CHICAS
centre for health informatics, computing, and statistics

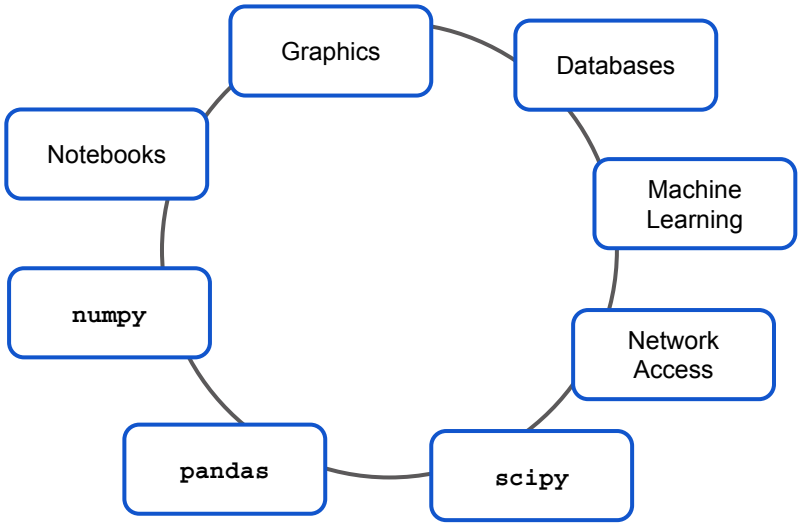
Scientific Python

Barry Rowlingson

Lancaster University
Medical School

Python is being used a lot for science… find out why and how!

Parts



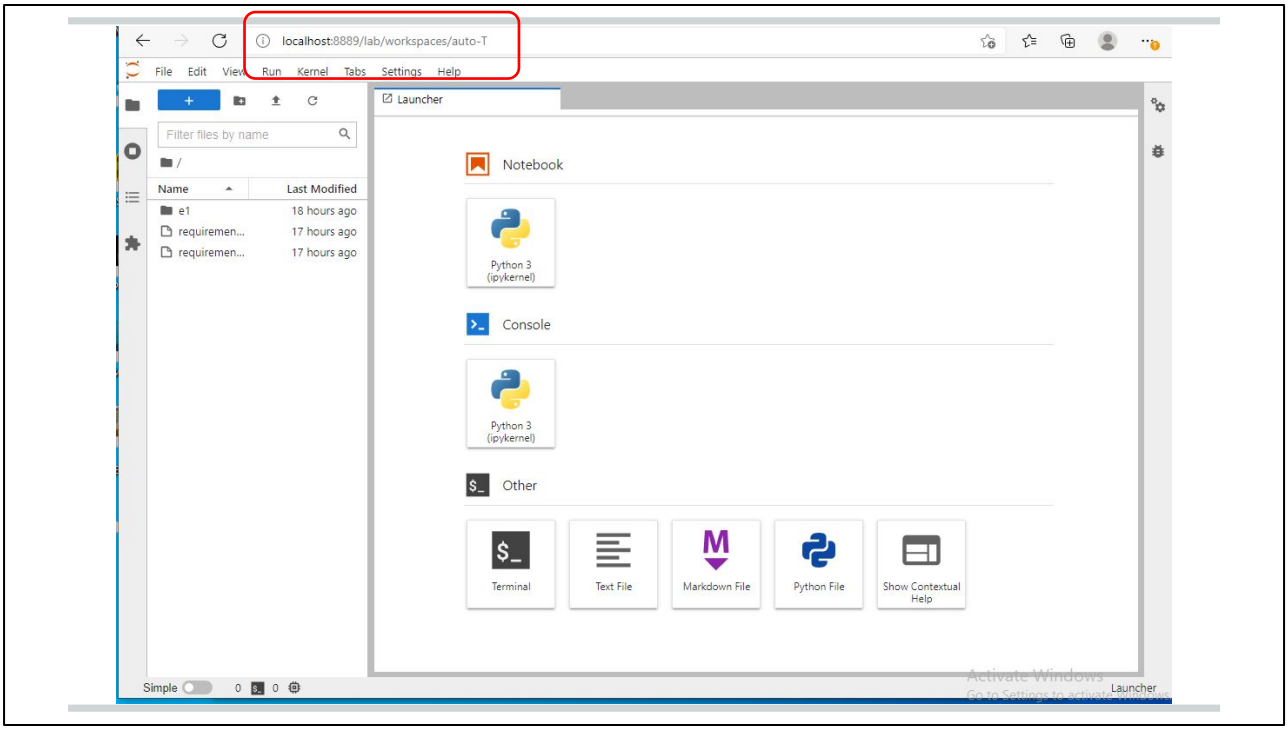
The scientific infrastructure for python revolves round a few key packages and a few key concepts. Most of the work is done in interactive notebooks, and involves the usual scientific graphics, databases, network access, statistics, machine learning etc, and there are package for all of that. The fundamental packages on which most python science build are numpy for numeric calculation, pandas for data-frame operations, and scipy for general scientific calculation.

