PRINT: Python bootcamp 2020

Matplotlib basics

Johanna Hartke (jhartke@eso.org)

This tutorial/lecture is modelled after the introductory tutorial from matplotlib (https://matplotlib.org/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py). If in doubt, have a look at the matplotlib documentation ;)

What packages do we need?

For now, only

- [matplotlib (https://matplotlib.org)](https://matplotlib.org)
- [numpy (https://numpy.org)](https://numpy.org) (and strictly speaking, not even this).

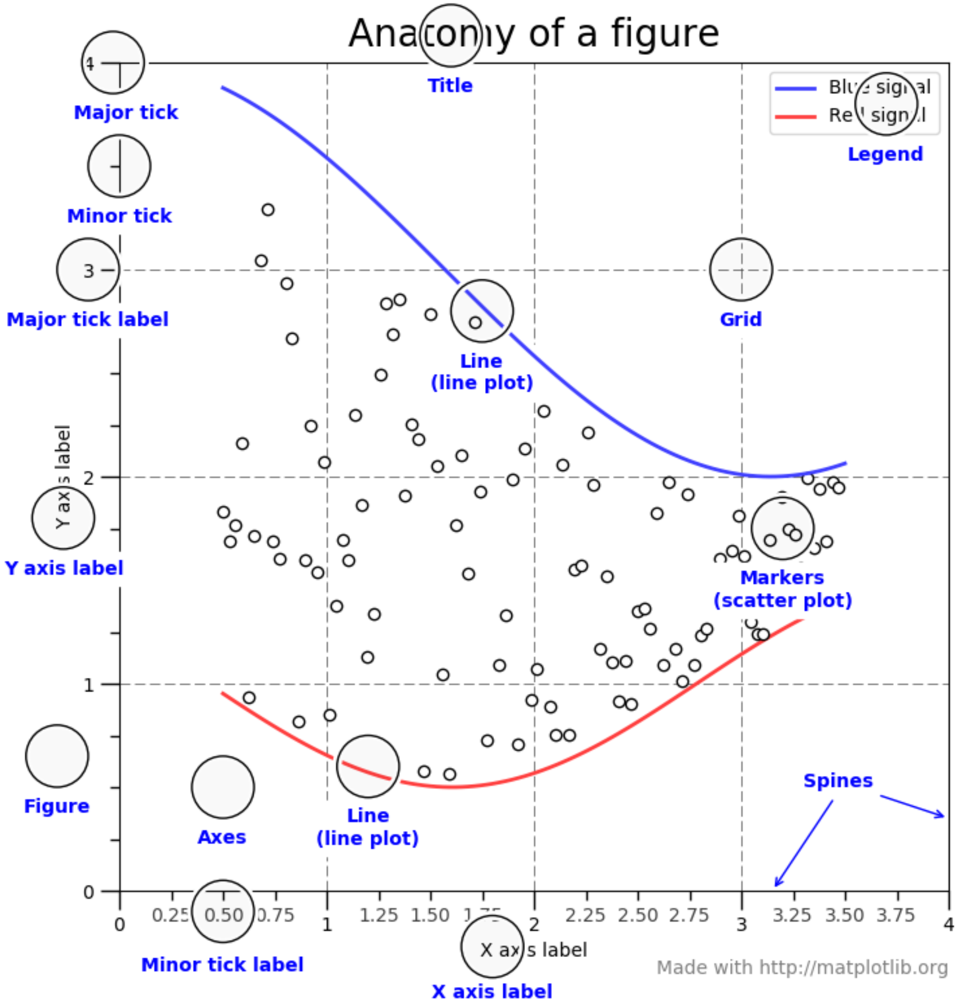
Later, you might want to have a look at

- [seaborn (https://seaborn.pydata.org)](https://seaborn.pydata.org)
- [bokeh (https://docs.bokeh.org/en/latest/#)](https://docs.bokeh.org/en/latest/#)

In [17]:

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

Basic figure specifications



The Figure

- Top level container for all the plot elements such as
  - axes
  - artists (titles, legends, etc)
  - canvas
- Different ways to create a new figure with pyplot:

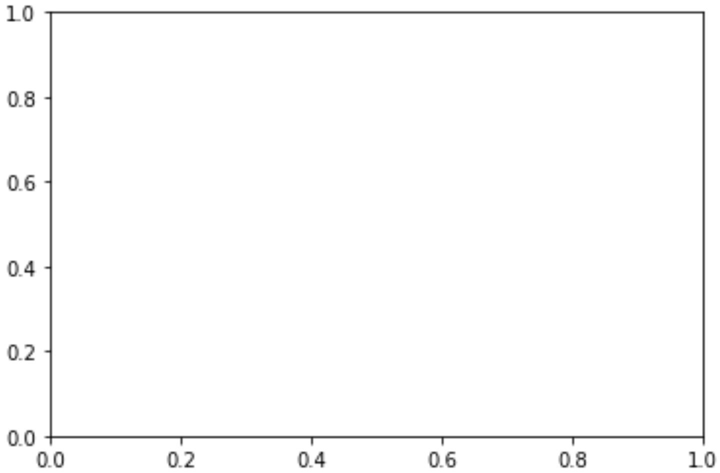
In [18]:

```
# empty figure, no axes
fig = plt.figure()
```

```
<Figure size 432x288 with 0 Axes>
```

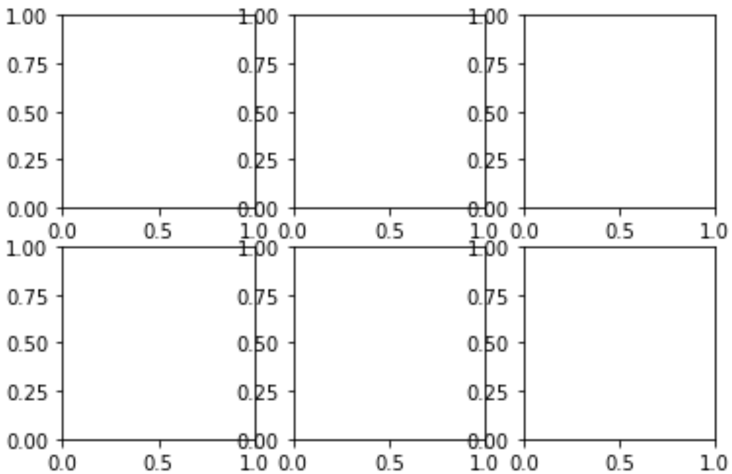
In [19]:

```
# figure with a single Axes
fig, ax = plt.subplots()
```



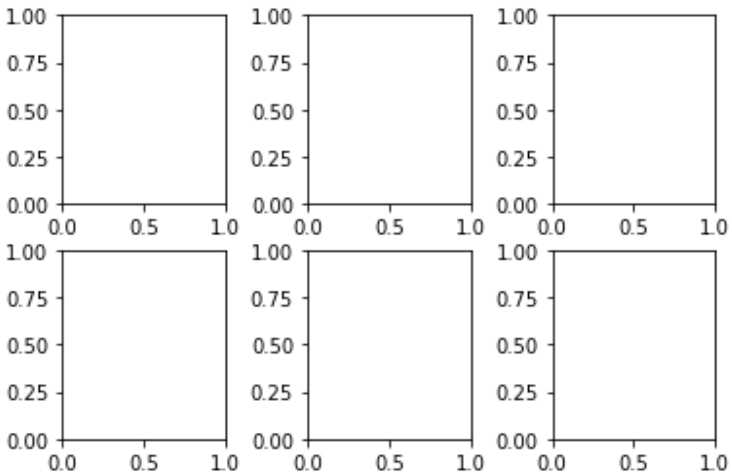
In [20]:

```
# figure with 2x3 grid of Axes
fig, axs = plt.subplots(2, 3)
```



In [21]:

```
fig, axs = plt.subplots(2, 3)
fig.subplots_adjust(hspace=0.25, wspace=0.5) # adjust the space between subplots
```



But what are Axes?

```
help(plt.Axes)
```

"The class `Axes` contains most of the figure elements."

