Astroinformatics I (Semester 1 2024)

Python Data Exploration & Visualization

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

June 17, 2024

Motivation

Data exploration is a **critical step** in any data analysis project. It helps us understand the dataset, identify patterns, and gain insights.

Without spending significant time on understanding the data and its patterns one cannot expect to build efficient predictive models. Data exploration takes a major chunk of time in a science project.

In this class, we will see the various steps involved in data exploration through general concepts and Python code snippets.

Libraries

Earlier we saw how to do basic math with Python. We also used various data types that are available already within Python (such as integers, floats, strings, lists).

However, once a code requires more **sophisticated analytical tools** (especially for astronomical processes), it becomes apparent that the built-in Python functions are not sufficient. Luckily, there are hundreds of functions that have been written to accomplish these tasks, most of which are organized into what are called libraries.

Definition

A library is a maintained collection of functions which can be installed and imported into a Python code to be used.

Most Python distributions come with a lot of these libraries included, and the installation of new libraries is generally straightforward.

Libraries

There are 4 key libraries that we will discuss in detail in this course: numpy, matplotlib, astropy, scipy.

NumPy is an extremely versatile library of functions to do things Python itself can't. For example, Python doesn't provide trigonometric functions.
That's where NumPy comes in.

Libraries

There are 4 key libraries that we will discuss in detail in this course: numpy, matplotlib, astropy, scipy.

NumPy is an extremely versatile library of functions to do things Python itself can't. For example, Python doesn't provide trigonometric functions.
That's where NumPy comes in.

Matplotlib is a library with functions dedicated to plotting data and making graphs.

Libraries

There are 4 key libraries that we will discuss in detail in this course: numpy, matplotlib, astropy, scipy.

NumPy is an extremely versatile library of functions to do things Python itself can't. For example, Python doesn't provide trigonometric functions.
That's where NumPy comes in.

Matplotlib is a library with functions dedicated to plotting data and making graphs.

Astropy is a library with functions specifically for astronomical applications:
we will e.g. use it to import `fits` files (containing science images and tables).

Libraries

There are 4 key libraries that we will discuss in detail in this course: numpy, matplotlib, astropy, scipy.

NumPy is an extremely versatile library of functions to do things Python itself can't. For example, Python doesn't provide trigonometric functions.
That's where NumPy comes in.

Matplotlib is a library with functions dedicated to plotting data and making graphs.

Astropy is a library with functions specifically for astronomical applications:
we will e.g. use it to import `fits` files (containing science images and tables).

Scipy is a library that contains special functions that are often used in science.

Libraries

How to find the right library and function within?

Since there are thousands of these functions, instead of memorizing them all, the best way to learn is to Google or query Stack Exchange for the type of function you are looking for, and you'll find a library containing the function you need. The ones you use most often will then become second nature.

Modules

A module is simply a file containing Python definitions, functions, and statements. Putting code into modules is useful because of the ability to import the module functionality into your code, for instance:

```
import astropy
import astropy.table
data = astropy.table.Table.read('my_table.fits')
```

You will find `import` in almost every Python script.

Packages

A **package** is just a way of collecting related modules together within a single tree-like hierarchy. Very complex packages like NumPy or SciPy have hundreds of individual modules so putting them into a directory-like structure keeps things organized and avoids name collisions. For example here is a partial list of sub-packages available within SciPy:

`scipy.fftpack`	Discrete Fourier Transform algorithms
`scipy.stats`	Statistical Functions
`scipy.lib`	Python wrappers to external libraries
`scipy.lib.blas`	Wrappers to BLAS library
`scipy.lib.lapack`	Wrappers to LAPACK library
`scipy.integrate`	Integration routines
`scipy.linalg`	Linear algebra routines
`scipy.sparse.linalg`	Sparse Linear Algebra
`scipy.sparse.linalg.eigen`	Sparse Eigenvalue Solvers

Importing Libraries


```
#Example: Importing libraries
import numpy
import astropy
import astropy.table
import astropy.io.fits
```

The dot notation of some imports is associated with classes. Some libraries are huge, so it is best to only load the functions you really need.

Importing Libraries


```
#Example: Importing libraries
import numpy
import astropy
import astropy.table
import astropy.io.fits
```

The dot notation of some imports is associated with classes. Some libraries are huge, so it is best to only load the functions you really need.

When importing libraries, one can name them whatever we want for the purposes of our code. A standard choices is:

```
import numpy as np
```

Importing Libraries

When using those functions, again the dot notation is used to let Python know from which libary the function you are calling is coming from.

```
#Example: sin function
import numpy
x = numpy.arange(100)
y = numpy.sin(x)
```

Installing Libraries

As mentioned above, most scientific distributions of Python (like Anaconda) come with important packages like numpy preinstalled. However, for most smaller packages, like astropy, or pyfits, or those for programs you are using written by other scientists, you will likely have to install them yourself. The easiest way is to use `pip`, a package installer already available on most computers.

Installing Libraries

As mentioned above, most scientific distributions of Python (like Anaconda) come with important packages like numpy preinstalled. However, for most smaller packages, like astropy, or pyfits, or those for programs you are using written by other scientists, you will likely have to install them yourself. The easiest way is to use `pip`, a package installer already available on most computers.

The easiest way to see if a package is in `pip` is to just try to `pip install` it, if it works then you're done:

```
$ pip install packagename
```

Installing Libraries

As mentioned above, most scientific distributions of Python (like Anaconda) come with important packages like numpy preinstalled. However, for most smaller packages, like astropy, or pyfits, or those for programs you are using written by other scientists, you will likely have to install them yourself. The easiest way is to use `pip`, a package installer already available on most computers.

The easiest way to see if a package is in `pip` is to just try to `pip install` it, if it works then you're done:

```
$ pip install packagename
```

If it says "not found" then you might have to look up online how to install it. Usually it is required to download an archive and running a Python script like the following:

```
$ python setup.py install
```

A final thought on working with packages

How do we know how a function actually works, what its inputs and outputs are?

A final thought on working with packages

How do we know how a function actually works, what its inputs and outputs are?

We can take a look at the documentation of that library or function (and I strongly recommend that).
We can, however, also do someting straight within the interpreter. Simply type

A final thought on working with packages

How do we know how a function actually works, what its inputs and outputs are?

We can take a look at the documentation of that library or function (and I strongly recommend that).
We can, however, also do someting straight within the interpreter. Simply type

```
[IN]: help(np.genfromtxt)
```

(plug in your function of choice) and Python will give you a helpful rundown of how the function works. To advance through the documentation, keep hitting Enter , or hit Q to exit out of it.

Loading Data

The first step in data exploration is to load data and identify variables:

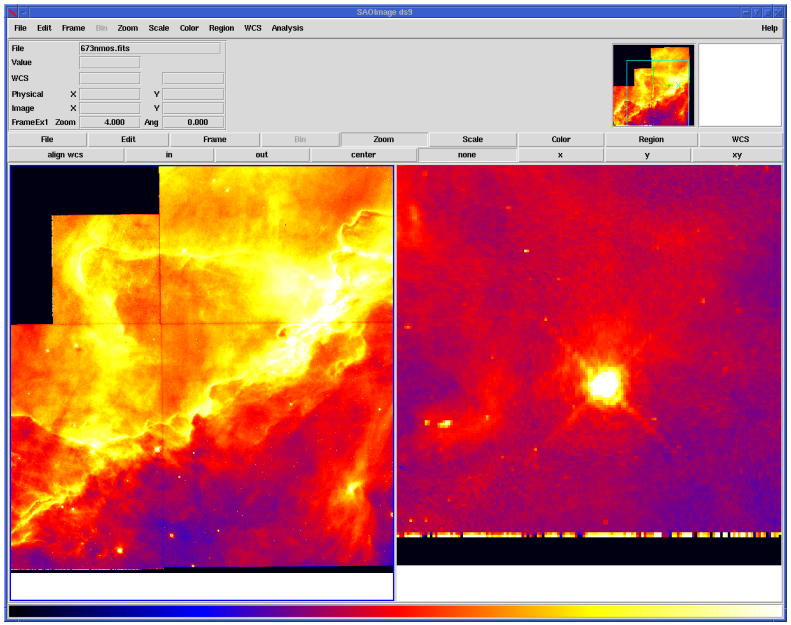
Data sources can vary: e.g. text files, databases, websites and remote services.

We have seen how to load tables in text format (`*.txt`, `*.csv`).
We will see now how to load files in another format that is commonly used in astronomy: **FITS files**.

Astronomical Fits Files

FITS (*Flexible Image Transport System*) is the standard astronomical data format endorsed by both NASA and the IAU. It is widely used in the astronomy community to store images and tables.
FITS can store scientific data sets consisting of multi- dimensional arrays (1-D spectra, 2-D tables, 2-D images or 3-D data cubes).

There exists a range of FITS file viewers; a list of such can be found here: `https://fits.gsfc.nasa.gov/fits_viewer.html`



Astronomical Fits Files

In addition to FITS viewers, there exits software packages allowing for image viewing, data analysis, and format conversion.

For Python, such packages allow for seemlessly integrating FITS files into scientific code. It should be noted that FITS is a very general data format that is used for many different types of astronomical data sets, so these packages are not

We are here demonstrating the functionality of the `astropy.io.fits` package which provides access to FITS files.

Loading Astronomical Fits Files in Python

To open fits files, we use `astropy.io.fits.open()`:

```
# Example: open an existing FITS file
from astropy.io import fits
fits_image_filename = 'my_fitsfile.fits'
hdul = fits.open(fits_image_filename)
```

The `open()` function takes either the relative or absolute file path. In addition, the function has several optional arguments which will be discussed later. The `open()` function returns an object called an HDUList which is a list-like collection of HDU objects. An HDU (Header Data Unit) is the highest level component of the FITS file structure, consisting of a header and (typically) a data array or table.

`hdul[0]` is the primary HDU, `hdul[1]` is the first extension HDU, etc. (if there are any extensions), and so on. It should be noted that astropy uses zero-based indexing when referring to HDUs and header cards, whereas the FITS standard uses one-based indexing.

Loading Astronomical Fits Files in Python

The HDUList has a useful method `HDUList.info()`, which summarizes the content of the opened FITS file:

```
hdul.info()
Filename: ...test0.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU     138   ()
  1  SCI           1 ImageHDU        61   (40, 40)   int16
  2  SCI           2 ImageHDU        61   (40, 40)   int16
  3  SCI           3 ImageHDU        61   (40, 40)   int16
  4  SCI           4 ImageHDU        61   (40, 40)   int16
```

Loading Astronomical Fits Files in Python

After you are done with the opened file, close it with the `HDUList.close()` method:

```
hdul.close()
```

Loading Astronomical Fits Files in Python

After you are done with the opened file, close it with the `HDUList.close()` method:

```
hdul.close()
```

You can avoid closing the file manually by using `open()` as context manager, just like we saw it previously for text files:

```
with fits.open(fits_image_filename) as hdul:
```

After exiting the with scope the file will be closed automatically. That is (generally) the preferred way to open a file in Python, because it will close the file even if an exception happens.

Loading Astronomical Fits Files in Python

So far we have accessed FITS files locally available on a computer. In addition to that, we can access `remote and cloud-hosted files`.

The `open()` function supports a `use_fsspec` argument which allows file paths to be opened using `fsspec`. The `fsspec` package supports a range of remote and distributed storage backends such as Amazon and Google Cloud Storage. For example, you can access a Hubble Space Telescope image located in the Hubble's public Amazon S3 bucket as follows:

```
# Location of a large Hubble archive image in Amazon S3 (213 MB)
url = "s3://stpubdata/hst/public/j8pu/j8pu0y010/j8pu0y010_drc.fits"

# Extract a 10-by-20 pixel cutout image
with fits.open(url, use_fsspec=True, fsspec_kwargs={"anon": True})
as hdul:

    cutout = hdul[1].section[10:20, 30:50]
```

Loading Astronomical Fits Files in Python

We have seen how to open FITS files. This is the more typical use case you will encounter.
Packages such as `astropy` also allow for creating FITS files.
The typical use case for creating FITS files is distributing and publishing astronomical catalogs.