Jupyter Lab Notebooks

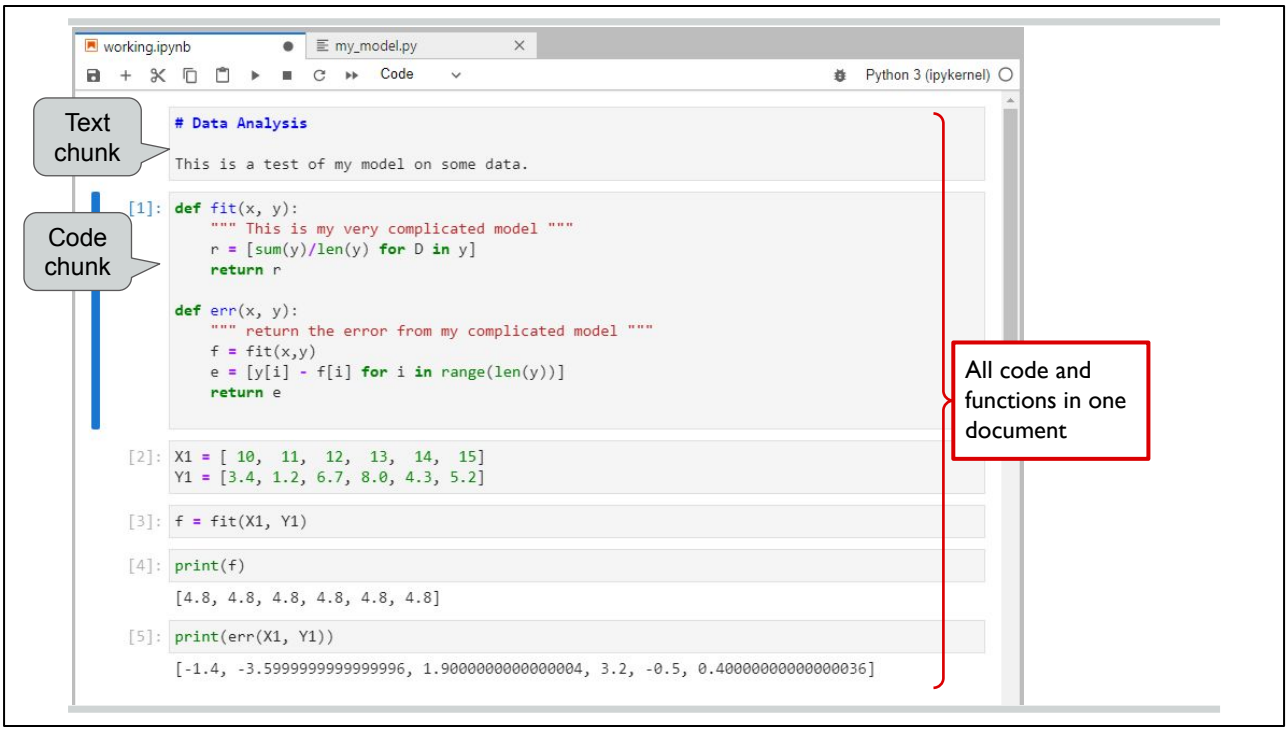
CHICAS centre for health informatics, computing, and statistics

```
> pip install jupyterlab
[lots happens]
> jupyter lab
[more things happen…]
```

