Design

One of the things that make data visualization fun and interesting is its artistic aspect. A beautiful visualization may not only be pleasing to the eyes, but also can be more effective and engaging in communicating the message. At the same time, the design principles are not arbitrary, but are based on the human perception and cognition.

To think about the importance of design in visualization, let us start with two visualization examples. The first one is "Gun deaths in Florida" from Reuters:

This graphic shows the number of murders committed using firearms in Florida for a couple of decades, highlighting the year (2005) when the "Stand Your Ground" law was enacted. It shows that the number of murders may be increasing after the law was enacted. It is hard to conclude anything from this single graph given the complex contexts that this graph does not show.

You may wonder, "wait, what do you mean by increasing? I think the number of murders is going down." Yes, that is the problem: this visualization is confusing! The y-axis is flipped—the top is 0 and the bottom is 1,000! This visualization is often mentioned as one of the most confusing visualizations.

Here is another (morbid) one:

It is similar to the previous one, in a sense that it flipped the y-axis and used the same red color. However, this visualization is extremely well-done and effective and conveying the message of "bloody toll"!

This visualization also lets us understand the idea behind the first visualization. "Ah, the first visualization was trying to have the same effect, representing the number of murders as the blood dripping down!" Actually, the creator of the first visualization said that they were directly inspired by the second one. But the problem is that it failed to do so (spectacularly), not because the data was bad, not because the idea was bad, not because bad visual encodings were used, but because of the design!

Readings (see Readings folder from General/Files for the pdf files)

- Tufte Data Ink ratio (chap 04 Tufte - Data-ink and graphical redesign)
- What Makes a Visualization Memorable? (what_makes_a_visualization_memorable.pdf)
- A tour through the visualization zoo (1794514.1805128.pdf)

Practical work

The aims are:

1. Learn about `matplotlib` 's colormaps, including the awesome `viridis` .
2. Learn how to adjust the design element of a basic plot in `matplotlib` .
3. Understand the differences between bitmap and vector graphics.
4. Learn what is SVG and how to create simple shapes in SVG.

First, import `numpy` and `matplotlib` libraries (don't forget the `matplotlib inline` magic command if you are using Jupyter notebook).

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Colors

For *quantitative* data we can specify the quantitative cases into *sequential* and *diverging*. "Sequential" means that the underlying value has a sequential ordering and the color also just needs to change sequentially and monotonically.

In the "diverging" case, there should be a meaningful anchor point. For instance, the correlation values may be positive or negative. Both large positive correlation and large negative correlation are important and the sign of the correlation has an important meaning. Therefore, we would like to stitch two sequential colormap together, one from zero to +1, the other from zero to -1.

Categorical (qualitative) colormaps

`numpy`

`numpy` is one of the most important packages in Python. As the name suggests it handles all kinds of numerical manipulations and is the basis of pretty much all scientific packages. Actually, a `pandas` "series" is essentially a `numpy` array and a dataframe is essentially a bunch of `numpy` arrays grouped together.

If you use it wisely, it can easily give you 10x, 100x or even 1000x speed-up, although `pandas` takes care of such optimization under the hood in many cases. If you want to study `numpy` more, check out the official tutorial and "From Python to Numpy" book:

- Numpy Quickstart tutorial (https://docs.scipy.org/doc/numpy/user/quickstart.html)
- From Python to Numpy (https://www.labri.fr/perso/nrougier/from-python-to-numpy/)

Plotting some trigonometric functions

Let's plot a sine and cosine function. By the way, a common trick to plot a function is creating a list of x coordinate values (evenly spaced numbers over an interval) first. `numpy` has a function called `linspace` (https://docs.scipy.org/doc/numpy-1.12.0/reference/generated/numpy.linspace.html) for that. By default, it creates 50 numbers that fill the interval that you pass.

In [2]:
```
np.linspace(start=0, stop=3)
```

Out[2]:
```
array([0.        , 0.06122449, 0.12244898, 0.18367347, 0.24489796,
       0.30612245, 0.36734694, 0.42857143, 0.48979592, 0.55102041,
       0.6122449 , 0.67346939, 0.73469388, 0.79591837, 0.85714286,
       0.91836735, 0.97959184, 1.04081633, 1.10204082, 1.16326531,
       1.2244898 , 1.28571429, 1.34693878, 1.40816327, 1.46938776,
       1.53061224, 1.59183673, 1.65306122, 1.71428571, 1.7755102 ,
       1.83673469, 1.89795918, 1.95918367, 2.02040816, 2.08163265,
       2.14285714, 2.20408163, 2.26530612, 2.32653061, 2.3877551 ,
       2.44897959, 2.51020408, 2.57142857, 2.63265306, 2.69387755,
       2.75510204, 2.81632653, 2.87755102, 2.93877551, 3.        ])
```

And a nice thing about `numpy` is that many operations just work with vectors.

In [3]:
```
np.linspace(0, 3, num=10)     # 10 numbers instead of 50

# notice how you do not need explicity write "start" & "stop" like the previous cell.
# Similarly, "num" is not needed.
# It exists here just to make the explanation clear
# and it can be good practice to include them to help the future readers of your coe (including yourself!)
```

Out[3]:
```
array([0.        , 0.33333333, 0.66666667, 1.        , 1.33333333,
       1.66666667, 2.        , 2.33333333, 2.66666667, 3.        ])
```

If you want to apply a function to every value in a vector, you simply pass that vector to the function.