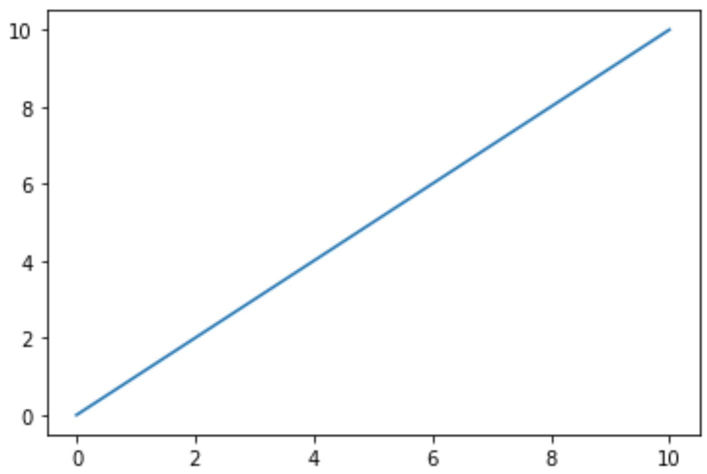
- Basically: **a plot**
- Data space
- `Axis`, i.e. x and y
- *Yes, it is quite easy to initially be confused between Axis and Axes*
- Set the title, x and y limits, labels

In [22]:

```python
# Let's start with a simple plot
# generate some data
xdata = np.linspace(0, 10, 100)
# create the figure
fig = plt.figure()
# add the Axes
ax = fig.add_subplot(111)
# simple plot
ax.plot(xdata, xdata)
```

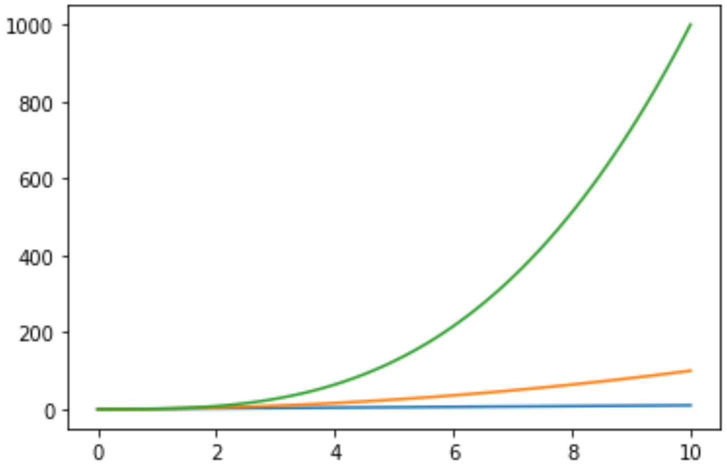
Out[22]:

```
[<matplotlib.lines.Line2D at 0x116c2a450>]
```



In [23]:
```
# We can easily add more lines
# create the figure
fig = plt.figure()
# add the Axes
ax = fig.add_subplot(111)
# simple plot
ax.plot(xdata, xdata)
ax.plot(xdata, xdata**2)
ax.plot(xdata, xdata**3)
```

Out[23]:
```
[<matplotlib.lines.Line2D at 0x116b02210>]
```



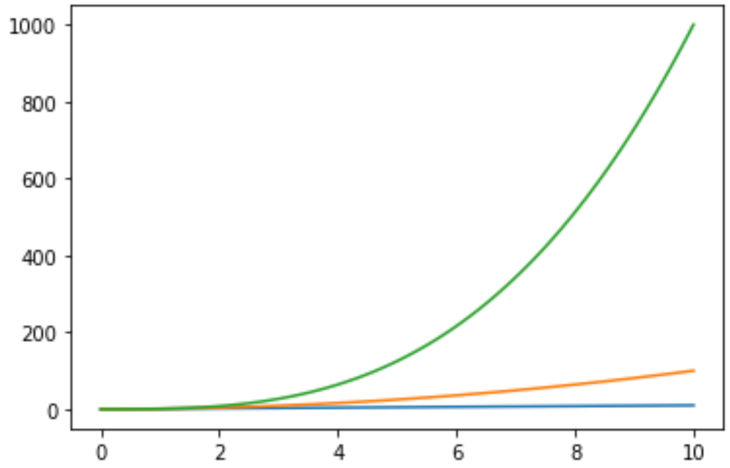
Something is missing...

If we wanted to send this Figure to the Journal of Simple Analytic Functions, it would still need (at least)

- Axis labels
- A legend
- Title (optional)

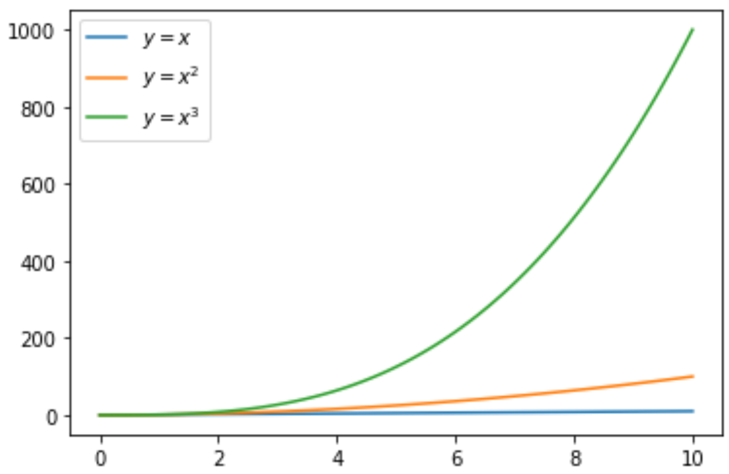
In [24]:
```python
# create the figure
fig = plt.figure()
# add the Axes
ax = fig.add_subplot(111)
# simple plot
ax.plot(xdata, xdata, label='$y=x$') # each plot gets a label, $$ brings us to TeX mode
ax.plot(xdata, xdata**2, label='$y=x^2$')
ax.plot(xdata, xdata**3, label='$y=x^3$')
```

Out[24]:
```
[&lt;matplotlib.lines.Line2D at 0x11634dd90&gt;]
```



In [25]:
```python
# create the figure
fig = plt.figure()
# add the Axes
ax = fig.add_subplot(111)
# simple plot
ax.plot(xdata, xdata, label='$y=x$') # each plot gets a label, $$ brings us to TeX mode
ax.plot(xdata, xdata**2, label='$y=x^2$')
ax.plot(xdata, xdata**3, label='$y=x^3$')
ax.legend() # in order to show the legend, we need to call it
```

Out[25]:
```
<matplotlib.legend.Legend at 0x1162b7d10>
```

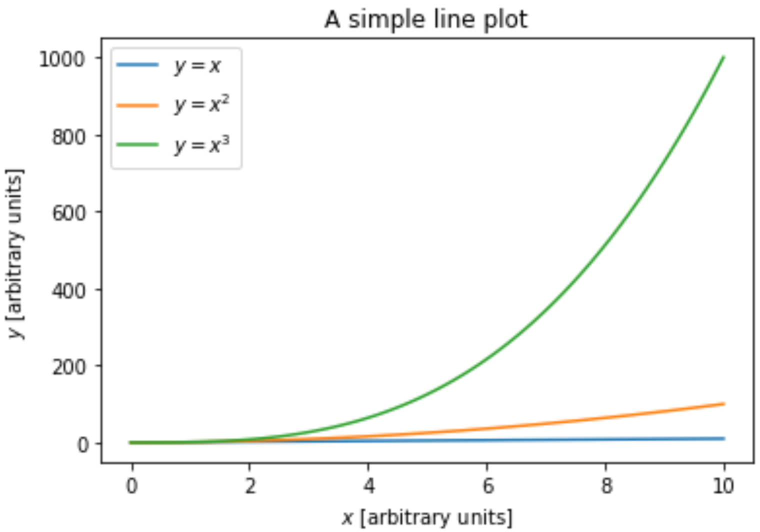


In [26]:

```
# create the figure
fig = plt.figure()
# add the Axes
ax = fig.add_subplot(111)
# simple plot
ax.plot(xdata, xdata, label='$y=x$') # each plot gets a label, $$ brings us to TeX mode
ax.plot(xdata, xdata**2, label='$y=x^2$')
ax.plot(xdata, xdata**3, label='$y=x^3$')
ax.legend() # in order to show the legend, we need to call it
ax.set_xlabel('$x$ [arbitrary units]')
ax.set_ylabel('$y$ [arbitrary units]')
ax.set_title('A simple line plot')
```