A **complete documentation** of astropy's FITS functionality, including on how to **create FITS files**, can be found online:

`https://docs.astropy.org/en/stable/io/fits/`

Functional Programming

We have already used `functions`:

When we import a library like `numpy` and then call e.g. `np.sin()`, we are using a function someone else wrote and included within the `numpy` library.
We will now see how we can **write** these functions ourselves, thus creatin **user-defined functions**.

Functional Programming

There are two main reasons to use functions:

Code Readability: Functions break up code into individually testable, easily-debuggable pieces.

Code Reusability: Functions enable us to reuse code in a convenient way.

Functional Programming

There are two main reasons to use functions:

Code Readability: Functions break up code into individually testable, easily-debuggable pieces.

Code Reusability: Functions enable us to reuse code in a convenient way.

An **example** that illustrates this:

Imagine a **data extraction pipeline** for producing a 1D spectrum of galaxies from 2D CCD image data stored in multiple of FITS files.
One would want to write a function to read fits files into your code and perform the data reduction (such as sky subtractions), then a function to identify the spectral orders in your images and one to extract (collapse) the 2D data into a single dimension.

Writing Functions

In Python, the **function declaration** syntax is as follows:

```
def function_name(arguments):
    # function body

    return
```

Here, we have te following elements:

- `def` keyword used to declare a function
- `function_name` any name given to the function
- `arguments` any value passed to function
- `return` optional, returns value from a function

A Python function may or may not **return a value**.
Note: The return statement also denotes that the function has ended. Any code after the return statement is not executed.

Writing Functions

Let's now see an example.

```
def load_img(fname):
    with fits.open(fname) as hdulist:
    img = hdulist[0].data
    return img
```

This function to load a file assumes the following:

- the file we load is a FITS file
- the library was already imported with `import astropy.io.fits as fits`
- within the FITS file, the science image is located in the first extension.

Writing Functions

Let's now see an example.

```
def load_img(fname):
    with fits.open(fname) as hdulist:
    img = hdulist[0].data
    return img
```

This function to load a file assumes the following:

- the file we load is a FITS file
- the library was already imported with `import astropy.io.fits as fits`
- within the FITS file, the science image is located in the first extension.

This code isn't very flexible but might make sense given the above criteria hold.

Writing Functions


```
def load_img(fname):
    with fits.open(fname) as hdulist:
      img = hdulist[0].data
      return img
```

Using this example, we can go through the elements of a function declaration introduced above in more detail.

The function declaration starts with a `def` keyword, followed by a space and then the function name. Next we have a parenthesis set, containing the names of the arguments for using the function. The argument names are our choice as long as we are consistent about their use within the function. These names only matter within the function and are discarded after (more on this later on). Finally, we have a colon, and an indented block comprising the code we want to run when the function is called, followed by a return statement which specifies what the function returns to our global code when it's done.

Writing Functions

We can add more flexibility to the above function:

```
def load_img(fname,extension=0):
  with fits.open(fname) as hdulist:
    img = hdulist[extension].data
    return img
```

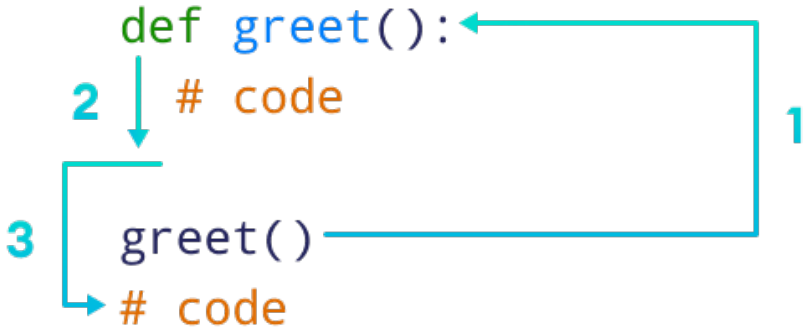
Now in addition to a filename `fname`, an optional extension `extension` can be given. The default value of `extension` is 0.

Now, to use this function, we need to call it.
Here's how we can call the `load_img()` function in Python:

```
load_img('imagefile.fits',1)
load_img('survey.fits',0)
```

Writing Functions

When a function is called, the control of the program goes to the function definition. All code inside the function is executed. The control of the program jumps to the next statement after the function call.



An important aspect of functions is that variables defined and used within a function are what is known as **local in scope**: those variables are created when the function is called and destroyed once the output is returned, so those values are not retrievable outside the function.

In contrast, **global variables** (defined outside of functions) are accessible from within the function, regardless of them being an argument or not. However, relying on this is considered **bad programming style**. Instead, a function should only depend on the variables listed as arguments, so the function could be moved to any other code without breaking it.

Variable Scope in Functions

As an example, we examine the following block of code:

```
constant = 5

def load_img(fname,extension=0):
     with fits.open(fname) as hdulist:
          img = hdulist[extension].data + constant
          return img
```

What would happen if we ran the above block?

Variable Scope in Functions

As an example, we examine the following block of code:

```
constant = 5

def load_img(fname,extension=0):
     with fits.open(fname) as hdulist:
          img = hdulist[extension].data + constant
          return img
```

What would happen if we ran the above block?

The function would have no problem adding the value of `constant` to our image, since it is defined globally in our code (outside the function). But if I copied and pasted this function into a different script, which didn't have a `constant` variable defined, I'd get an error. Even scarier, if I did have one defined but for a completely different reason in a different part of the code, I'd never know what happens.

Documentation

One important way to ensure we are doing all this correctly, as well as making things easier on ourselves later and on anyone else who may use our function is **good documentation**.

This is slightly different than commenting our code to describe what's going on (though we should do this).

Documentation is a **built-in feature** of the way Python does functions. The way to set documentation for a function is to place it inside triple quotes right at the top of our function definition, as follows:

Documentation


```
def load_img(fname,extension=0):
'''A function to quickly extract the data extension of a fits file
INPUTs:
fname (string): path/file name of the fits file to be loaded
extension (int) [default: 0]: extension to index
RETURNS:
img (array_like): data attribute queried'''

with fits.open(fname) as hdulist:
     img = hdulist[extension].data
     return img
```

Once having set documentation this way, you can run the command

```
help(load_img)
```

from Python/iPython to show the function's documentation.

Function Arguments

Setting optional arguments, args, and kwargs Optional Arguments
Functions can take **optional arguments**. They must come after the non-default variables and they must have a default value.
We can set arguments to be optional by setting the arguments of the `def` call equal to some default values. An example:

```
def somefunction(var1,var2,var3=1,var4='cat'):
     output = str(var1+var2+var3) + var4
     return output
```

Function Arguments

So far the number of arguments, despite some might be optional, was fixed: We specified a numer of inputs. We can use the `*args` and `**kwargs` (keyword arguments) commands to create variable numbers of arguments of a function. An example:

```
def test_function(farg,*args):
     print 'formal argument: ', farg
     print ' length of args: ', len(args)
     for arg in args:
          print 'new arg:', arg
```

Function Arguments

So far the number of arguments, despite some might be optional, was fixed: We specified a numer of inputs. We can use the `*args` and `**kwargs` (keyword arguments) commands to create variable numbers of arguments of a function. An example:

```
def test_function(farg,*args):
     print 'formal argument: ', farg
     print ' length of args: ', len(args)
     for arg in args:
          print 'new arg:', arg
```

What happens here is the following:
The formal argument `farg` is read in like a normal argument. We could have any number of these. But we've specified the last argument as `*args`, which tells Python to expect some uknown number of inputs after this. Within the function, one then can iterate through the list of extra inputs to do things with them individually.

Function Arguments

`**kwargs` (keyword arguments) are very similar to `*args` in the way that they let you pass a variable number of extra variables to the function. The difference is, when you feed those extra arguments into the function, you individually give each a new keyword by setting it equal to it in the function call. Then, instead of putting all the extra arguments into a list, they are put into a dictionary where each value is linked to the key and can be accessed via dictionary style slicing. An example:

```
def kwarg_examplefunction(farg, **kwargs):
     print 'formal argument: ', farg
     for key in kwargs:
          print 'argument: ', kwargs[key]
```

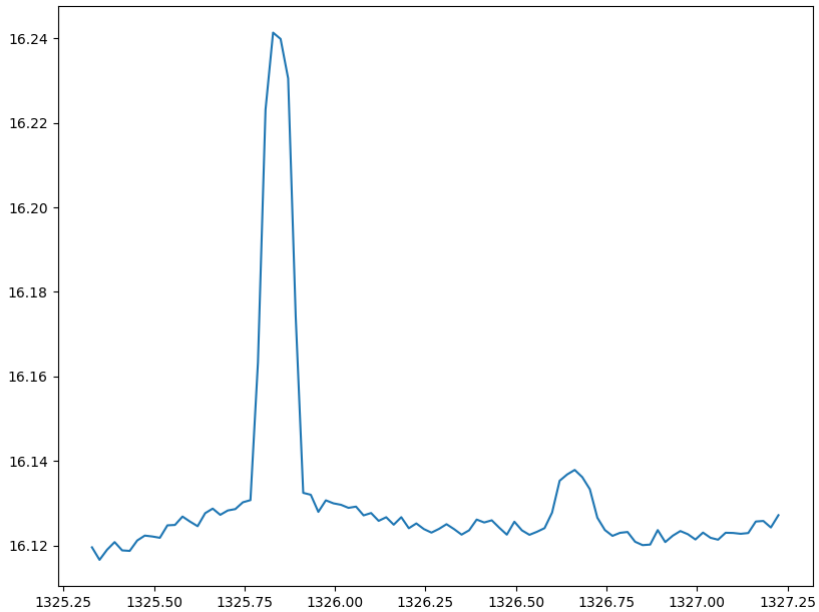
Plotting

Let's start with the basics. Say we have an **independent variable** (like time), and a **measured (dependent) variable** (like position). This type of data could easily be read in from a 2-column text file and then plotted against each other, plotting e.g. a **light curve**.

```
# example: Plotting x vs. y
import matplotlib.pyplot as plt
import numpy as np

file_name = '/research/timeseries.txt'
data = np.loadtxt(file_name)
times = data[:,0]
positions = data[:,1]

# Now let's plot the data
plt.plot(times,positions)
plt.show()
```

Plotting

We get the following plot:



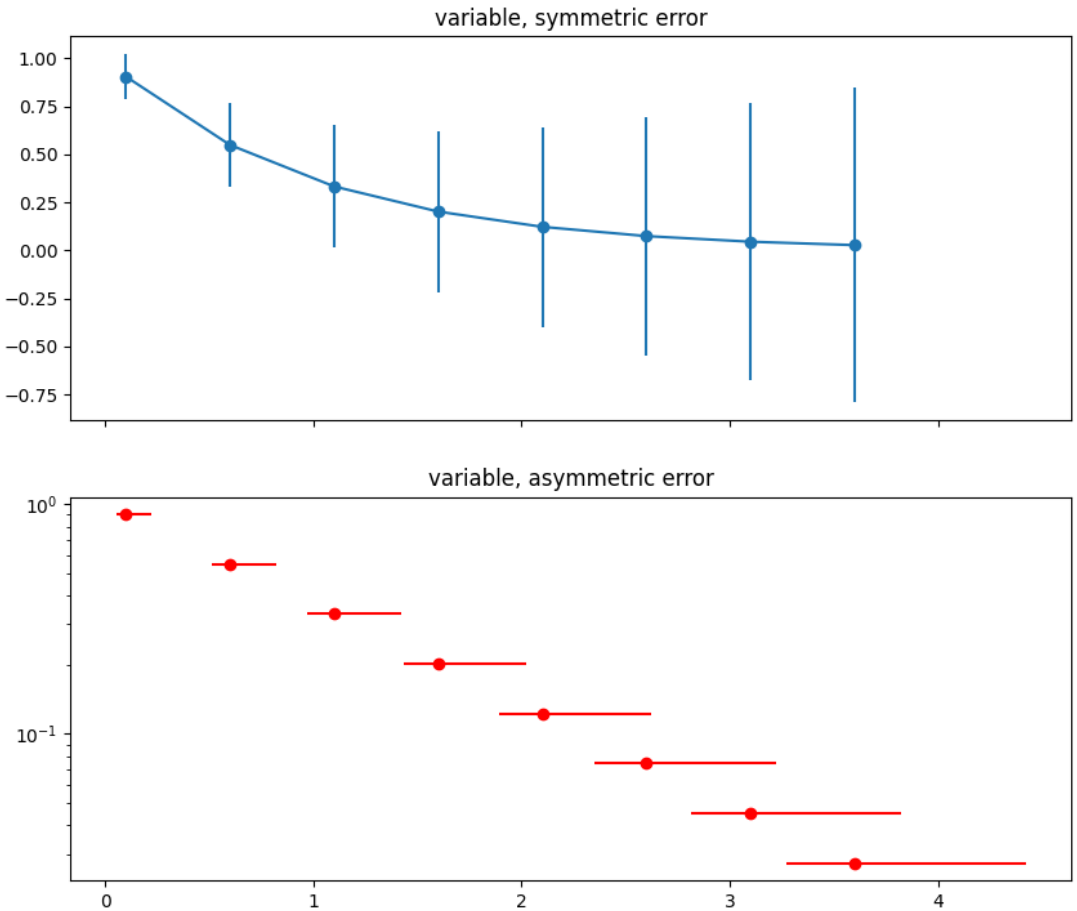
When executing the code, you notice that the default way Python plots is by plotting the data connected with blue lines. We however want to plot the individual data points.

In the following we will see an example that plots measurements along with their errors, and plots everything in individual data points:

Plotting


```
# example: Plotting x vs. y with error bars
import matplotlib.pyplot as plt
import numpy as np

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)
# example error bar values that vary with x-position
error = 0.1 + 0.2 * x

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True)
ax0.errorbar(x, y, yerr=error, fmt='-o')
ax0.set_title('variable, symmetric error')

# error bar values w/ different -/+ errors
lower_error = 0.4 * error
upper_error = error
asymmetric_error = [lower_error, upper_error]

ax1.errorbar(x, y, xerr=asymmetric_error, fmt='ro')
ax1.set_title('variable, asymmetric error')
ax1.set_yscale('log')
plt.show()
```

Plotting



Plotting

When using `plt.plot` and similar such as `errorbar` or `scatter`, there are optional settings you can specify, mainly color and line/ point style.
If you choose a discrete symbol (like `'o'` for circles or `'+'` for plusses), then Python won't connect them automatically.
You can use a matplotlib shortcut to simultaniously choose a color and linestyle as follows:

```
plt.plot(times, positions, 'ro')
```

This would plot the discrete points as red circles.
You can also specify the size of the symbol by including the argument `ms`, e.g. `ms=10`

More on this can be found in the `matplotlib` documentation.

Plotting

So far we have displayed plots by using simply a

```
plt.show()
```

at the end. In many cases it is better to save plots, and doing so as a vectorized `*.pdf` file.
For this purpose, the plotting code needs to be modified as follows:

```
import matplotlib
matplotlib.use('PDF')
import matplotlib.pyplot as plt

# here comes your code to create the plot

plt.savefig('plot.pdf)
```

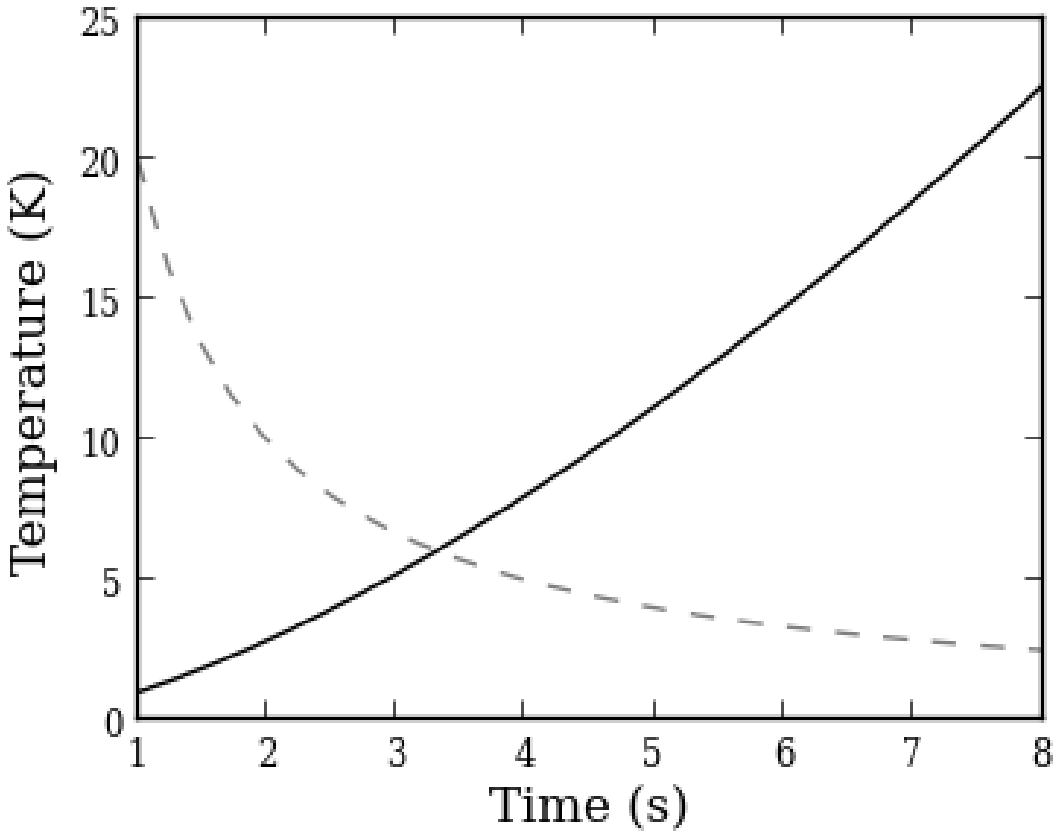
Lines and Markers

A matplotlib *line* is a object containing a set of points and various attributes describing how to draw those points. Optionally, the points are drawn with *markers* and the connections between points with various *styles* of line (including no connecting line at all). See the following example:

```
# Example: plotting with different styles

import matplotlib.pyplot as plt
import numpy as np

plt.rc('font', family='serif')
plt.rc('xtick', labelsize='x-small')
plt.rc('ytick', labelsize='x-small')

fig = plt.figure(figsize=(4, 3))

x = np.linspace(1., 8., 30)
plt.plot(x, x ** 1.5, color='k', ls='solid')
plt.plot(x, 20/x, color='0.50', ls='dashed')
plt.xlabel('Time (s)')
plt.ylabel('Temperature (K)')
plt.show()
```

Lines and Markers

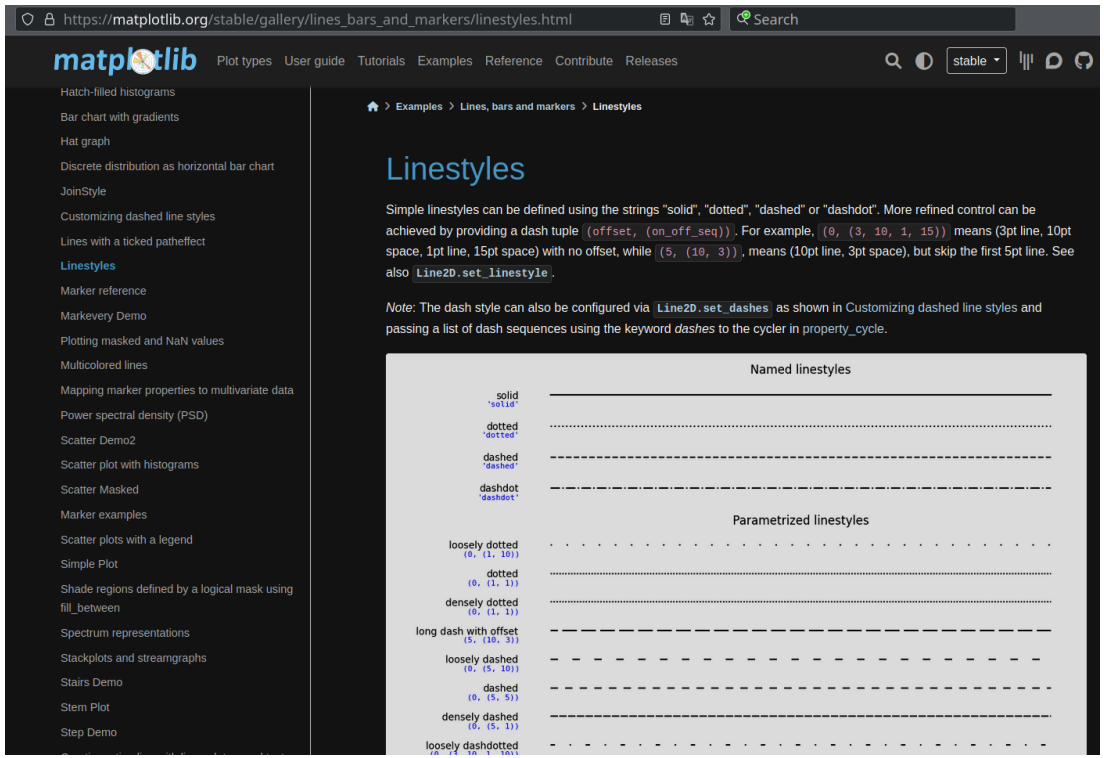
The resulting plot is the following:



Lines and Markers

Matplotlib offers a huge range of line and marker styles.
As usual with Python, everything is well documented:

`https://matplotlib.org/stable/api/markers_api.html`
`https://matplotlib.org/stable/gallery/lines_bars_and_markers/linestyles.html`

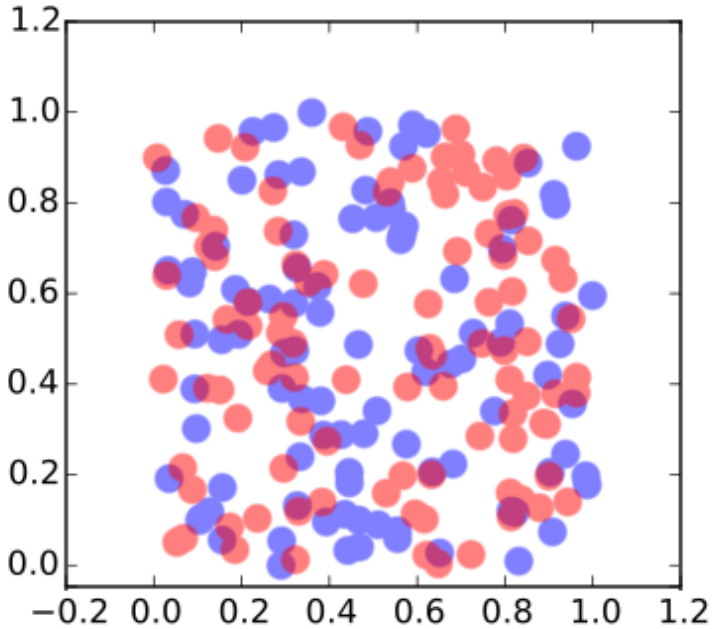


Colors

Matplotlib also offers to **customize colors**. A range of named colors is available, but it is also possible to specify colors as e.g. RGB. In addition, colors can have an *alpha value* to specify its **transparency**, where 0 is fully transparent and 1 is fully opaque.