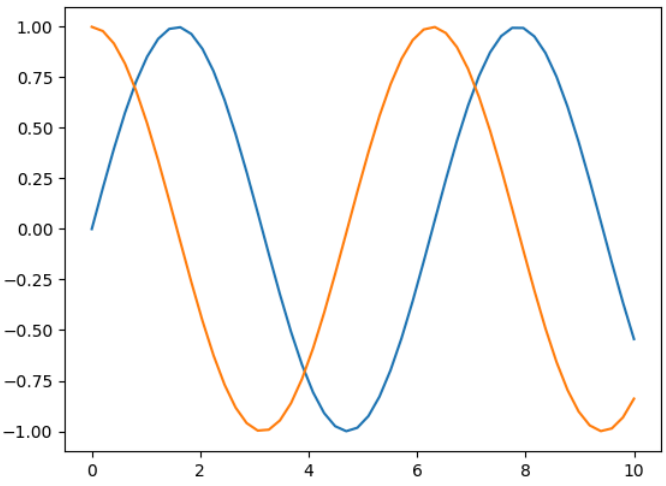
In [4]:
```
np.sin(np.linspace(0, 3, 10))
```

Out[4]:
```
array([0.        , 0.3271947 , 0.6183698 , 0.84147098, 0.9719379 ,
       0.99540796, 0.90929743, 0.72308588, 0.45727263, 0.14112001])
```

Q: Let's plot `sin` and `cos`

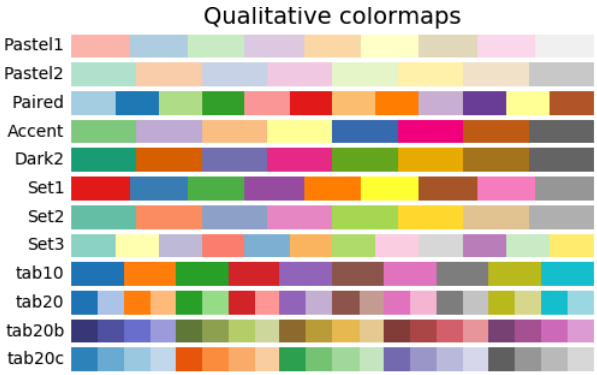
In [5]:
```
x = np.linspace(0, 10)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
```

Out[5]: `[<matplotlib.lines.Line2D at 0x2f715b70310>]`



`matplotlib` picks a pretty good color pair by default! Orange-blue pair is colorblind-safe.

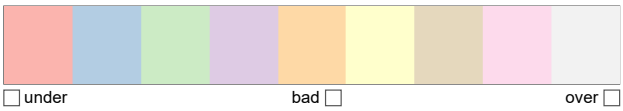
`matplotlib` has many qualitative (categorical) colorschemes. https://matplotlib.org/users/colormaps.html (https://matplotlib.org/users/colormaps.html)



You can access them through the following ways:

In [6]:
```
plt.cm.Pastel1
```

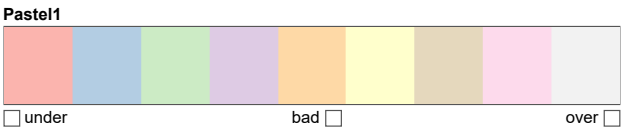
Out[6]:



or

In [7]:
```
pastel1 = plt.get_cmap('Pastel1')
pastel1
```

Out[7]:



You can also see the colors in the colormap in RGB.

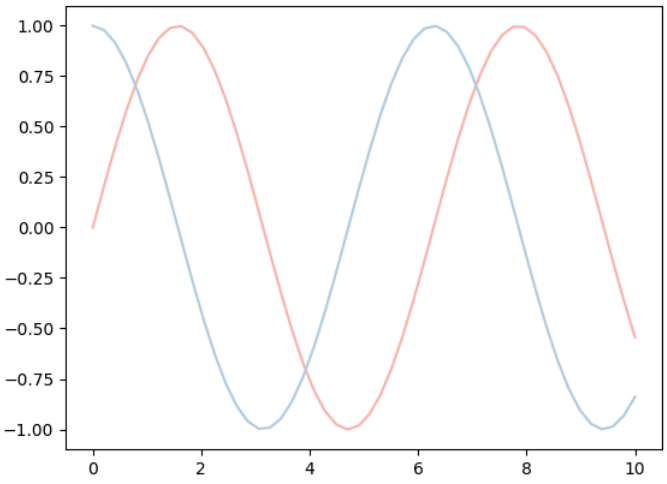
In [8]:
```
pastel1.colors
```

Out[8]:
```
((0.984313725490196, 0.7058823529411765, 0.6823529411764706),
 (0.7019607843137254, 0.803921568627451, 0.8901960784313725),
 (0.8, 0.9215686274509803, 0.7725490196078432),
 (0.8705882352941177, 0.796078431372549, 0.8941176470588236),
 (0.996078431372549, 0.8509803921568627, 0.6509803921568628),
 (1.0, 1.0, 0.8),
 (0.8980392156862745, 0.8470588235294118, 0.7411764705882353),
 (0.9921568627450981, 0.8549019607843137, 0.9254901960784314),
 (0.9490196078431372, 0.9490196078431372, 0.9490196078431372))
```