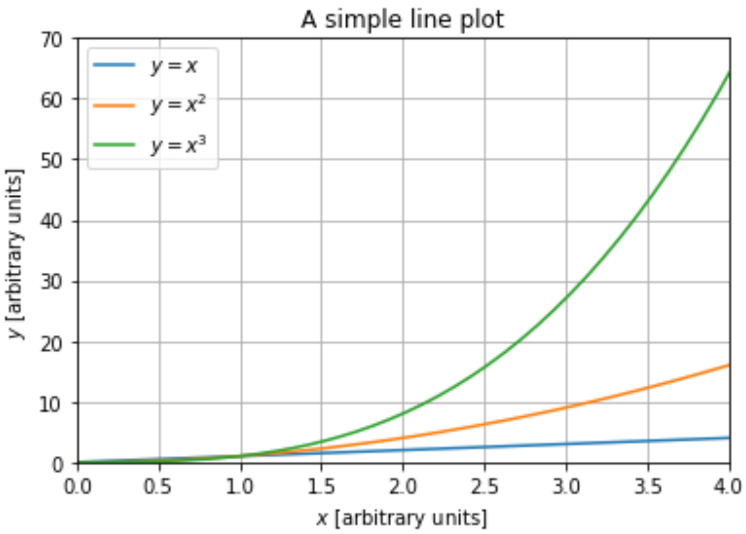
Out[26]:

```
Text(0.5, 1.0, 'A simple line plot')
```



In [27]:

```
# create the figure
fig = plt.figure()
# add the Axes
ax = fig.add_subplot(111)
# simple plot
ax.plot(xdata, xdata, label='$y=x$') # each plot gets a label, $$ brings us to TeX mode
ax.plot(xdata, xdata**2, label='$y=x^2$')
ax.plot(xdata, xdata**3, label='$y=x^3$')
ax.legend() # in order to show the legend, we need to call it
ax.set_xlabel('$x$ [arbitrary units]')
ax.set_ylabel('$y$ [arbitrary units]')
ax.set_title('A simple line plot')
ax.set_xlim(0, 4) # control the x-axis limits
ax.set_ylim(0, 70)
ax.grid('on') # grid
```



Controlling the appearance of a line plot

There are a myriad of things you can change to your liking:

- Markers
- Colour
- Line width
- Line style
- ...
- The documentation is your friend...

In [28]:
```
help(plt.plot)
```

```
Help on function plot in module matplotlib.pyplot:

plot(*args, scalex=True, scaley=True, data=None, **kwargs)
    Plot y versus x as lines and/or markers.

    Call signatures::

        plot([x], y, [fmt], *, data=None, **kwargs)
        plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

    The coordinates of the points or line nodes are given by *x*, *y*.

    The optional parameter *fmt* is a convenient way for defining basic
    formatting like color, marker and linestyle. It's a shortcut string
    notation described in the *Notes* section below.

    >>> plot(x, y)        # plot x and y using default line style and color
    >>> plot(x, y, 'bo')  # plot x and y using blue circle markers
    >>> plot(y)           # plot y using x as index array 0..N-1
    >>> plot(y, 'r+')     # ditto, but with red plusses

    You can use `.Line2D` properties as keyword arguments for more
    control on the appearance. Line properties and *fmt* can be mixed.
    The following two calls yield identical results:

    >>> plot(x, y, 'go--', linewidth=2, markersize=12)
    >>> plot(x, y, color='green', marker='o', linestyle='dashed',
    ...      linewidth=2, markersize=12)

    When conflicting with *fmt*, keyword arguments take precedence.


    **Plotting labelled data**

    There's a convenient way for plotting objects with labelled data (i.e.
```

There's a convenient way for plotting objects with labelled data (i.e.
data that can be accessed by index ``obj['y']``). Instead of giving
the data in *x* and *y*, you can provide the object in the *data*
parameter and just give the labels for *x* and *y*::

```
>>> plot('xlabel', 'ylabel', data=obj)
```

All indexable objects are supported. This could e.g. be a `dict`, a
`pandas.DataFame` or a structured numpy array.

Plotting multiple sets of data

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call `plot` multiple times.
  Example:

  ```
  >>> plot(x1, y1, 'bo')
  >>> plot(x2, y2, 'go')
  ```

- Alternatively, if your data is already a 2d array, you can pass it
  directly to *x*, *y*. A separate data set will be drawn for every
  column.

  Example: an array ``a`` where the first column represents the *x*
  values and the other columns are the *y* columns::

  ```
  >>> plot(a[0], a[1:])
  ```

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]*
  groups::

  ```
  >>> plot(x1, y1, 'g^', x2, y2, 'g-')
  ```

  In this case, any additional keyword argument applies to all
  datasets. Also this syntax cannot be combined with the *data*
  parameter.

```
By default, each line is assigned a different style specified by a
'style cycle'. The *fmt* and line property parameters are only
necessary if you want explicit deviations from these defaults.
Alternatively, you can also change the style cycle using the
'axes.prop_cycle' rcParam.


Parameters
----------
x, y : array-like or scalar
    The horizontal / vertical coordinates of the data points.
    *x* values are optional and default to `range(len(y))`.

    Commonly, these parameters are 1D arrays.

    They can also be scalars, or two-dimensional (in that case, the
    columns represent separate data sets).

    These arguments cannot be passed as keywords.

fmt : str, optional
    A format string, e.g. 'ro' for red circles. See the *Notes*
    section for a full description of the format strings.

    Format strings are just an abbreviation for quickly setting
    basic line properties. All of these and more can also be
    controlled by keyword arguments.

    This argument cannot be passed as keyword.

data : indexable object, optional
    An object with labelled data. If given, provide the label names to
    plot in *x* and *y*.

    .. note::
        Technically there's a slight ambiguity in calls where the
        second label is a valid *fmt*. `plot('n', 'o', data=obj)`
        could be `plt(x, y)` or `plt(y, fmt)`. In such cases,
```

```
        the former interpretation is chosen, but a warning is issued.
        You may suppress the warning by adding an empty format string
        `plot('n', 'o', '', data=obj)`.

    Other Parameters
    ----------------
    scalex, scaley : bool, optional, default: True
        These parameters determined if the view limits are adapted to
        the data limits. The values are passed on to `autoscale_view`.

    **kwargs : `.Line2D` properties, optional
        *kwargs* are used to specify properties like a line label (for
        auto legends), linewidth, antialiasing, marker face color.
        Example::

        >>> plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
        >>> plot([1,2,3], [1,4,9], 'rs',  label='line 2')

        If you make multiple lines with one plot command, the kwargs
        apply to all those lines.

        Here is a list of available `.Line2D` properties:

      agg_filter: a filter function, which takes a (m, n, 3) float array and a
dpi value, and returns a (m, n, 3) array
      alpha: float
      animated: bool
      antialiased or aa: bool
      clip_box: `.Bbox`
      clip_on: bool
      clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
      color or c: color
      contains: callable
      dash_capstyle: {'butt', 'round', 'projecting'}
      dash_joinstyle: {'miter', 'round', 'bevel'}
      dashes: sequence of floats (on/off ink in points) or (None, None)
      drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-p
ost'}, default: 'default'
```

