[1]:
```
%matplotlib inline
import matplotlib

# make figures bigger:
import numpy as np
import matplotlib.pyplot as plt
import random
import scipy, scipy.stats

nicered = "#E6072A"
niceblu = "#424FA4"
nicegrn = "#6DC048"

# make figures better:
font = {'weight':'normal','size':20}
matplotlib.rc('font', **font)
matplotlib.rc('figure', figsize=(7.0, 5.0))
#matplotlib.rc('xtick.major', pad=10) # xticks too close to border!
matplotlib.rc('xtick', labelsize=16)
matplotlib.rc('ytick', labelsize=16)
matplotlib.rc('legend',**{'fontsize':16})

import warnings
warnings.filterwarnings('ignore')
```

DS1 Lecture 21

James Bagrow, james.bagrow@uvm.edu, http://bagrow.com

Last time:

1. Basic principles of graphics and visualizations

Today:

1. Maps
2. Curves
3. Colors

Maps

Many data science problems and reports can benefit from drawing maps.

- May find yourself drawing maps!

Let's give a brief overview of how to do some mapping in Python.

Drawing maps is surprisingly hard!

You need to take your map data (collections of latitude,longitude pairs) and project them from the sphere (actually the earth's shape is called a "geoid") onto the 2D plane.

There is a companion package to matplotlib called Cartopy

- It handles the map projection business
- It contains a bunch of data for the shapes of countries, coastlines, states, etc.

Here's an example:

[2]:
```
import cartopy.crs as ccrs # crs - coordinate reference system

ax = plt.axes(projection=ccrs.Orthographic(central_longitude=-100,central_latitude=40))

ax.coastlines()
ax.gridlines();
```

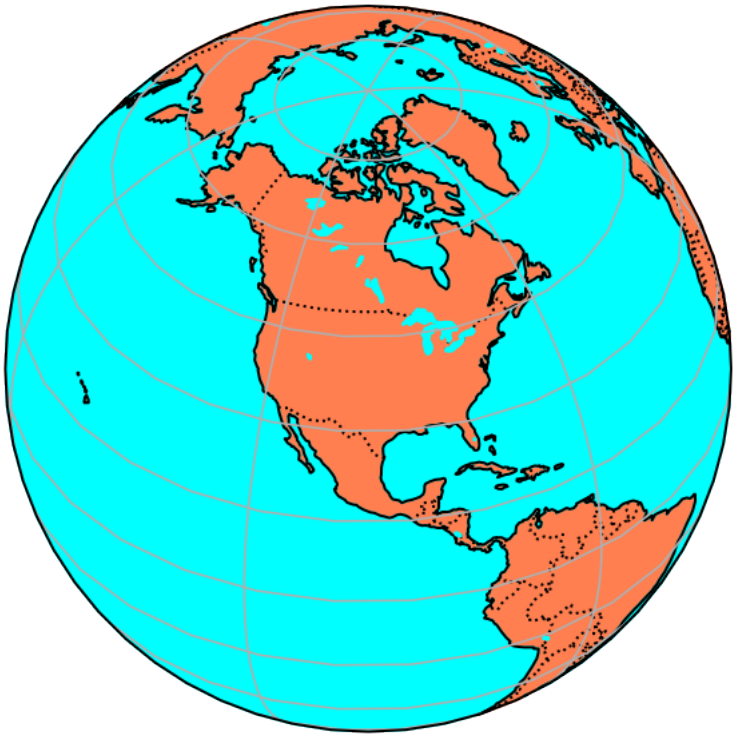
Cartopy works by extending the matplotlib `Axes` class into a `GeoAxes` class that supports additional, map-specific functionality:

[3]:
```
print(type(ax))
```

```
<class 'cartopy.mpl.geoaxes.GeoAxesSubplot'>
```

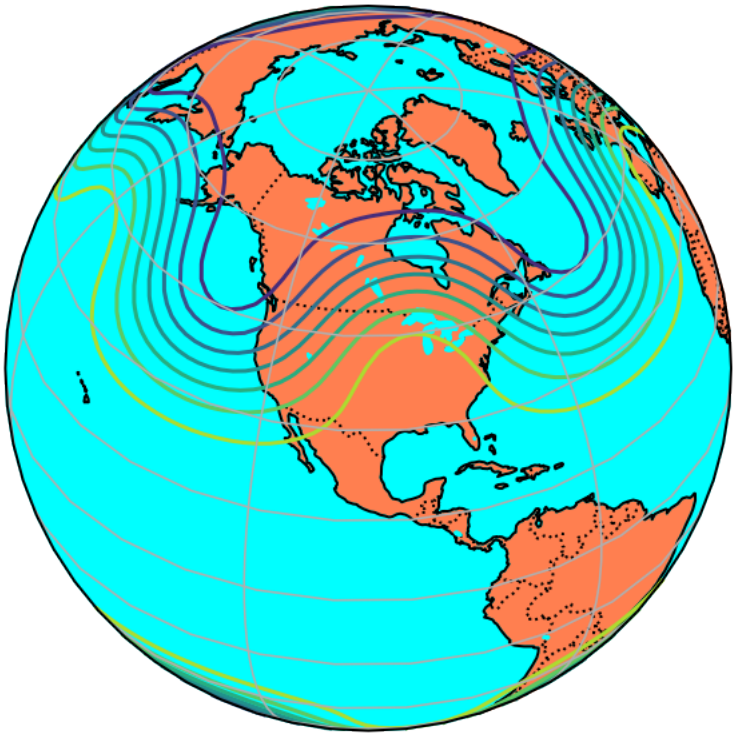
[4]:
```
import cartopy.feature as cfeature

plt.figure(figsize=(9,5))
ax = plt.axes(projection=ccrs.Orthographic(central_longitude=-100,central_latitude=40))

ax.coastlines()
ax.gridlines()

ax.add_feature(cfeature.BORDERS, linestyle=':')
```

```
ax.add_feature(cfeature.LAND,  color='coral')
ax.add_feature(cfeature.OCEAN, color='aqua')
ax.add_feature(cfeature.LAKES, color='aqua');
```



[5]:
```
plt.figure(figsize=(9,5))
ax = plt.axes(projection=ccrs.Orthographic(central_longitude=-100,central_latitude=40))

ax.set_global()
ax.coastlines()
ax.gridlines()

ax.add_feature(cfeature.BORDERS, linestyle=':')
ax.add_feature(cfeature.LAND,  color='coral')
ax.add_feature(cfeature.OCEAN, color='aqua')
ax.add_feature(cfeature.LAKES, color='aqua');


# some "data":
nlats = 73; nlons = 145; delta = 2.*np.pi/(nlons-1)
lats = (0.5*np.pi-delta*np.indices((nlats,nlons))[0,:,:])
lons = (delta*np.indices((nlats,nlons))[1,:,:])
wave = 0.75*(np.sin(2.*lats)**8*np.cos(4.*lons))
mean = 0.5*np.cos(2.*lats)*((np.sin(2.*lats))**2 + 2.)
lons = np.rad2deg(lons)
lats = np.rad2deg(lats)
```

```
# add as contour lines:
plt.contour(lons, lats, wave+mean,
            transform=ccrs.PlateCarree());
                        # ^^ coord. sys for use with lat/lon
# https://scitools.org.uk/cartopy/docs/latest/tutorials/understanding_transform.html
```



Map projections.

The core of cartopy is the **coordinate reference system** class:

```
import cartopy.crs as ccrs # crs - coordinate reference system

ax = plt.axes(projection=ccrs.Orthographic(central_longitude=-100,central_latitude=40))
```

Here the `ccrs.Orthographic` lets us translate from, say, the latitudes and longitudes used when mapping to cartesian (xy) coordinates expected by matplotlib for plotting.