To get the first and second colors, you can use either ways:

In [9]:
```
plt.plot(x, np.sin(x), color=plt.cm.Pastel1(0))
plt.plot(x, np.cos(x), color=pastel1(1))
```

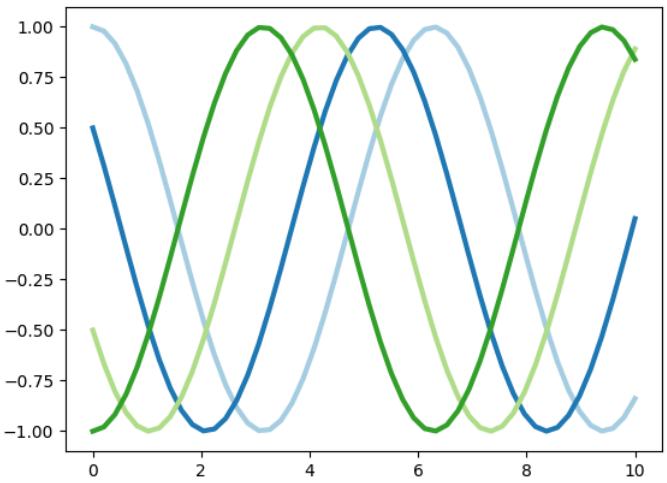
Out[9]:
```
[<matplotlib.lines.Line2D at 0x2f717d6e8f0>]
```



Q: pick a qualitative colormap and then draw four different curves with four different colors in the colormap.

Note that the colorschemes are not necessarily colorblindness-safe nor lightness-varied! Think about whether the colormap you chose is a good one or not.

In [27]:
```
for i in range(4):
    plt.plot(x, np.cos(x+i*np.pi/3), linewidth=3, color=plt.cm.Paired(i))
```



Quantitative colormaps

Take a look at the tutorial about image processing in `matplotlib` : http://matplotlib.org/users/image_tutorial.html (http://matplotlib.org/users/image_tutorial.html)

We can also display an image using quantitative (sequential) colormaps. Use the snake image or use other image of your liking.

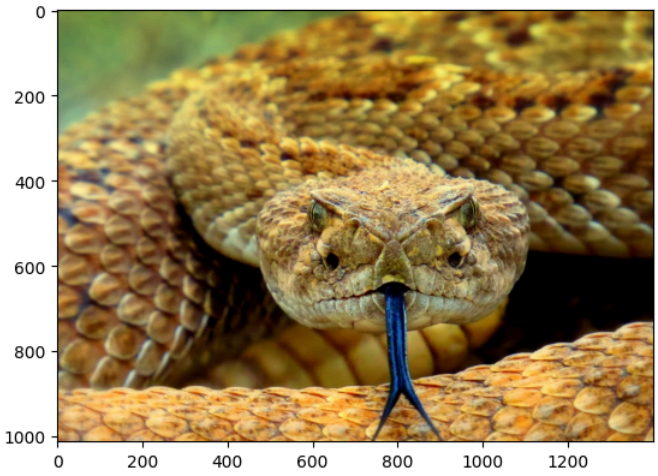
Check out `imread()` (http://matplotlib.org/api/image_api.html#matplotlib.image.imread) function that returns an `numpy.array()` .

In [28]:
```
import matplotlib.image as mpimg
```

In [30]:
```
img = mpimg.imread('sneakySnake.jpg')
```

In [31]:
```
plt.imshow(img)
```

Out[31]:
```
<matplotlib.image.AxesImage at 0x2f71be634c0>
```



How is the image stored?

In [32]:
```
img
```

Out[32]:
```
array([[[129, 153,  67],
        [128, 152,  66],
        [128, 152,  66],
        ...,
        [150, 114,  30],
        [150, 113,  32],
        [149, 112,  31]],

       [[129, 153,  67],
        [128, 152,  66],
        [128, 152,  66],
        ...,
        [147, 111,  27],
        [147, 110,  29],
        [147, 110,  29]],

       [[129, 153,  67],
        [128, 152,  66],
        [128, 152,  66],
        ...,
        [146, 109,  28],
        [145, 108,  28],
        [145, 108,  28]],

       ...,

       [[185, 124,  61],
        [182, 121,  58],
        [180, 118,  59],
        ...,
        [212, 182, 109],
        [211, 181, 109],
        [211, 181, 109]],

       [[181, 120,  57],
        [180, 118,  57],
        [179, 117,  60],
        ...,
        [212, 182, 109],
        [211, 181, 109],
        [211, 181, 109]],

       [[180, 118,  57],
        [179, 117,  56],
        [178, 116,  59],
        ...,
        [213, 183, 110],
        [211, 181, 109],
        [211, 181, 109]]], dtype=uint8)
```

`shape()` method lets you know the dimensions of the array.

```
In [33]: np.shape(img)
```

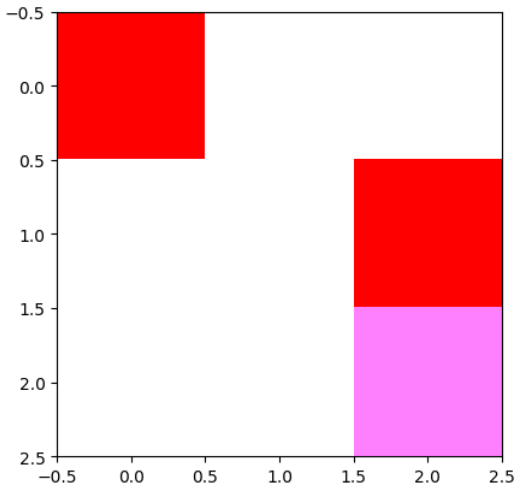
```
Out[33]: (1013, 1400, 3)
```

This means that `img` is a three-dimensional array with 219 x 329 x 4 numbers. If you look at the image, you can easily see that 219 and 329 are the dimensions (height and width in terms of the number of pixels) of the image. What is 4?

