Matplotlib for Storytellers

By: Alexander Clark

This version: November 16, 2021

Contents

Code

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

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

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

Preface

Technical Notes and Prerequisites

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

Why Matplotlib?

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

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

Good Visualization is Like Good Writing

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

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

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

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

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

The important point here is that the unimaginative application of ready-made visualizations, just like phrases, can conceal your meaning from yourself, not to mention your audience, and create a monotonous presentation of bar chart after bar chart.

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

Resources and Inspiration

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

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

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

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

Text Organization

Continuing the parallel to writing, I have built this text around two main parts: Prose and Poetry, though the distinction between prose and poetry is surely less exact than the division I've created. Prose, or Part I, focuses on the fundamentals of customizing plots through the object-oriented interface. This section attempts to be reasonably thorough in breadth while providing only a minimal effective dose in depth. Then, after a mathematical interlude in Part II, we reach poetry in Part **??**. 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 III, multi-dimensional scaling and ternary plots.

Part I

Prose

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

Chapter 1

The Object-oriented Interface

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

The MATLAB-style interface looks like the following.

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

The object-oriented interface looks like this.

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

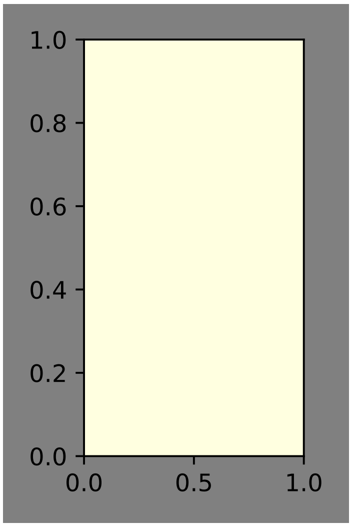
There is no such thing as a free lunch, so you will observe this interface requires more code to do the same exact thing. Its virtues will be more apparent later. Object-oriented programming (OOP) also requires some new vocabulary. OOP might be contrasted with procedural programming as another common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

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

1.1 Figure, Axes

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

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

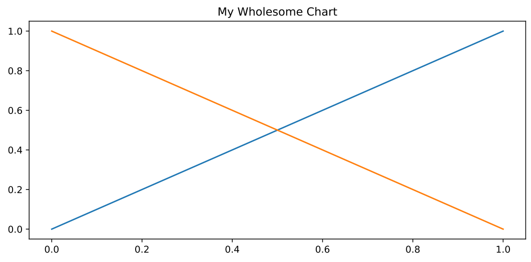


The axes object, named `ax` by convention, gets more use in most programs. In place of `plt.plot()`, you'll use `ax.plot()`. Similary, `plt.hist()` is replaced with `ax.hist()` to create a histogram. If you have experience with the MATLAB interface, you might get reasonably far with the object-oriented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

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

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

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



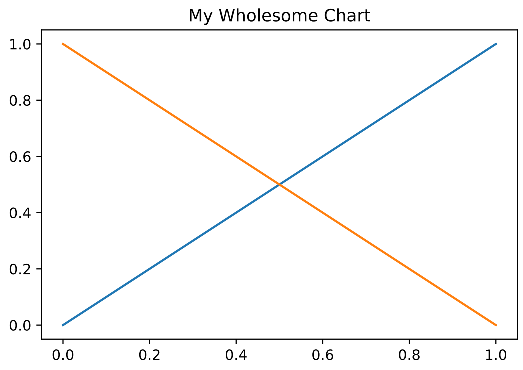
Axes methods `set_xlim()` and `get_xlim()` behave just like `set_title()` and `get_title()`, but note there is no attribute simply accessible with `ax.xlim`, so the existence of getters and setters is the more fundamental pattern.[1]

1.2 Mixing the Interfaces

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

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

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

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

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

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

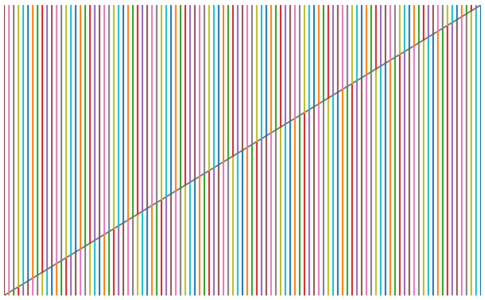


In the above, we started with MATLAB and then converted to object-oriented. We can also go in the opposite direction, though it's not always ideal, especially when working with subplots. Below, we start with our figure and axes objects, and then revert back to the MATLAB style with the `axvline()` functions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