How to do so can be found in the documentation:

`https://matplotlib.org/stable/users/explain/colors/colors.html`
`https://matplotlib.org/stable/gallery/color/named_colors.html`



Histograms with matplotlib

A **histogram** is a graphical representation of numerical data distribution. The x axis represents the bin ranges while the y axis gives information about frequency. Python's Matplotlib Library provides us with an easy way to create histograms.

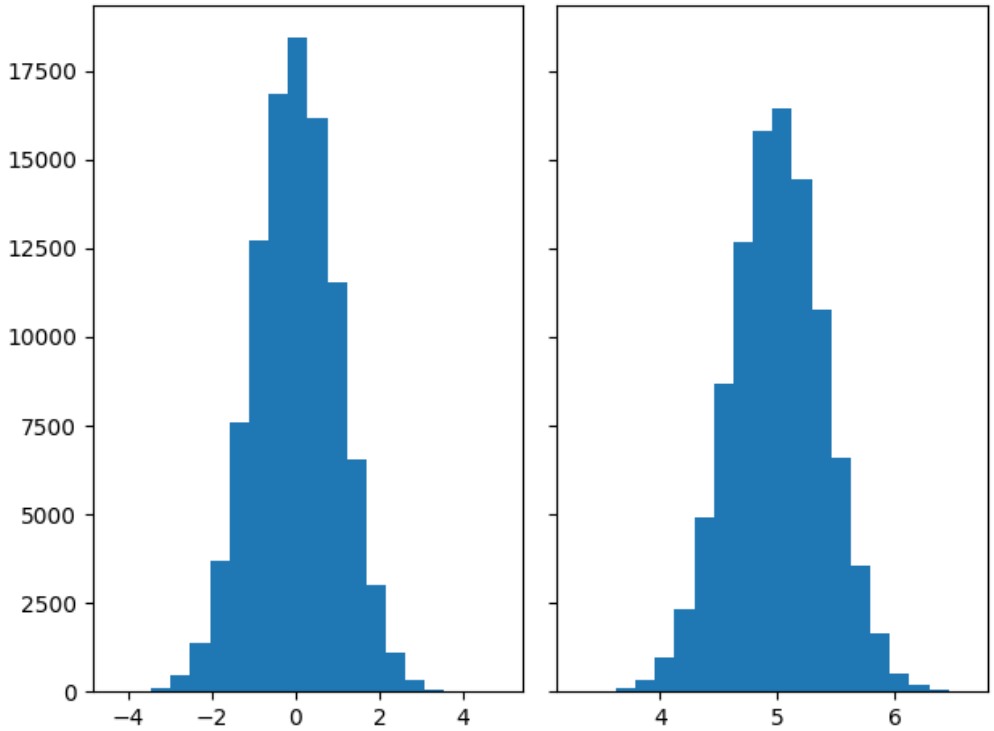
To generate a 1D histogram we only need a single vector of numbers. In the following example, we generate both test data and histograms.

Histograms with matplotlib


```
#example: Generate data and plot a simple histogram

import matplotlib.pyplot as plt
import numpy as np

N_points = 100000
n_bins = 20

# pseudo-random generator
rng = np.random.default_rng()

# Generate two normal distributions
dist1 = rng.standard_normal(N_points)
dist2 = 0.4 * rng.standard_normal(N_points) + 5

fig, axs = plt.subplots(1, 2, sharey=True, tight_layout=True)

# We can set the number of bins with the *bins* keyword argument.
axs[0].hist(dist1, bins=n_bins)
axs[1].hist(dist2, bins=n_bins)

plt.show()
```

Plotting 3D Data

The resulting plot is the following:



You can find more on this in the documentation:

`https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html`

Figures and Axes

When running `plot()` above, matplotlib actually generated two distinct entities needed: a **figure**, and at least one or multiple **axes**. Think of the axes as what is actually being plotted (the lines we see there), and the figure as the "canvas" behind it in the background that the axes sit on.

(Those axes should not be confused with coordinate axes.)

If you wanted to adjust e.g. the plot limits, that would be an axes action; in contrast, changing width and/or height of the overall image file (*.png, *.pdf) would be a figure command.

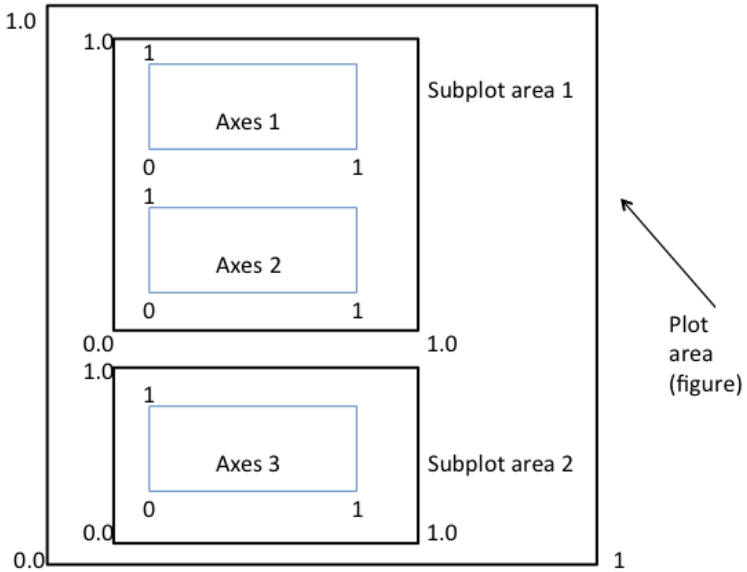
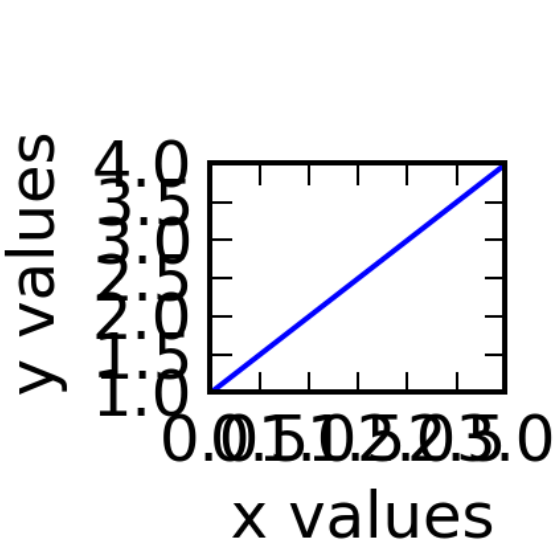


Figure Size

In many cases we want to **explicitly set the figure size**.
Doing so ensures that label fonts will be the right size compared to the content of the figure, and the axis having the right aspect ratio.
Setting the figure size explicitly also ensures that if the defaults change, your plot will not.



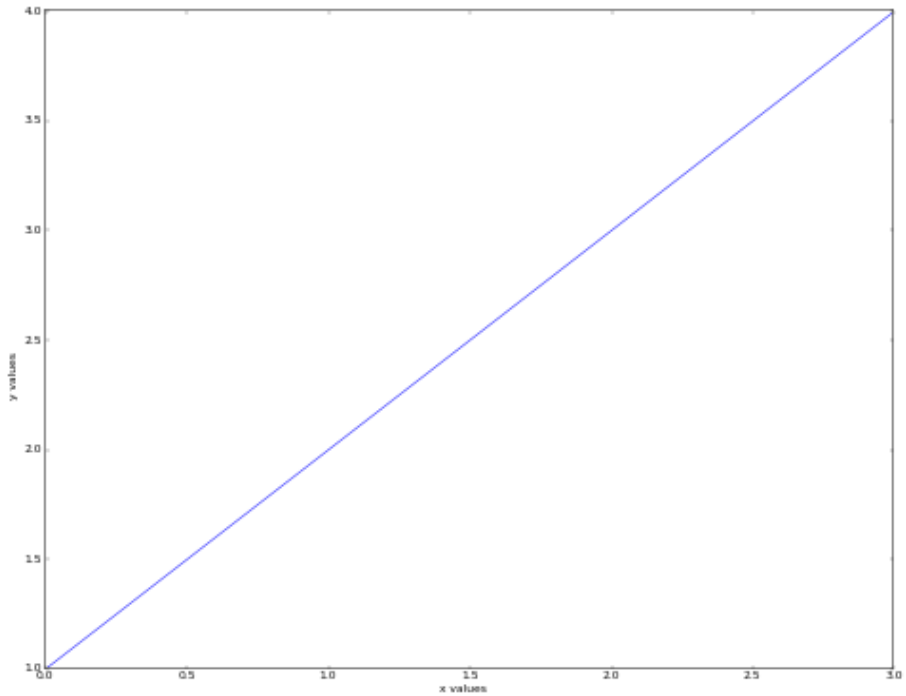


Fig: Two plots where the figure size is not optimal.

Figure Size

The following plot has the right size for half-page width plots:

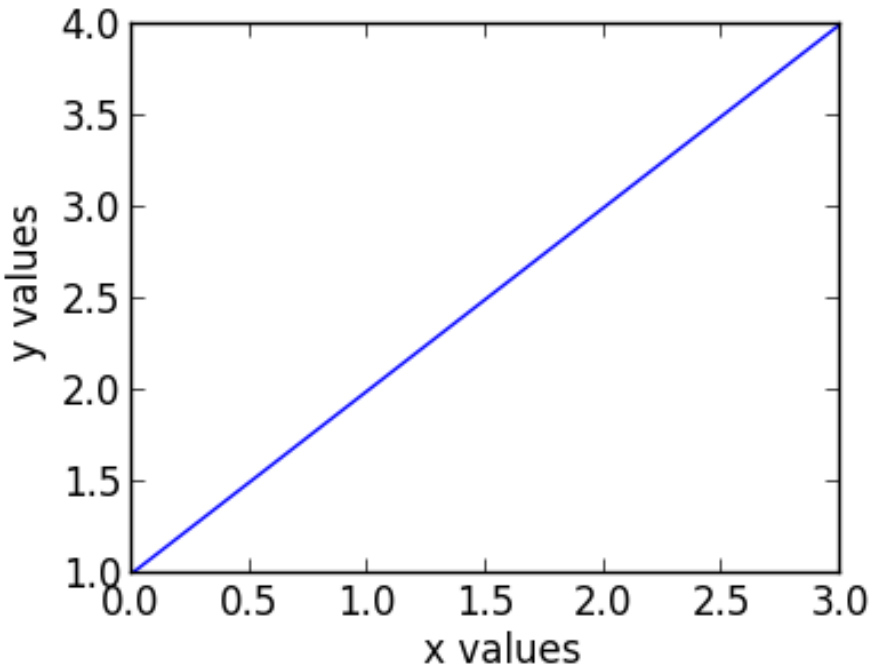
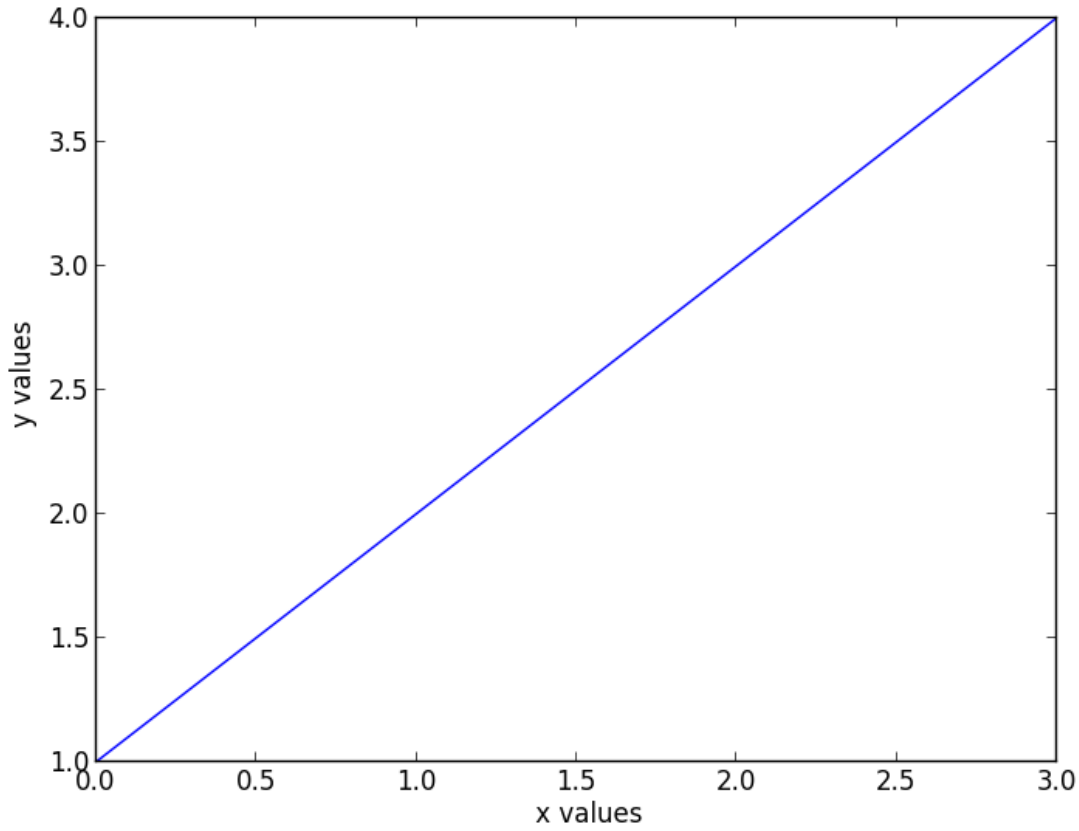


Figure Size

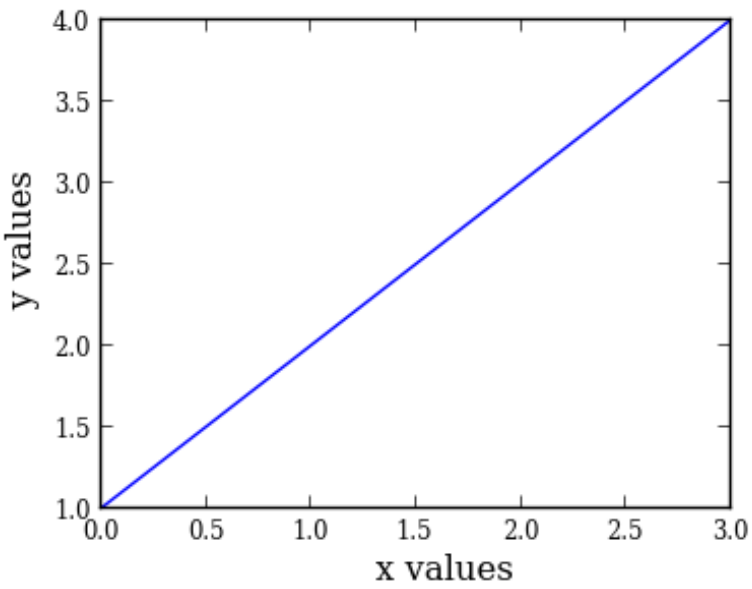
... and for full-page width plots:



Font Size

In the following we see how to chose a different font size/ style for the axis labels and the tick labels (by default these are the same, and are set to medium). The tick label size can often be reduced compared to the default:

```
plt.rc('font', family='serif')
plt.rc('xtick', labelsize='x-small')
plt.rc('ytick', labelsize='x-small')

fig = plt.figure(figsize=(4, 3))
ax = fig.add_subplot(1, 1, 1)
plt.plot([1, 2, 3, 4])
ax.set_xlabel('The x values')
ax.set_ylabel('The y values')
```



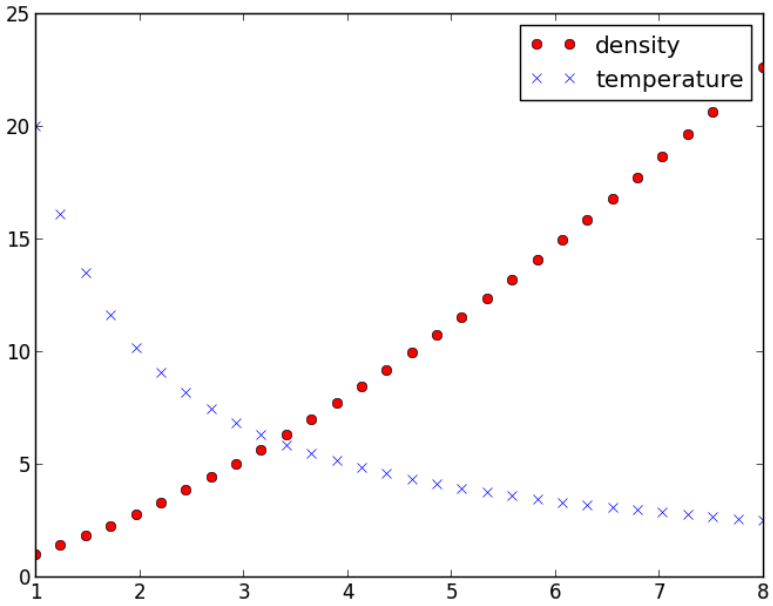
Legends

Legends and color bars are essential parts of many plots.
Adding a **legend** to a plot is straightforward. First, whenever calling a plotting routine for which you want the results included in the legend, add the `label=` argument. Then call the `legend` method:

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
x = np.linspace(1., 8., 30)
ax.plot(x, x ** 1.5, 'ro', label='density')
ax.plot(x, 20/x, 'bx', label='temperature')

ax.legend() #call the legend method
```

Legends

This is the resulting plot:

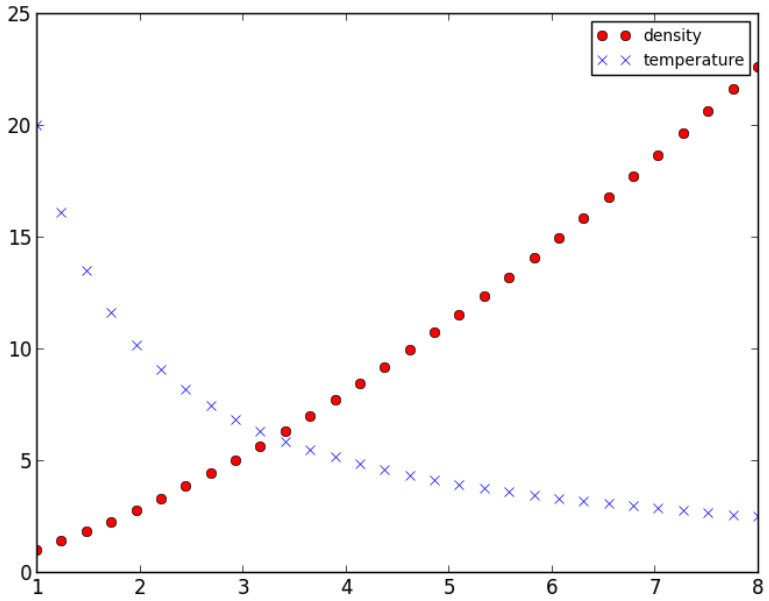


Legends

Note that you can control the font size (and other properties) in a legend with the following `rc` parameter:

```
plt.rc('legend', fontsize='small')
```

This produces the following plot:

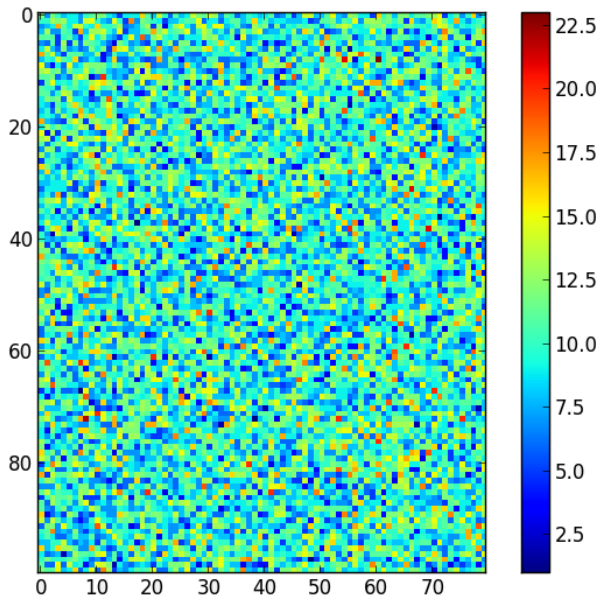


Colorbars

Adding a **colorbar** involves capturing the handle to the `imshow` object:

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
image = np.random.poisson(10., (100, 80))
i = ax.imshow(image, interpolation='nearest')
fig.colorbar(i)
```

This is the resulting plot:



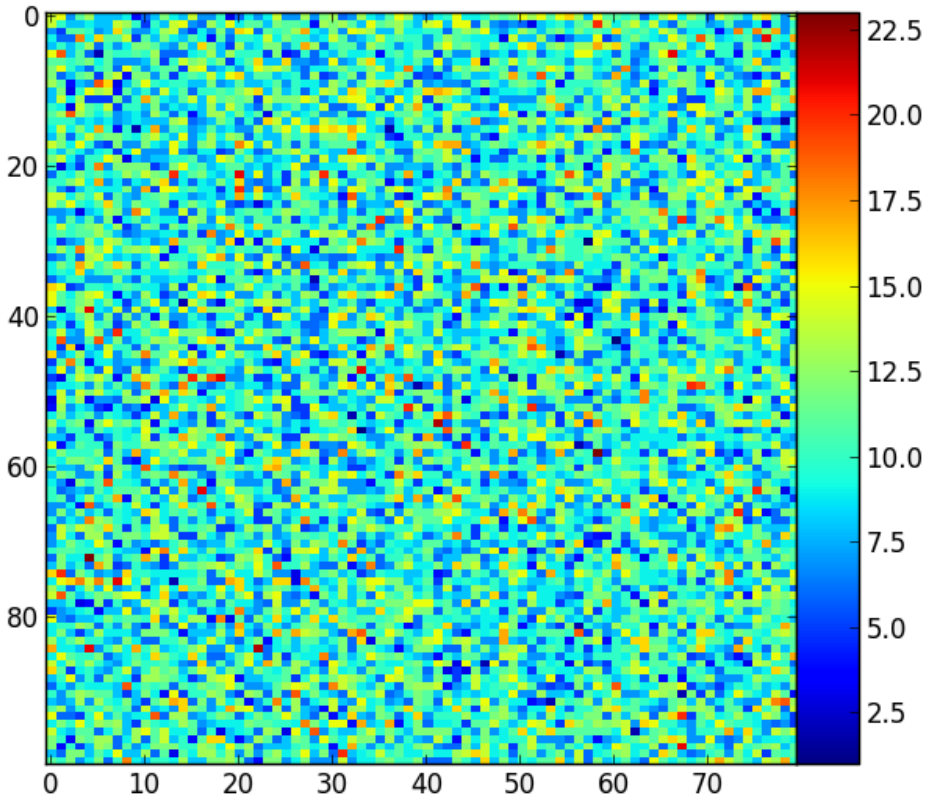
Colorbars

Note that in the above example, the colorbar box automatically takes space from the axes to which it is attached. If you want to **customize exactly** where the colorbar appears, you can define a set of axes, and pass it to colorbar using the `cax=` argument and setting `aspect='auto'`:

```
fig = plt.figure()
ax = fig.add_axes([0.1,0.1,0.6,0.8])
image = np.random.poisson(10., (100, 80))
i = ax.imshow(image, aspect='auto', interpolation='nearest')

colorbar_ax = fig.add_axes([0.7, 0.1, 0.05, 0.8])
fig.colorbar(i, cax=colorbar\_ax)
```

Colorbars

This is the resulting plot:



Annotating Text

In many cases it makes sense to add **annoations** to a plot to highlight specific parts of a plot.