We can actually create our own small image to investigate. Let's create a 3x3 image.

```
In [34]: myimg = np.array([ [[1,0,0,1], [1,1,1,1], [1,1,1,1]],
                            [[1,1,1,1], [1,1,1,1], [1,0,0,1]],
                            [[1,1,1,1], [1,1,1,1], [1,0,1,0.5]] ])
         plt.imshow(myimg)
```

```
Out[34]: <matplotlib.image.AxesImage at 0x2f71c2ea0e0>
```



Q: Play with the values of the matrix, and explain what are each of the four dimensions (this matrix is 3x3x4) below.

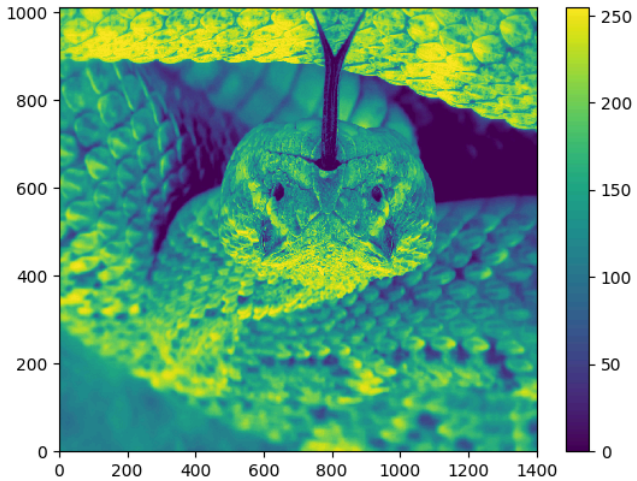
Write your answer here

Applying other colormaps

Let's assume that the first value of the four dimensions represents some data of your interest. You can obtain `height x width x 1` matrix by doing `img[:,:,0]` , which means give me the all of the first dimension (`:`), all of the second dimension (`:`), but only the first one from the last dimension (`0`).

```
In [35]: plt.pcolormesh(img[:,:,0], cmap=plt.cm.viridis)
         plt.colorbar()
```

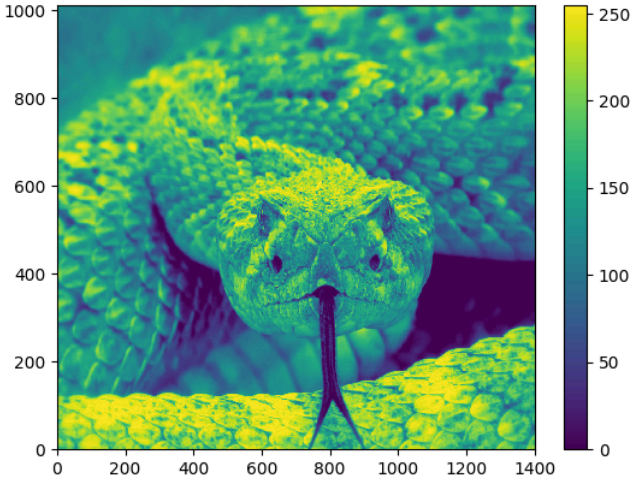
```
Out[35]: <matplotlib.colorbar.Colorbar at 0x2f71cd222f0>
```



Q: Why is it flipped upside down? Take a look at the previous `imshow` example closely and compare the axes across these two displays. Let's flip the figure upside down to show it properly. This function `numpy.flipud()` (http://docs.scipy.org/doc/numpy/reference/generated/numpy.flipud.html) may be handy.

In [36]:
```
plt.pcolormesh(np.flipud(img[:,:,0]), cmap=plt.cm.viridis)
plt.colorbar()
```

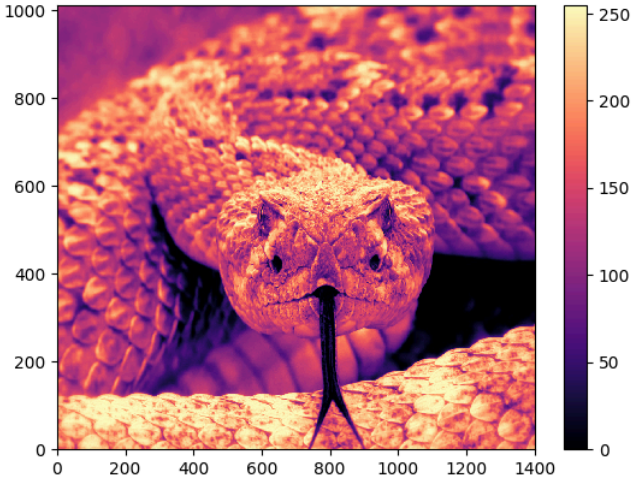
Out[36]: <matplotlib.colorbar.Colorbar at 0x2f71c767ca0>



Q: Try another sequential colormap here.