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

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

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

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

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

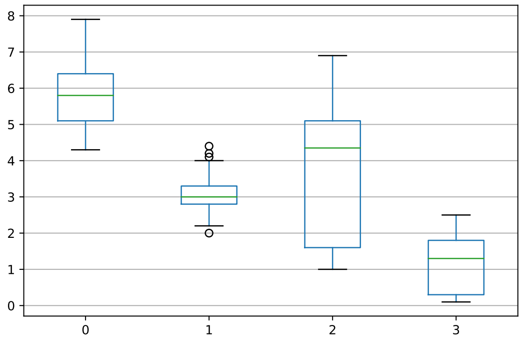
Matplotlib is also integrated into pandas, with a `plot()` method for both Series and DataFrame objects, among other functionalities. There is excellent documentation available.[2] These plots can be mixed with the object-oriented interface. You can use a plot

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



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

```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('iris_box.pdf')
```



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

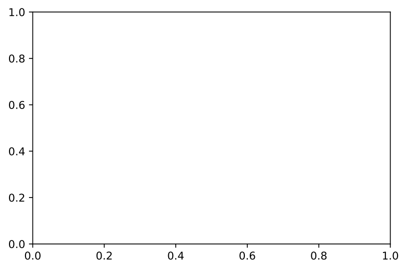
Chapter 2

Axes Appearance, Ticks, and Grids

2.1 Axis Aspect and Limits

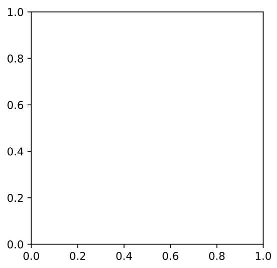
The most basic plot is the empty plot.

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



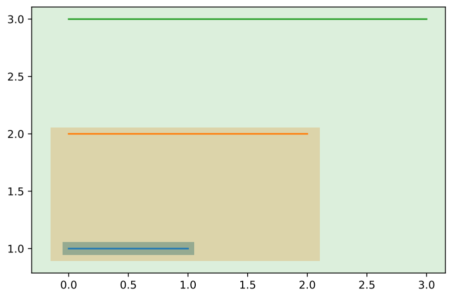
You'll notice this defaults to plotting the square region between data points (0,0) and (1,1). However, the plot is not square by default. That is to say the *aspect* is not one, where the aspect is the ratio of height to width. This can be changed with the axes method `set_aspect()`. For equal scaling,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

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



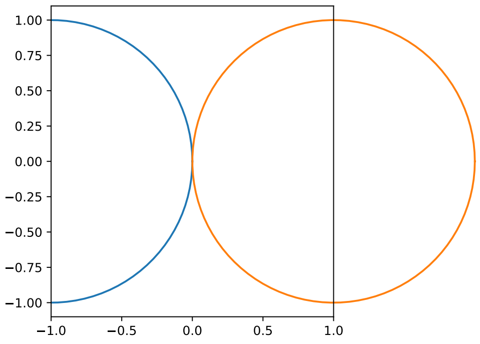
As we already covered in Chapter 1, the x and y limits can be adjusted with axes methods `set_xlim()` and `set_ylim()`, taking a sequence for the minimum and maximum values. If you don't explicitly set the limits, matplotlib will set the limits automatically based on the data. You can retrieve those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

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

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

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



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)
```

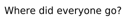


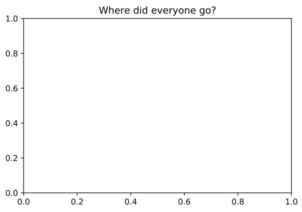
2.2 Axis Lines and Spines

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.

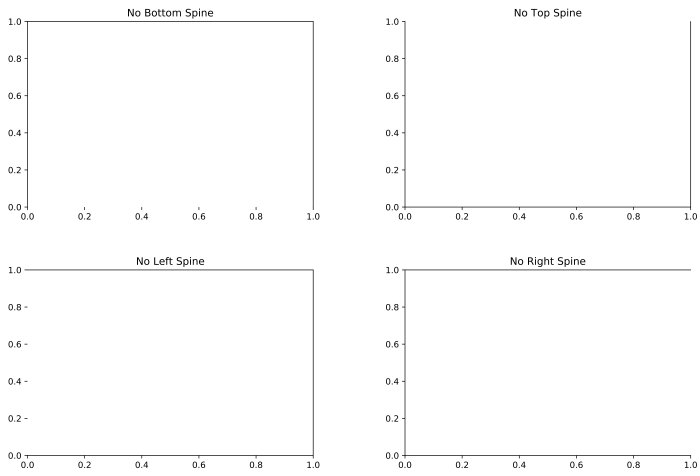
```
fig, ax = plt.figure(), plt.axes()
ax.set_title("Where did everyone go?")
ax.axis('off')
```





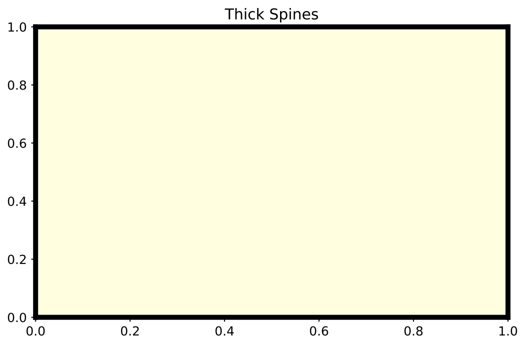
Next, we can access and modify specific spines through `ax.spines`, which returns an `OrderedDict`. Access a specific spine using the appropriate key: `"left"`, `"right"`, `"top"`, or `"bottom"`. A spine can be toggled on or off by passing the appropriate boolean value to `set_visible()`.

