# Design module Lab Assignment

**The aims of this lab 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 [15]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

## Colors

**We discussed 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 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**
- **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` for that. By default, it creates 50 numbers that fill the interval that you pass.**

In [ ]:

```
np.linspace(0, 3)
```

Out[ ]:

```
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,
```

```
0.6122449 , 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 [ ]:

```
np.linspace(0, 3, 10)      # 10 numbers instead of 50
```

Out[ ]:

```
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 [ ]:

```
np.sin(np.linspace(0, 3, 10))
```

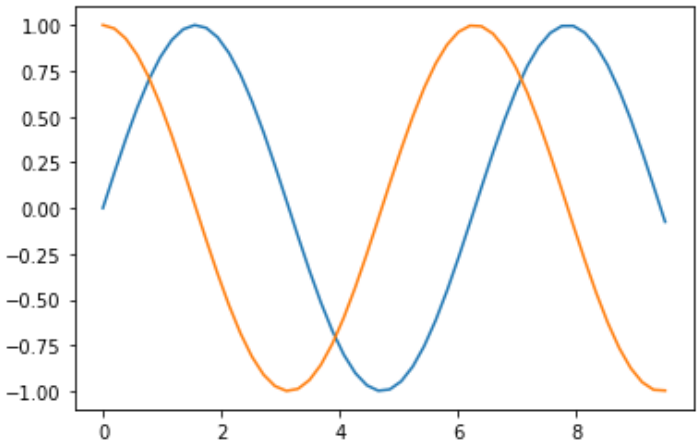
Out[ ]:

```
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`**

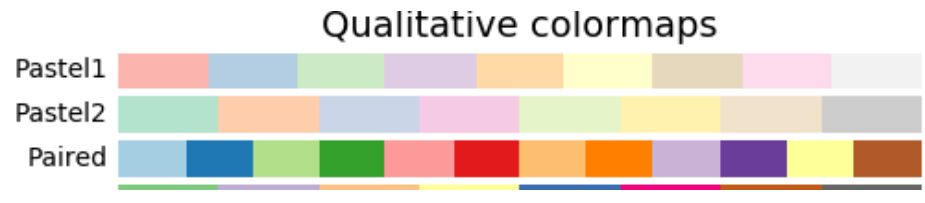
In [ ]:

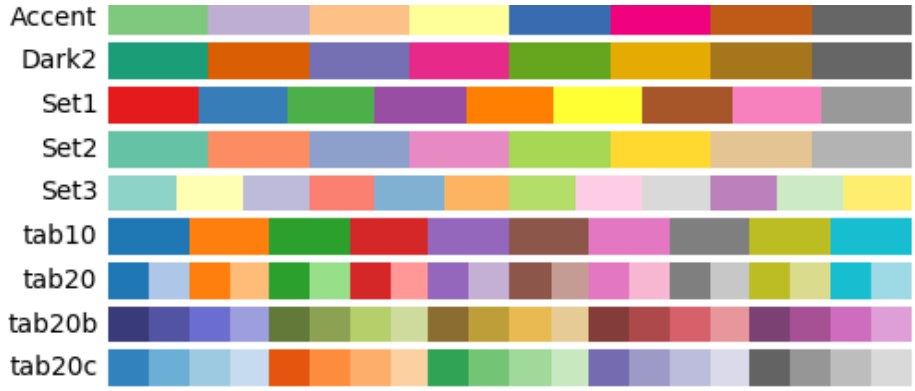
```
# TODO: put your code here
x = np.linspace(0, 9.5, 50)
y = np.sin(x)
z = np.cos(x)
plt.plot(x,y, x, z)
plt.show()
```




`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](#).**

`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 [ ]:

```
plt.cm.Pastel1
```

Out[ ]:

```
<matplotlib.colors.ListedColormap at 0x11bdfb048>
```

**or**

In [ ]:

```
pastel1 = plt.get_cmap('Pastel1')
pastel1
```

Out[ ]:

```
<matplotlib.colors.ListedColormap at 0x7fbee1c84490>
```

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

In [ ]:

```
pastel1.colors
```

Out[ ]:

```
((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:**

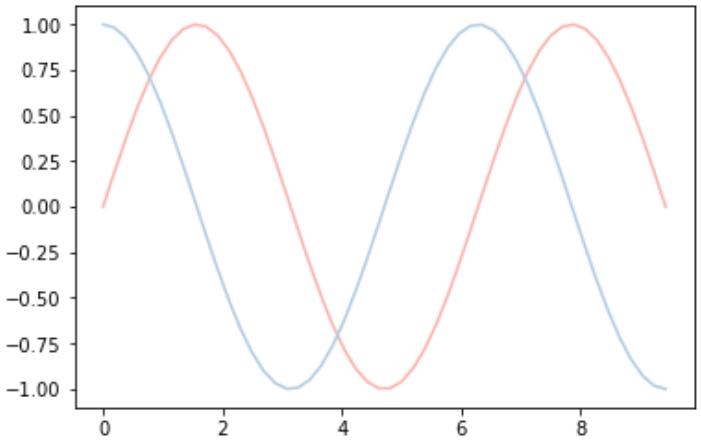
In [ ]:

```
plt.plot(x, np.sin(x), color=plt.cm.Pastel1(0))
plt.plot(x, np.cos(x), color=pastel1(1))
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x11c960ac8>]
```

[<matplotlib.lines.Line2D at 0x11c960ac6>]

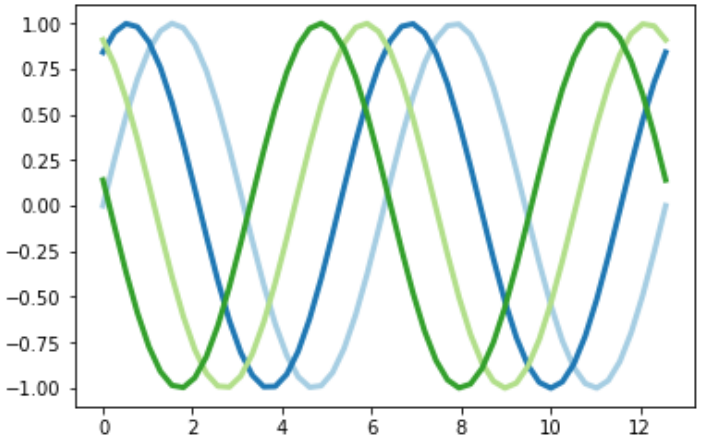



**Q: pick a qualitative colormap and then draw four different curves with four different colors in the colormap.**

**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 [ ]:

```
# TODO: put your code here
```

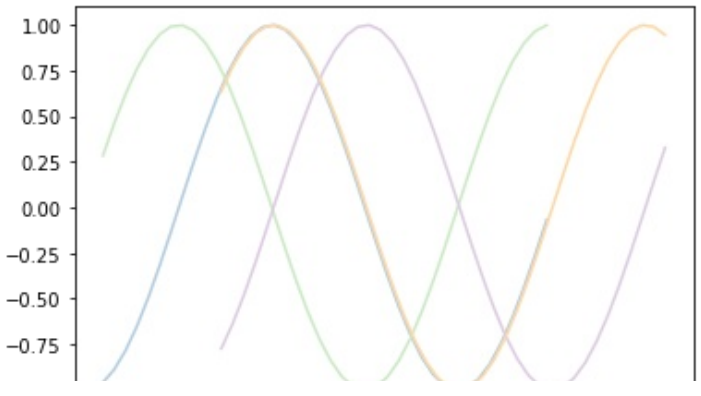


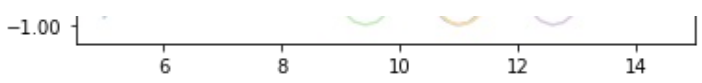

In [ ]:

```
x = np.linspace(5, 12.5, 40, endpoint = True)
y = np.sin(x)
y1 = np.cos(x)
y2 = np.sin(x + .4)
y3 = np.cos(x + .4)

plt.plot( x, y, color = pastel1(1))
plt.plot( x, y1, color = pastel1(2))
plt.plot( x + 2, y2, color = pastel1(3))
plt.plot( x + 2, y3, color= pastel1(4))
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x7fbed5cb2d50>]
```