This translation means putting a 3D sphere onto a 2D plane. This is called **projection** and there are many ways to do it.

All map projections must introduce some degree of **distortion**, especially when they try to show the entire globe. For example, the "mercator" projection used to be very common:

Mercator introduces **crazy** distortion at the north and south poles. For example, Greenland looks to be the same size as Africa. In fact, Africa is **14 times bigger**.

- Here's an interactive visualization showing the distortion: The True Size of...

A different projection, the Winkel Tripel, is better:

This is the current choice for the National Geographic Society.

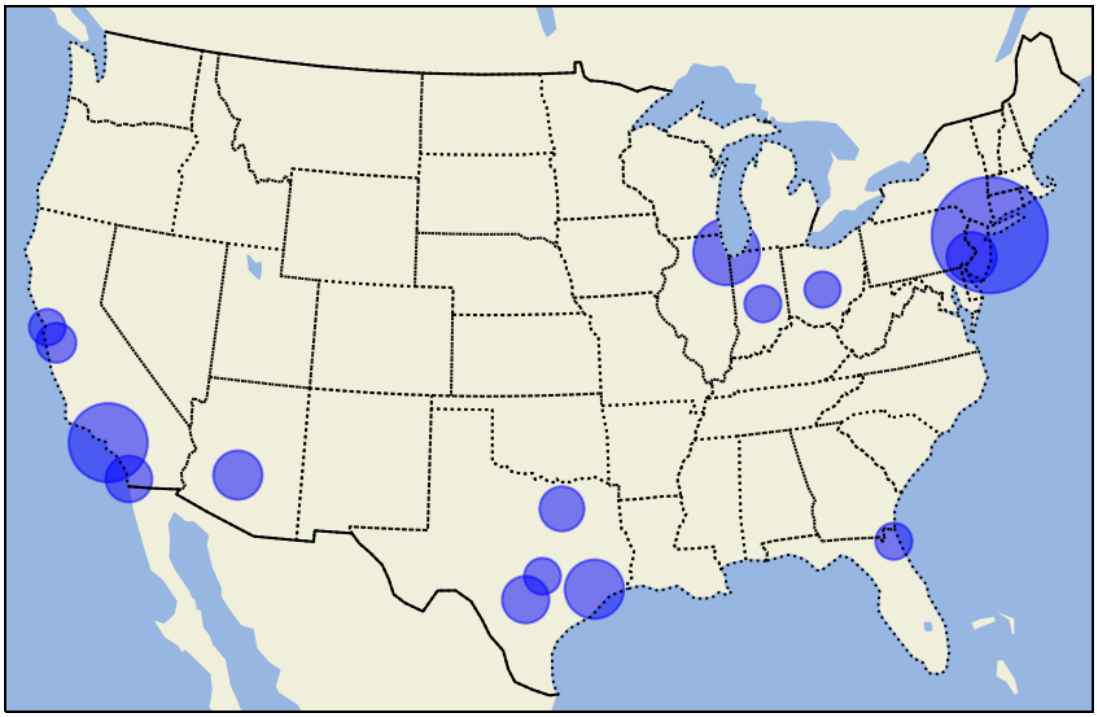
Cartopy incorporates the math for many projections. The previous example used the "orthographic" projection. This is very accurate because it only shows the visible portion of the globe.

Here's another mapping example showing **US cities and their populations**.

- This time we're going to draw only the US using yet another type of project called the Lambert conformal conic projection (LCC). This projection looks strange when showing the entire earth, but is relatively accurate over a small range, such as the continental US.
- The call to `ccrs` requires different input variables depending on the type of projection being used. For example, the LCC projection requires (lat,lon) pairs defining the **lower left** (ll) and **upper right** of the "figure".

[6]:
```
# Data of city location (logitude,latitude) and population

city2pop={'New York':8244910,'Los Angeles':3819702,
'Chicago':2707120,'Houston':2145146,
'Philadelphia':1536471,'Pheonix':1469471,
'San Antonio':1359758,'San Diego':1326179,
'Dallas':1223229,'San Jose':967487,
'Jacksonville':827908,'Indianapolis':827908,
'Austin':820611,'San Francisco':812826,
'Columbus':797434} # dictionary of the populations of each city

city2lat={'New York':40.6643,'Los Angeles':34.0194,
'Chicago':41.8376,'Houston':29.7805,
'Philadelphia':40.0094,'Pheonix':33.5722,
'San Antonio':29.4724,'San Diego':32.8153,
'Dallas':32.7942,'San Jose':37.2969,
'Jacksonville':30.3370,'Indianapolis':39.7767,
'Austin':30.3072,'San Francisco':37.7750,
'Columbus':39.9848} # dictionary of the latitudes of each city

city2lon={'New York':-73.9385,'Los Angeles':-118.4108,
'Chicago':-87.6818,'Houston':-95.3863,
'Philadelphia':-75.1333,'Pheonix':-112.0880,
'San Antonio':-98.5251,'San Diego':-117.1350,
'Dallas':-96.7655,'San Jose':-121.8193,
'Jacksonville':-81.6613,'Indianapolis':-86.1459,
'Austin':-97.7560,'San Francisco':-122.4183,
'Columbus':-82.9850} # dictionary of the longitudes of each city


plt.figure(figsize=(9,5))
ax = plt.axes(projection=ccrs.
 ↪AlbersEqualArea(central_longitude=-100,central_latitude=40))

#ax.set_global()
ax.set_extent( (-121, -74, 25, 49.5), crs=ccrs.PlateCarree() )


ax.add_feature(cfeature.BORDERS, linestyle='-')
ax.add_feature(cfeature.LAND, )# color='coral')
ax.add_feature(cfeature.OCEAN, )#color='aqua')
```

```
ax.add_feature(cfeature.LAKES, )#color='aqua');
ax.add_feature(cfeature.STATES, linestyle=':');


max_size=2000
for city in city2lon:
    lo, la = city2lon[city],city2lat[city]
    marker_size = max_size*city2pop[city]/city2pop['New York']
    plt.scatter(lo,la, marker_size,
                marker='o',color='b', alpha=0.5,
                transform=ccrs.PlateCarree())
```



The shortest path between two points on a globe is not a line but a segment of a circle known as a great circle.

- Cartopy, with the right coordinate transform, let's us draw these arcs:

[7]:
```
plt.figure(figsize=(9,5))
ax = plt.axes(projection=ccrs.Robinson() )

ax.add_feature(cfeature.BORDERS, linestyle='-', lw=0.5)
ax.add_feature(cfeature.LAND, )# color='coral')
ax.add_feature(cfeature.OCEAN, )#color='aqua')
ax.add_feature(cfeature.LAKES, )#color='aqua');

# New York
nylat = 40.78;
nylon = -73.98
```

```
# London
londlat = 51.53;
londlon = 0.08

ax.set_global()

plt.plot([nylon, londlon], [nylat, londlat],
         color='blue', linewidth=2, marker='o',
         transform=ccrs.Geodetic(),
         )

plt.plot([nylon, londlon], [nylat, londlat],
         color='red', linestyle='-', zorder=0,
         transform=ccrs.PlateCarree(),
         );
```