```python
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)
    fig.show()
```



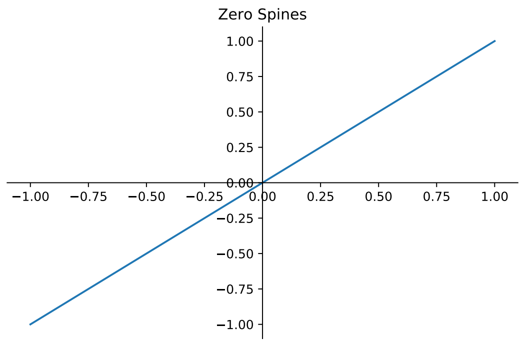
Other spine modifications might be their width and color. Again, we access a particular spine and then make use of setter methods, `set_color` and `set_linewidth` in particular.

```python
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)
```



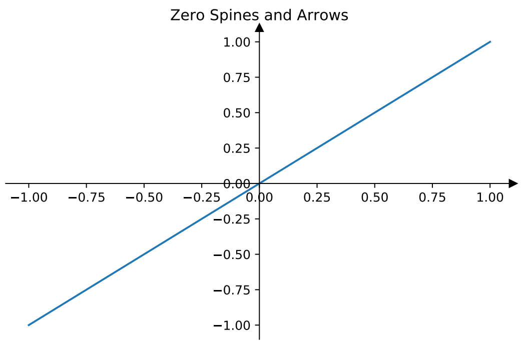
It's easy to get this far imagining that spines are simply the pieces of the box enclosing your plot. But they don't have to enclose the plot if we alter them with the `set_position` method. Below, we set the bottom spine to be along the usual x-axis and the left spine to be along the usual y-axis by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



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)
```

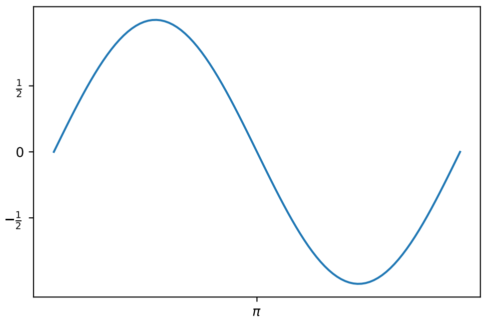


The tick labels do clutter the graph above. This can be solved after we cover Section 2.3. Knaflic 2015 recommends removing the top and right spines as part of the imperative to declutter and remove unnecessary chart border. I think it is arguable. I'm used to default spines enclosing the data. Removing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

```python
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"$\pi$"])
```

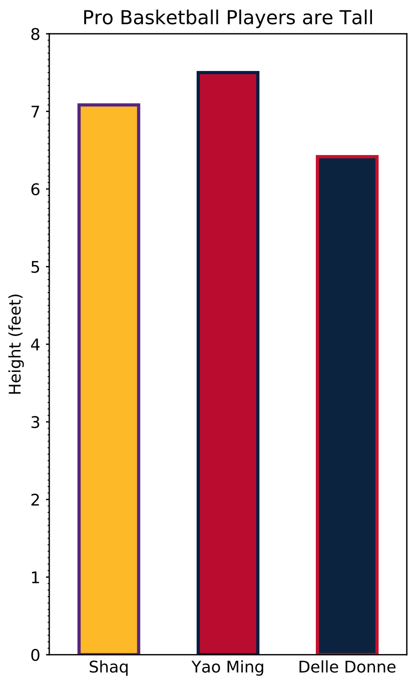


To remove the ticks entirely, simply pass an empty array to `set_ticks()`. To customize the appearance of your axis ticks and the labels, use the `set_tick_params` axis method. Parameters include `direction`, `width`, `length`, `color`, `pad`, `rotation`, `labelsize`, `labelcolor`

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`.

```python
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")
```



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.