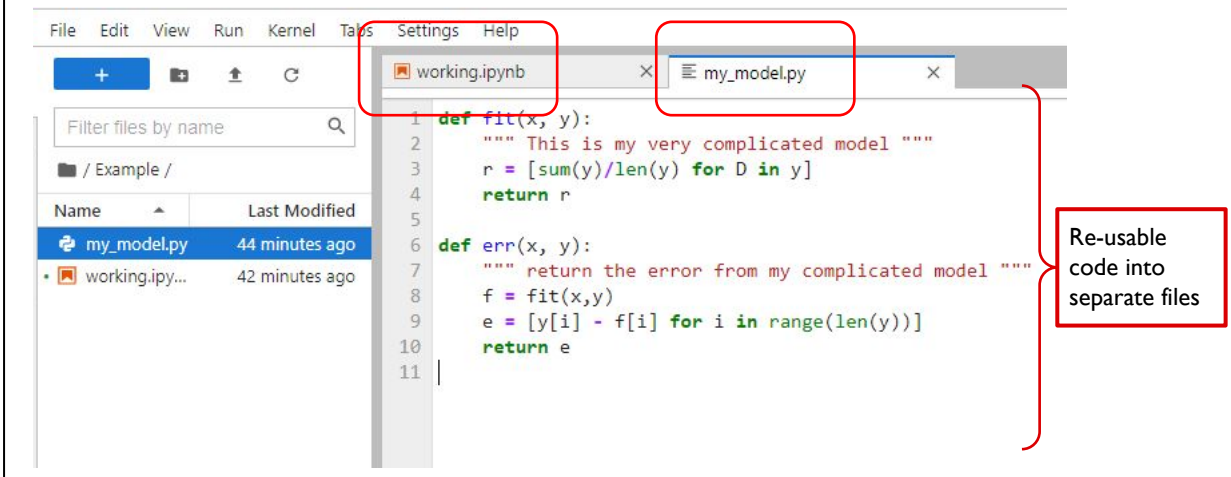
Jupyter lab notebooks have become the de-facto standard for working on science applications. Install using pip, and run...



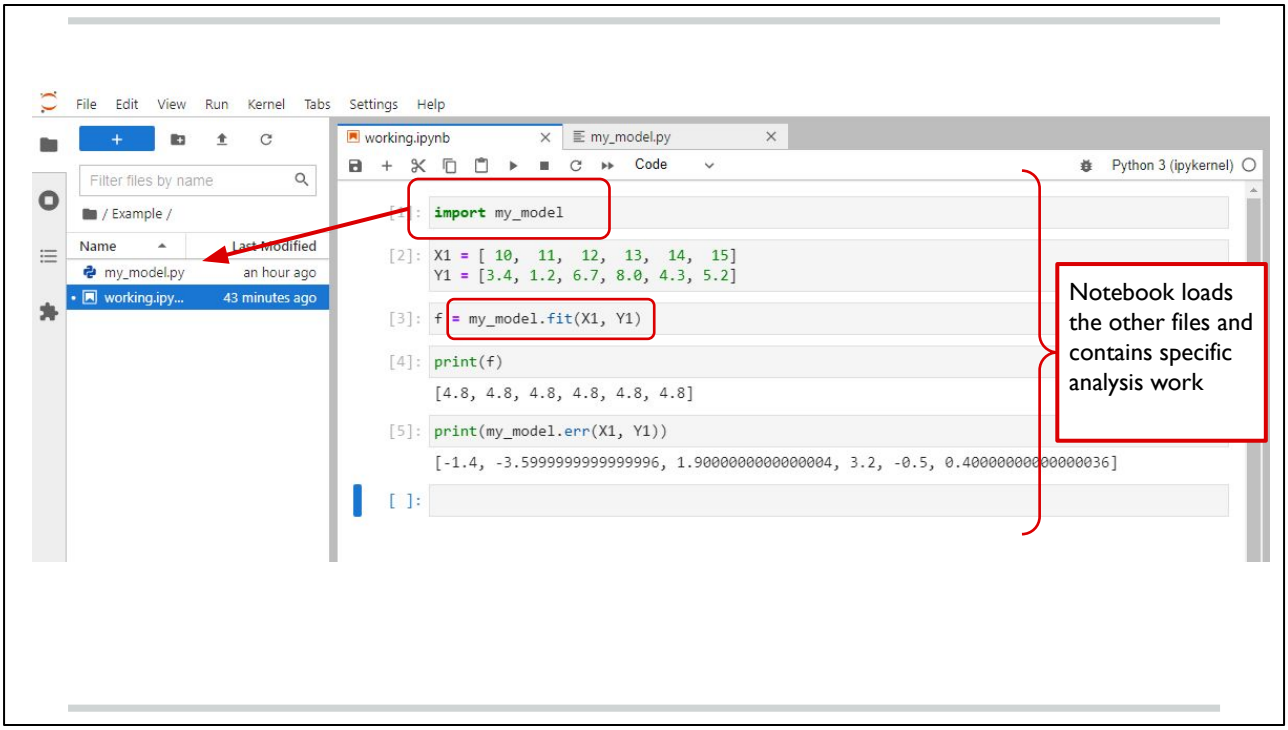
The notebook interface appears in a browser via a local web server which isn't exposed to the outside internet, but it is possible to run jupyter lab servers on remote machines and access them across the internet, securely by password. There is also work on collaborative notebook editing so multiple people can worn on the same document, but that's still a work in flux at this time.