## Quantitative colormaps

**Take a look at the tutorial about image processing in `matplotlib`:**
**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 or use other image of your liking.**

**Check out `imread()` function that returns an `numpy.array()`.**

In [2]:

```
import matplotlib.image as mpimg
```

In [13]:

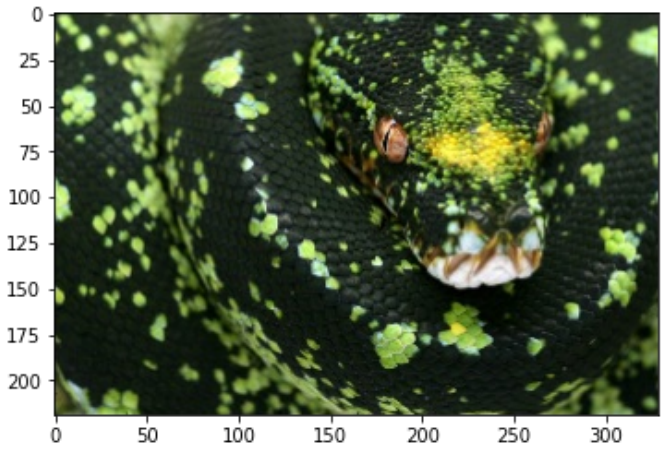
```
img = mpimg.imread('sneakySnake.png')
```

In [60]:

```
plt.imshow(img)
```

Out[60]:

```
<matplotlib.image.AxesImage at 0x7fc3a3782290>
```




**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, 0.1764706 , 1.        ],
        [0.1882353 , 0.23921569, 0.18039216, 1.        ],
        [0.21960784, 0.29803923, 0.21176471, 1.        ]],

       [[0.1882353 , 0.25490198, 0.1764706 , 1.        ],
        [0.18431373, 0.2627451 , 0.16470589, 1.        ],
        [0.3019608 , 0.40392157, 0.2509804 , 1.        ],
        ...,
        [0.18431373, 0.21176471, 0.18039216, 1.        ],
        [0.18039216, 0.21960784, 0.18431373, 1.        ],
        [0.17254902, 0.22745098, 0.1764706 , 1.        ]],
```

```
[[0.18431373, 0.2509804 , 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, 0.1764706 , 1.        ],
 [0.1764706 , 0.22352941, 0.18431373, 1.        ]],

...,

[[0.3372549 , 0.4117647 , 0.1764706 , 1.        ],
 [0.29803923, 0.38039216, 0.14509805, 1.        ],
 [0.28235295, 0.35686275, 0.13725491, 1.        ],
 ...,
 [0.1254902 , 0.1764706 , 0.09411765, 1.        ],
 [0.10196079, 0.14901961, 0.07450981, 1.        ],
 [0.06666667, 0.10980392, 0.05882353, 1.        ]],

[[0.28235295, 0.3529412 , 0.15294118, 1.        ],
 [0.25490198, 0.3254902 , 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, 0.2784314 , 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?**

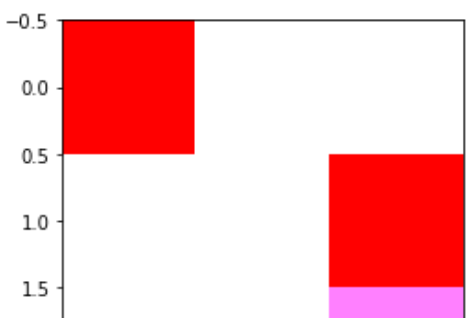
**We can actually create our own small image to investigate. Let's create a 3x3 image.**

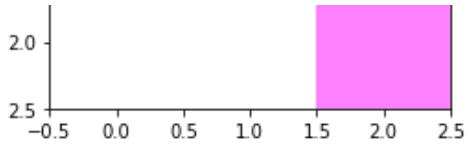
In [9]:

```
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 0x7fc3a54e4590>
```