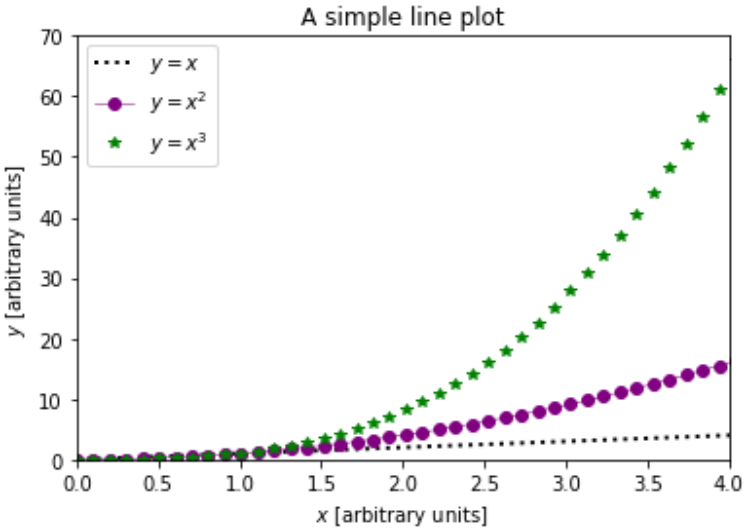
```
      figure: `.Figure`
      fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
      gid: str
      in_layout: bool
      label: object
      linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
      linewidth or lw: float
      marker: marker style
      markeredgecolor or mec: color
      markeredgewidth or mew: float
      markerfacecolor or mfc: color
      markerfacecoloralt or mfcalt: color
      markersize or ms: float
      markevery: None or int or (int, int) or slice or List[int] or float or
(float, float)
      path_effects: `.AbstractPathEffect`
      picker: float or callable[[Artist, Event], Tuple[bool, dict]]
      pickradius: float
      rasterized: bool or None
      sketch_params: (scale: float, length: float, randomness: float)
      snap: bool or None
      solid_capstyle: {'butt', 'round', 'projecting'}
      solid_joinstyle: {'miter', 'round', 'bevel'}
      transform: `matplotlib.transforms.Transform`
      url: str
      visible: bool
      xdata: 1D array
      ydata: 1D array
      zorder: float

    Returns
    -------
    lines
        A list of `.Line2D` objects representing the plotted data.

    See Also
    --------
    scatter : XY scatter plot with markers of varying size and/or color (
```

```
    sometimes also called bubble chart).

Notes
-----
**Format Strings**

A format string consists of a part for color, marker and line::

    fmt = '[marker][line][color]'

Each of them is optional. If not provided, the value from the style
cycle is used. Exception: If ``line`` is given, but no ``marker``,
the data will be a line without markers.

Other combinations such as ``[color][marker][line]`` are also
supported, but note that their parsing may be ambiguous.

**Markers**

=============    ===============================
character        description
=============    ===============================
``'.'``          point marker
``','``          pixel marker
``'o'``          circle marker
``'v'``          triangle_down marker
``'^'``          triangle_up marker
``'<'``          triangle_left marker
``'>'``          triangle_right marker
``'1'``          tri_down marker
``'2'``          tri_up marker
``'3'``          tri_left marker
``'4'``          tri_right marker
``'s'``          square marker
``'p'``          pentagon marker
``'*'``          star marker
``'h'``          hexagon1 marker
``'H'``          hexagon2 marker
```

```
``'+'``           plus marker
``'x'``           x marker
``'D'``           diamond marker
``'d'``           thin_diamond marker
``'|'``           vline marker
``'_'``           hline marker
=============    ===============================

**Line Styles**

=============    ===============================
character        description
=============    ===============================
``'-'``           solid line style
``'--'``          dashed line style
``'-.'``          dash-dot line style
``':'``           dotted line style
=============    ===============================

Example format strings::

    'b'    # blue markers with default shape
    'or'   # red circles
    '-g'   # green solid line
    '--'   # dashed line with default color
    '^k:'  # black triangle_up markers connected by a dotted line

**Colors**

The supported color abbreviations are the single letter codes

=============    ===============================
character        color
=============    ===============================
``'b'``           blue
``'g'``           green
``'r'``           red
``'c'``           cyan
```

```
``'m'``          magenta
``'y'``          yellow
``'k'``          black
``'w'``          white
=============    ===============================

and the ``'CN'`` colors that index into the default property cycle.

If the color is the only part of the format string, you can
additionally use any  `matplotlib.colors` spec, e.g. full names
(``'green'``) or hex strings (``'#008000'``).
```

```
In [29]: # create the figure
         fig = plt.figure()
         # add the Axes
         ax = fig.add_subplot(111)
         # simple plot
         ax.plot(xdata, xdata, label='$y=x$', ls=':', lw=2, c='black')
         ax.plot(xdata, xdata**2, label='$y=x^2$', ls='-',
                 lw=0.5, marker='o', color='purple')
         ax.plot(xdata, xdata**3, '*', c='g',
                 label='$y=x^3$') # shorthand notation
         ax.legend()
         ax.set_xlabel('$x$ [arbitrary units]')
         ax.set_ylabel('$y$ [arbitrary units]')
         ax.set_title('A simple line plot')
         ax.set_xlim(0, 4)
         ax.set_ylim(0, 70)

Out[29]: (0, 70)
```



Excercise 1

Create a figure that contains one axis showing

- a sine and a cosine function, with the x-axis ranging from -π to π and the y-axis from -1.05 to 1.05.
- The figure should have meaningful axis labels and a legend that does not overlap with the plot.
- The two plotted lines have to be distinguishable not only by their colour.
- Labels and legend have to be rendered in TeX
- Save the figure to disk (Hint: matplotlib documentation (https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.savefig.html))

There are two ways to use Matplotlib.

For the plot above, we have used the **object-oriented style**.

- We explicitely created figures and axes, and then called methods on them.

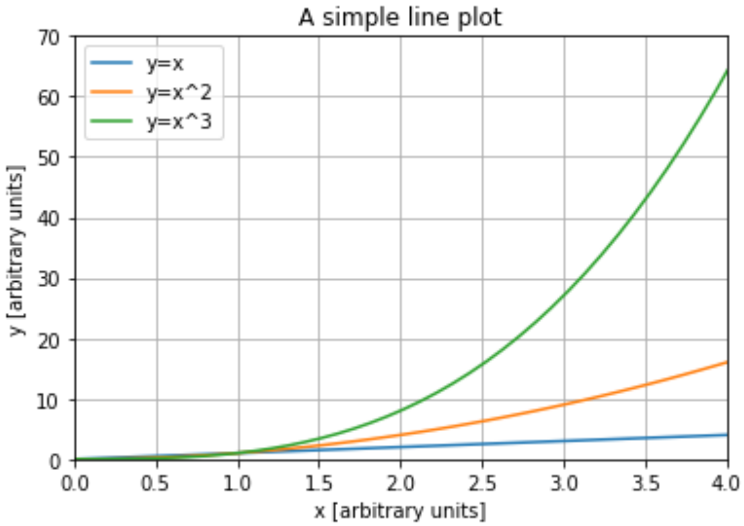
We could also have used the so-called **pyplot-style**

- Pyplot automatically creates and manages the figures and axes.

I personally prefer the object-oriented style, as it is easier to keep control over complicated figures. Below is an example of the same figure in pyplot style.