From the main interface you can start new notebooks based on the python “kernel” which is the language used to execute code in the notebook. R-based notebooks, and other languages, can be used. The notebook is a mix of code chunks and text chunks written in markdown. There’s various ways of working in notebooks, one is to keep all your code in a notebook so you have a self-contained file that you can share and other people can run.



Alternatively you can split out anything that might be useful elsewhere into a module, and maybe write some tests and documentation for this….



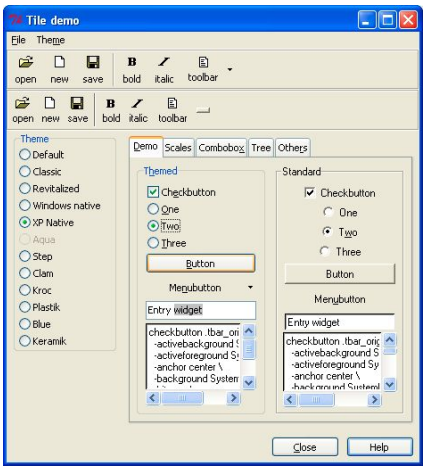
And then you can import this into your notebook. This keeps your notebook “cleaner” and means you can share and re-use the other code.

Graphics and Interfaces

CHICAS centre for health informatics, computing, and statistics

Cross-platform native standalone GUI applications:

- `tkinter`
- `wxpython`



Graphics are core to science and Python has various options for graphical applications. You can build standalone GUI applications that run across platforms that use the system's native widget toolkit, for example this is a demo of all the various controls you can put in an application shown in a (slightly old now) Windows system. The same code could run on a Mac or Linux machine with the dialog automatically styled as other applications on the platform. This is all done by the "tkinter" module, but other user interface modules are available such as wxwidgets.

Drawing - graphics.py

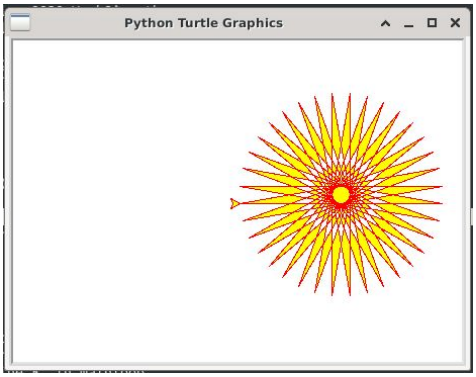
CHICAS centre for health informatics, computing, and statistics

```python
from graphics import *

def main():
    col_arr=["violet","indigo","blue","green","yellow","orange","red"]
    workArea = GraphWin('Rainbow Circle', 300, 300) # give title and dimensions
    x=workArea.getWidth()/2 # get x of middle of drawing area
    y=workArea.getHeight()/2 # get y of middle of drawing area

    i=0
    while i<len(col_arr):
        cir=Circle(Point(x, y), 10+10*i)# draw circle with center at middle of drawing area
        cir.setOutline(col_arr[i]) #get a next outline color from color array
        cir.setWidth(4)#set outline width
        cir.draw(workArea)#draw the current circle
        i+=1 # increment counter for iteration

    message = Text(Point(workArea.getWidth()/2, 250), 'Click to Exit')
    message.draw(workArea)
    workArea.getMouse()# get mouse to click on screen to exit
    workArea.close() # close the drawing window

main()
```