Matplotlib offers convenient ways on how to annotate plots, with and without arrows or lines.

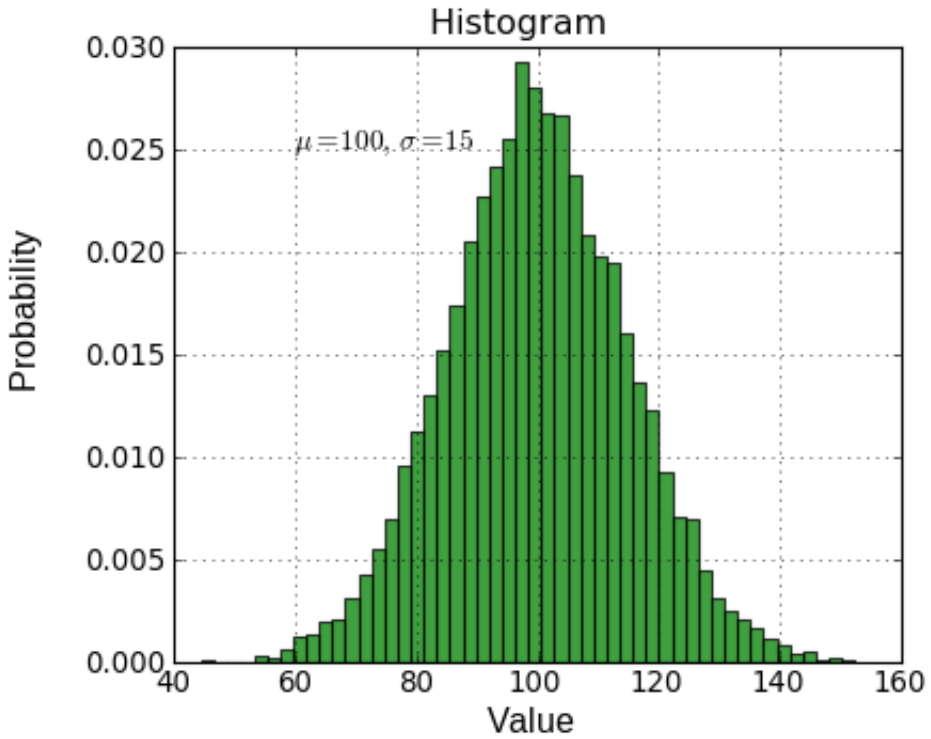
The `text()` command can be used to add text in an arbitrary location, and the `xlabel()`, `ylabel()` and `title()` are used to add text in the indicated locations. See the following example where we annotate a histogram:

Annotating Text


```
mu, sigma = 100, 15
x = np.random.normal(mu, sigma, size=10000)
plt.clf()

# the histogram of the data
histvals, binvals, patches =
   plt.hist(x, bins=50, normed=True, facecolor='g', alpha=0.75)

plt.xlabel('Value')
plt.ylabel('Probability')
plt.title('Histogram')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
```

Colorbars

This is the resulting plot:



Annotating Text

All of the `text()` commands return a `matplotlib.text.Text` instance. Just as with with lines above, you can customize the properties by passing keyword arguments into the text functions or using `setp()`:

```
t = plt.xlabel('my data', fontsize=14, color='red')
```

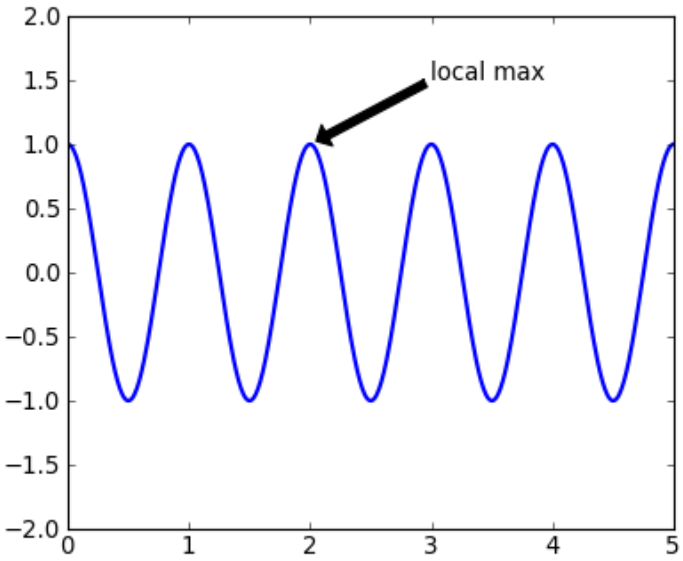
Annotating Text

The basic `text()` command above places text at an arbitrary position on the axes.
We can, however, also **add a line or arrow** to annotate a specific feature of the plot using `arrowprops`:

```
plt.clf()
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*pi*t)
lines = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05))
plt.ylim(-2,2)
```

Colorbars

This is the resulting plot:



Symbols and Equations with LaTeX

When using symbols and equations in a plot we want to make sure they look the same as in the context, such as text, the plot is shown.
For this purpose, matplotlib accepts LaTeX expressions in any text expression. For example to write the expression $\sigma_i = 15$ in the title:

```
plt.title(r'$\sigma_i=15$')
```

Symbols and Equations with LaTeX

When using symbols and equations in a plot we want to make sure they look the same as in the context, such as text, the plot is shown.
For this purpose, matplotlib accepts LaTeX expressions in any text expression. For example to write the expression $\sigma_i = 15$ in the title:

```
plt.title(r'$\sigma_i=15$')
```

The `r` signifies that a raw string follows so backslashes are not interpreted as Python escapes. Matplotlib comes with built-in LaTeX to allow symbols across platforms. When LaTeX is installed, it is possible to use this LaTeX installation by setting one of the `rc`. See the following example:

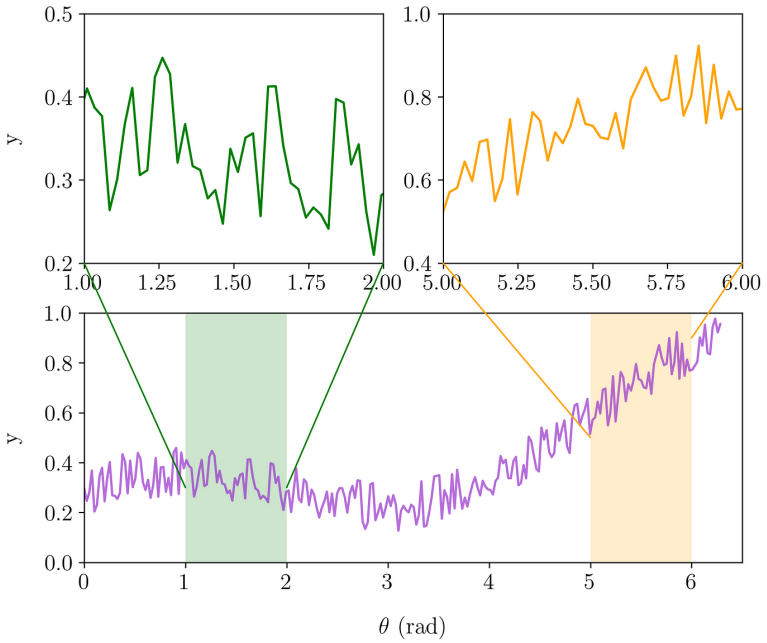
```
# put this line before any text output
plt.rc('text', usetex=True)
```

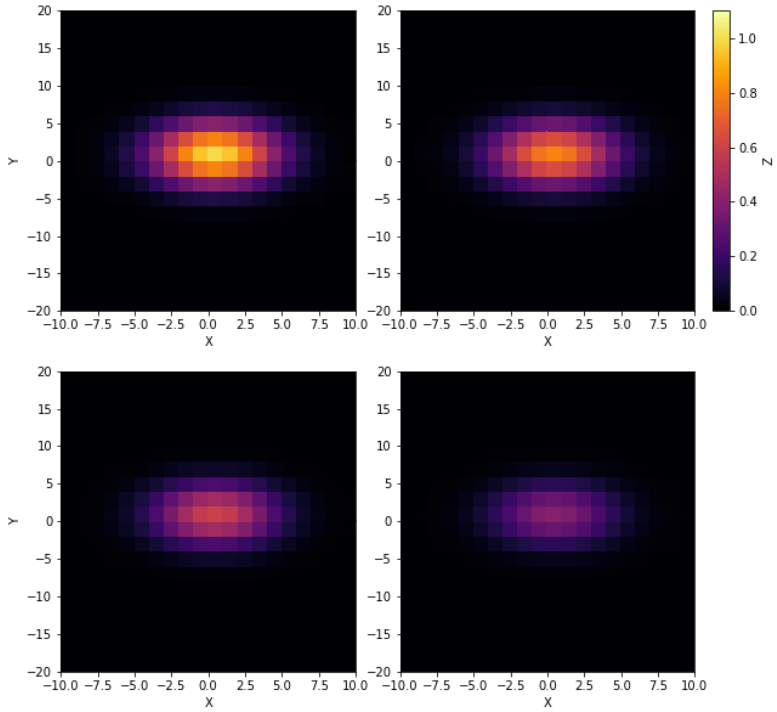
More information on this can be found in the `mathtext` and `usetex` documentation.

Creating Subplots

Subplots are a way of putting multiple axes objects within one figure object, allowing you to make multipanel plots.

Mathplotlib allows for many ways to control the appearance of such plots, like in the following examples:





Axes

We have already seen how to make axes in matplotlib. We **customize** them now in order to create subplots.

The easiest way to make a set of axes in a matplotlib figure is to use the `subplot` command which biggest advantage is that it allows users to easily handle multiple figures/axes without getting confused as to which one is currently active. For example:

```
fig1 = plt.figure()
fig2 = plt.figure()
ax1 = fig1.add_subplot(1, 1, 1)
ax2 = fig2.add_subplot(2, 1, 1)
ax3 = fig2.add_subplot(2, 1, 2)
```

This defines two figures, one with two sets of axes, and one with one set of axes. We use `ax1.plot(...)` to plot to the subplot in `fig1`, and `ax2.plot(...)` and `ax3.plot(...)` to plot in the top and bottom subplots of `fig2` respectively.

Axes

We have already seen how to make axes in matplotlib. We **customize** them now in order to create subplots.

