W4 Lab Assignment

The aims of today's lab is

1. Learn about `matplotlib`'s colormaps, including the awesome `vidiris`.
2. Learn how to adjust the design element of a basic plot in `matplotlib`.

First, import `numpy` and `matplotlib` libraries (don't forget the `matplotlib inline` magic command).

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

Colors

We discussed about using colors for *categorical* and *quantitative* data. We can further 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 in a sequential manner without any breaking point. In the "diverging" case, there is a meaning anchor point in the quantity. 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

first some `numpy` basics

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(0, 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, 10) + 1    # 10 numbers instead of 50
```

```
Out[3]: array([ 1.        ,  1.33333333,  1.66666667,  2.        ,  2.33333333,
                2.66666667,  3.        ,  3.33333333,  3.66666667,  4.        ])
```

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

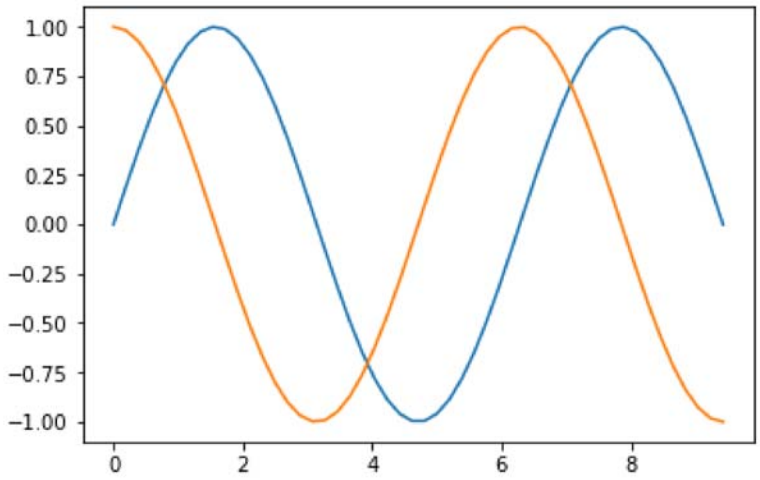
```
In [4]: np.sin(np.linspace(0, 3, 10))
```

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

Let's plot `sin` and `cos`

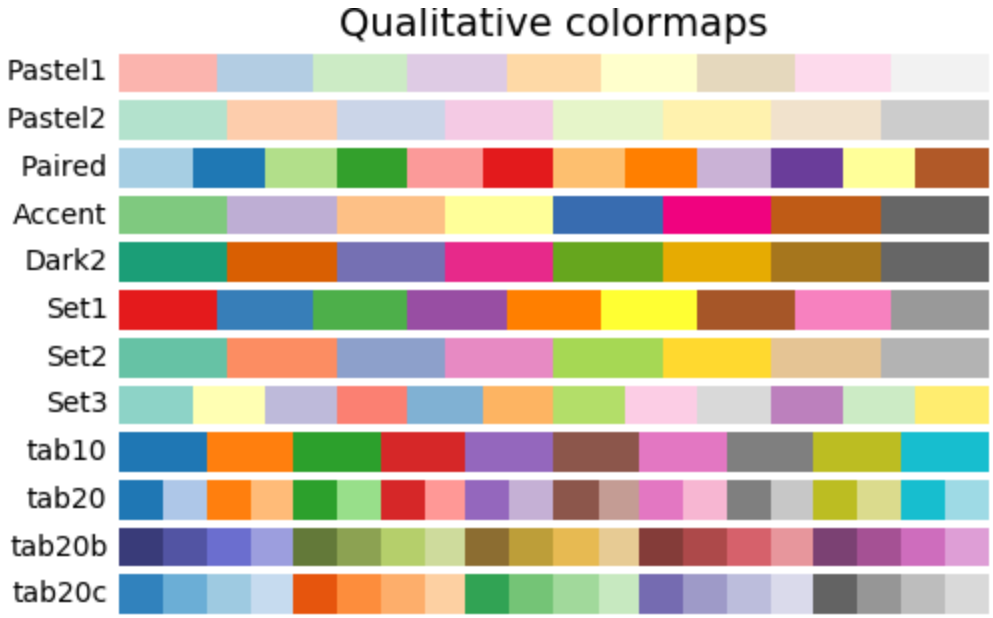
```
In [5]: x = np.linspace(0, 3*np.pi)
        plt.plot(x, np.sin(x))
        plt.plot(x, np.cos(x))
```

```
Out[5]: [<matplotlib.lines.Line2D at 0x2a1a78429b0>]
```



`matplotlib` picks a pretty good color pair by default! Orange-blue pair is colorblind-safe and it is like the color pair of every movie (https://www.google.com/search?q=why+every+movie+orange+blue).

`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 [12]: plt.cm.Paired
```

```
Out[12]: <matplotlib.colors.ListedColormap at 0x2a1a71dd908>
```

```
In [13]: paired1 = plt.get_cmap('Paired')
         paired1
```

```
Out[13]: <matplotlib.colors.ListedColormap at 0x2a1a71dd908>
```