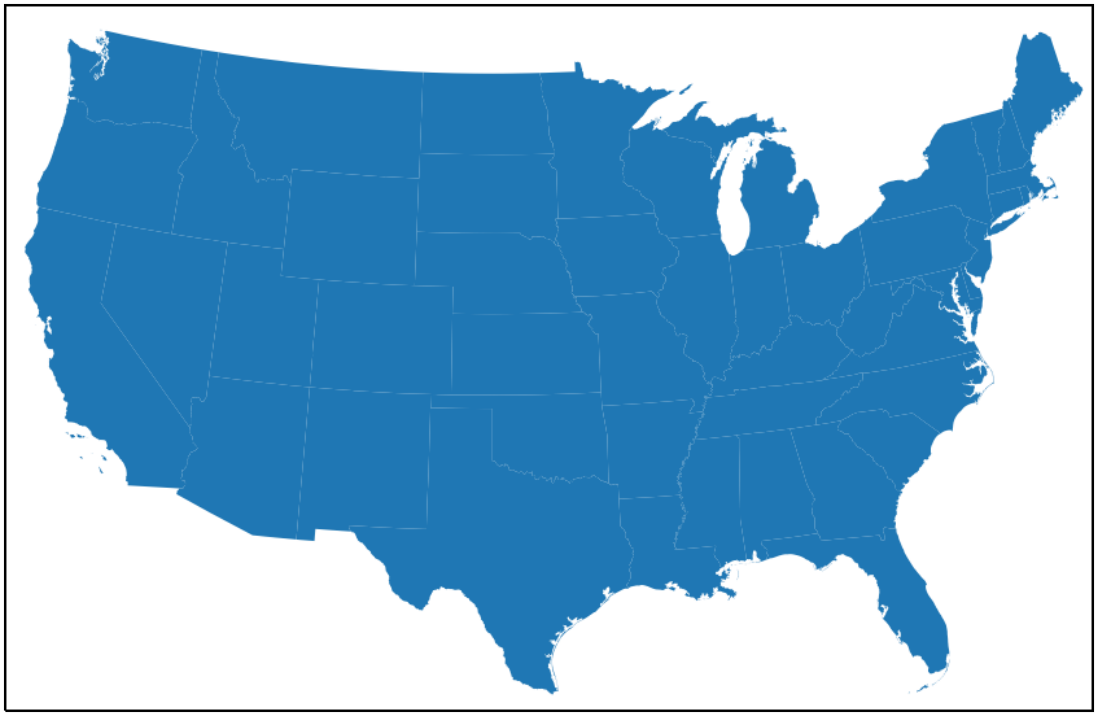
Notice how we need to use `Geodetic` to capture the spherical nature...

These arcs can be used to visualize travel or communication data, for example.

External map data – Shapefiles

[8]:
```
plt.figure(figsize=(9,5))
ax = plt.axes(projection=ccrs.
 ↪AlbersEqualArea(central_longitude=-100,central_latitude=40))

#ax.set_global()
ax.set_extent( (-121, -74, 25, 49.5), crs=ccrs.PlateCarree() )
               # ^^^ Will only show lower 48

from cartopy.io.shapereader import Reader
```

```
from cartopy.feature import ShapelyFeature

fname = 'gz_2010_us_040_00_500k.shp' #,'states',drawbounds=False)

shape_feature = ShapelyFeature(Reader(fname).geometries(),
                               ccrs.PlateCarree())
ax.add_feature(shape_feature)
plt.show()
```



This example reads what is called a **shapefile**, a standard way to store map data (which is surprisingly nontrivial data).

Shapefiles often come in triples. For this example:

```
gz_2010_us_040_00_500k.dbf
gz_2010_us_040_00_500k.shp
gz_2010_us_040_00_500k.shx
```

You almost always want to find shapefiles online. I found these shape files from the US census: http://www2.census.gov/geo/tiger/GENZ2010/

Example: Choropleth Let's use these shapes to draw a map of the USA where each state's color represents its **population density**.

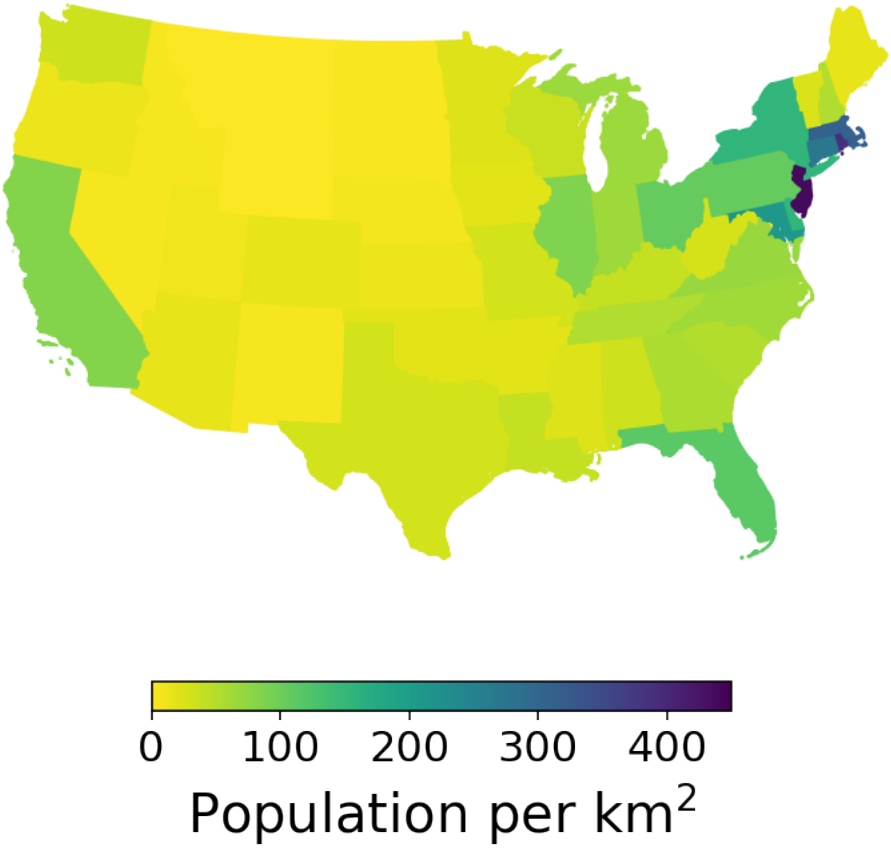
First, the data:

[9]:
```
# population density by state from
# http://en.wikipedia.org/wiki/List_of_U.S._states_by_population_density
popdensity = {
'New Jersey': 438.00,
'Rhode Island': 387.35,
'Massachusetts': 312.68,
'Connecticut': 271.40,
'Maryland': 209.23,
'New York': 155.18,
'Delaware': 154.87,
'Florida': 114.43,
'Ohio': 107.05,
'Pennsylvania': 105.80,
'Illinois': 86.27,
'California': 83.85,
'Hawaii': 72.83,
'Virginia': 69.03,
'Michigan': 67.55,
'Indiana': 65.46,
'North Carolina': 63.80,
'Georgia': 54.59,
'Tennessee': 53.29,
'New Hampshire': 53.20,
'South Carolina': 51.45,
'Louisiana': 39.61,
'Kentucky': 39.28,
'Wisconsin': 38.13,
'Washington': 34.20,
'Alabama': 33.84,
'Missouri': 31.36,
'Texas': 30.75,
'West Virginia': 29.00,
'Vermont': 25.41,
'Minnesota': 23.86,
'Mississippi': 23.42,
'Iowa': 20.22,
'Arkansas': 19.82,
'Oklahoma': 19.40,
'Arizona': 17.43,
'Colorado': 16.01,
'Maine': 15.95,
'Oregon': 13.76,
'Kansas': 12.69,
'Utah': 10.50,
'Nebraska': 8.60,
'Nevada': 7.03,
'Idaho': 6.04,
'New Mexico': 5.79,
'South Dakota': 3.84,
'North Dakota': 3.59,
'Montana': 2.39,
'Wyoming': 1.96,
'Alaska': 0.42}
```

(In practice, you would load these data from a file.)