If you just want to draw some lines and points on a canvas, there's a few packages for that in the PyPI index. Here's "graphics.py" which lets you create a graphic window, and draw things on it various colours.

Drawing - turtle

```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```



Slightly more specialised, turtle graphics was designed as an educational system for programming where the system controlled a robot “turtle” and the code instructed the turtle to move, turn, or draw in various colours. A quick look like this - move, turn, move, turn etc - results in a star like this.

Scientific Graphics

CHICAS centre for health informatics, computing, and statistics

- `matplotlib`
  - Low-level graphics and basic charts
- `seaborn`
  - High-level graphics and scales
- `plotnine`
  - Grammar of Graphics system
- `bokeh`
  - Interactive web graphics

But what we want for science is “Scientific Graphics” right? Data visualisations! Python has packages for that. Matplotlib is a low-ish level charting package that will do the basic scatter and line and histogram plots. Seaborn builds on matplotlib to give easier control over appearance. There’s been a couple of grammar of graphics packages, the only one that seems active now is plotnine. These three all produce graphics either in a new window or as an inline image in a notebook. The “bokeh” package creates plots that are designed for the web using HTML and Javascript, and can incorporate interactive controls.

matplotlib

CHICAS centre for health informatics, computing, and statistics

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter([4,6,5,3,4,5,2],[6,5,8,3,2,5,3])
<matplotlib.collections.PathCollection object at 0x7f0916eabfd0>
>>>
```

Nothing happens!

Basic matplotlib. It has a few sub-modules, the most useful one is pyplot which is conventionally imported as “plt” to save typing. Next I call the “scatter” method with a list of X and a list of Y coordinates. Where’s my plot?

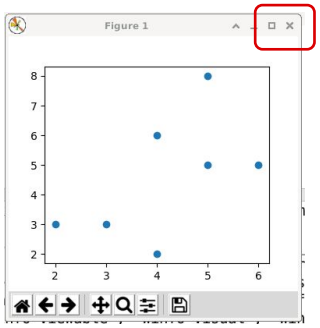
The matplotlib.pyplot object

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter([4,6,5,3,4,5,2],[6,5,8,3,2,5,3])
<matplotlib.collections.PathCollection object at 0x7f0916eabfd0>
>>>
```

Nothing happens!

```
>>> plt.show()
```

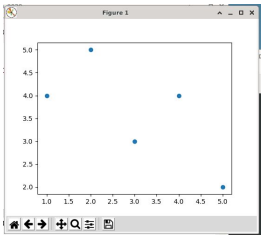
My prompt has gone!!



To show the graph I need to call the show method of plot. There it is. But I don't get my python prompt back. Its stuck until I close the window. But then the window is gone...

Interactive mode

```
>>> plt.ion()
>>> plt.scatter([1,2,3,4,5],[4,5,3,4,2])
<matplotlib.collections.PathCollection object at 0x7f486c3a5be0>
>>>
```



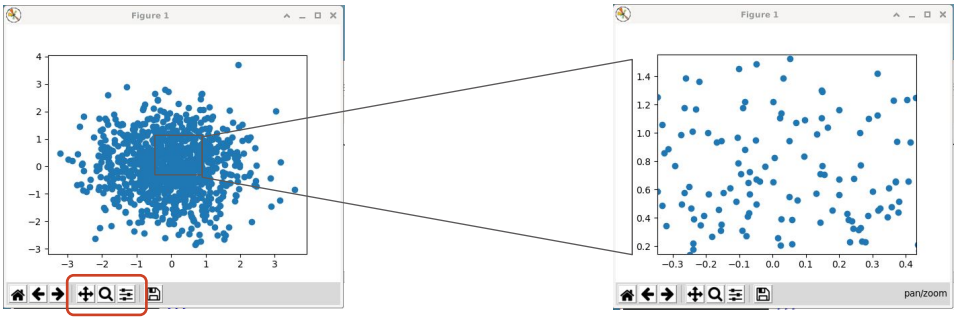