**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** This matrix is 3 by 3 by 4

**3: Three blocks from -.5 to -.5 is the height**

**3: Three blocks from -.5 to -.5 is the width**

**4: Each block consists of 4 values**

## Applying other colormaps

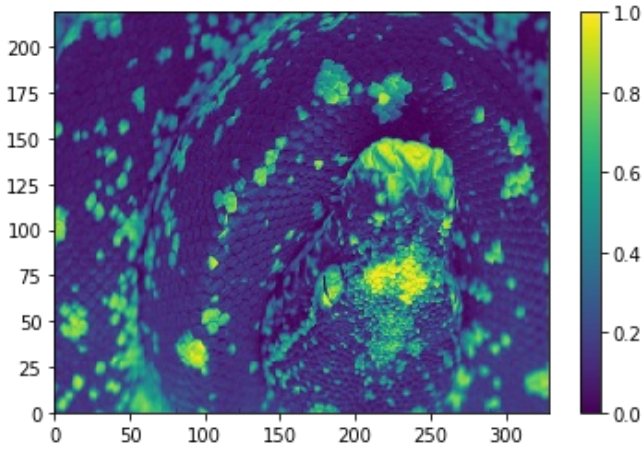
**Let's assume that the first value of the four dimensions represents some data of your interest. You can obtain `height x width x 1` matrix by doing `img[:,:,0]`, which means give me the all of the first dimension ( `:` ), all of the second dimension ( `:` ), but only the first one from the last dimension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

In [68]:

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

Out[68]:

```
<matplotlib.colorbar.Colorbar at 0x7fc3a31d3ed0>
```




**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()` may be handy.**

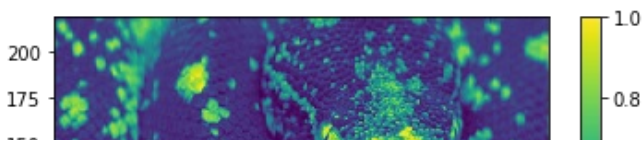
**This is happening because the first array is begging from the lower right of the image**

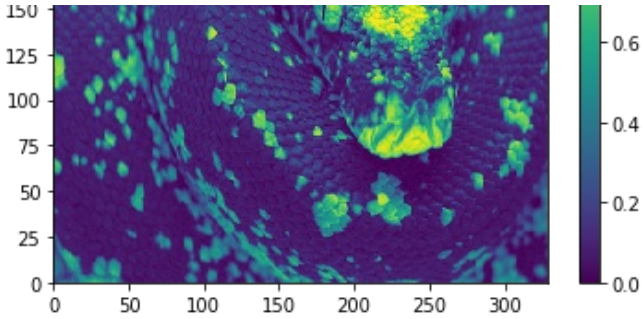
In [70]:

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

Out[70]:

```
<matplotlib.colorbar.Colorbar at 0x7fc3a309ff10>
```





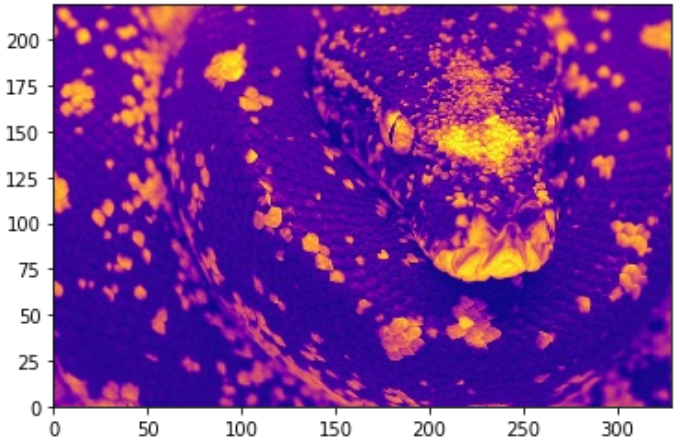

**Q: Try another sequential colormap here.**

In [76]:

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

Out[76]:

```
<matplotlib.collections.QuadMesh at 0x7fc3a37cfd90>
```



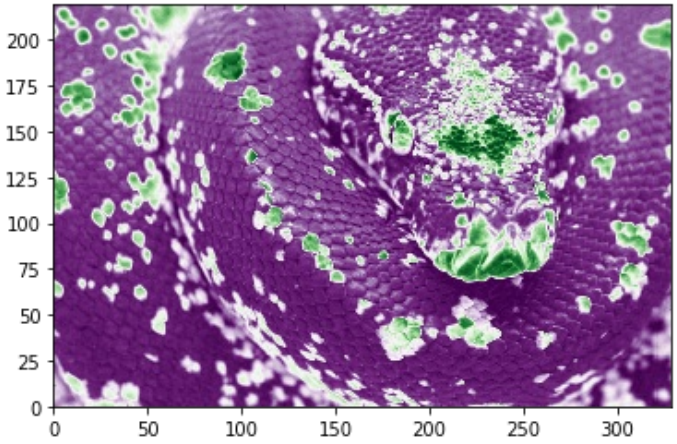

**Q: Try a diverging colormap, say `coolwarm`.**

In [77]:

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

Out[77]:

```
<matplotlib.collections.QuadMesh at 0x7fc3a47e5c90>
```




**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 There is a colormap designed specifically for the *oxygen level*, which has three regimes.**

## Adjusting a plot

**First of all, always label your axes!**

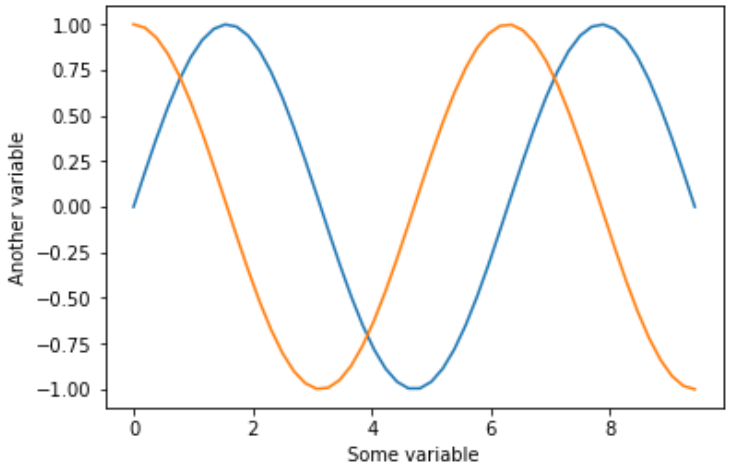
https://flowingdata.com/2012/06/07/always-label-your-axes/


In [ ]:

```
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[ ]:

```
[<matplotlib.lines.Line2D at 0x11d9d23c8>]
```



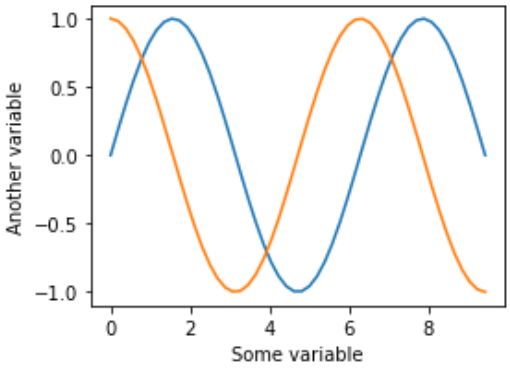

**You can change the size of the whole figure by using `figsize` option. You specify the horizontal and vertical dimension in *inches*.**

In [ ]:

```
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[ ]:

```
[<matplotlib.lines.Line2D at 0x11ca899e8>]
```