Now the map:

[10]:
```
import matplotlib.colors

plt.figure(figsize=(9,5))
ax = plt.axes(projection=ccrs.
 ↪AlbersEqualArea(central_longitude=-100,central_latitude=40))
#ax.set_global()
ax.set_extent( (-121, -74, 25, 49.5), crs=ccrs.PlateCarree() )
               # ^^^ Will only show lower 48

# set up colormap for the choropleth
cmap = plt.cm.viridis_r # Blues_r hot_r
vmin = 0; vmax = 450 # set range of pop densities.


# loop over states in the shapefile:
for rec in Reader(fname).records():
    attrs = rec.attributes
    geo = rec.geometry

    statename = attrs['NAME']
    try:
        pop = popdensity[statename]
    except KeyError:
        pop = 0.0

    # calling colormap with value between 0 and 1 returns
    # rgba(lpha) value (keep only rgb):
    this_color = cmap((pop-vmin)/(vmax-vmin))[:3]

    # append to map:
    shape_feature = ShapelyFeature(geo, ccrs.PlateCarree())
    ax.add_feature(shape_feature, color=this_color)
    # adding many shapes separately may be slow


# add colorbar:
mm = plt.cm.ScalarMappable(cmap=cmap)
mm.set_array([vmin,vmax])
plt.colorbar(mm, label="Population per km$^2$",
             ticks=[0,100,200,300,400],
             orientation="horizontal", fraction=0.04,
            )

# remove box around map:
ax.outline_patch.set_visible(False)

plt.show()
```



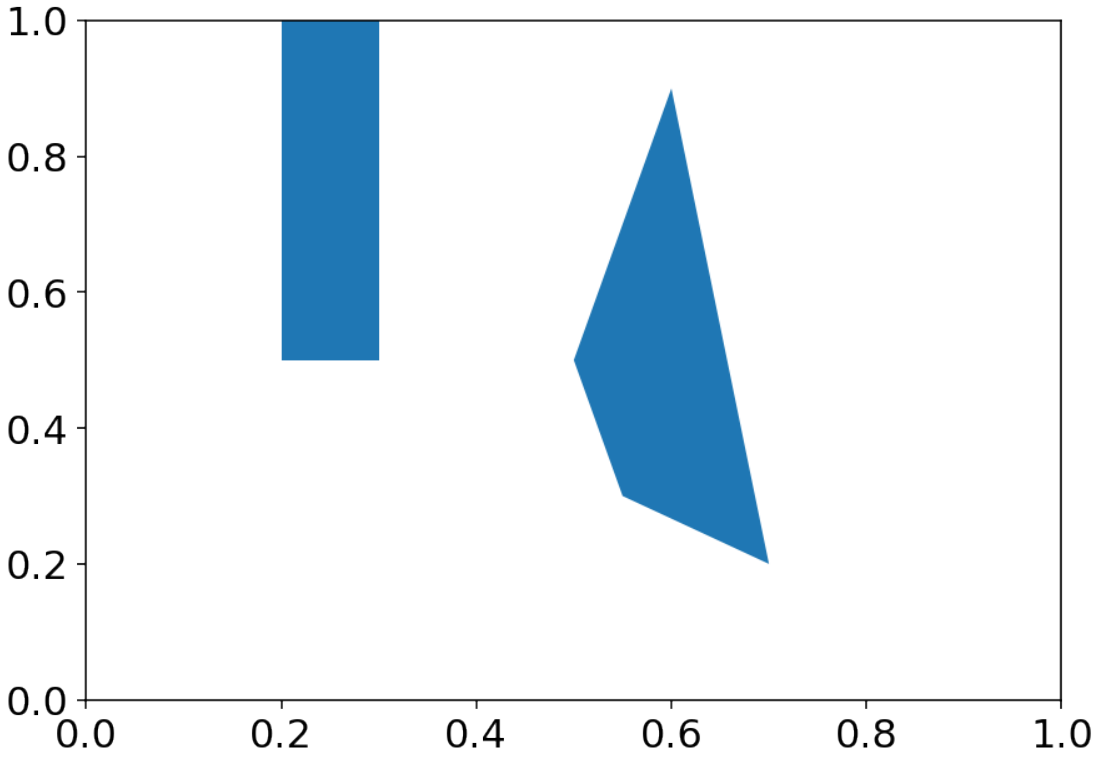
Drawing with the computer

In a sense, drawing is similar to plotting: If you want to draw a shape you need to manage the (x,y) coordinates of all the points that define that shape.

- Lines, rectangles, polygons, circles, curves are define with xy coordinates on the **image**.
- You are in charge of defining how these image coordinates relative to whatever drawing or plotting library you are using.
- Matplotlib adds things like x- and y-axes automatically.

[11]:
```
fig = plt.figure()
ax  = fig.gca()

# create a rectangle instance
rect = matplotlib.patches.Rectangle( (0.2,0.5), width=0.1, height=1.2)

# now we add the Rectangle to the Axes
ax.add_patch(rect)


# now let's add a crazy polygon:
```

```
vertices = [
 (0.5,0.5),
 (0.6,0.9),
 (0.7,0.2),
 (0.55,0.3)
]
poly = matplotlib.patches.Polygon( vertices )
ax.add_patch(poly)


plt.show()
```



Curves

Given a small number of **low-level** drawing commands (draw point, draw line segment, draw filled region) you can **build up** to pretty much any more advanced drawing level. For example, to draw a circle of radius r centered at x_0, y_0 you only need to draw a number of line segments:

```
x_prev = x0 + r*cos(0) # first point of circle
y_prev = y0 + r*sin(0)

for th in [th1,th2, ..., 2*pi]:
    x = x0 + r*cos(th)
    y = y0 + r*sin(th)
```

```
    LINE(x_prev, y_prev, x, y)

    x_prev = x
    y_prev = y
```

However, almost all drawing packages/toolboxes/frameworks/etc. also provide high-level **convenience** functions for typical tasks. Drawing a circle is very common so in practice you almost never have to write something like the above loop. Instead:

```
CIRCLE(x0, y0, r) # much shorter!
```

The specific function equivalents `LINE` and `CIRCLE` will of course depend on your particular drawing library.

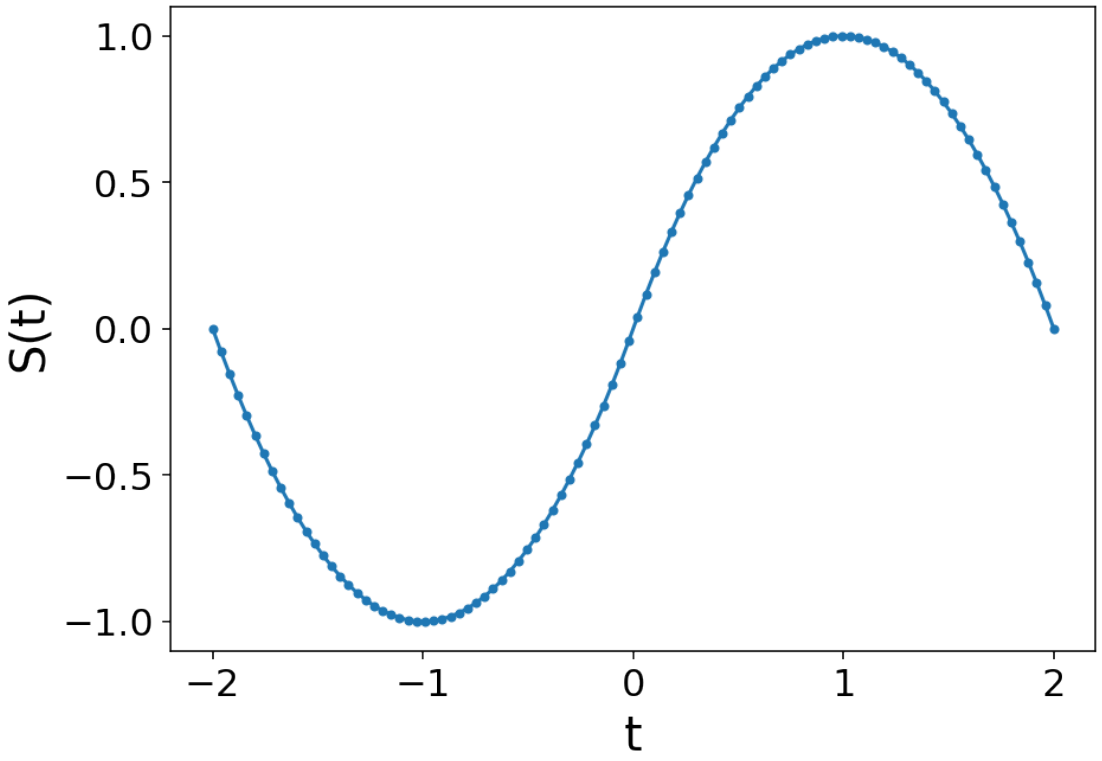
You can always visualize any **curve** by drawing a number of short line segments, with increasing accuracy as the number of segments increases. However, most drawing systems have a **different way** of encoding curves (or "splines").