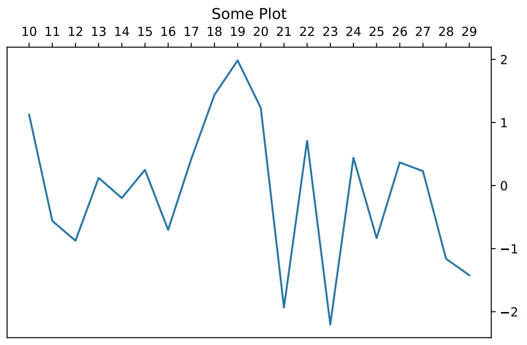
With a function like $\sin x$, ticks might most naturally be placed at multiples of π. This can be accomplished by the below.

```
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))
```

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()`.

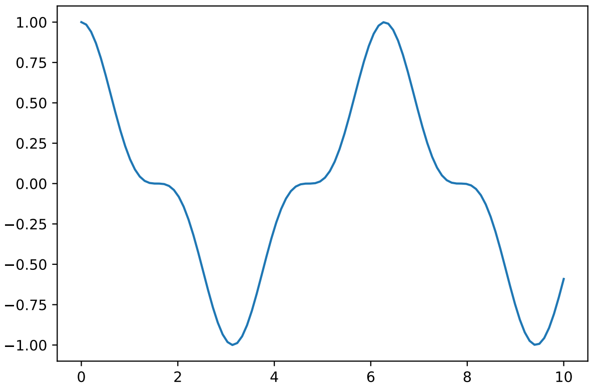
```python
fig, ax = plt.figure(), plt.axes()
x = np.arange(10,30,1)
y = np.random.normal(size = len(x))
ax.plot(x,y)

# 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")
```



2.4 Grids

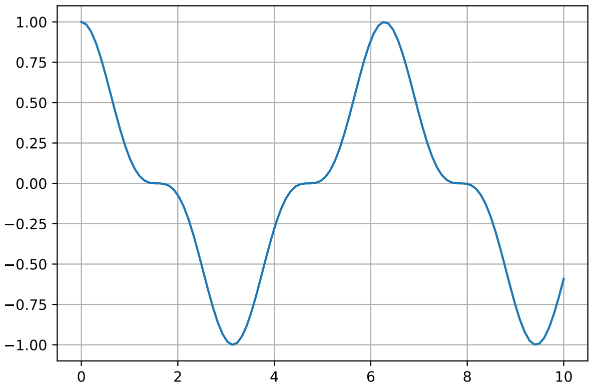
Including gridlines in a plot is generally discouraged (Knaflic 2015, Schwabish 2021). It's clutter that won't spark joy. Perhaps we could stop here, with the instruction to run `ax.grid(False)` as in the code below.

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



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

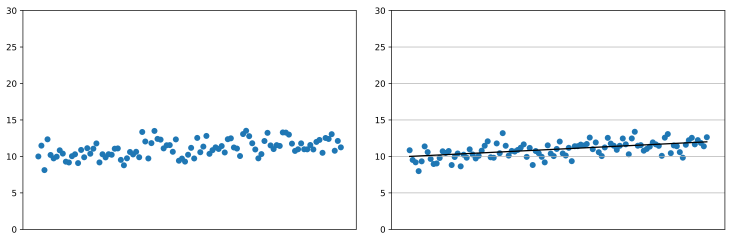
```
fig, ax = plt.figure(), plt.axes()
x = np.linspace(0,10,100)
ax.plot(x, np.cos(x)**3)
ax.grid(True)
```



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()`.

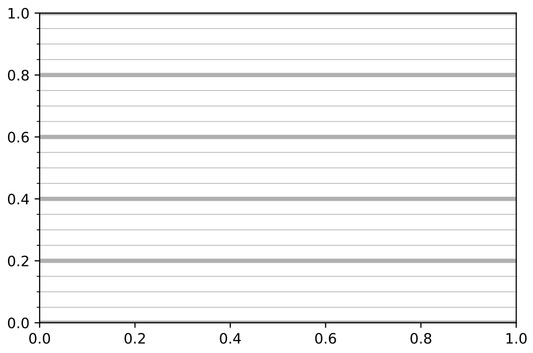
```
fig, ax = plt.figure(), plt.axes()

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([])
```

```
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')
```