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

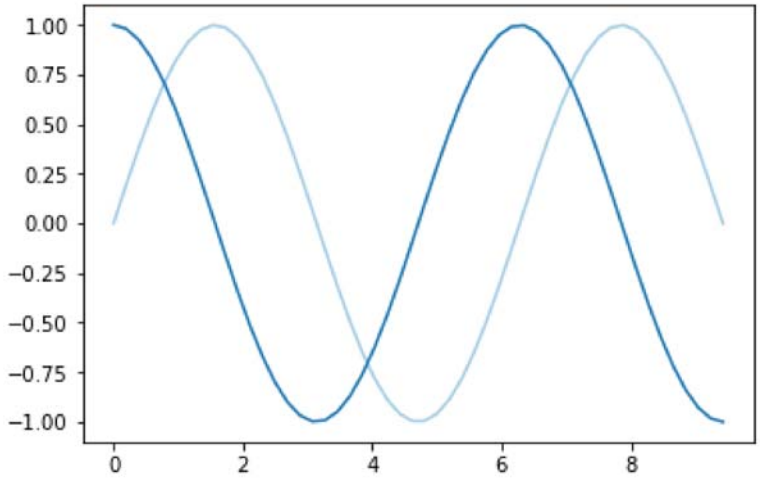
In [14]:
```
paired1.colors
```

Out[14]:
```
((0.6509803921568628, 0.807843137254902, 0.8901960784313725),
 (0.12156862745098039, 0.47058823529411764, 0.7058823529411765),
 (0.6980392156862745, 0.8745098039215686, 0.5411764705882353),
 (0.2, 0.6274509803921569, 0.17254901960784313),
 (0.984313725490196, 0.6039215686274509, 0.6),
 (0.8901960784313725, 0.10196078431372549, 0.10980392156862745),
 (0.9921568627450981, 0.7490196078431373, 0.43529411764705883),
 (1.0, 0.4980392156862745, 0.0),
 (0.792156862745098, 0.6980392156862745, 0.8392156862745098),
 (0.41568627450980394, 0.23921568627450981, 0.6039215686274509),
 (1.0, 1.0, 0.6),
 (0.6941176470588235, 0.34901960784313724, 0.1568627450980392))
```

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

In [15]:
```
plt.plot(x, np.sin(x), color=paired1(0))
plt.plot(x, np.cos(x), color=paired1(1))
```

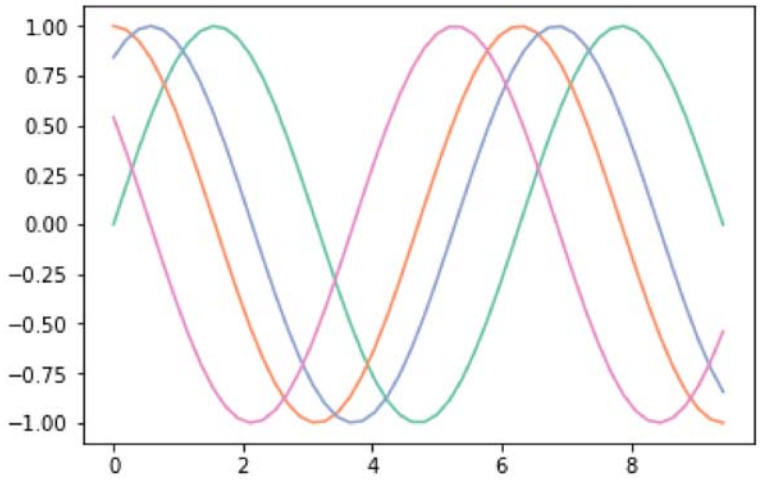
Out[15]:
```
[<matplotlib.lines.Line2D at 0x2a1a7ebbcf8>]
```



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 based on the criteria that we discussed.

In [18]:
```
# TODO: put your code here
cm =plt.get_cmap('Set2')

plt.plot(x, np.sin(x), color=cm(0))
plt.plot(x, np.cos(x), color=cm(1))

plt.plot(x, np.sin(x+1), color=cm(2))
plt.plot(x, np.cos(x+1), color=cm(3))
```

Out[18]:
```
[<matplotlib.lines.Line2D at 0x2a1a810ac50>]
```



Quantitative colormaps

http://matplotlib.org/users/image_tutorial.html (http://matplotlib.org/users/image_tutorial.html)

We can also display an image using quantitative (sequential) colormaps. Download the image of a snake: https://github.com/yy/dviz-course/blob/master/m05-design/sneakySnake.png (https://github.com/yy/dviz-course/blob/master/m05-design/sneakySnake.png) 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 [4]:
```
import matplotlib.image as mpimg
```

In [6]:
```
img = mpimg.imread('sneakySnake.png')
```

In [21]:
```
plt.imshow(img)
```

Out[21]:
```
<matplotlib.image.AxesImage at 0x2a1a7f11518>
```

How is the image stored?