Consider this function:

$$S(t) = \begin{cases} (t+1)^2 - 1 & -2 \leq t < 0 \\ 1 - (t-1)^2 & 0 \leq t \leq 2 \end{cases}$$

Let's plot $S(t)$:

[12]:
```
def S(t):
    if  -2 <= t < 0:
        return (t+1)**2 - 1
    elif 0 <= t <= 2:
        return 1 - (t-1)**2
    else:
        return float("nan")

x = []
y = []
for t in np.linspace(-2,2,100):
    x.append(t)
    y.append( S(t) )

plt.plot(x,y, '.-')
plt.xlabel("t")
plt.ylabel("S(t)")
plt.show()
```



The function `S(t)` encodes a nice, smooth curve to "**infinite**" precision. When we go to draw it we can choose the smoothness of the curve by the number of values of `t` we pick.

- This gives us a compact **functional form** for a curve.

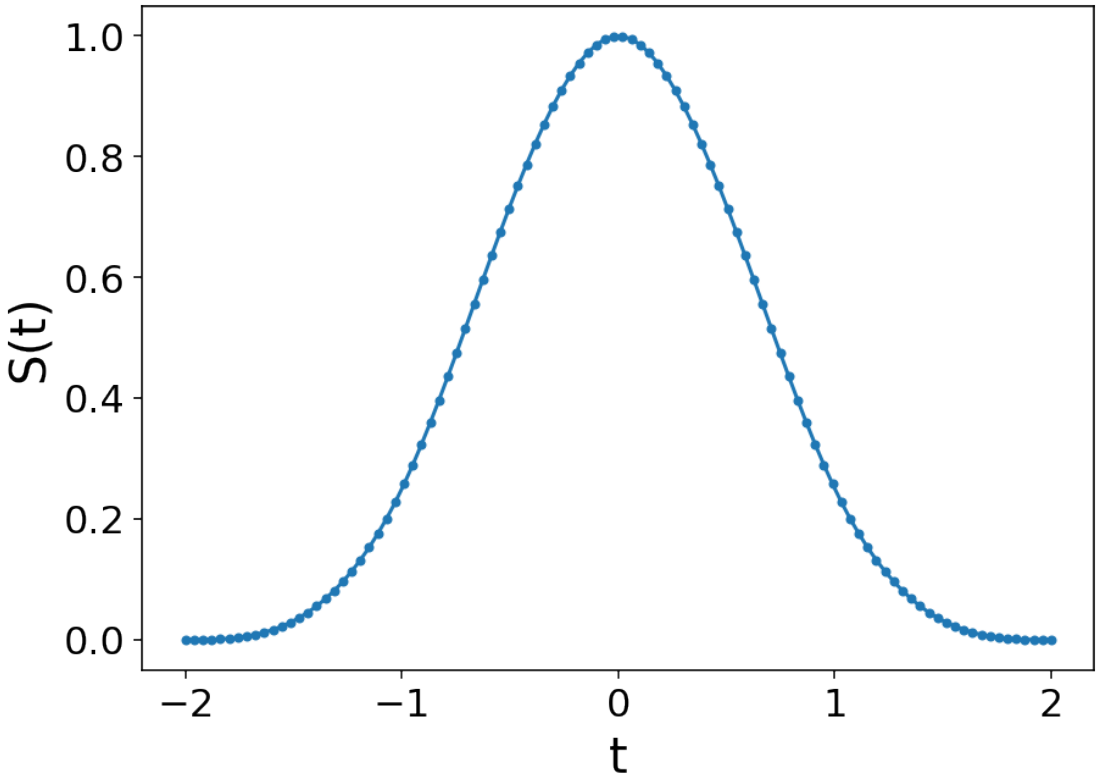
A more extreme example:

$$S(t) = \begin{cases} \frac{1}{4}(t+2)^3 & -2 \leq t \leq -1 \\ \frac{1}{4}\left(3\,|t|^3 - 6t^2 + 4\right) & -1 \leq t \leq 1 \\ \frac{1}{4}(2-t)^3 & 1 \leq t \leq 2 \end{cases}$$

And to draw it:

[13]:
```
def S(t):
    if  -2 <= t < -1:
        return 0.25 * (t+2)**3
    elif -1 <= t < 1:
        return 0.25 * (3*abs(t)**3 - 6*t**2 + 4)
    elif 1 <= t <= 2:
        return 0.25 * (2 - t)**3
    else:
        return float("nan")

x = []
y = []
for t in np.linspace(-2,2,100):
```

```
    x.append(t)
    y.append( S(t) )

plt.plot(x,y, '.-')
plt.xlabel("t")
plt.ylabel("S(t)")
plt.show()
```



"OK," you may be thinking, "big deal: those are just (almost) a sin function and a gaussian. Why do I need to build these crazy piece-wise functions?"

These functions are **polynomials**. This makes them very easy to work with and the computer can work with them very efficiently.

- Done any **curve-fitting**? It is easy to fit polynomials (not just lines) to data. Much easier than sin or exp, etc.

These piecewise, smooth polynomials are often called **splines**. They are so easy to work with and well understood that computer scientists decided to use them to **encode curves in early graphical software**. It's been the standard ever since.

- "Spline" was originally a term for a flexible ruler used to draw or cut curves when building ships.

Of course, looking at the previous $S(t)$'s, they don't look so easy. Where do the equations come from?

Bezier curves

A bezier curve is a way of writing a polynomial as a **parametric curve** that is particularly convenient for drawing.

The simplest bezier curve is a straight line. Imagine a straight line connecting two points $\vec{P}_0 = (x_0, y_0)$ and $\vec{P}_1 = (x_1, y_1)$. We can write down this line as

$$B(t) = \vec{P}_0 + t\left(\vec{P}_1 - \vec{P}_0\right)$$

where t is in $[0, 1]$. This t parameterizes the curve, telling us how far along the line we are.

Cool animation:

OK, so what??

* We can combine such linear interpolations to draw much more complex curves:

First, take the linear interpolation between P_0 and P_1 and the interpolation between P_1 and P_2. As t increases a point (Q_0, Q_1) will move along each line. Draw a line (shown in green) between those two points. A point moving along this moving line segment traces out a **quadratic bezier curve**.

- We can control the P_i's (aka control points) to tune the shape of the final curve.

To make more involved shapes keep repeating this process:

By the way, if you've ever used a vector drawing program like adobe illustrator, you may have seen these before:

The pen tool lays down a bezier curve (or spline) and the little "handles" that you drag around are the control points that guide the shape of the curve.

- While you click around the computer is drawing the polynomial(s) for you.