```
>>> plt.close()
>>>
```

When running python from the command line it helps to use interactive mode via calling the “ion” (interactive - on) method. Then a plot will appear on creation and you’ll get your prompt back. So if you get stuck without a prompt you’ll know what the problem is. To close a window, call the close method.

Interaction

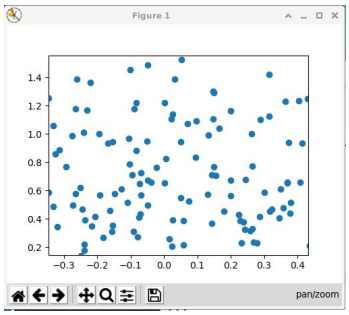
CHICAS centre for health informatics, computing, and statistics

```
>>> import random
>>> x = [random.gauss(0,1) for i in range(1000)]
>>> y = [random.gauss(0,1) for i in range(1000)]
>>> p = plt.scatter(x,y)
>>>
```

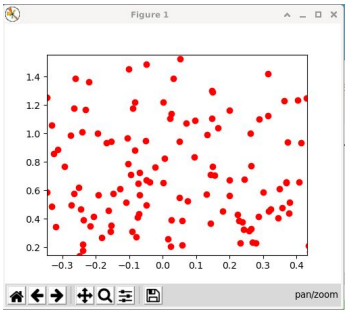


Python's plots are not static images. Far from it. For one, there's some controls that let you zoom in to see detail, such as in this busy cluster of 1000 points. But also the plotting methods return an object that you can work with. Here I've stored it in "p".

Changing plots




```
>>> p.set_color("red")
>>>
```

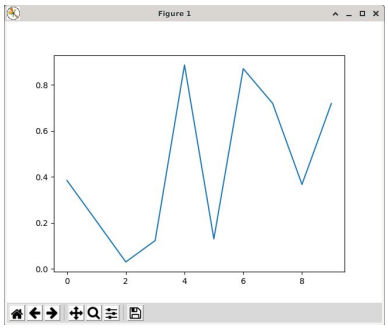


Then I can methods like “set_color” to change the points to red. This gives us a lot of power and control over plots.

Plot one line...

```
>>> x = range(10)
>>> y = [random.uniform(0,1) for i in x]
>>> plt.plot(x,y)
[<matplotlib.lines.Line2D object at 0x7f486593d100>]
```

Y values between 0 and 1

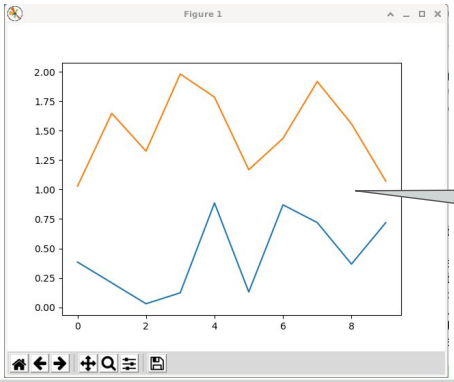


We can also add things to the current plot and see the change instantly. Here's a single line plot.

...adding another one.

```
>>> y2 = [random.uniform(1,2) for i in x]
>>> plt.plot(x,y2)
[<matplotlib.lines.Line2D object at 0x7f48658d42b0>]
```

Y values between 1 and 2



Plot rescales to fit

If I generate some more random numbers and call the plot method again, the plot rescales to fit the new line.

Figures and Axes

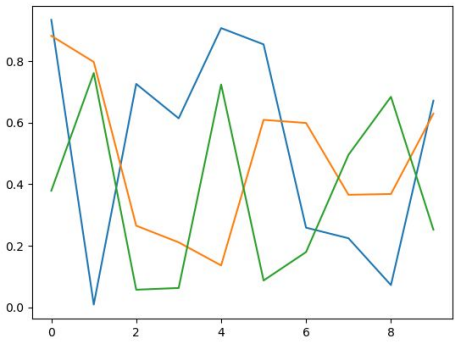
CHICAS centre for health informatics, computing, and statistics