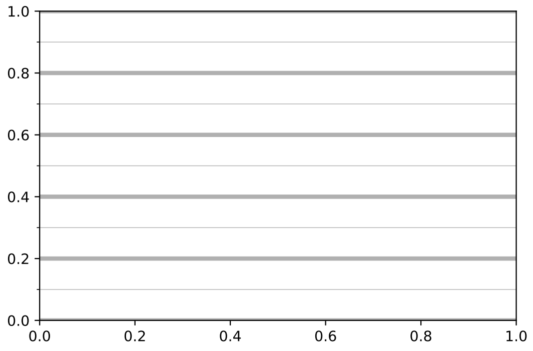
What we learned previously about locating ticks in Section 2.3 can be reapplied here, as seen in the examples further below. The location of gridlines and ticks can be set by the `set_major_locator()` and `set_minor_locator()` methods. `ax.grid()` is used to display the gridlines, but note it features 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()
    )
```



The above used an auto locator, and now we set the locations manually using `MultipleLocator()`.

```
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))
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Chapter 3

Colors

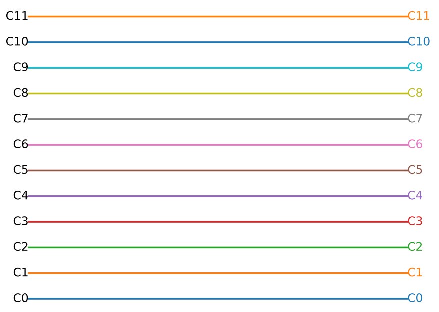
Methods like `plot` and `text` include a color parameter, which we've already made use of. While you can get pretty far simply using `color = 'blue'`, you might also make use of colormaps or set your own colors using hex strings or RGB(A) tuples.

3.1 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.

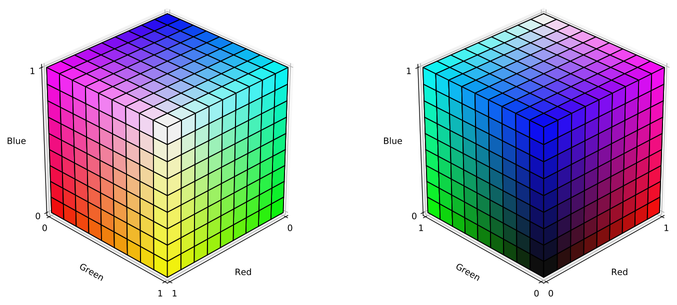
```
fig, ax = plt.figure(), plt.axes()
for i in range(12):
    # Plot color automatically cycles through color map
    ax.plot( [0,1], np.ones(2)*i)

    # 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')
```



3.2 Red, Green, Blue, Alpha

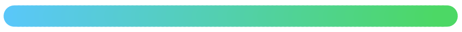
An RGB color is given by three values, specifying the amount of red, green, and blue. In matplotlib, these values are between zero and one (you might also see RGB values between zero and 255 elsewhere). These colors live inside a cube, as a particular color is a triple $(r, g, b) \in [0, 1]^3$.



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
```

```python
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')
```

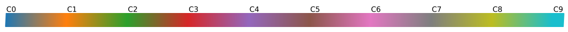


Any color can be made lighter by averaging it with white, $(1,1,1)$, or darker by averaging it with black $(0,0,0)$. We can also find the inverse of an RGB color by simply subtracting that triple from $(1,1,1)$. RGBA tuples are very similar, adding a fourth $a$lpha value for the opacity.

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'`.

```python
# 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)

# How many color changes
segments = 100

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


for c in range(n_colors - 1):
    color1 = mpl.colors.ColorConverter().to_rgb(
    color_strings[c])
    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, 'C'+str(c), va = 'bottom', size =
    12, ha = 'center')

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

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



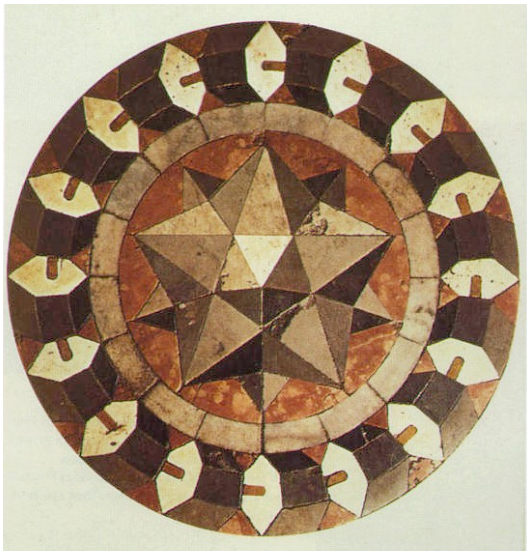
Color Cube Code

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

```python
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from itertools import product

fig = plt.figure(figsize = (6,6), constrained_layout=True)
ax = plt.axes(projection='3d')

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

# 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]):
                #print(x)
                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)
                #break

            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)
# Increase distance so labels are not cut off
ax.dist = 12
```

Part II

Mathematical Interlude



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

Chapter 4

Math

In Part **??**, 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 5.2, which can't be fashioned with a simple call to `plt.Circle()`.