In [31]:
```
plt.plot(xdata, xdata, label='y=x')
plt.plot(xdata, xdata**2, label='y=x^2')
plt.plot(xdata, xdata**3, label='y=x^3')
plt.xlabel('x [arbitrary units]')
plt.ylabel('y [arbitrary units]')
plt.title('A simple line plot')
plt.legend()
plt.grid('on')
plt.xlim(0, 4)
plt.ylim(0, 70)
```

Out[31]:
```
(0, 70)
```



Scatter plots

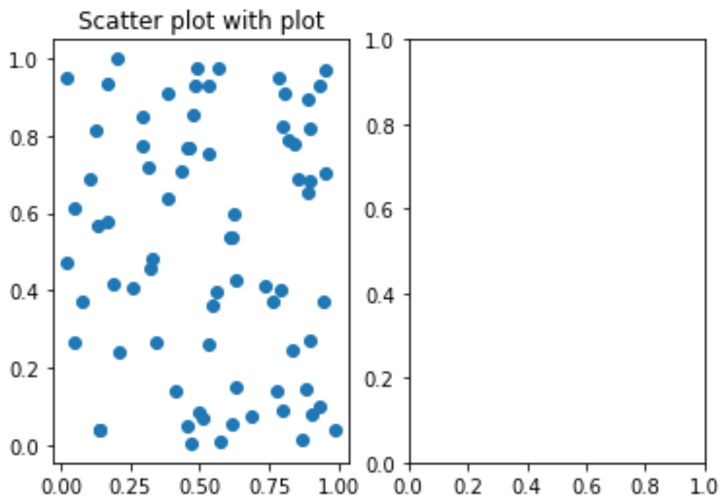
- Method 1: `plot` and set a specific marker
    - Good for simple scatter plots
- Method 2: `scatter`
    - More room for customisation

In [32]:

```
# create some random data with numpy
N = 75
x, y = np.random.rand(2, N)
c = np.random.randint(1, 25, size=N)
s = np.random.randint(10, 150, size=N)
```

In [33]:
```
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax1.plot(x, y, 'o')
ax1.set_title('Scatter plot with plot')
```

Out[33]:
```
Text(0.5, 1.0, 'Scatter plot with plot')
```

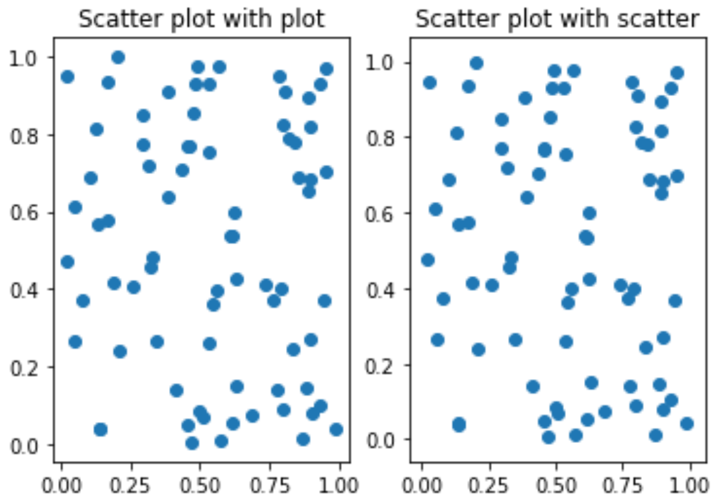


In [34]:

```
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax1.plot(x, y, 'o')
ax1.set_title('Scatter plot with plot')
ax2.scatter(x, y)
ax2.set_title('Scatter plot with scatter')
```

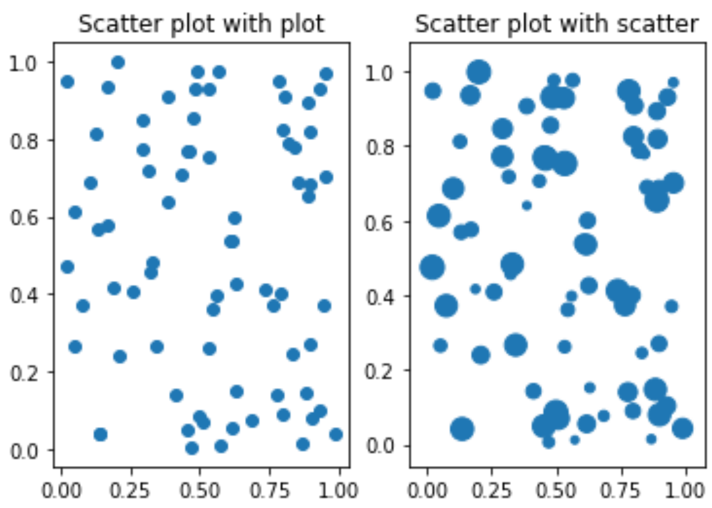
Out[34]:

```
Text(0.5, 1.0, 'Scatter plot with scatter')
```



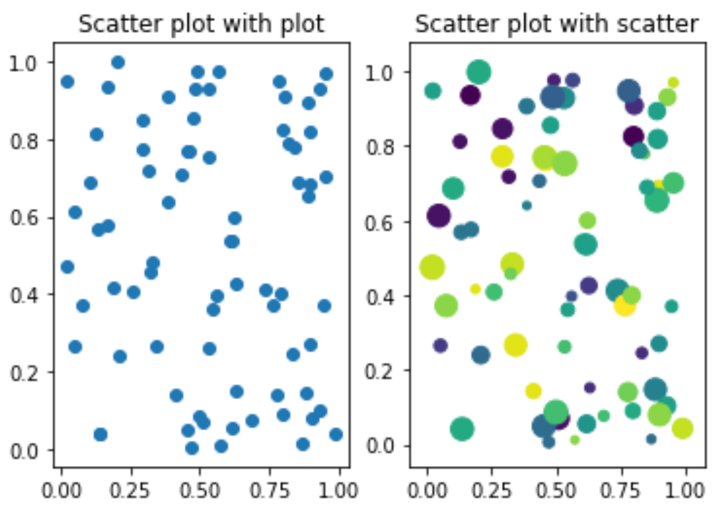
In [35]:
```
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax1.plot(x, y, 'o')
ax1.set_title('Scatter plot with plot')
ax2.set_title('Scatter plot with scatter')
ax2.scatter(x, y, s=s)
```

Out[35]:
```
<matplotlib.collections.PathCollection at 0x116324d10>
```



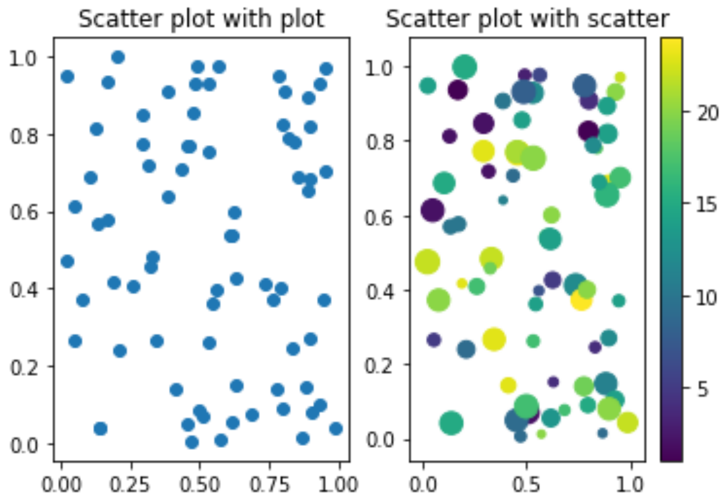
In [36]:
```
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax1.plot(x, y, 'o')
ax1.set_title('Scatter plot with plot')
ax2.set_title('Scatter plot with scatter')
ax2.scatter(x, y, s=s, c=c)
```