In [38]:
```
plt.pcolormesh(np.flipud(img[:,:,0]), cmap=plt.cm.magma)
plt.colorbar()
```

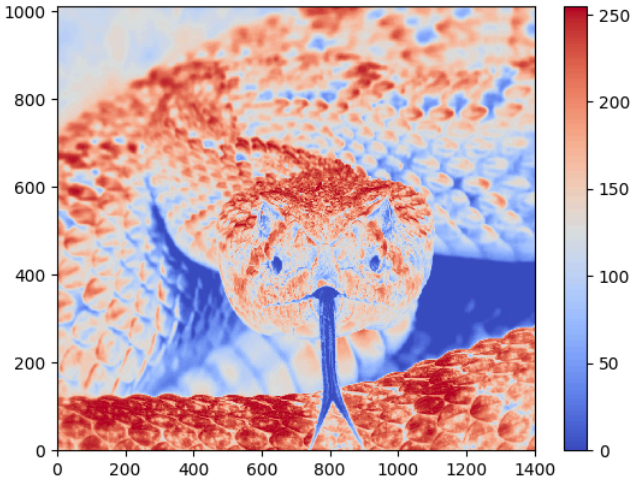
Out[38]: <matplotlib.colorbar.Colorbar at 0x2f71be805b0>



Q: Try a diverging colormap, say `coolwarm` .

In [39]:
```
plt.pcolormesh(np.flipud(img[:,:,0]), cmap=plt.cm.coolwarm)
plt.colorbar()
```

Out[39]: <matplotlib.colorbar.Colorbar at 0x2f71acb26b0>



Although there are clear choices such as `viridis` for quantitative data, you can come up with various custom colormaps depending on your application. For instance, take a look at this video about colormaps for Oceanography: https://www.youtube.com/watch?v=XjHzLUnHeM0 (https://www.youtube.com/watch?v=XjHzLUnHeM0) There is a colormap designed specifically for the *oxygen level*, which has three regimes.

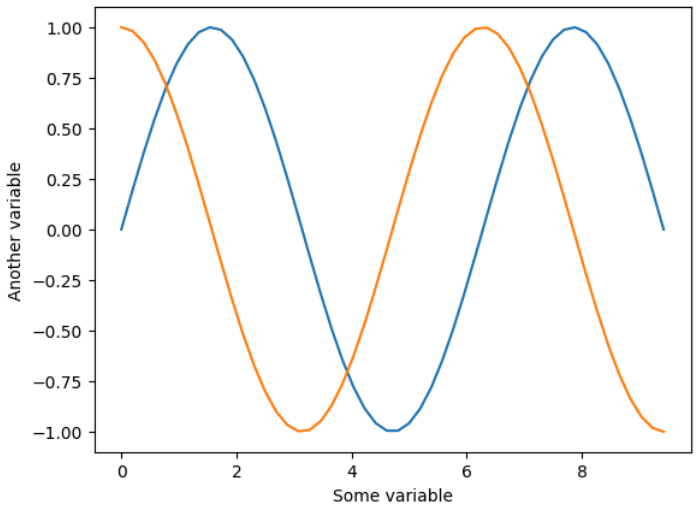
Adjusting a plot

First of all, always label your axes!

https://flowingdata.com/2012/06/07/always-label-your-axes/ (https://flowingdata.com/2012/06/07/always-label-your-axes/)

In [40]:
```
x = np.linspace(0, 3*np.pi)

plt.xlabel("Some variable")
plt.ylabel("Another variable")
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
```

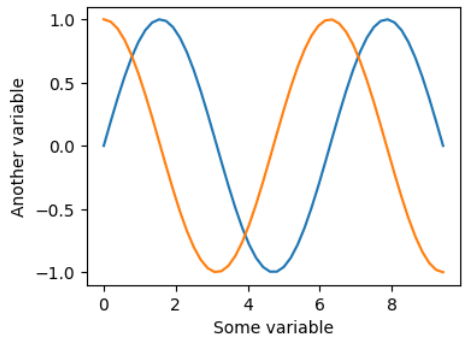
Out[40]:
```
[<matplotlib.lines.Line2D at 0x2f71ac12860>]
```



You can change the size of the whole figure by using `figsize` option. You specify the horizontal and vertical dimension in *inches*.

In [41]:
```
plt.figure(figsize=(4,3))
plt.xlabel("Some variable")
plt.ylabel("Another variable")
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
```

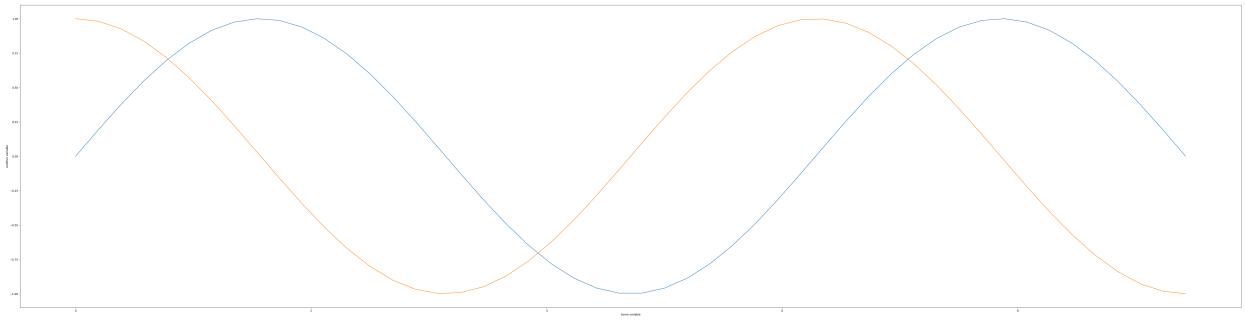
Out[41]: [<matplotlib.lines.Line2D at 0x2f71a98f730>]



A very common mistake is making the plot too big compared to the labels and ticks.