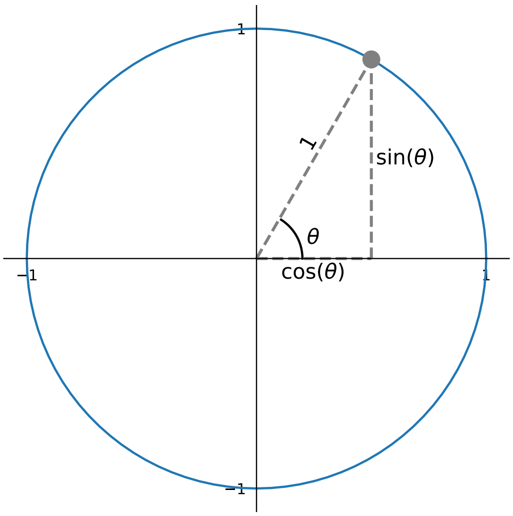
4.1 Circles

4.1.1 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.



```python
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 spine_mod.py
ax.set_xticks([-1, 1])
ax.set_yticks([-1, 1])
```

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 4.1.2, we consider how to do the same, but for non-unit circles.

4.1.2 Non-unit Circles

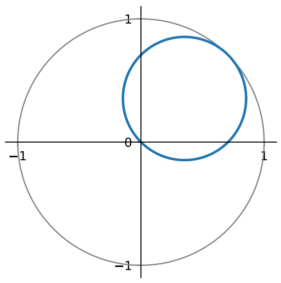
The unit circle has a radius of one and it's centered at the origin. How do we obtain coordinates for other circles? There are two steps to change the radius and shift a circle off the origin.

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 spine_mod.py

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



4.1.3 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.

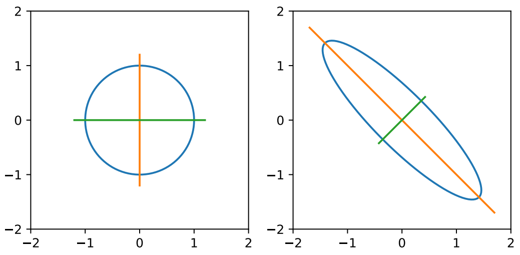
```
theta = np.pi / 4
rotation_matrix = np.matrix([[np.cos(theta), -np.sin(
    theta)], [np.sin(theta), np.cos(theta)]])

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
```

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

```
# 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
```

```python
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)
```

```python
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')
```



4.2 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

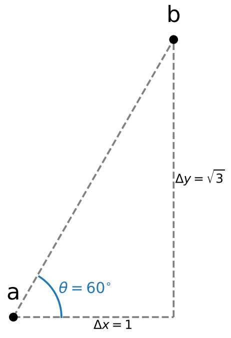
$$\sin\theta = \frac{\text{opposite}}{\text{hypotenuse}}$$
$$\cos\theta = \frac{\text{adjacent}}{\text{hypotenuse}}$$
$$\tan\theta = \frac{\text{opposite}}{\text{adjacent}},$$

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.

```python
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)

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)
```


b
a
$\theta = 60^\circ$
$\Delta y = \sqrt{3}$
$\Delta x = 1$

Chapter 5

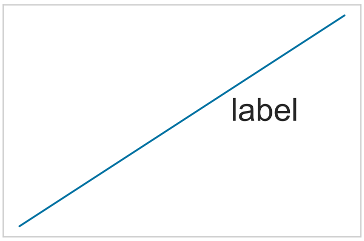
Applications

5.1 Sloping Text

Suppose you'd like to label a line.

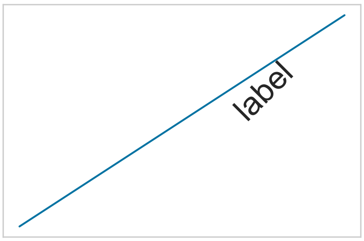
For a sloped line, you might rather the text sit parallel to the line instead of suffering the below.

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

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



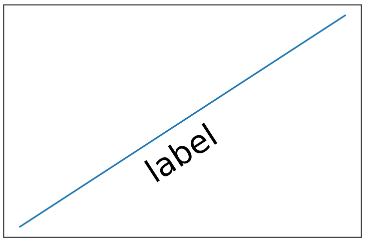
The `rotation` argument can help if you know the right angle in degrees. Here the angle is 45 degrees or $\frac{\pi}{4}$ radians. So we modify the second line to be `plt.text(0.65, 0.5, 'label', size = 30, rotation = 45)`.