Here's an interactive version of the previous animations:

- Animated bezier curves

In principle that seems OK, but in practice isn't it going to be a lot of math to compute the curves?

- Yes, but fortunately you've almost always got built in functions for drawing the bezier curve given a collection of $P = (x, y)$ points!

In matplotlib, for example, you can draw a bezier curve using a `Path`:

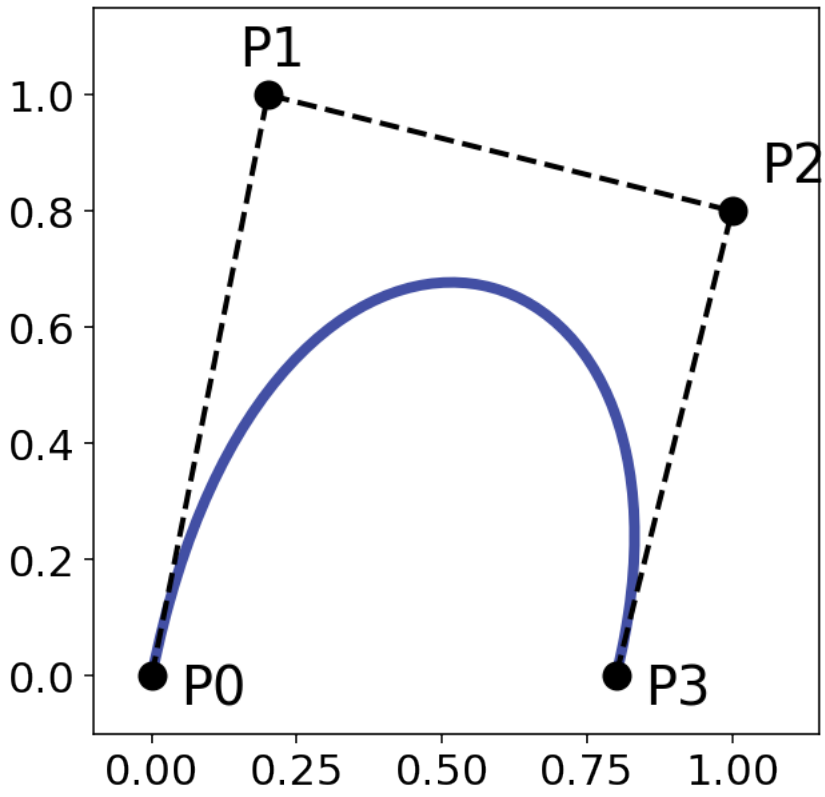
[14]:
```
from matplotlib.path import Path


# define the geometry of a path:
verts = [(0.0, 0.0), # P0
         (0.2, 1.0), # P1
         (1.0, 0.8), # P2
         (0.8, 0.0), # P3
        ]

codes = [Path.MOVETO, # put the "pen" at P0
         Path.CURVE4, # and draw bezier curves
         Path.CURVE4, # from P0 to P1, to P2
```

```
         Path.CURVE4, # to P3.
        ]

path = Path(verts, codes) # build path from vertices & command codes
```

This defined a `path` object that we encodes the geometry of a curve. Matplotlib (and many other packages) let you define "paths" geometrically and then use a separate object for drawing/visualization, typically called a "patch".

- Let's plot that `path` using a patch:

[15]:
```
import matplotlib.patches as patches


# build stylizeable "patch" to visualize geometric "path":
patch = patches.PathPatch(path, facecolor='none',
                                edgecolor=niceblu, lw=4)


ax = plt.gca() # gca = get current plot's "axes"

ax.add_patch(patch)

# draw control points:
xs, ys = zip(*verts)
ax.plot(xs, ys, 'o--', lw=2, color='black', ms=10)


# label the control points:
ax.text( 0.05, -0.05, 'P0')
ax.text( 0.15,  1.05, 'P1')
ax.text( 1.05,  0.85, 'P2')
ax.text( 0.85, -0.05, 'P3')

# resize the plot:
ax.set_xlim(-0.1, 1.15)
ax.set_ylim(-0.1, 1.15)
ax.set_aspect("equal")
plt.show()
```



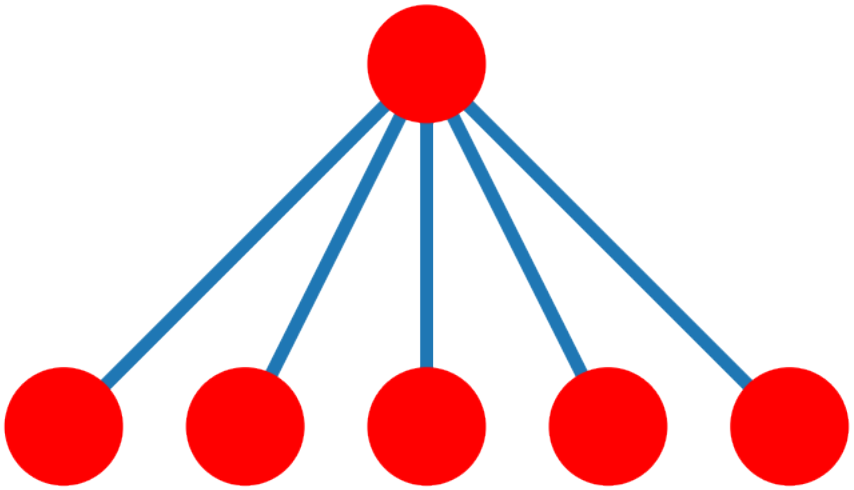
We're programming the curve!

Example Application

Here's a fun little example. Suppose we want to draw a flow chart:

[16]:
```
from matplotlib.lines   import Line2D
from matplotlib.patches import Circle
ax = plt.gca()

r = 0.08

# draw top circle
circ = Circle((0.5,0.75),r, color="red", zorder=2)
ax.add_patch(circ)

for i in range(5):
    # line from top circle to bottom
    line = Line2D( [0.5, i/4.0], [0.75,0.25], linewidth=5, zorder=1 )
    ax.add_line(line)

    # draw bottom circle
    circ = Circle((i/4.0,0.25),r, color="red", zorder=2)
    ax.add_patch(circ)
```

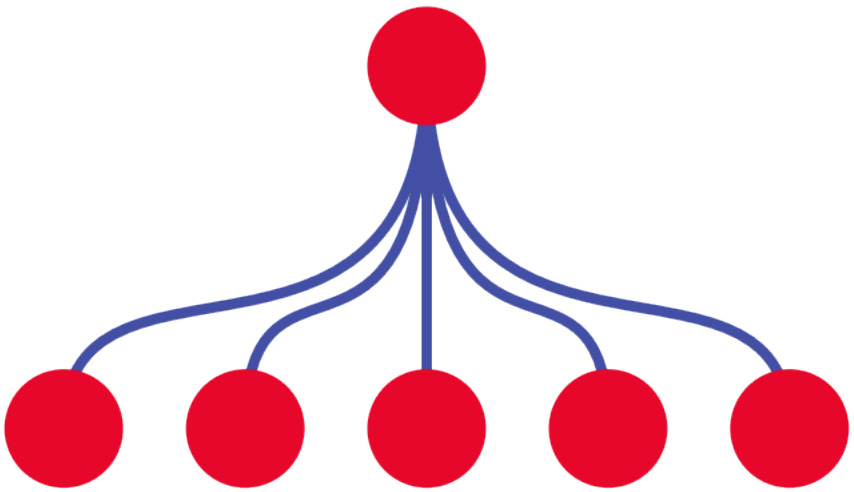
```
ax.set_aspect('equal', "datalim")
ax.axis('off')
plt.show()
```



Instead of straight line connectors we can program in some curves!