The easiest way to make a set of axes in a matplotlib figure is to use the `subplot` command which biggest advantage is that it allows users to easily handle multiple figures/axes without getting confused as to which one is currently active. For example:

```
fig1 = plt.figure()
fig2 = plt.figure()
ax1 = fig1.add_subplot(1, 1, 1)
ax2 = fig2.add_subplot(2, 1, 1)
ax3 = fig2.add_subplot(2, 1, 2)
```

The `add_subplot` arguments are the number of rows, columns, and the subplot ID, between 1 number of columns $\times$ number of rows.
Explicitly setting the figure size ensures that the fonts for the labels will be the right size compared to the content of the figure.

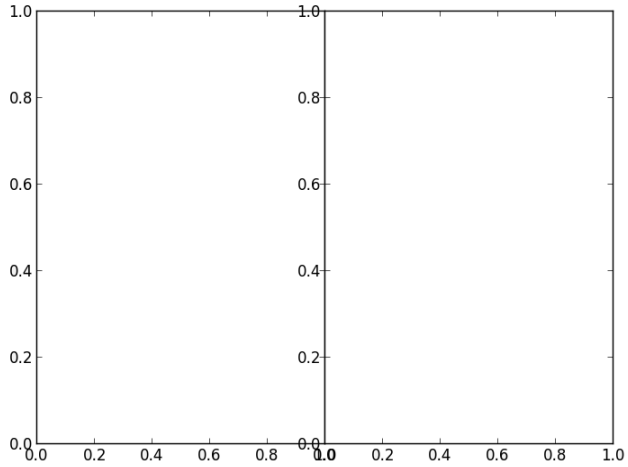
Axes and Subplots

While it is possible to **adjust the spacing** between the subplots using `subplots_adjust`, or use the `gridspec` functionality for more advanced subplotting, it is often easier to just use the more general `add_axes` method instead of `add_subplot`.

The `add_axes` method takes a list of four values, which are `xmin`, `ymin` (the coordinates of the lower left corner of the subplot), `dx`, and `dy` (the width and height of the subplot) for the subplot, where `xmin`.

All values are specified in relative units (where 0 is left/bottom and 1 is top/right). For example:

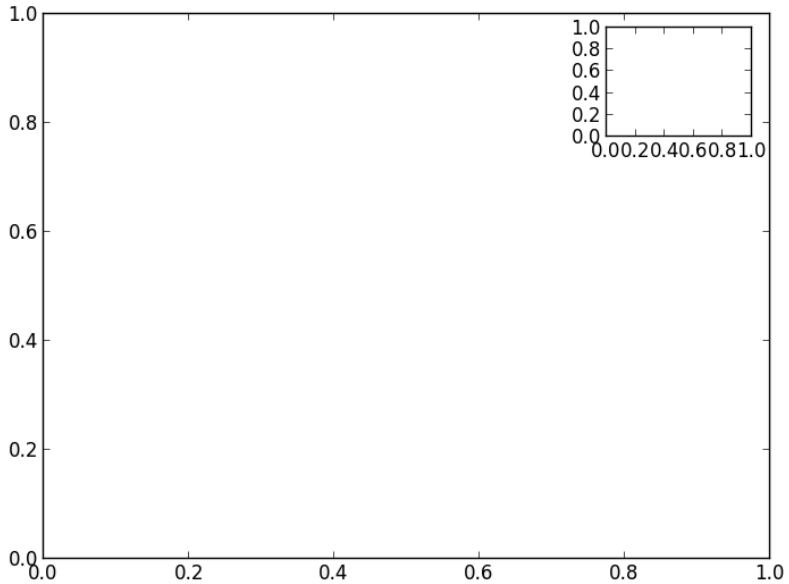
```
fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.1, 0.4, 0.8])
ax2 = fig.add_axes([0.5, 0.1, 0.4, 0.8])
```



Axes and Subplots

Note that is also allows us to easily make inset plots:

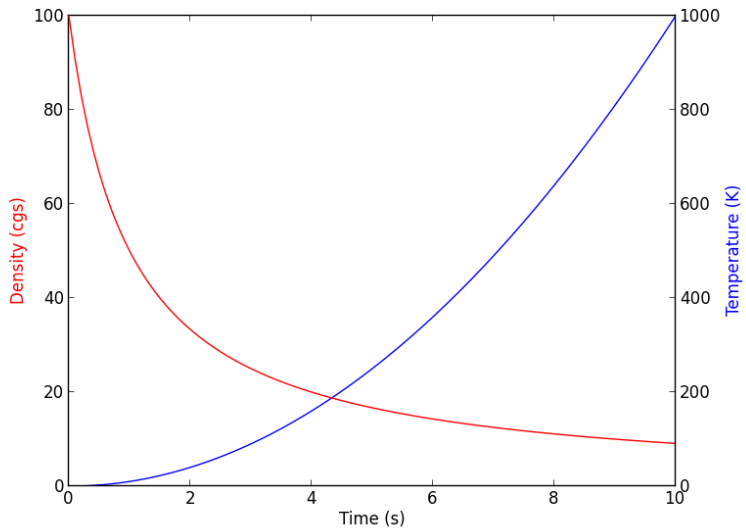
```
fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax2 = fig.add_axes([0.72, 0.72, 0.16, 0.16])
plt.show()
```



Axes and Subplots

In some cases, it can be desirable to show two different x axes (e.g. distance and redshift), or two different y axes (e.g. two different quantities such as density and temperature). Matplotlib provides an easy way to create **twin axes**. For example:

```
fig = plt.figure()
ax1 = fig.add_subplot(1, 1, 1)
ax2 = ax1.twinx()
t = np.linspace(0., 10., 100)
ax1.plot(t, t ** 2, 'b-')
ax2.plot(t, 1000 / (t + 1), 'r-')
ax1.set_ylabel('Density (cgs)', color='red')
ax2.set_ylabel('Temperature (K)', color='blue')
ax1.set_xlabel('Time (s)')
```



Similarly, `twiny` returns a second set of axes sharing the y-axis, but with a separate x-axis.

Plotting 2D Arrays

Earlier we discussed 2-D arrays. The easiest way to think about a 2-D array in terms of plotting is an image with colors assigned based on value. Each pixel is a value within the array. Some pixels might have low numbers (not bright), others higher (very bright).
(This makes much sense, since images taken by telescopes are simply 2D CCD pixel arrays counting how many photons hit each pixel and returning a 2-D array with the totals).

Plotting 2D Arrays

Earlier we discussed 2-D arrays. The easiest way to think about a 2-D array in terms of plotting is an image with colors assigned based on value. Each pixel is a value within the array. Some pixels might have low numbers (not bright), others higher (very bright).
(This makes much sense, since images taken by telescopes are simply 2D CCD pixel arrays counting how many photons hit each pixel and returning a 2-D array with the totals).

Lets say we used astropy to read in a fits image, and turn it into a 2-D array. Now we have a two dimensional numpy array, with `array[0,0]` giving the number of photons in the top-left pixel, and so forth. We can plot it like the following:

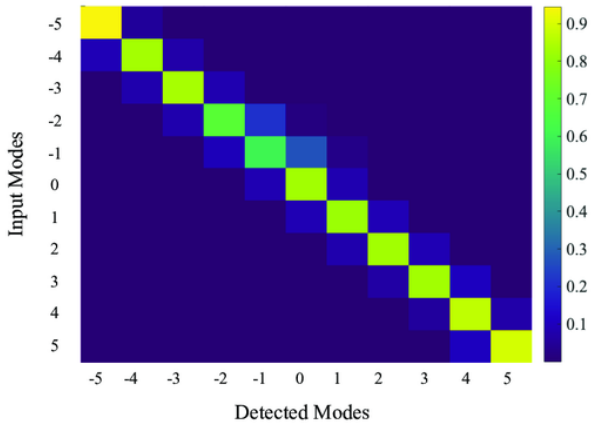
```
plt.imshow(array, cmap='gray_r')
plt.show()
```

In this example, we chose a `cmap (color map)` to be `gray_r`.
Note: You don't have to call `plt.figure()` before using `imshow`.

Plotting 2D Arrays

Typical usages of plots of 2D arrays are the following:

cross-correlation matrices

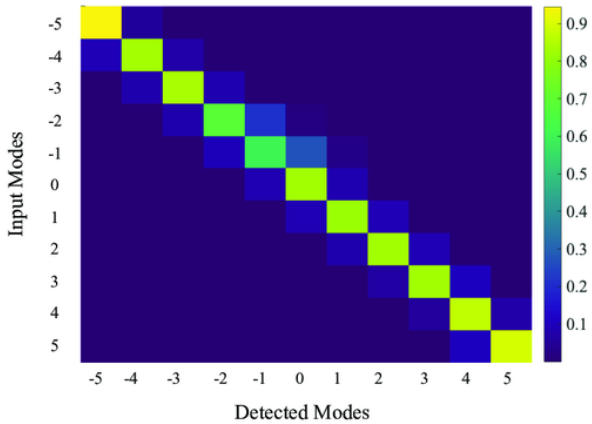


In this example, the fraction of detected modes vs. input modes is plotted, representing conditional probabilities.

Plotting 2D Arrays

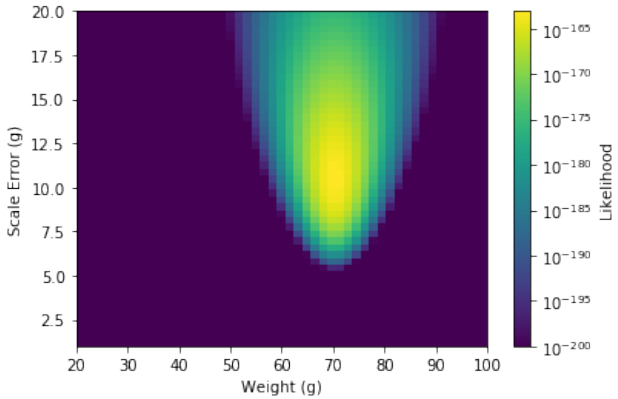
Typical usages of plots of 2D arrays are the following:

cross-correlation matrices



In this example, the fraction of detected modes vs. input modes is plotted, representing conditional probabilities.

heatmaps to show likelihoods



The likelihood is the joint probability of measured data as a function of model parameters. The arg max (over the parameters) of the likelihood function then serves as a point estimate for the most likely parameter vector. In this example, the likelihood of parameter value combinations for the parameters *scale error* and *weight* are plotted.

Plotting 2D Arrays

An important note regarding coordinate systems:

The convention amongst astronomers and scientists in general is the **origin of an image** is in the lower left hand corner.

Plotting 2D Arrays

An important note regarding coordinate systems:

The convention amongst astronomers and scientists in general is the **origin of an image** is in the lower left hand corner.

However, `imshow` plots data as a matrix, with (0,0) in the upper left corner.

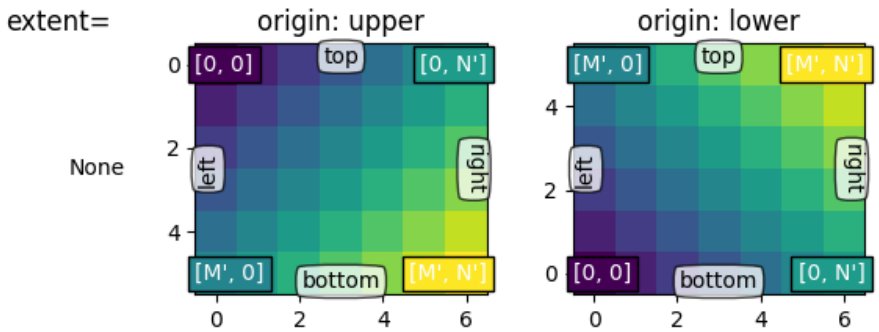
Plotting 2D Arrays

An important note regarding coordinate systems:

The convention amongst astronomers and scientists in general is the **origin of an image** is in the lower left hand corner.

However, `imshow` plots data as a matrix, with (0,0) in the upper left corner.