In [42]:
```
plt.figure(figsize=(80, 20))
plt.xlabel("Some variable")
plt.ylabel("Another variable")
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
```

Out[42]: [<matplotlib.lines.Line2D at 0x2f71aa9a860>]

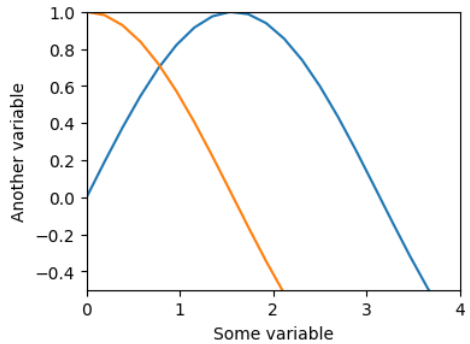


If you shrink this plot into a reasonable size, you cannot read the labels anymore! Actually this is one of the most common comments that I provide to my students!

You can adjust the range using `xlim` and `ylim`

In [43]:
```
plt.figure(figsize=(4,3))
plt.xlabel("Some variable")
plt.ylabel("Another variable")
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.xlim((0,4))
plt.ylim((-0.5, 1))
```

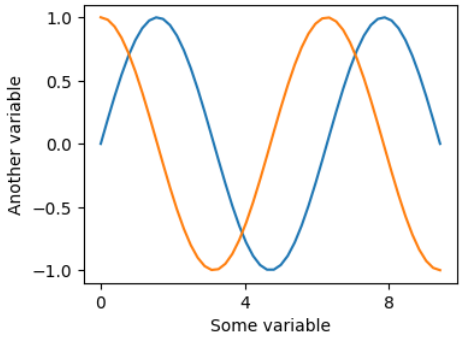
Out[43]: (-0.5, 1.0)



You can adjust the ticks.

In [44]:
```
plt.figure(figsize=(4,3))
plt.xlabel("Some variable")
plt.ylabel("Another variable")
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.xticks(np.arange(0, 10, 4))
```

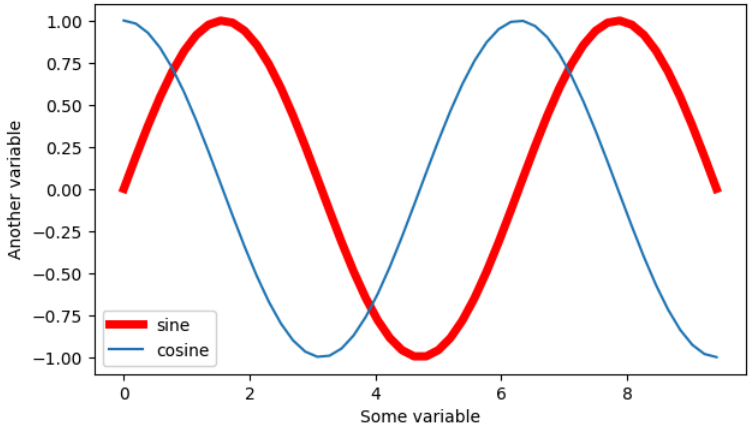
Out[44]:
```
([<matplotlib.axis.XTick at 0x2f717f28a00>,
  <matplotlib.axis.XTick at 0x2f717f29030>,
  <matplotlib.axis.XTick at 0x2f717f29a50>],
 [Text(0, 0, '0'), Text(4, 0, '4'), Text(8, 0, '8')])
```



colors, linewidth, and so on.

In [45]:
```
plt.figure(figsize=(7,4))
plt.xlabel("Some variable")
plt.ylabel("Another variable")
plt.plot(x, np.sin(x), color='red', linewidth=5, label="sine")
plt.plot(x, np.cos(x), label='cosine')
plt.legend(loc='lower left')
```

Out[45]:
```
<matplotlib.legend.Legend at 0x2f71ab8dc00>
```



For more information, take a look at this excellent tutorial: https://www.labri.fr/perso/nrougier/teaching/matplotlib/matplotlib.html (https://www.labri.fr/perso/nrougier/teaching/matplotlib/matplotlib.html)