```
>>> x = range(10)
>>> y1 = [random.uniform(0,1) for i in x]
>>> y2 = [random.uniform(0,1) for i in x]
>>> y3 = [random.uniform(0,1) for i in x]
```

Create a 1x1 subplot and return figure and axes

```
>>> fig, ax = plt.subplots()
>>> ylines = ax.plot(x, y1, x, y2, x, y3)
```

Add three lines to the axes and save.



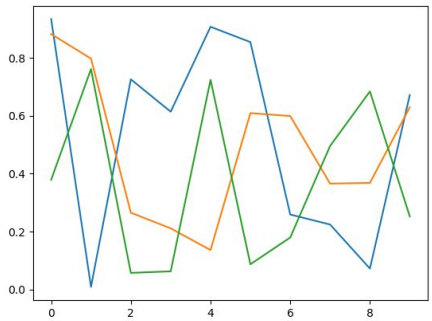

```
>>> ylines
[<matplotlib.lines.Line2D object at 0x7f4865577c40>,
 <matplotlib.lines.Line2D object at 0x7f486545aa30>,
 <matplotlib.lines.Line2D object at 0x7f486545a730>]
```

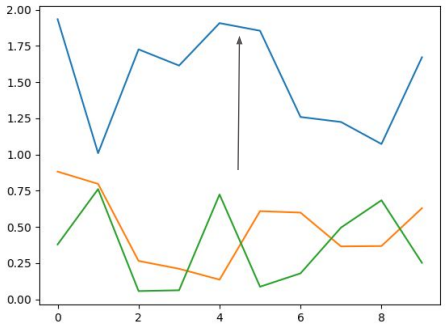
The “plt” object does have a lot of methods attached to it, so often you’ll see a plot created using an empty call to subplots() which returns two things that are conventionally unpacked into “figure” and “axes”. The “axes” refers to a plot with a set of X-Y axes, and the figure is the entire graphic space. So to add some lines I now call the plot method of the axes object. Here I’m adding three lines, and you can see the method returns a list of three objects that represent the lines.

Modify on-the-fly

CHICAS centre for health informatics, computing, and statistics

```
>>> ylines[0].set_data(x, [y + 1 for y in y1])
>>> ax.relim()
>>> ax.autoscale_view()
```





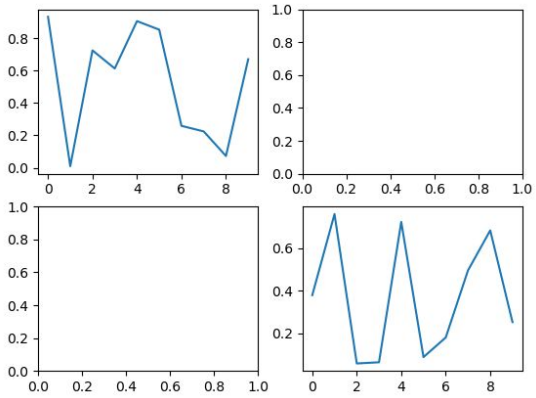
Those lines objects have methods for changing the colour, style etc, and also you can change the data of the line. Here I'm adding 1 to the Y coordinates, and after telling python that it needs to update its axes, you can see the new data. I suspect the rescaling can be done automatically but not sure how...

Multiple subplots

centre for health informatics, computing, and statistics

Create a list of lists of "axes":

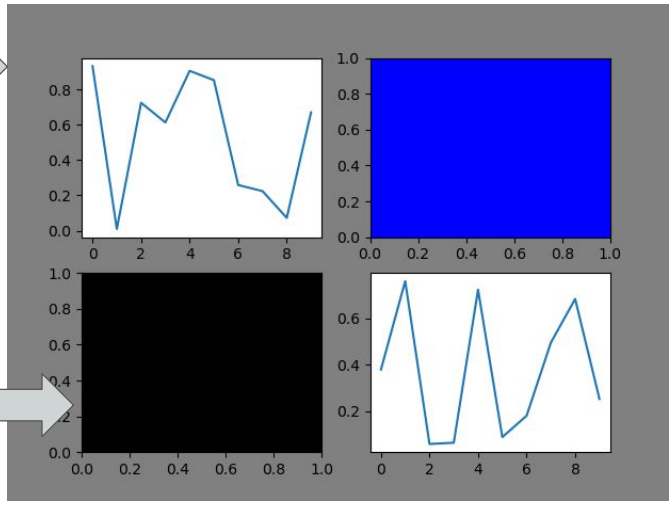
```
>>> fig, ax = plt.subplots(2,2)
>>>
>>> ax[0][0].plot(x, y1)
[<matplotlib.lines.Line2D object at 0x7f48
>>> ax[1][1].plot(x, y3)
[<matplotlib.lines.Line2D object at 0x7f48
```



Calling plt.subplot() sets up a single plot in a figure, but you can divide the figure into multiple axis sets. Here's a 2x2 layout. The "ax" object is a list of lists, so I can index into it twice to get individual axes objects and plot into them.

Figs and Axes