In [7]:
```
img
```

Out[7]:
```
array([[[ 0.15294118,  0.21568628,  0.14117648,  1.        ],
        [ 0.16470589,  0.22745098,  0.15686275,  1.        ],
        [ 0.17254902,  0.24705882,  0.14509805,  1.        ],
        ...,
        [ 0.1882353 ,  0.22352941,  0.17647059,  1.        ],
        [ 0.1882353 ,  0.23921569,  0.18039216,  1.        ],
        [ 0.21960784,  0.29803923,  0.21176471,  1.        ]],

       [[ 0.1882353 ,  0.25490198,  0.17647059,  1.        ],
        [ 0.18431373,  0.26274511,  0.16470589,  1.        ],
        [ 0.3019608 ,  0.40392157,  0.25098041,  1.        ],
        ...,
        [ 0.18431373,  0.21176471,  0.18039216,  1.        ],
        [ 0.18039216,  0.21960784,  0.18431373,  1.        ],
        [ 0.17254902,  0.22745098,  0.17647059,  1.        ]],

       [[ 0.18431373,  0.25098041,  0.18039216,  1.        ],
        [ 0.24705882,  0.34117648,  0.2       ,  1.        ],
        [ 0.41960785,  0.5529412 ,  0.31764707,  1.        ],
        ...,
        [ 0.2       ,  0.23921569,  0.19607843,  1.        ],
        [ 0.17254902,  0.21176471,  0.17647059,  1.        ],
        [ 0.17647059,  0.22352941,  0.18431373,  1.        ]],

       ...,
       [[ 0.33725491,  0.41176471,  0.17647059,  1.        ],
        [ 0.29803923,  0.38039216,  0.14509805,  1.        ],
        [ 0.28235295,  0.35686275,  0.13725491,  1.        ],
        ...,
        [ 0.1254902 ,  0.17647059,  0.09411765,  1.        ],
        [ 0.10196079,  0.14901961,  0.07450981,  1.        ],
        [ 0.06666667,  0.10980392,  0.05882353,  1.        ]],

       [[ 0.28235295,  0.35294119,  0.15294118,  1.        ],
        [ 0.25490198,  0.32549021,  0.12156863,  1.        ],
        [ 0.25882354,  0.32156864,  0.12941177,  1.        ],
        ...,
        [ 0.06666667,  0.08627451,  0.0627451 ,  1.        ],
        [ 0.05882353,  0.07843138,  0.05490196,  1.        ],
        [ 0.0627451 ,  0.08235294,  0.06666667,  1.        ]],

       [[ 0.20784314,  0.27450982,  0.09803922,  1.        ],
        [ 0.21568628,  0.26666668,  0.08627451,  1.        ],
        [ 0.24313726,  0.27843139,  0.10196079,  1.        ],
        ...,
        [ 0.05098039,  0.05882353,  0.05490196,  1.        ],
        [ 0.06666667,  0.07450981,  0.07058824,  1.        ],
        [ 0.07450981,  0.08235294,  0.07843138,  1.        ]]], dtype=float32)
```