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`. Try experimenting by replacing the `x2,y2` values to see this works for any angle.

```python
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 = np.array([[0,0]]) # doesn't matter what
    pair you use.
plot_angle = ax.transData.transform_angles(np.array((
    true_angle,)),

    dummy_array)[0]

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

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

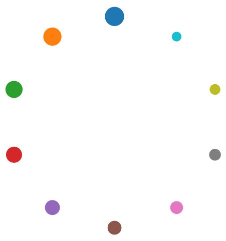


5.2 Circular Arrangements

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.

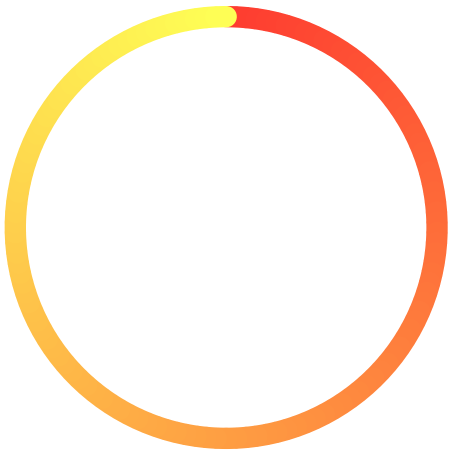
```python
n_points = 10
pie_angle = 360/n_points # angle of each slice
starting_angle = 90

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')
```

This code produces the following.



The below makes similar use of trigonometry to create a circle colored according to a gradient, like in Chapter 3. 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')
```



5.3 Network Graphs

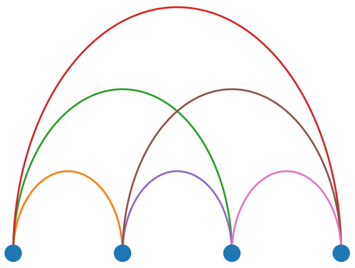
Networks are represented mathematically as graphs—a set of vertices and edges between them. In drawing a graph, there are many drawing algorithms available. For large networks or sophisticated algorithms, you should use something off the shelf in a package like nxviz. For a small network, you might avoid 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()
```

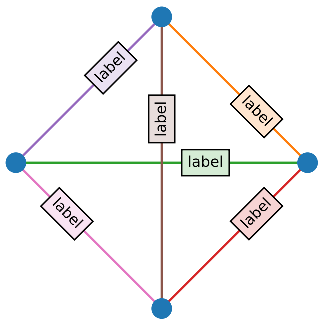
```
x = np.linspace(0,1,4)
ax.plot(x, np.zeros(4),
        marker = 'o',
        linestyle = '',
        markersize = 13)

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:
        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)
```



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)
```

```python
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 = 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')
```

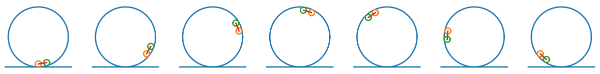


5.4 Tony Hawk's Vertical Loop

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)
```

```python
    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)
```



Part III

Special Topics

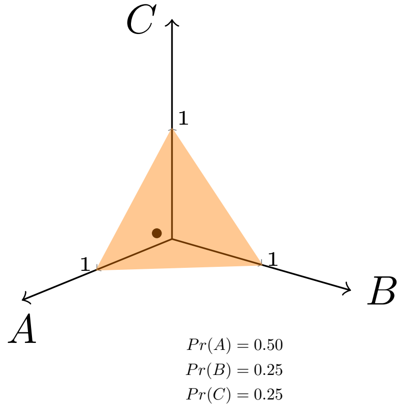
A Virgin with a Unicorn by Domenichino (Public Domain)

Chapter 6

Ternary Plots

6.1 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 $\left\{(p_1, p_2, p_3) \in \mathbb{R}^3 : p_1 + p_2 + p_3 = 1 \text{ and } p_i \in [0,1] \text{ for } i = 1,2,3\right\}$.

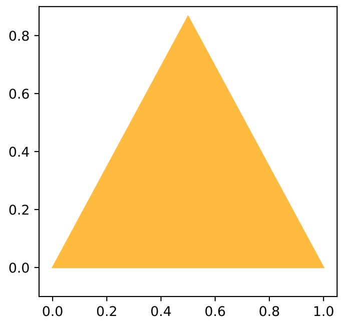


I encountered these diagrams in a few economics courses. Professor Bill Sandholm made particularly memorable use of these diagrams in his courses and research in evolutionary game theory.

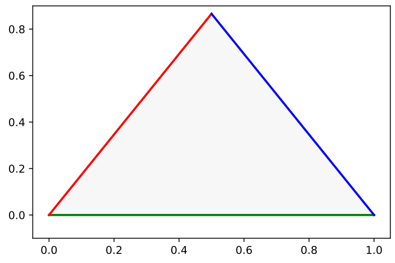
These plots aren't the most natural selection for inclusion in this text. It's a personal indulgence and a favor to other game theorists.

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')
```