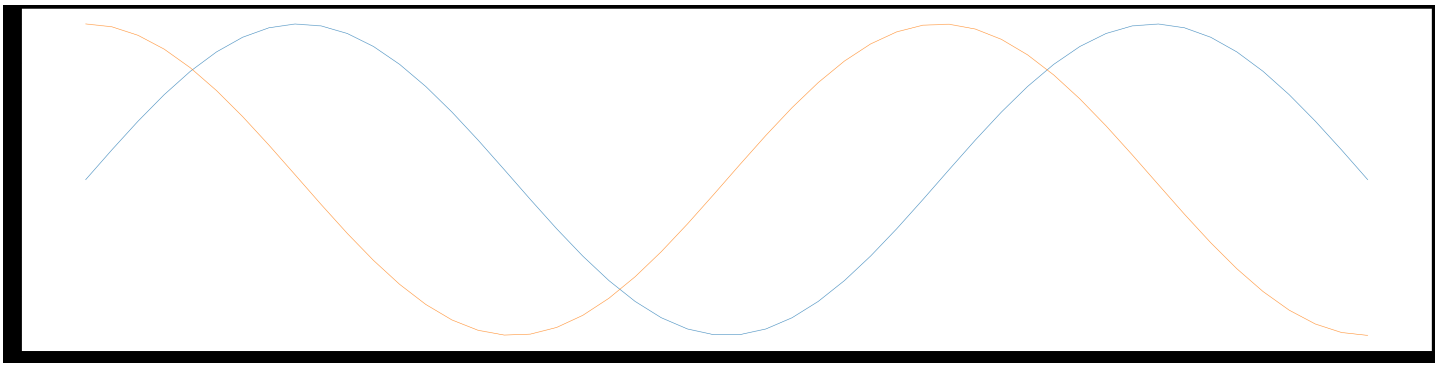

**A very common mistake is making the plot too big compared to the labels and ticks.**

In [ ]:

```
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[ ]:

```
[<matplotlib.lines.Line2D at 0x11d3a4160>]
```



**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!**

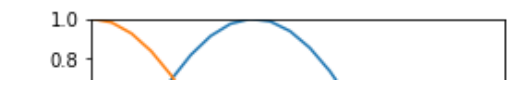
**You can adjust the range using `xlim` and `ylim`**

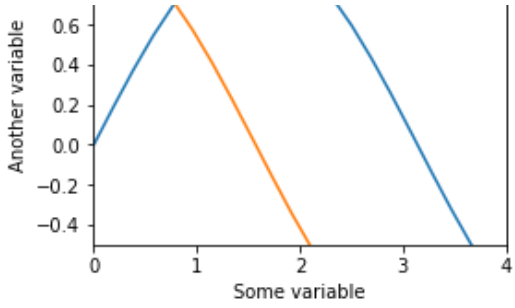
In [ ]:

```
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[ ]:

```
(-0.5, 1)
```





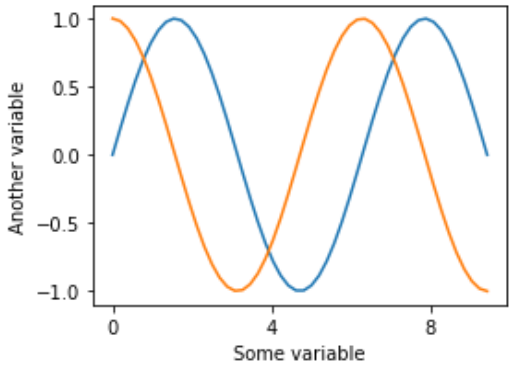

**You can adjust the ticks.**

In [ ]:

```
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[ ]:

```
([<matplotlib.axis.XTick at 0x11cae7550>,
  <matplotlib.axis.XTick at 0x11cacac50>,
  <matplotlib.axis.XTick at 0x11caca128>],
 <a list of 3 Text xticklabel objects>)
```



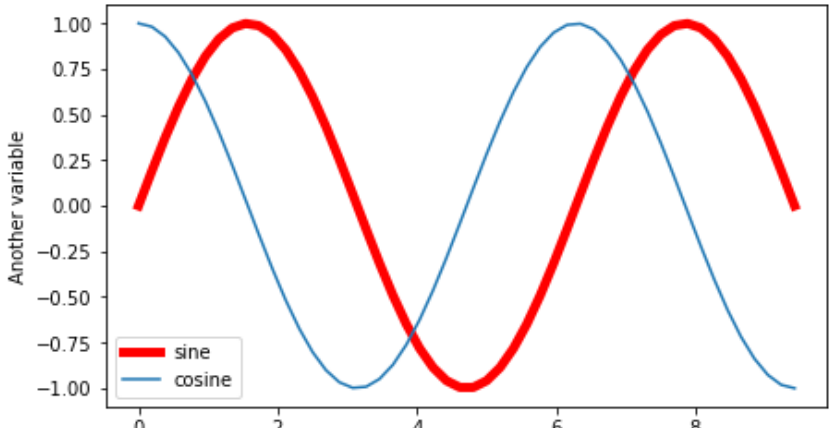

**colors, linewidth, and so on.**

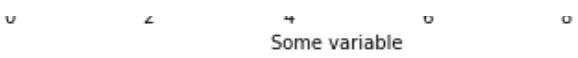
In [ ]:

```
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[ ]:

```
<matplotlib.legend.Legend at 0x11d199a90>
```






**For more information, take a look at this excellent tutorial:**
**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 [108]:

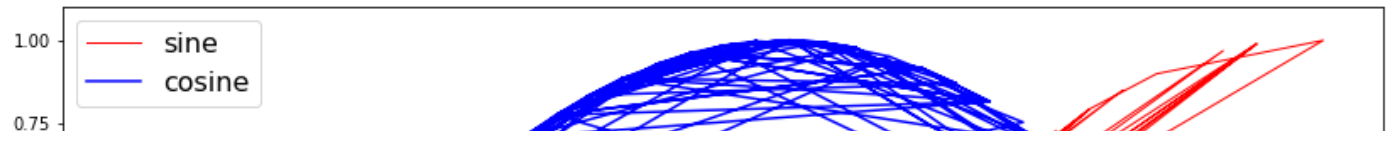
```
# TODO: put your code here
from vega_datasets import data

data = data.iris()

print(data)

plt.figure(figsize=(14,8))
plt.xlabel("SepalLength")
plt.ylabel("SinCosSepalLength")
plt.plot(data.sepalLength, np.sin(data.sepalLength), color='red', linewidth=1, label="si
ne")
plt.plot(data.sepalLength, np.cos(data.sepalLength), label='cosine', color="blue")
plt.legend(loc="lowerleft", prop={'size': 16})

#Red was chosen for sin and blue for cosine to create an easy and simple distinction, wit
h a legend provided label the two lines.
#This dataset provided a nice line so it was chosen. The legend was made larger to be mor
e readable.
```

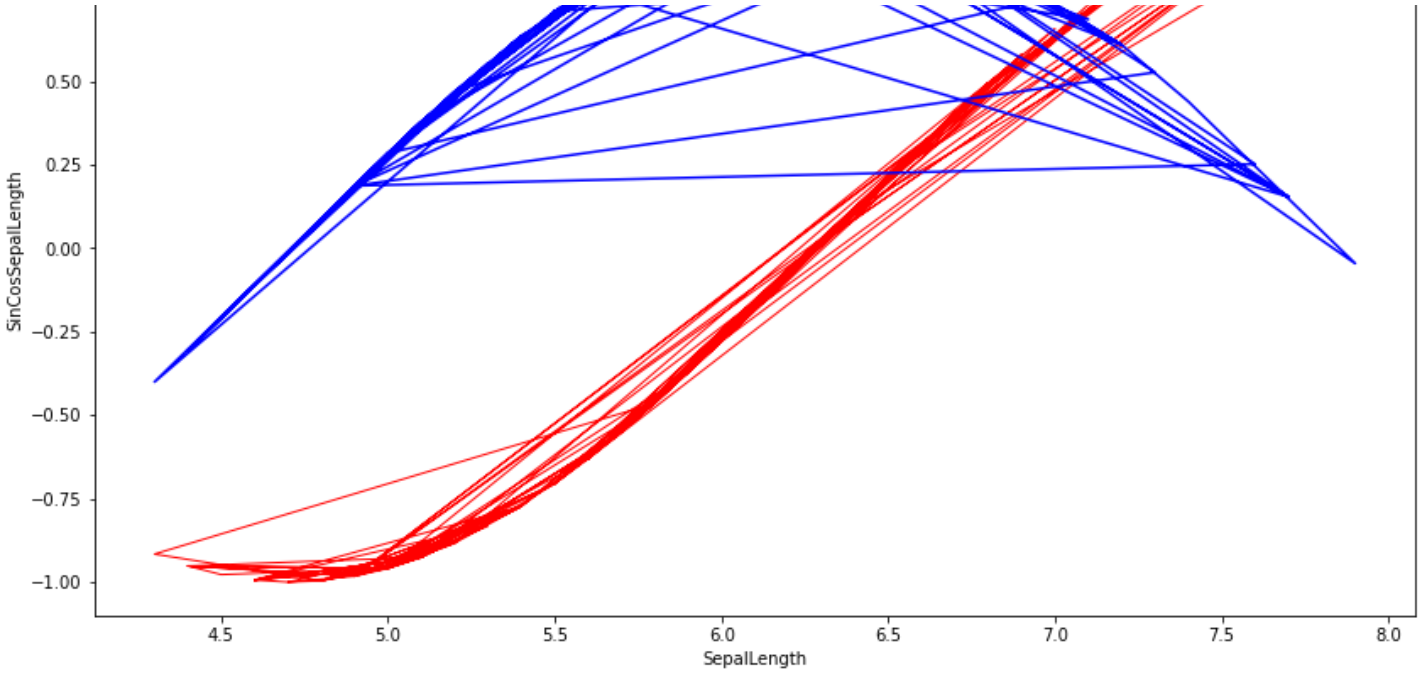
```
     sepalLength  sepalWidth  petalLength  petalWidth    species
0            5.1         3.5          1.4         0.2     setosa
1            4.9         3.0          1.4         0.2     setosa
2            4.7         3.2          1.3         0.2     setosa
3            4.6         3.1          1.5         0.2     setosa
4            5.0         3.6          1.4         0.2     setosa
..           ...         ...          ...         ...        ...
145          6.7         3.0          5.2         2.3  virginica
146          6.3         2.5          5.0         1.9  virginica
147          6.5         3.0          5.2         2.0  virginica
148          6.2         3.4          5.4         2.3  virginica
149          5.9         3.0          5.1         1.8  virginica

[150 rows x 5 columns]
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:13: MatplotlibDeprecationWar
ning: Unrecognized location 'lowerleft'. Falling back on 'best'; valid locations are
 best
 upper right
 upper left
 lower left
 lower right
 right
 center left
 center right
 lower center
 upper center
 center
This will raise an exception in 3.3.
  del sys.path[0]
```