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

In [8]:
```
np.shape(img)
```

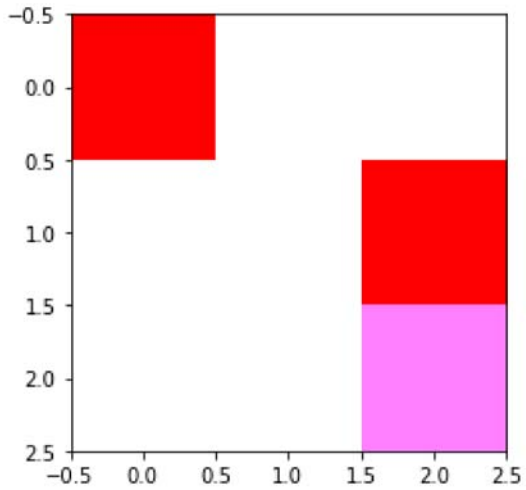
Out[8]:
```
(219, 329, 4)
```

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 [9]:
```
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[9]: `<matplotlib.image.AxesImage at 0x20de20dedd8>`



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

Each inner list represents a pixel. This image is an RGBA image with 4 values per inner list: R(ed), G(reen), B(lue) and A(lfa) or transparency.

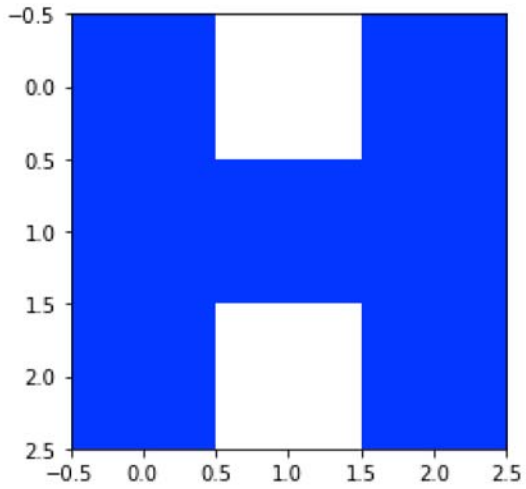
I would like to see the colors in light blue as follows

In [30]:
```
img1 = np.array([ [[0,.2,1,.99], [1,1,1,1], [0,.2,1,.99]],
                  [[0,.2,1,.99], [0,0.2,1,.99], [0,.2,1,.99]],
                  [[0,.2,1,.99], [1,1,1,1], [0,0.2,1,.99]] ])
np.shape(img1)
```

Out[30]: `(3, 3, 4)`

In [31]:
```
plt.imshow(img1)
```

Out[31]:
```
<matplotlib.image.AxesImage at 0x20de29a5828>
```

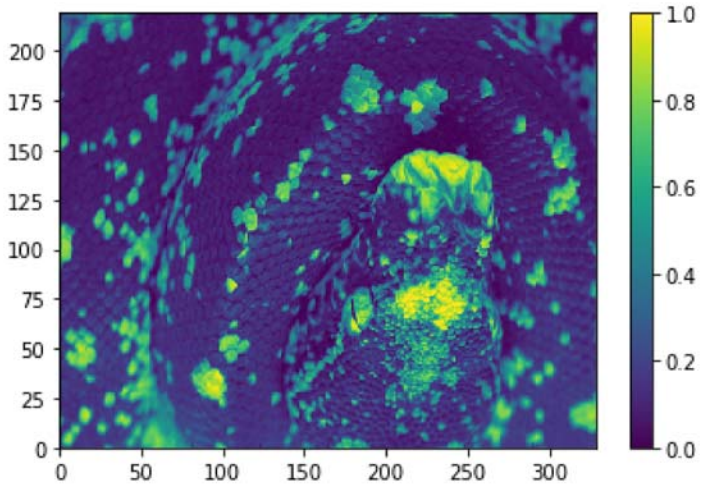


Applying other colormaps