[17]:
```
ax = plt.gca()

r = 0.08

x_top = 0.5
y_top = 0.75
y_bot = 0.25

# draw top circle
circ = Circle((x_top,y_top),r, color=nicered, zorder=2)
ax.add_patch(circ)

for i in range(5):
    x_bot = i/4.0

    # set up the path:
    verts = [(x_top, y_top),                    # P0
             (x_top, y_bot ),                   # P1
             (x_bot, y_bot+0.6*(y_top-y_bot) ), # P2 ***
```

```
             (x_bot, y_bot),                    # P3
            ]

    codes = [Path.MOVETO,
             Path.CURVE4,
             Path.CURVE4,
             Path.CURVE4,
            ]

    path = Path(verts, codes)
    patch = patches.PathPatch(path, facecolor='none', zorder=1,
                                    edgecolor=niceblu, lw=4)
    ax.add_patch(patch)


    # draw bottom circle
    circ = Circle((x_bot,y_bot),r, color=nicered, zorder=2)
    ax.add_patch(circ)


ax.set_aspect('equal', "datalim")
ax.axis('off')
plt.show()
```



Colors

You may have seen commands like this:

```
plt.plot(X,Y, '-', color="red")
```

and thought, "Oh ok, a red line.". But what about this:

```
plt.plot(X,Y, 'o-', color='#FF00AA', markerfacecolor="#00FF00")
```

Those strings, which you've likely seen before if you do any web design, are **hexadecimal numbers** representing **RGB** (red green blue) colors. This is known as a hex triplet. The leading "#" is a standard convention for denoting a triple.

A hex number is base-16. It ranges **from 0 to 9** and then **from A to F**. Base-16 is convenient on the computer when working with bytes and it lets you represent a number between 0 and 256 with two digits, where as base-10 could only represent numbers between 0 and 99 with two digits.

The six-digits in a hex triple let us define the color channels for red, green, and blue:

`#RRGGBB`

So the color pure red, sometimes denoted `RGB(1.0,0,0)` is `#FF0000`. The first `FF` the largest value possible, while the other two channels are `00` since there is no green or blue in the color.

- Since hex triples are base-16, people often write the color scale going from 0 to 255 instead of 0 to 1, so pure red would then be `RGB(255,0,0)` with the same hex representation.
- This can sometimes cause problems. If a function wants the color channels to be in [0,1] and you pass a channel value of 128 (which represents 50%), the function may round that down to 1, the largest value it assumes can exist.

To the wikipedias!

Color math

The way to represent color on a computer is non-trivial and there are lots of different color systems beside RGB.

RGB is convenient because media that transmit light (such as TVs) use red, green, and blue pixels. Let's see how a modern computer display actually works, **it's cool!**

So the computer display is just an array of red, green, and blue pixels in close proximity. Colored light gets **mixed**. This is called additive mixing:

This is different from what paint and pigment does, which is called subtractive mixing:

Notice how the circles are "cyan", "magenta", and "yellow" and their intersections are red, green, and blue? This is why high-end graphic design doesn't use RGB colors but instead uses CMYK: it more accurately models the ink in a printing press.

- We won't use CMYK.

Our brains automatically mix light additively:

There are only three colors in that image!

Mixing colors using math

Additive mixing is easy, it's just addition. Let's mix two colors. All we do is sum up the three color channels elementwise and then round down any numbers that are too big:

[18]:
```
c1 = (1.0, 0.0, 0.0) # pure red in RGB
c2 = (0.0, 0.0, 1.0) # pure blue

cS = ( c1[0]+c2[0], c1[1]+c2[1], c1[2]+c2[2] )

# round down to 1.0:
cS = ( min([cS[0],1]),
       min([cS[1],1]),
       min([cS[2],1]) )


print(cS) # should be 100% red and 100% blue
```

```
(1.0, 0.0, 1.0)
```

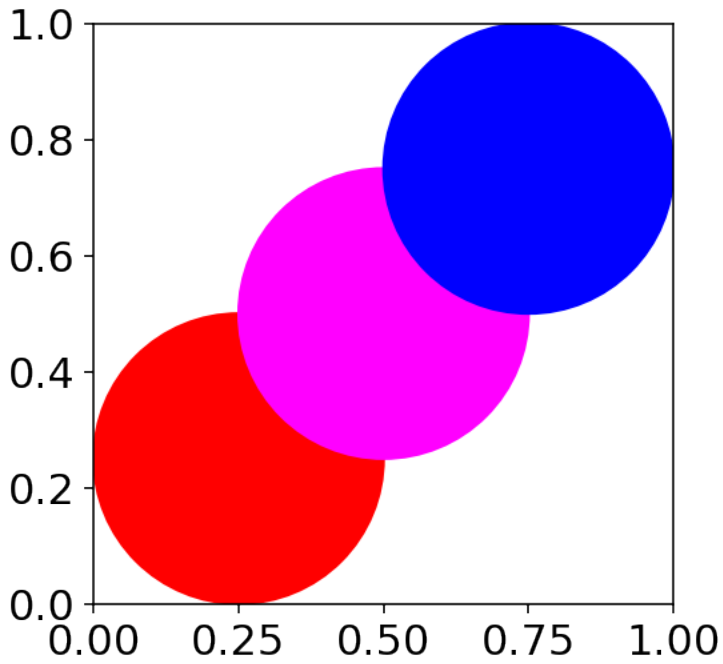
[19]:
```
def rgb_to_rgb256(rgb):
    """Map [0,1] rgb to [0,255] rgb."""
    r,g,b = rgb
    return ( int(255*r), int(255*g), int(255*b) )


def rgb256_to_hex(rgb):
    """Make hex triple from rgb"""
    return '#%02X%02X%02X' % rgb

print(cS)

hex_triple = rgb256_to_hex( rgb_to_rgb256(cS) )

print(hex_triple)
```

```
(1.0, 0.0, 1.0)
#FF00FF
```

Let's see what we've got:

[20]:
```
%matplotlib inline

h1 = rgb256_to_hex( rgb_to_rgb256(c1) )
h2 = rgb256_to_hex( rgb_to_rgb256(c2) )

#           Circle((x0,  y0  ),  r ,          )
circle1=plt.Circle((0.25,0.25),0.25, color=h1)
circle3=plt.Circle((0.75,0.75),0.25, color=h2)
circle2=plt.Circle((0.50,0.50),0.25, color=hex_triple)

plt.clf()
fig = plt.gcf()
fig.gca().add_artist(circle1)
fig.gca().add_artist(circle2)
```

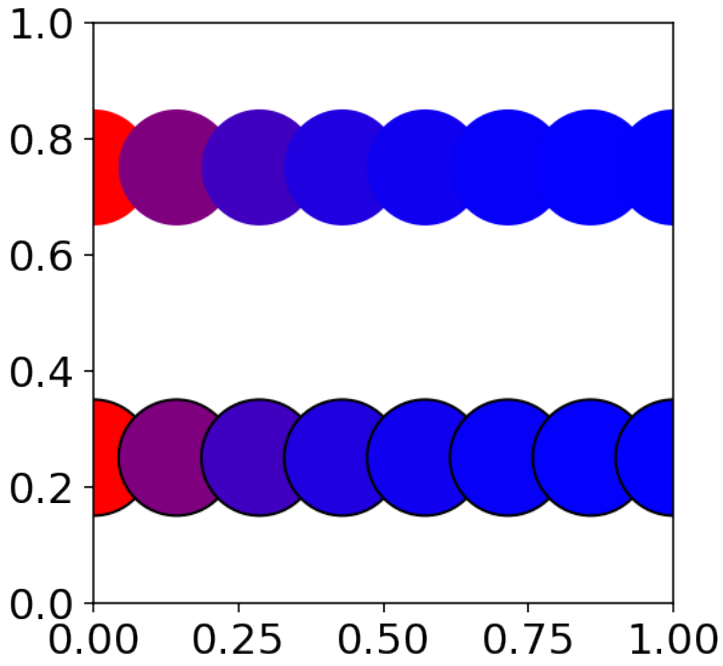
```
fig.gca().add_artist(circle3)
fig.gca().set_aspect('equal')
plt.show()
```