Out[36]:
```
<matplotlib.collections.PathCollection at 0x1167846d0>
```



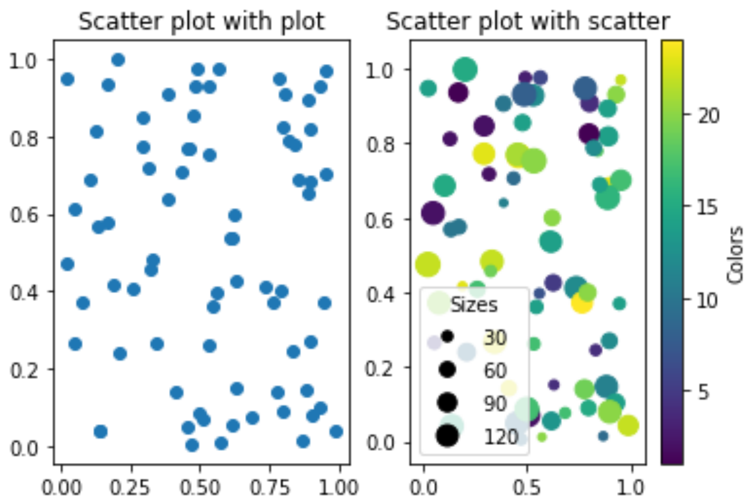
In [37]:

```
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax1.plot(x, y, 'o')
ax1.set_title('Scatter plot with plot')
ax2.set_title('Scatter plot with scatter')
sc = ax2.scatter(x, y, s=s, c=c)
cbar = fig.colorbar(sc)
```



In [38]:
```
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax1.plot(x, y, 'o')
ax1.set_title('Scatter plot with plot')
ax2.set_title('Scatter plot with scatter')
sc = ax2.scatter(x, y, s=s, c=c)
cbar = fig.colorbar(sc)
cbar.set_label('Colors')
ax2.legend(*sc.legend_elements(prop='sizes', num=5), title='Sizes')
```

Out[38]:
```
<matplotlib.legend.Legend at 0x116611c90>
```

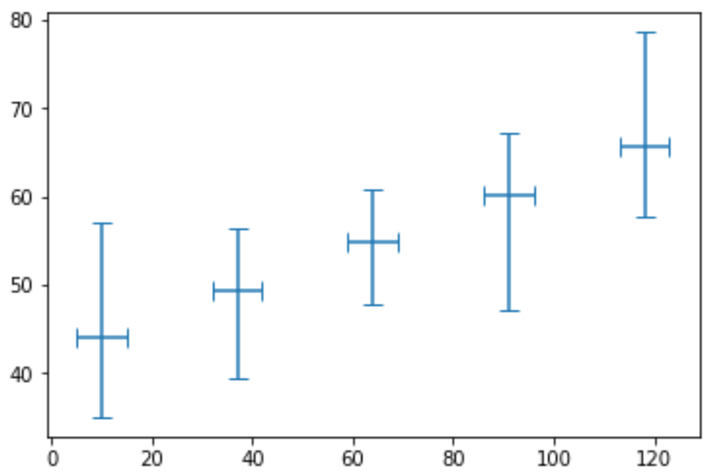


Error bars

You will not get around them ;)

In [39]:
```
# Again, we'll first "fabricate" some data
xdata = np.arange(10, 120, 27)
xerr = 5
ydata = xdata*0.2 + 42.
yerr_lo = np.random.randint(5, 15, len(ydata))
yerr_hi = np.random.randint(5, 15, len(ydata))

# and then plot
fig = plt.figure()
ax = fig.add_subplot(111)
ax.errorbar(xdata, ydata,
            xerr=xerr, yerr=[yerr_lo, yerr_hi],
            fmt=' ', capsize=5)
```

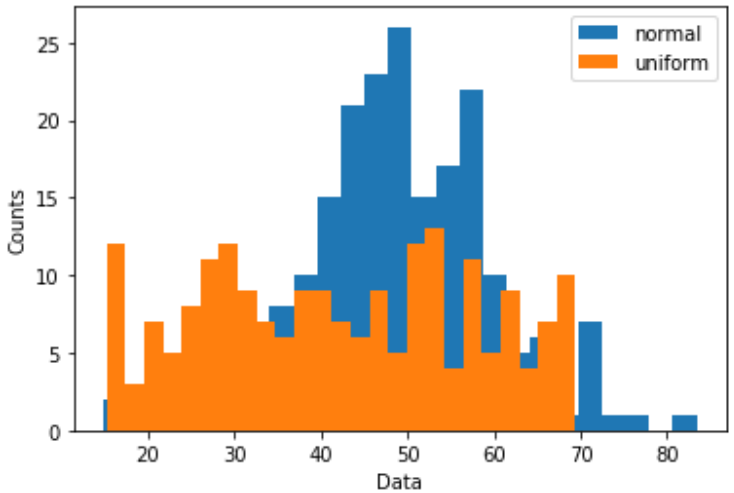
Out[39]:
```
<ErrorbarContainer object of 3 artists>
```



Histograms

In [40]:
```
# generating some random distributions
data1 = np.random.normal(50, 10, 200)
data2 = np.random.uniform(15, 70, 200)

# plot
fig = plt.figure()
ax1 = fig.add_subplot(111)
N, bins, patches = ax1.hist(data1, bins=25, label='normal')
N, bins, patches = ax1.hist(data2, bins=25, label='uniform')
ax1.legend()
ax1.set_ylabel('Counts')
ax1.set_xlabel('Data')
```

Out[40]:
```
Text(0.5, 0, 'Data')
```

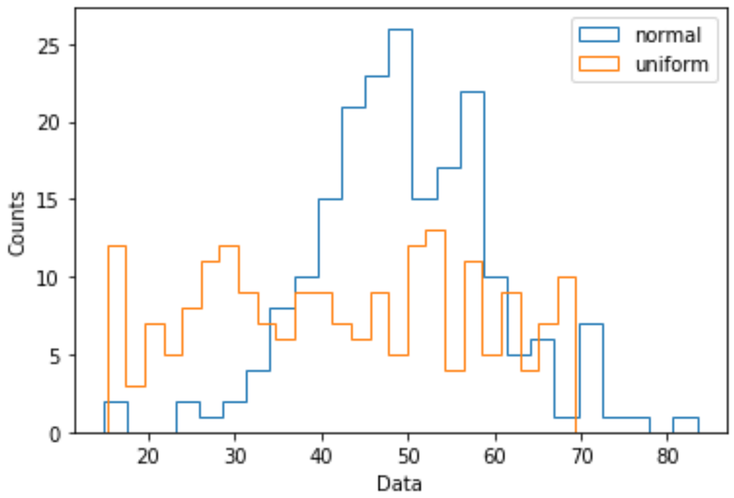


In [41]:

```
# plot
fig = plt.figure()
ax1 = fig.add_subplot(111)
N, bins, patches = ax1.hist(data1, bins=25, label='normal', histtype='step')
N, bins, patches = ax1.hist(data2, bins=25, label='uniform', histtype='step')
ax1.legend()
ax1.set_ylabel('Counts')
ax1.set_xlabel('Data')
```

Out[41]:

```
Text(0.5, 0, 'Data')
```

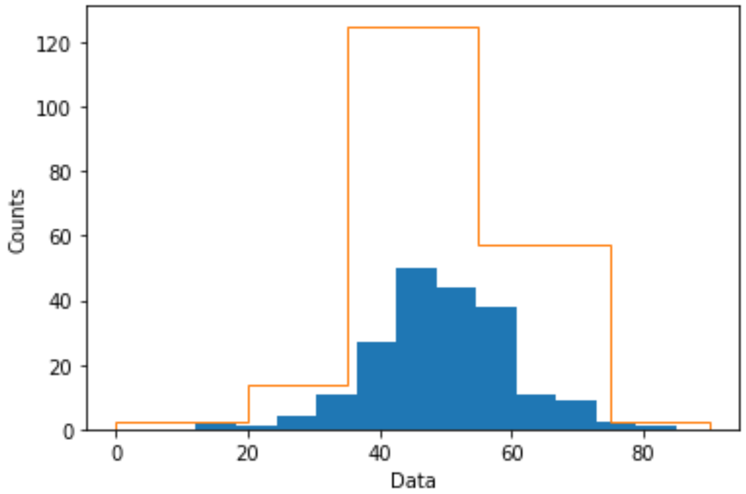


In [53]:

```
# custom bins
bins1 = np.linspace(0, 85, 15)
bins2 = np.array([0, 20, 35, 55, 75, 90])

fig = plt.figure()
ax1 = fig.add_subplot(111)
N, bins, patches = ax1.hist(data1, bins=bins1, histtype='stepfilled')
N, bins, patches = ax1.hist(data1, bins=bins2, histtype='step')
ax1.set_ylabel('Counts')
ax1.set_xlabel('Data')
```