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 [32]:
```
plt.pcolormesh(img[:,:,0], cmap=plt.cm.viridis)
plt.colorbar()
```

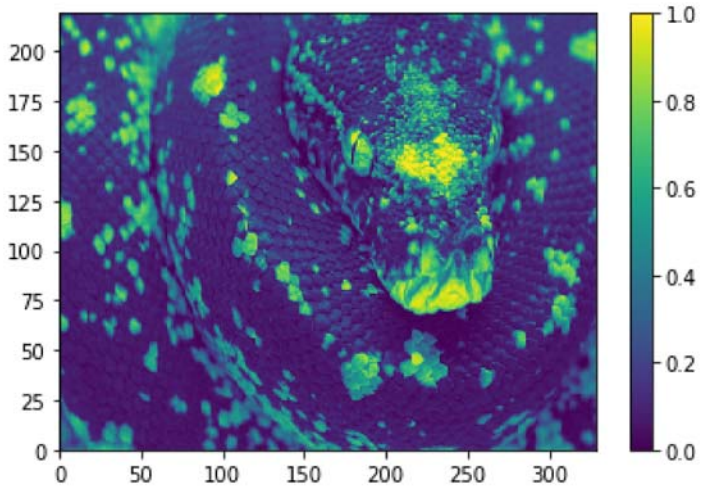
Out[32]:
```
<matplotlib.colorbar.Colorbar at 0x20de294a320>
```



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) is handy.

In [33]:
```
plt.pcolormesh(np.flipud(img[:,:,0]), cmap=plt.cm.viridis)
plt.colorbar()
```

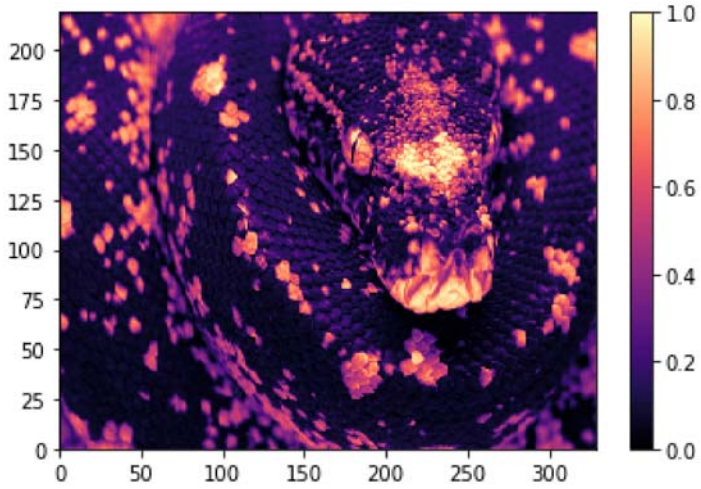
Out[33]: <matplotlib.colorbar.Colorbar at 0x20de3f2c588>



Q: Try another sequential colormap here.

In [36]:
```
# TODO: put your code here
plt.pcolormesh(np.flipud(img[:,:,0]), cmap=plt.cm.magma)
plt.colorbar()
```

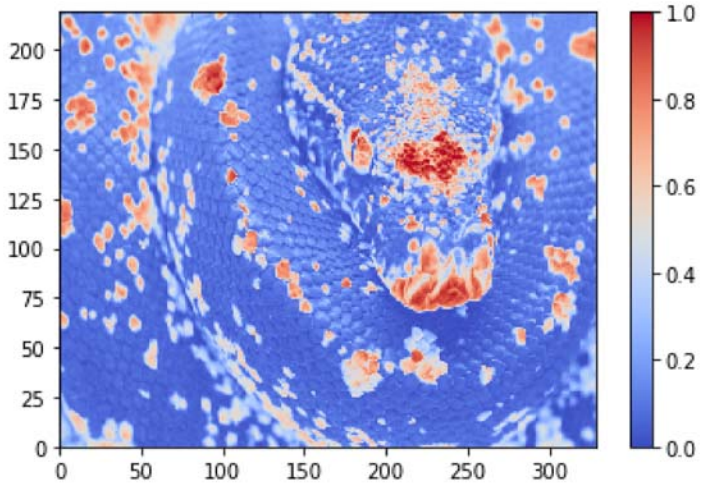
Out[36]: <matplotlib.colorbar.Colorbar at 0x20de490c828>



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

In [37]:
```
# TODO: put your code here
plt.pcolormesh(np.flipud(img[:,:,0]), cmap=plt.cm.coolwarm)
plt.colorbar()
```

Out[37]:
```
<matplotlib.colorbar.Colorbar at 0x20de4d69828>
```