[21]:
```
from matplotlib.lines   import Line2D
from matplotlib.patches import Circle


color = (255,0,0) # pure red
h = rgb256_to_hex(color)
print(0, h, color)

ax = plt.gca()
r_circ = 0.1

circ = Circle((0/10.0,0.25),r_circ, color=h, ec='black')
ax.add_patch(circ)
circ = Circle((0/10.0,0.75),r_circ, color=h, ec='none')
ax.add_patch(circ)


for i in range(7): # draw seven circles

    r,g,b = color

    r = (r +   0)/2.0 # average with blue, rgb(0,0,255)
    g = (g +   0)/2.0
    b = (b + 255)/2.0
```

```
    color = (int(r),int(g),int(b))

    h = rgb256_to_hex(color)
    print(i+1, h, color)

    # draw two circles
    x_circ = (i+1)/7.0

    circ = Circle( (x_circ, 0.25), r_circ,
                   color=h, ec="black", lw=1.0)
    ax.add_patch(circ)

    circ = Circle( (x_circ, 0.75), r_circ,
                   color=h, ec="none",  lw=1.0)
    ax.add_patch(circ)

ax.set_aspect('equal')
plt.show()
```

```
0 #FF0000 (255, 0, 0)
1 #7F007F (127, 0, 127)
2 #3F00BF (63, 0, 191)
3 #1F00DF (31, 0, 223)
4 #0F00EF (15, 0, 239)
5 #0700F7 (7, 0, 247)
6 #0300FB (3, 0, 251)
7 #0100FD (1, 0, 253)
```



Color spaces

The (R,G,B) tuples can be thought as defining a **space**:

The XYZ euclidean dimensions map to RGB.

Euclidean coordinates are not the only way to describe 3D space. There are also spherical and **cylindrical coordinates**:

Hue, Saturation, Value

We can use cylindrical coordinates for colors. This is known as HSV (HSB, HSL) colors.

- A color is still three numbers:
  1. **Hue**: The "color" is a single number between 0 and 360 degrees;
  2. **Saturation**: How "vibrant" the color is, between 0 and 1;
  3. **Value**: How "bright" the color is, also between 0 and 1.

A saturation of 0 corresponds to 'white', while a value of 0 corresponds to 'black'.

Look familiar?

For a fixed saturation, we get a 2D color space:

Python provides a nice module, `colorsys` for converting between color systems. Here's an example converting some HSV colors to RGB

[22]:
```
import colorsys # h in [0,1] for this module, not [0,360]

print(colorsys.hsv_to_rgb(1.0, 0.0,0.0 )) # black?
print(colorsys.hsv_to_rgb(0.5, 1.0,1.0 ))
print(colorsys.hsv_to_rgb(1.0, 1.0,1.0 ))
```

```
(0.0, 0.0, 0.0)
(0.0, 1.0, 1.0)
(1.0, 0.0, 0.0)
```

OK cylinders are great. So what?

Unlike RGB, HSV **separates** color and brightness/lightness. This lets us do certain operations more conveniently.

- For example, say we are plotting **five curves** and we want to use colors that are as **distinct as possible** so it's easy to tell the curves apart.

In RGB we need to carefully pick (r,g,b) values very far apart. But in HSV all we need to do is pick five values of H that are **evenly spaced between 0 and 360 degrees**:

[23]:
```
num_colors = 5.0
hue = 0.0
sat, val = 1.0, 1.0
while hue < 1.0:
    rgb = list( colorsys.hsv_to_rgb(hue, 1.0, 1.0) )

    hex = rgb256_to_hex(rgb_to_rgb256(rgb))

    print(hue, "-->", hex, rgb)

    hue += 1.0/num_colors;
```

```
0.0 --> #FF0000 [1.0, 0.0, 0.0]
0.2 --> #CBFF00 [0.7999999999999998, 1.0, 0.0]
0.4 --> #00FF66 [0.0, 1.0, 0.40000000000000036]
0.6000000000000001 --> #0065FF [0.0, 0.39999999999999947, 1.0]
0.8 --> #CC00FF [0.8000000000000007, 0.0, 1.0]
```

These colors are pretty much guaranteed to be as distinct as possible for a given number of colors. Of course, if you have hundreds of colors they will be forced to be very close to one another.

HSV is also nice for darkening or lightening a color without changing its saturation, just change V.

- If you take all the RGB channels and multiply them by a constant (like 0.5) you tend to **both darken the color and decrease its saturation!**

Useful functions

Here's some useful functions you may want to use.

[24]:
```python
def distinguishable_colors(num, sat=1.0, val=1.0):
    """Generate a list of `num' rgb hexadecimal color strings. The strings are
    linearly spaced along hue values from 0 to 1, leading to `num' colors with
    maximally different hues.

    Example:
    >>> print(distinguishable_colors(5))
    ['#ff0000', '#ccff00', '#00ff66', '#0066ff', '#cc00ff']
    """
    list_colors = [];
    hue = 0.0
    while abs(hue - 1.0) > 1e-4:
        rgb = list( colorsys.hsv_to_rgb(hue, sat, val) )
        list_colors.append( rgb_to_hex(rgb) )
        hue += 1.0/num;
    return list_colors


def rgb_to_hex(rgb):
    """Convert an rgb 3-tuple to a hexadecimal color string.

    Example:
    >>> print(rgb_to_hex((0.50,0.2,0.8)))
    #8033cc
    """
    return '#%02x%02x%02x' % tuple([round(x*255) for x in rgb])


def hex_to_rgb(hexrgb):
    """ Convert a hexadecimal color string to an rgb 3-tuple.

    Example:
    >>> print(hex_to_rgb("#8033CC"))
    (0.502, 0.2, 0.8)
    """
    hexrgb = hexrgb.lstrip('#')
```

```
    lv = len(hexrgb)
    return tuple(round(int(hexrgb[i:i+lv/3], 16)/255.0,4) for i in range(0, lv, lv/3))


def darken_hex(hexrgb, factor=0.5):
    """Take an rgb color of the form #RRGGBB and darken it by `factor' without
    changing the color. Specifically the RGB is converted to HSV and V ->
    V*factor.

    Example:
    >>> print(darken_hex("#8033CC"))
    '#401966'
    """
    rgb = hex_to_rgb(hexrgb)
    hsv = list(colorsys.rgb_to_hsv(*rgb))
    hsv[2] = hsv[2]*factor
    rgb = colorsys.hsv_to_rgb(*hsv)
    return rgb_to_hex(rgb)


def darken_rgb(rgb, factor=0.5):
    """Take an rgb 3-tuple and darken it by `factor', approximately
    preserving the hue.

    Example:
    >>> print(darken_rgb((0.5,0.2,0.7)))
    (0.251, 0.098, 0.3529)
    """
    hexrgb = darken_hex(rgb_to_hex(rgb), factor=factor)
    return hex_to_rgb(hexrgb)
```

Color blindness

Remember that some fraction of your audience may not be able to distinguish all colors equally well. Often red and green appear the same for color blind people.

- It's worth taking the time to make sure your graphic, drawing, plot, visualization, is readable **without color**. I will briefly put my monitor into *greyscale* mode just to check if my plot curves are distinct, for example.

There are tools for this as well:

- Color Oracle
- Color Filter
- Colblindor

Summary

Graphics involve math too!

1. Curves and colors are *mathemagical!*