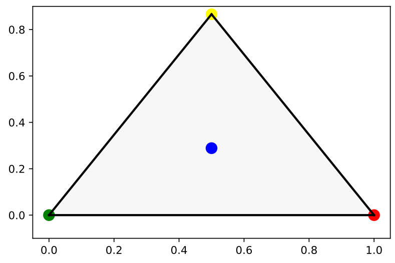


```
# Create the Plot
scale = 1 # length of the sides
figure, tax = ternary.figure(scale=scale)

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



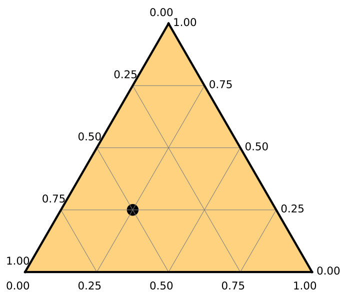
Next, let's add some points with the `scatter` method, which works as you might expect.

```
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)
```



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')
```



6.2 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.

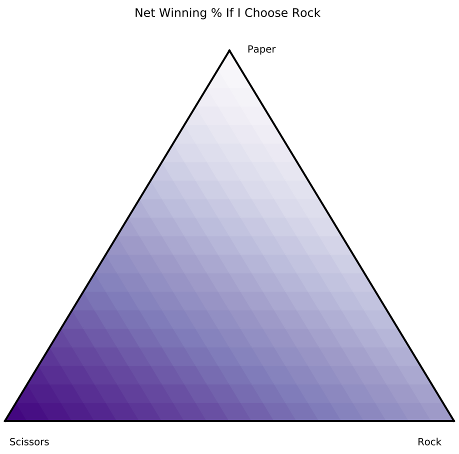
First, we'll construct a heatmap to show the net winning percentage from choosing a particular action depending on the opponent's probability distribution over the three actions, so that a point in the simplex is that opponent's strategy and the color represents how often you win. The following is a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

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

    return net
```

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

```
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()
```

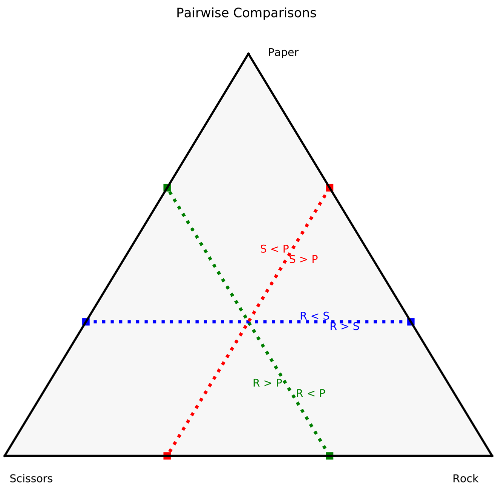


The natural extension of the above might be to repeat the above, which supposes you choose rock, for the actions paper and scissors. This can be done straightforwardly, by changing the default action in the `winning_pct` function. For the game theorist, the more interesting question is, given my opponent's 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)
```

```python
# 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', 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()
```

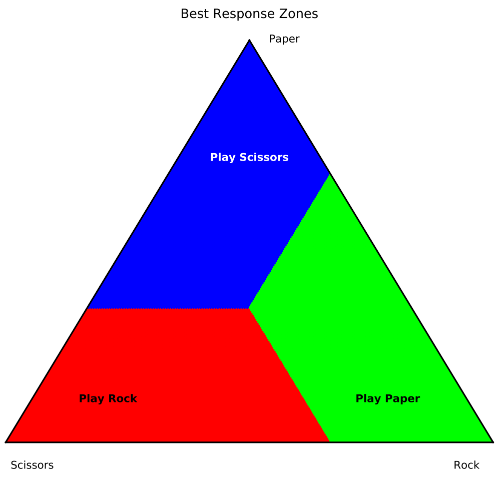


We can color best responses zones by using the `heatmap` method, though it will take some work. First, we have some pure Python coding to do. Recall that `heatmap` requires a correspondence of points and the colors you want. We won't use a colormap, but we'll pass the colors in explicitly as RGBA values.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

```
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.)
```

```
# 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()
```



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.

Bibliography

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

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

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

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

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

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

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

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

About the Author

His name is Alexander. He works at Peloton Interactive and is an adjunct lecturer 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. ✌