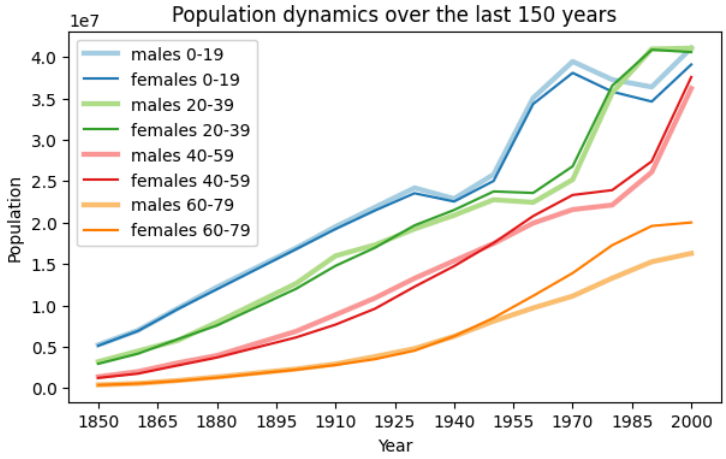
Q: Now, pick an interesting dataset (e.g. from `vega_datasets` package) and create a plot. Adjust the size of the figure, labels, colors, and many other aspects of the plot to obtain a nicely designed figure. Explain your rationales for each choice.

In [ ]:
```
from vega_datasets import data
df = data.population()
df.head()
```

Out[ ]:

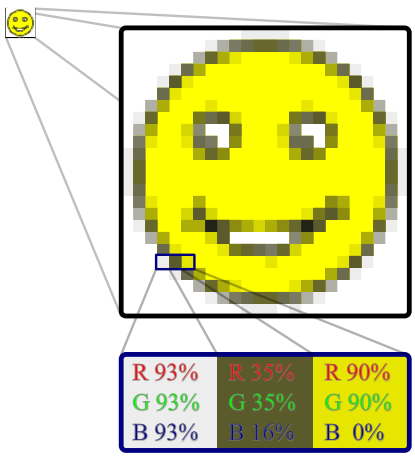
	year	age	sex	people
0	1850	0	1	1483789
1	1850	0	2	1450376
2	1850	5	1	1411067
3	1850	5	2	1359668
4	1850	10	1	1260099

In [126]:
```
plt.figure(figsize=(7,4)) # a bit wider than 4/3 to accomodate the timeline

plt.title('Population dynamics over the last 150 years')
plt.xlabel("Year")
plt.ylabel("Population")

plt.xticks(np.arange(1850, 2001, 15)) # pyplot default makes it 20 years tick which looks scarce

cmap = plt.cm.Paired # since we have m-f categorized dataset, we'll give
# each age class a hue and use different saturation for each sex

for i in range(4):
    m = df.query(f"{20*i} <= age < {20*(i+1)} & sex==1").groupby(by=[df.year]).people.sum()
    f = df.query(f"{20*i} <= age < {20*(i+1)} & sex==2").groupby(by=[df.year]).people.sum()
    # make the male lines thicker to compensare for their transparency
    plt.plot(m, color=cmap(2*i), label=f'males {20*i}-{20*(i+1)-1}', linewidth=3)
    plt.plot(f, color=cmap(2*i+1), label=f'females {20*i}-{20*(i+1)-1}')
plt.legend()
plt.show()
```

