1)[-len(rejection_x):],
                color = 'C1',
                alpha = 0.2)
# label
ax.text(2.2, stats.norm.pdf(2), 'Type I Error')

# Draw critical value line through both plots
trans = mpl.transforms.blended_transform_factory(ax.transData, fig.transFigure)
ax.plot([critical_value, critical_value],
        [0, 1],
        color='black',
        linestyle = 'dotted',
        lw=2,
        transform = trans,
        clip_on= False,
        zorder = 99)

# Draw type II error region
fail_to_reject_region = x[x <= critical_value]
ax1.fill_between(fail_to_reject_region,
                 stats.norm.pdf(x, 2, 1)[0:len(fail_to_reject_region)],
                 color = 'C4',
                 alpha = 0.2)
ax1.text(.1, stats.norm.pdf(.3, 2, 1), 'Type II Error', ha = 'right')
```

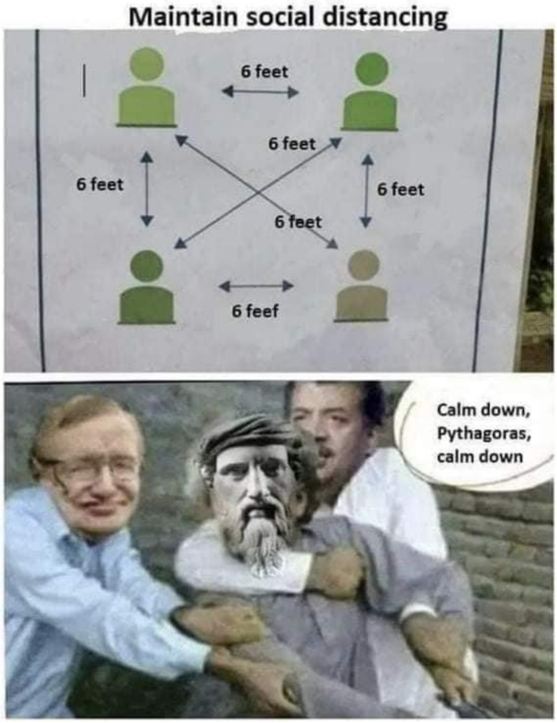
error-stack.py



Chapter 15

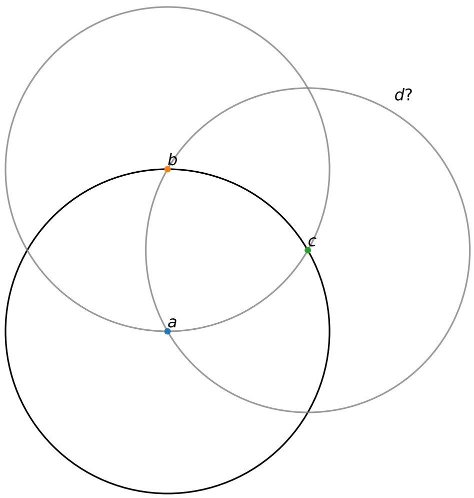
Multi-dimensional Scaling

Multi-dimensional scaling is a kind of dimensionality reduction, allowing one to translate a set of items and their predetermined pairwise distances into an arrangement of points in Cartesian space, typically in two dimensions. This is not a trivial task. Consider the meme below, which references social distancing guidance from the coronavirus pandemic. We have four individuals, forming six possible pairs, and any two individuals must maintain a six-foot distance. The graphic shows impossible right triangles that violate the Pythagorean Theorem when the distances are all exactly six feet. Indeed, there is no way to arrange four individuals in two-dimensional space so that they are all exactly six feet apart. This highlights one difficulty in multi-dimensional scaling—we must accept some error in our output.

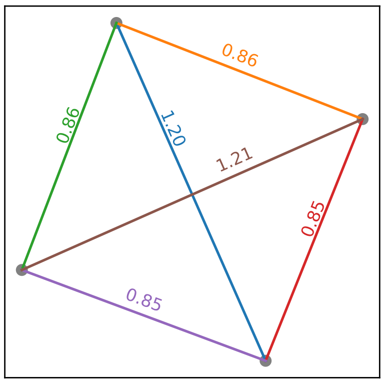


The following diagram helps show why no alternative arrangement could be perfect. Each point a, b, c is in the middle of a circle of six-foot radius. For b to be six feet away from a, b must lie on the circle centered at a. Similarly, for b to be six feet away from c, b must also lie on the circle centered at c. For three points, this arrangement is possible. But if we introduce d, d must lie on all three of the circles centered at a, b, and c. But there is no point where all three circles intersect.[1]

[1] The code is (link TK).



Suppose we have four points, with unit pairwise distances for any two distinct points. Applying multi-dimensional scaling creates what is roughly a square.



TKTK

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.

Scruton, R. (2015). Poetry and truth. In J. Gibson (Ed.), *The philosophy of poetry.* Oxford 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. ✌