Out[108]:

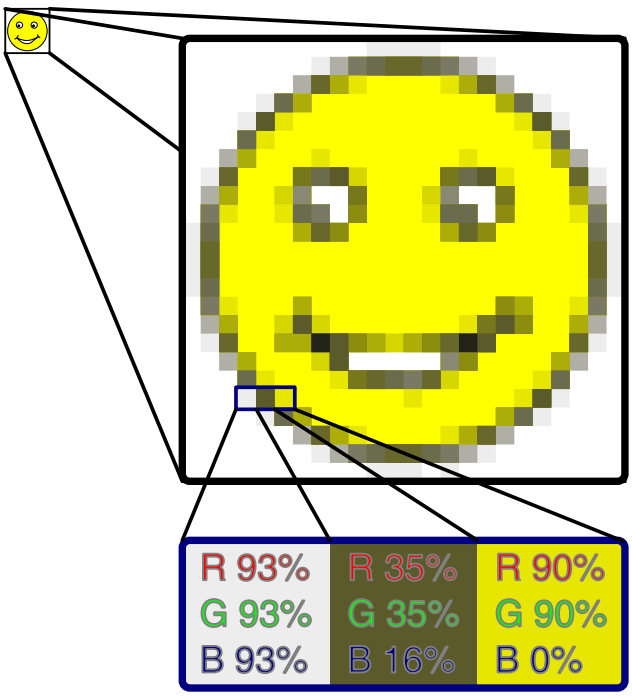
```
<matplotlib.legend.Legend at 0x7fc3a10b08d0>
```






## 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.**

In [ ]:

```
```

**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, vector**

graphics won't lose quality when zooming in.

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. 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.

In [ ]:

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, line, polygon (this can be used to create triangles), and path (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.

drawing this? Color it in the way you like. Upload this file to canvas.

In [108]:

```
<!DOCTYPE html>
<html>
<body>

<svg height="250" width="500">
  <polygon points="200,20 300,220 100,220" style="fill:white;stroke:black;stroke-width:5"
/>
   <circle cx="200" cy="150" r="50" fill="white" stroke="black" stroke-width="5"/>
   <line x1="200" x2="200" y1="220" y2="20" fill="yellow" stroke="black" stroke-width="5
"/>
</svg>

</body>
</html>
```