Out[53]:

```
Text(0.5, 0, 'Data')
```

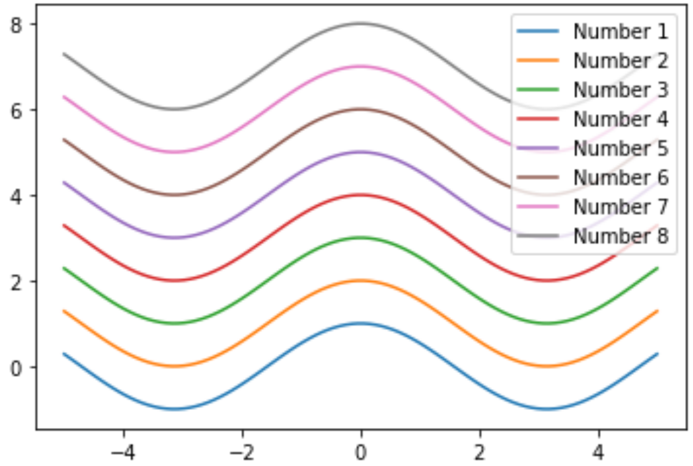


Python functionalities

- Use python syntax to make plotting more efficient
- Loops
- Conditionals
- Lists
- ...

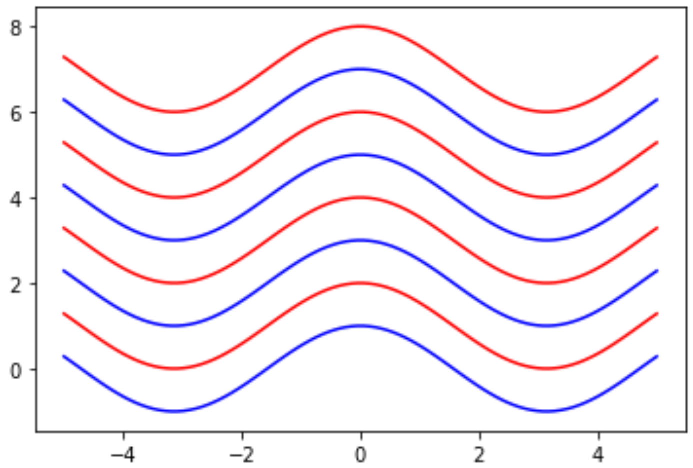
In [59]:
```
xrange = np.linspace(-5, 5, 100)

fig = plt.figure()
ax1 = fig.add_subplot(111)
for i in range(8):
    ax1.plot(xrange, np.cos(xrange)+i, label='Number '+str(i+1))
ax1.legend()
```

Out[59]: <matplotlib.legend.Legend at 0x11800dfd0>



In [63]:

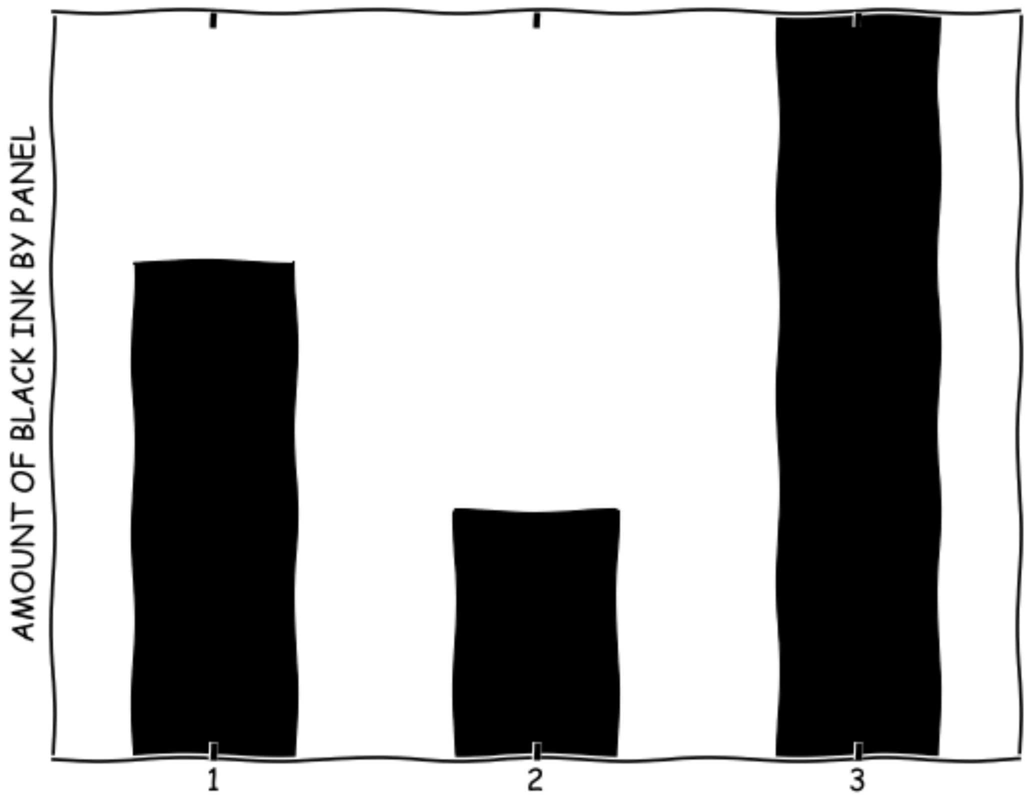
```
fig = plt.figure()
ax1 = fig.add_subplot(111)
for i in range(8):
    # colour code even-odd
    if i%2==0:
        ax1.plot(xrange, np.cos(xrange)+i, label='Number '+str(i+1), color='b')
    else:
        ax1.plot(xrange, np.cos(xrange)+i, label='Number '+str(i+1), color='r')
```



Fun with ~~Flags~~ Style Sheets

In [95]:

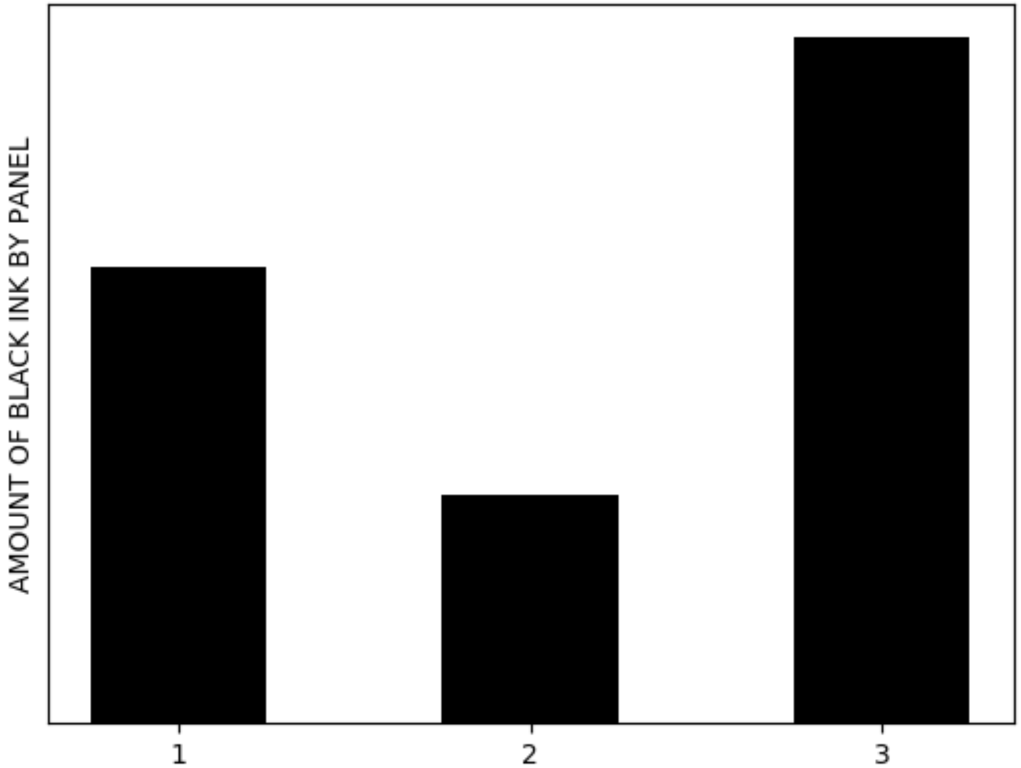
```
bars = [1,2,3]
heights = [2,1,3]

with plt.xkcd():
    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax1.bar(bars, heights, width=0.5, color='k')
    ax1.set_xticks([1, 2, 3])
    ax1.set_ylabel('AMOUNT OF BLACK INK BY PANEL')
    ax1.set_yticks([])
```