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.

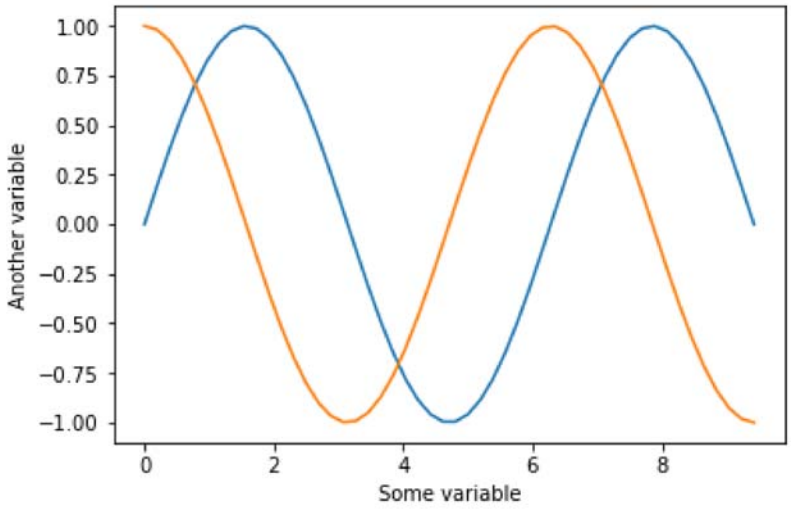
Adjusting a plot

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 [38]:
```
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))
```

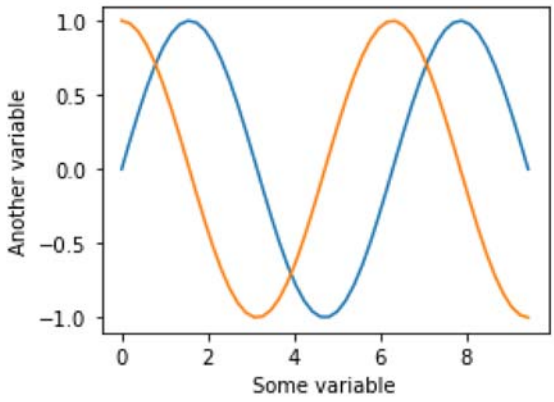
Out[38]: `[<matplotlib.lines.Line2D at 0x20de5215940>]`



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

In [39]:
```
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))
```

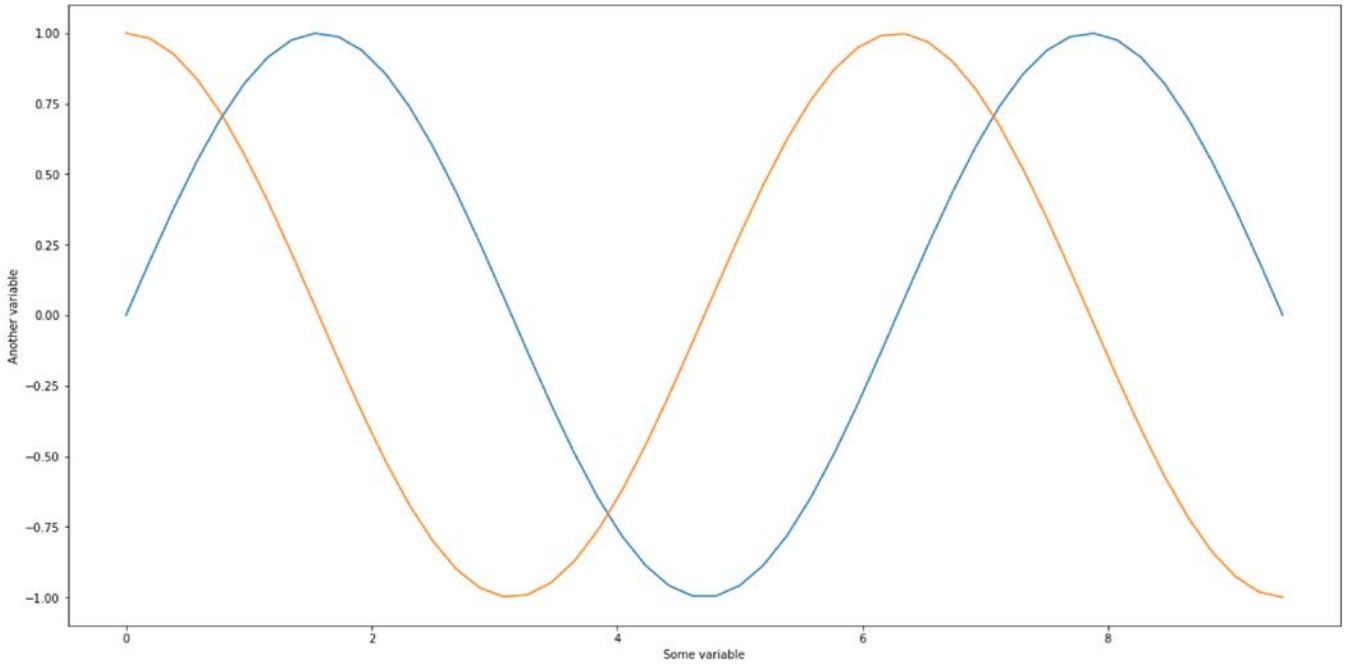
Out[39]: `[<matplotlib.lines.Line2D at 0x20de4d73668>]`



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

In [45]:
```
plt.figure(figsize=(20, 10))
plt.xlabel("Some variable")
plt.ylabel("Another variable")
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
```

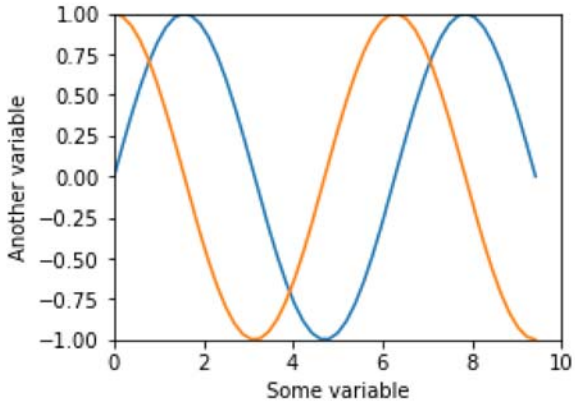
Out[45]:
```
[<matplotlib.lines.Line2D at 0x20de2acdb38>]
```



Once 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 [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.xlim((0,10))
plt.ylim((-1, 1))
```