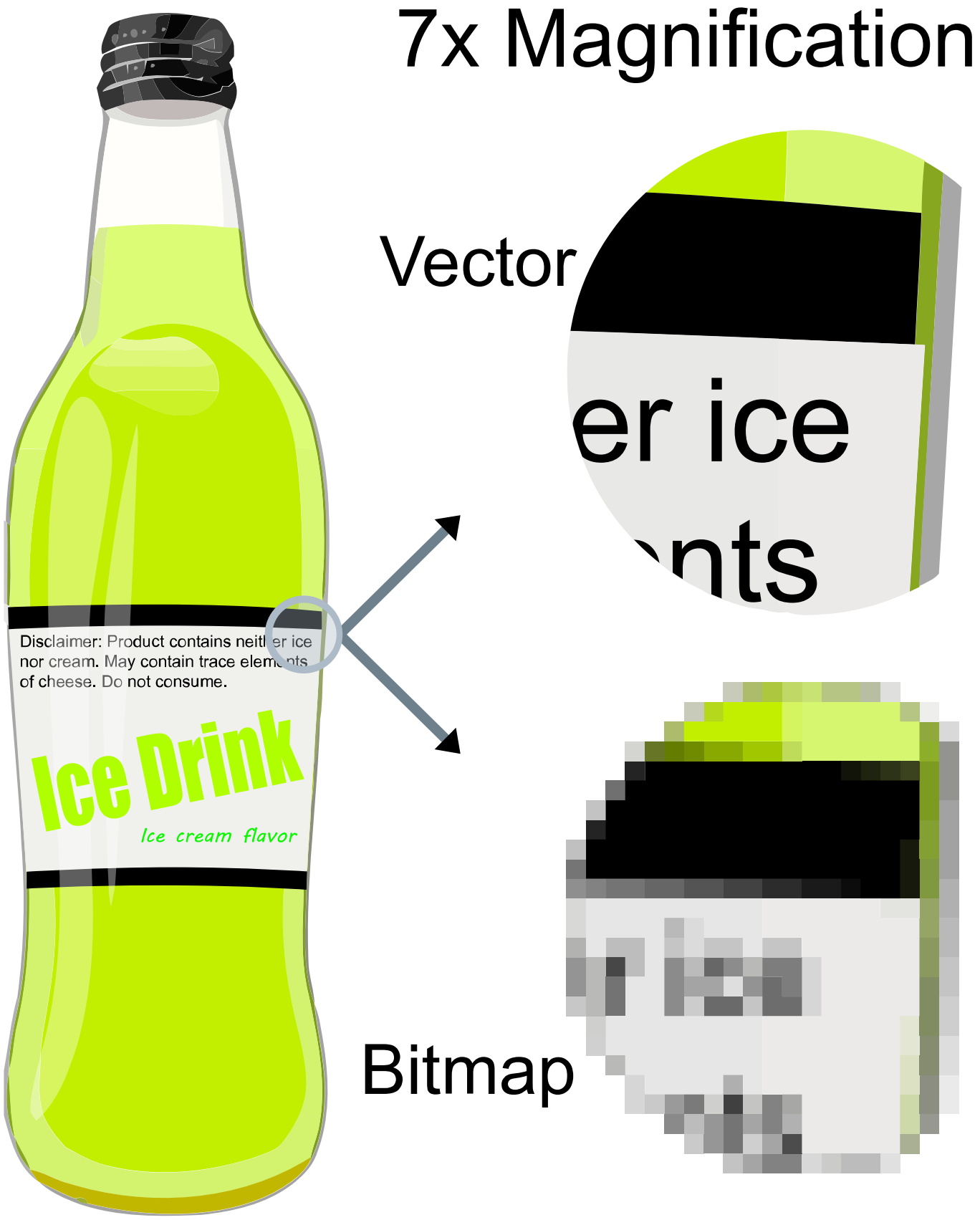


SVG

First of all, think about various ways to store an image, which can be a beautiful scenary or a geometric shape. How can you efficiently store them in a computer? Consider pros and cons of different approaches. Which methods would work best for a photograph? Which methods would work best for a blueprint or a histogram?



There are two approaches. One is storing the color of each pixel as shown above. This assumes that each pixel in the image contains some information, which is true in the case of photographs. Obviously, in this case, you cannot zoom in more than the original resolution of the image (~~if you're not in the movie~~). Also if you just want to store some geometric shapes, you will be wasting a lot of space. This is called **raster graphics**.



Another approach is using **vector graphics**, where you store the *instructions* to draw the image rather than the color values of each pixel. For instance, you can store "draw a circle with a radius of 5 at (100,100) with a red line" instead of storing all the red pixels corresponding to the circle. Compared to raster graphics (https://en.wikipedia.org/wiki/Raster_graphics), vector graphics (https://en.wikipedia.org/wiki/Vector_graphics) won't lose quality when zooming in.

Since a lot of data visualization tasks are about drawing geometric shapes, vector graphics is a common option. Most libraries allow you to save the figures in vector formats.

On the web, a common standard format is SVG (http://www.w3schools.com/svg/). SVG stands for "*Scalable Vector Graphics*". Because it's really a list of instructions to draw figures, you can create one even using a basic text editor. What many web-based drawing libraries do is simply writing down the instructions (SVG) into a webpage, so that a web browser can show the figure. The SVG format can be edited in many vector graphics software such as Adobe Illustrator and Inkscape. Although we rarely touch the SVG directly when we create data visualizations, I think it's very useful to understand what's going on under the hood. So let's get some intuitive understanding of SVG.

You can put an SVG figure by simply inserting a `<svg>` tag in an HTML file. It tells the browser to reserve some space for a drawing. For example,

```
<svg width="200" height="200">
   <circle cx="100" cy="100" r="22" fill="yellow" stroke="orange" stroke-width="5"/>
</svg>
```

This code creates a drawing space of 200x200 pixels. And then draw a circle of radius 22 at (100,100). The circle is filled with yellow color and *stroked* with 5-pixel wide orange line. That's pretty simple, isn't it? Place this code into an HTML file and open with your browser. Do you see this circle?

Another cool thing is that, because `svg` is an HTML tag, you can use `CSS` to change the styles of your shapes. You can adjust all kinds of styles using `CSS`:

```
<head>
<style>
.krypton_sun {
  fill: red;
  stroke: orange;
  stroke-width: 10;
}
</style>
</head>
<body>
<svg width="500" height="500">
   <circle cx="200" cy="200" r="50" class="krypton_sun"/>
</svg>
</body>
```

This code says "draw a circle with a radius 50 at (200, 200), with the style defined for `krypton_sun`". The style `krypton_sun` is defined with the `<style>` tag.

There are other shapes in SVG, such as ellipse (http://www.w3schools.com/graphics/svg_ellipse.asp), line (http://www.w3schools.com/graphics/svg_line.asp), polygon (http://www.w3schools.com/graphics/svg_polygon.asp) (this can be used to create triangles), and path (http://www.w3schools.com/graphics/svg_path.asp) (for curved and other complex lines). You can even place text with advanced formatting inside an `svg` element.

Exercise:

Let's reproduce the symbol for the Deathly Hallows (as shown below) with SVG. It doesn't need to be a perfect duplication (an equilateral triangle, etc), just be visually as close as you can. What's the most efficient way of drawing this? Color it in the way you like. Upload this file to canvas. Please Note: You have to upload the Deathly Hallows symbol as a separate HTML file to the canvas.

```
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
  <polygon points="100,20 10,190 190,190" style="fill:blue;stroke:black;stroke-width:5" />
  <circle cx="100" cy="135" r="50" stroke="black" fill="red" stroke-width="5"/>
  <line x1="100" y1="20" x2="100" y2="190" stroke="black" stroke-width="5" />
</svg>
```