In [6]:

```
plt.style.use('default')
bars = [1,2,3]
heights = [2,1,3]
fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.bar(bars, heights, width=0.5, color='k')
ax1.set_xticks([1, 2, 3])
ax1.set_ylabel('AMOUNT OF BLACK INK BY PANEL')
ax1.set_yticks([])
```

Out[6]: []



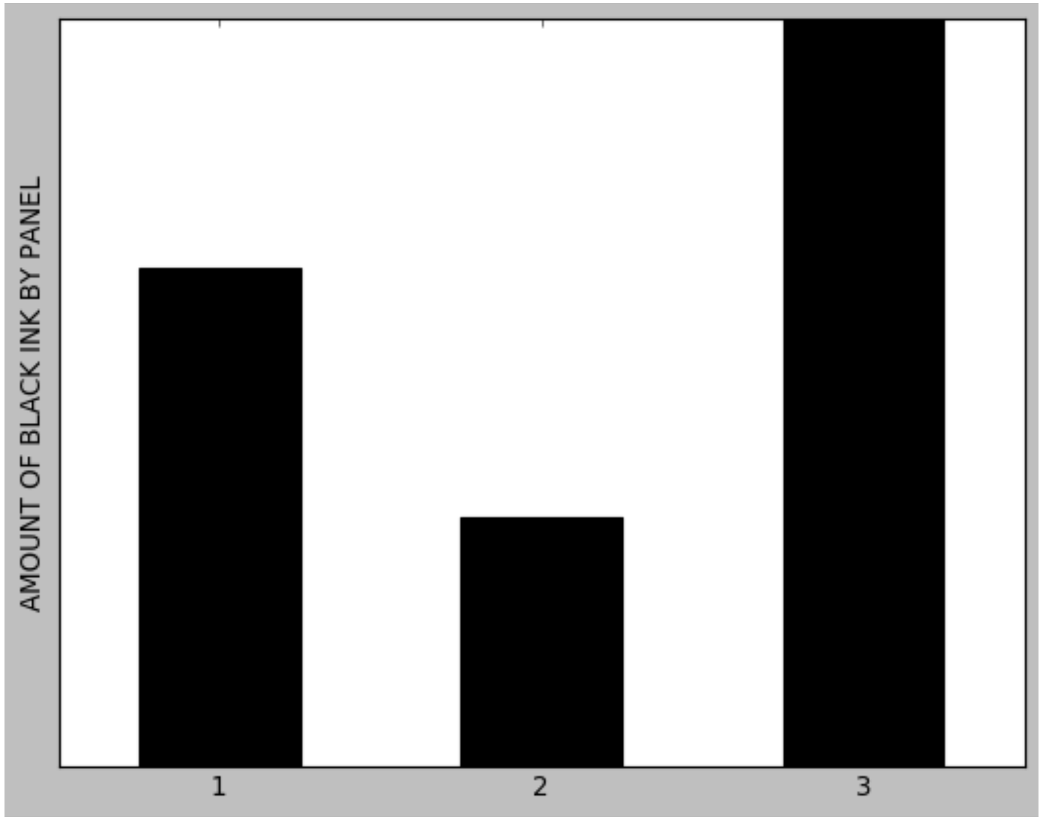
In [7]:
```
print(plt.style.available)
```

```
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight', 'seab
orn-whitegrid', 'classic', '_classic_test', 'fast', 'seaborn-talk', 'seaborn-d
ark-palette', 'seaborn-bright', 'seaborn-pastel', 'grayscale', 'seaborn-notebo
ok', 'ggplot', 'seaborn-colorblind', 'seaborn-muted', 'seaborn', 'Solarize_Lig
ht2', 'seaborn-paper', 'bmh', 'tableau-colorblind10', 'seaborn-white', 'dark_b
ackground', 'seaborn-poster', 'seaborn-deep', 'presentation', 'sansserif', 'pu
blication', 'publication_py3', 'sansserif_proposal']
```

In [94]:

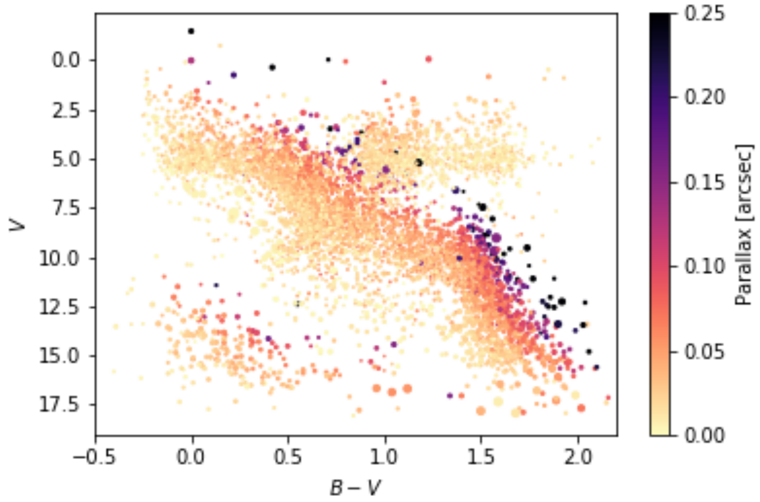
```
plt.style.use('classic')
bars = [1,2,3]
heights = [2,1,3]
fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.bar(bars, heights, width=0.5, color='k')
ax1.set_xticks([1, 2, 3])
ax1.set_ylabel('AMOUNT OF BLACK INK BY PANEL')
ax1.set_yticks([])
```

Out[94]: []



Excercise 2: Plot a Hertzsprung-Russell diagram

The goal of this excercise is to create a Hertzsprung-Russell diagram from tabular data.



Prelim: read in the data

Data can be found [here (http://burro.astr.cwru.edu/Academics/Astr221/HW/HW5/yaletrigplx.dat)](http://burro.astr.cwru.edu/Academics/Astr221/HW/HW5/yaletrigplx.dat). The columns are

- column 1: star ID number
- column 2: apparent V magnitude
- column 3: observed B-V color
- column 4: observed parallax (in arcsec)
- column 5: uncertainty in parallax (in milliarcsec)

2a) Plot the distribution of the observed parallaxes

- Determine a suitable bin size
- Make a histogram

2b) The actual HR diagram

- Plot magnitude versus colour
- As usual, in astronomy the magnitude axis is inverted
- Each point should be coloured by its respective parallax
- Use the information from 2a to determine appropriate colour cuts
- The final figure should have
  - sensible axis limits
  - axis labels
  - a labelled colour bar
  - a title
- Bonus: the size of each point should be inversely related to the measurement error of the parallax

In [ ]:

```

```