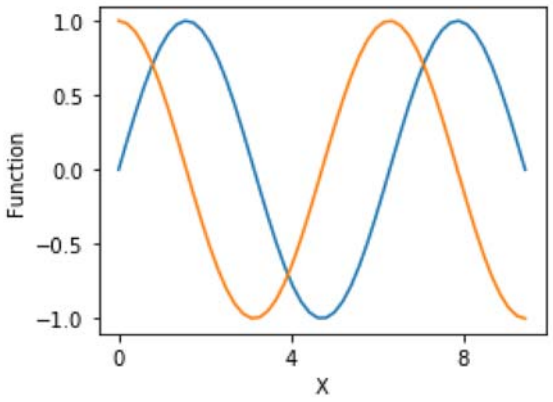
Out[44]:
```
(-1, 1)
```



You can adjust the ticks.

In [54]:
```
plt.figure(figsize=(4,3))
plt.xlabel("X")
plt.ylabel("Function")
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.xticks(np.arange(0, 10, 4))
```

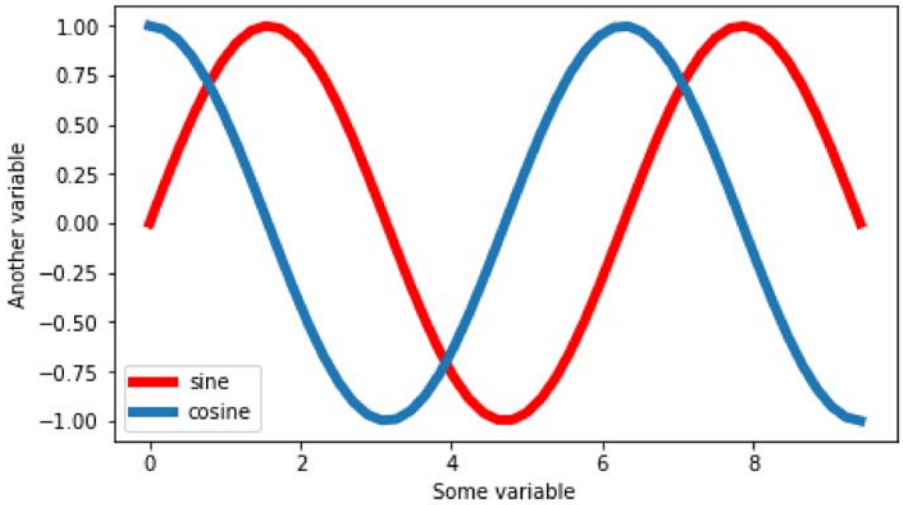
Out[54]:
```
([<matplotlib.axis.XTick at 0x20de5c75f60>,
  <matplotlib.axis.XTick at 0x20de5c25c18>,
  <matplotlib.axis.XTick at 0x20de5c8d550>],
 <a list of 3 Text xticklabel objects>)
```



colors, linewidth, and so on.

In [55]:
```
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', linewidth = 5)
plt.legend(loc='lower left')
```

Out[55]:
```
<matplotlib.legend.Legend at 0x20de58822b0>
```



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 [58]:
```
# TODO: put your code here
from vega_datasets import data
data.list_datasets()
```

Out[58]: ['7zip',
'airports',
'anscombe',
'barley',
'birdstrikes',
'budget',
'budgets',
'burtin',
'cars',
'climate',
'co2-concentration',
'countries',
'crimea',
'disasters',
'driving',
'earthquakes',
'ffox',
'flare',
'flare-dependencies',
'flights-10k',
'flights-200k',
'flights-20k',
'flights-2k',
'flights-3m',
'flights-5k',
'flights-airport',
'gapminder',
'gapminder-health-income',
'gimp',
'github',
'graticule',
'income',
'iowa-electricity',
'iris',
'jobs',
'la-riots',
'londonBoroughs',
'londonCentroids',
'londonTubeLines',
'lookup_groups',
'lookup_people',
'miserables',
'monarchs',
'movies',
'normal-2d',
'obesity',
'points',
'population',
'population_engineers_hurricanes',
'seattle-temps',
'seattle-weather',
'sf-temps',
'sp500',
'stocks',
'udistrict',
'unemployment',
'unemployment-across-industries',
'us-10m',
'us-state-capitals',
'weather',
'weball26',

```
 'wheat',
 'world-110m',
 'zipcodes']
```

In [62]:
```
df=data.gapminder()
```

In [63]:
```
df.head()
```

Out[63]:

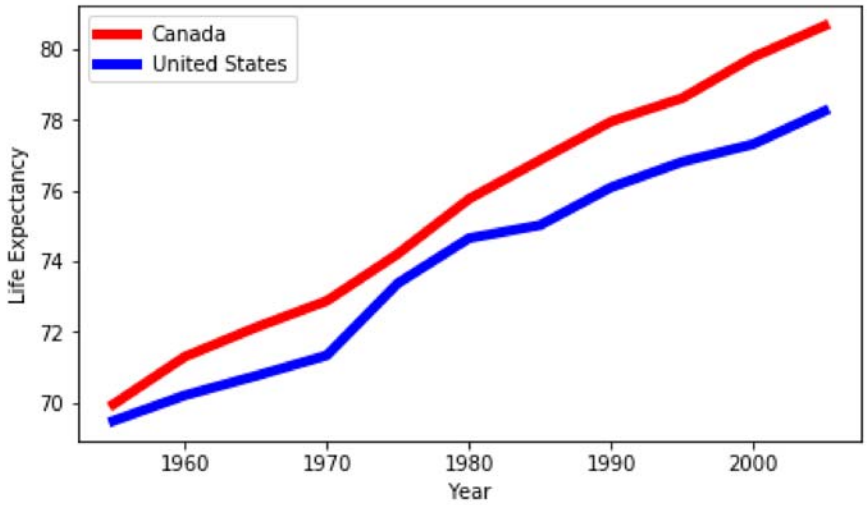
	cluster	country	fertility	life_expect	pop	year
0	0	Afghanistan	7.7	30.332	8891209	1955
1	0	Afghanistan	7.7	31.997	9829450	1960
2	0	Afghanistan	7.7	34.020	10997885	1965
3	0	Afghanistan	7.7	36.088	12430623	1970
4	0	Afghanistan	7.7	38.438	14132019	1975

In [66]:
```
import pandas as pd
```

In [77]:
```
#Compare life expectancy in Canada and US

df_canada=df.loc[df['country'] == 'Canada']
df_US=df.loc[df['country']=='United States']

l_exp_canada=df_canada["life_expect"].tolist()
year_canada=df_canada["year"].tolist()

l_exp_US=df_US["life_expect"].tolist()
year_US=df_US["year"].tolist()
```

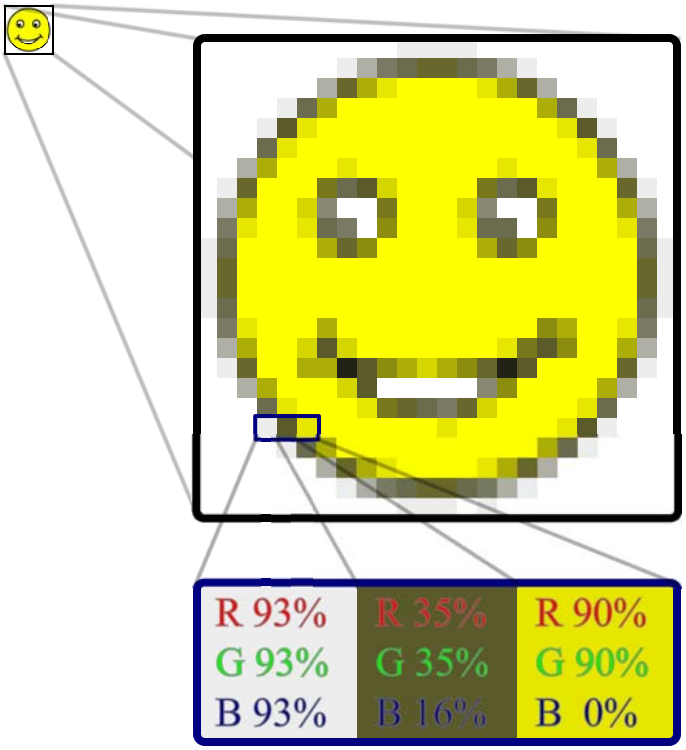
In [79]:
```
plt.figure(figsize=(7,4))
plt.xlabel("Year")
plt.ylabel("Life Expectancy")
plt.plot(year_canada, l_exp_canada, color='red',linewidth=5, label="Canada")
plt.plot(year_US, l_exp_US, color='blue',linewidth=5, label="United States")
plt.legend(loc='upper left')
```

Out[79]:
```
<matplotlib.legend.Legend at 0x20de6f6f048>
```



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**.

7x Magnification

Vector

Bitmap

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.

For more detail information about the difference between **raster** and **vector** images, including the conversion between the two types, read the following links:

1. Raster vs Vector (http://vector-conversions.com/vectorizing/raster_vs_vector.html)
2. Understanding the most commonly used image file formats (https://serialmentor.com/dataviz/image-file-formats.html)

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 writeing 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. 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.

In [83]:
```
#The most efficient way to draw the deathly hollows is to
#start with the cape as a polygon,
#add the stone as a circle and
#finish with the wand as a line:
```

In [ ]:
```
<!DOCTYPE html>
<html>
<style>
.hollows {
  fill: white;
  stroke: black;
  stroke-width: 5;
    }
</style>
<body>
 <svg width="600" height="600">
    <--Drawing the cape with poligon-->
    <polygon points="0,400 250,0 500,400" class="hollows" />

    <--Drawing the stone-->
    <circle cx="250" cy="260" r="135" class="hollows"/>

    <--Drawing the wand-->
    <line x1="250" y1="0" x2="250" y2="400"  class="hollows" />
  </svg>
</body>
</html>
```