To plot with (0,0) in the bottom left, use `origin='lower'` within the `plt.imshow` command. This, however, will flip the image vertically. Some astronomical images are upside down anyway (convention of north being up and east being left). But if your image was rightside up when plotted before, flip the array before plotting: `image = image[::-1]`



more on this can be found here: `https://matplotlib.org/stable/`

Plotting 2D Histograms

With `imshow` we assumed to already have a 2D array (index for the spacial information, value compnent for the color information).

In many other cases, however, we want to **plot a 2D histogram**.

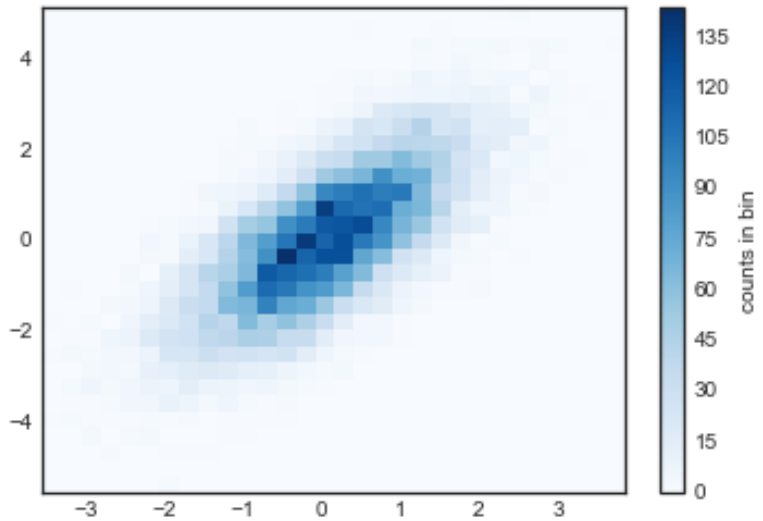


Figure: A two-dimensional histogram with `plt.hist2d`

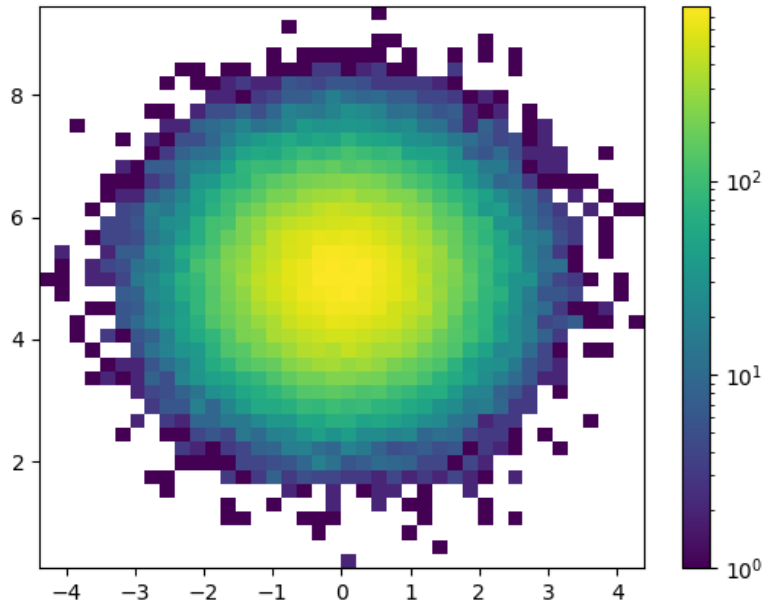
Plotting 2D Histograms

For how to do so, see the following example:

```
from matplotlib.colors import LogNorm
import matplotlib.pyplot as plt
import numpy as np

#generate normal distribution centered at x=0 and y=5
x = np.random.randn(100000)
y = np.random.randn(100000)+5

h = plt.hist2d(x, y, bins=40, norm=LogNorm())
plt.colorbar(h[3])
plt.show()
```

Plotting 2D Histograms

This produces the following plot of a 2D histogram with color bar:



Plotting 2D Histograms

Customizing a 2D histogram is similar to the 1D case - the bin size, normalization and other properties can be controlled.

```
# increase the number of bins on each axis
hist2d(dist1, dist2, bins=40)

# define normalization of the colors
hist2d(dist1, dist2, bins=40, norm=colors.LogNorm())

# define custom numbers of bins for each axis
hist2d(dist1, dist2, bins=(80, 10), norm=colors.LogNorm())
```

General Visualization Guidelines

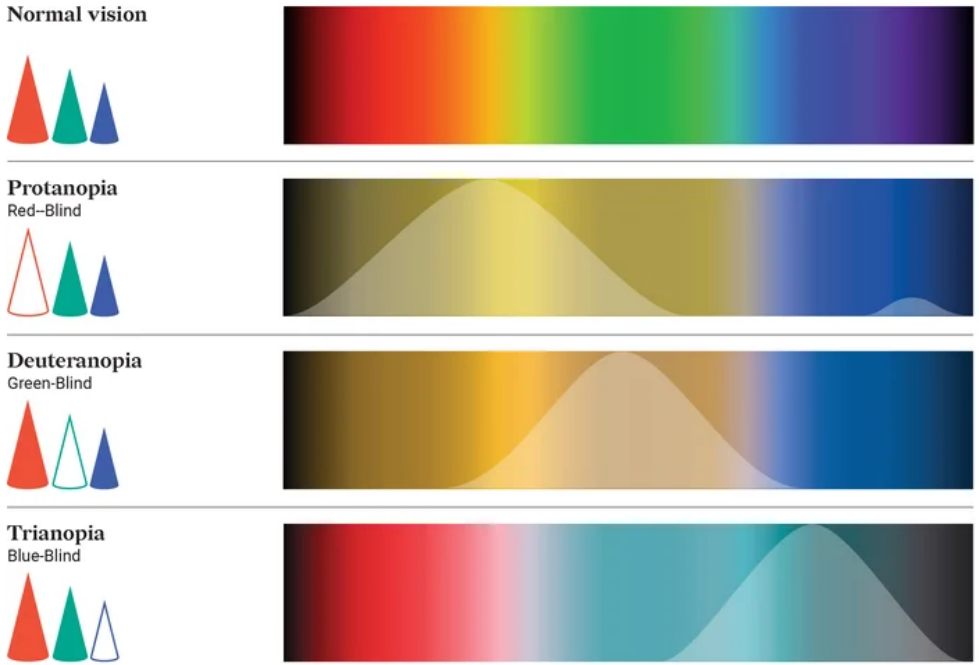
So far we have handled how to visualize data using Python libraries.

In addition to plotting, it is also important to **make plots easily readable**. We have already covered this to some extend when discussing figure and font sizes. Here, we take a look at more general visualization guidelines and how these can be implemented with Python.

General Visualization Guidelines

Color vision deficiencies are common.
There are around 300 million people in the world who are colorblind.
About 8% of men and 0.5% of women are colorblind, so making your chart color blind safe is a reasonable thing to do.

The most common form of color vision deficiency involves differentiating between red and green. However, avoiding color schemes with both red and green will not completely solving the problem:



General Visualization Guidelines

There are many information available on how to choose suitable color schemes for Python:

Matplotlib provides a range of different color maps:
`https://matplotlib.org/stable/users/explain/colors/colormaps.html`

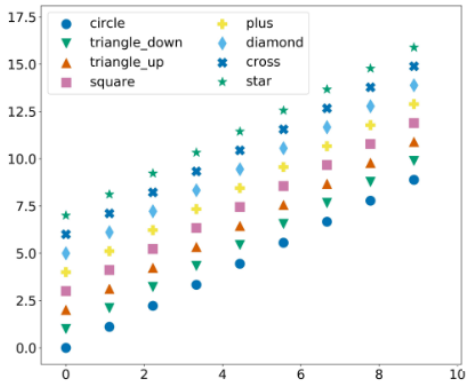
Colorblind is a Python computer vision library that converts images into a colorblind friendly version depending on the type of colorblindness. The three supported types of colorblindness/color weakness are:
`https://pypi.org/project/colorblind/`

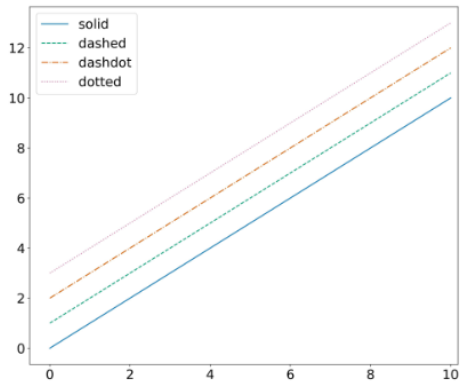
In addition, there are websites to check if a plot is understandable for everybody with online colour blindness simulators, like the one you can find at `www.color-blindness.com`.

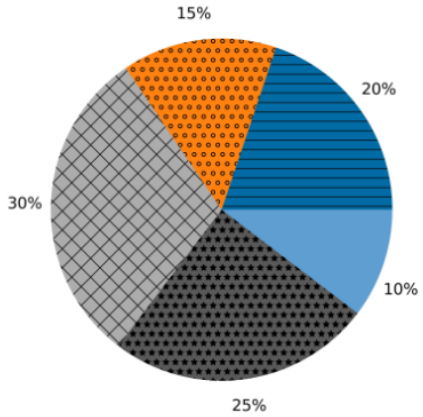
General Visualization Guidelines

Generally, it is recommended: Do not only rely on colors (this also makes printing easier!).

Always use different markers for your data points (circles, squares, triangles...), as well as different line styles (solid, dashed, dot dashed lines...). In case of pie charts or histograms, fill each section with patterns:



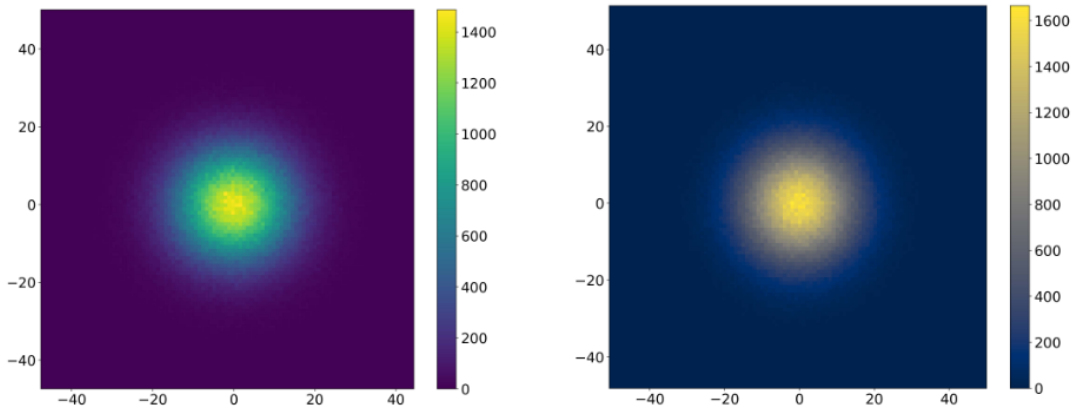




When giving an oral or written description, avoid describing the results of a plot based only on the colour and use the shapes instead (e.g. use "the dashed line represents..." instead of "the green line represents...").

General Visualization Guidelines

If that is not possible (e.g. with maps), try to use color blind (and printer) friendly palettes:



Summary

We have seen many ways on how to explore and visualize astronomical data.
With this, you can get an idea on how data you are working with look like.
In addition, you can create plots to represent your data in presentations and talks, thesis, and publications.

We will see more examples on doing so in this week's **tutorial session**.

An Outlook: Astronomical Packages

Next week, we will see how we can enhance our data analysis with using **astronomical packages** provided for Python.