```
>>> fig.set_facecolor("grey")
>>> ax[0][1].set_facecolor("blue")
>>> ax[1][0].set_facecolor("black")
```



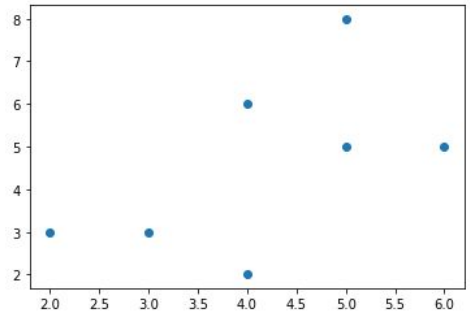
So methods of “fig” here apply to the whole figure, but methods of any element of “ax” refer to single subplots only. This kind of dynamic manipulation of data and attributes of plots makes for easy animations, for example you could animate one of these sub-plots without affecting any of the rest by calling methods on ax[0][0].

In Notebooks

CHICAS centre for health informatics, computing, and statistics

```
[1]: import matplotlib.pyplot as plt

     plt.scatter([4,6,5,3,4,5,2],[6,5,8,3,2,5,3])
```

```
[1]: <matplotlib.collections.PathCollection at 0x7f899f0e8fa0>
```



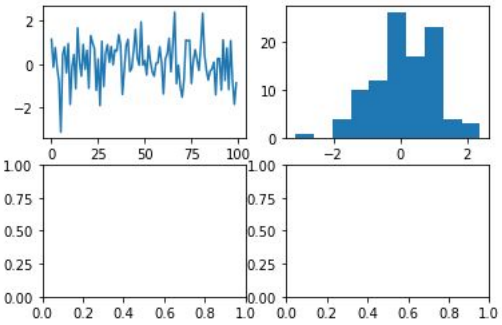
All the previous examples have been called from an interactive standalone python session. If you run from a jupyter lab notebook you get the plot appear when you run the chunk, but without any of the interaction buttons or the capability to modify the plot - you just have to run the chunk again.

```
[8]: import random
     x = range(100)
     y = [random.gauss(0,1) for i in x]
     fig, ax = plt.subplots(2,2)
     ax[0][0].plot(x,y)
     ax[0][1].hist(y, 10)
```

Create 2x2 subplots

Plot lines in one, histogram in the other

```
[8]: (array([ 1.,  0.,  4., 10., 12., 26., 17., 23.,  4.,  3.]),
      array([-3.14623132, -2.59194064, -2.03764996, -1.48335928, -0.9290686 ,
             -0.37477792,  0.17951276,  0.73380345,  1.28809413,  1.84238481,
              2.39667549]),
      <a list of 10 Patch objects>)
```



And you can do the same subplots as in a standalone python matplotlib window, here splitting into 2x2 and doing different graphs in each. That's the basics of matplotlib.

Numeric Python NumPy

CHICAS centre for health informatics, computing, and statistics

- Multi-Dimensional Numerical Data
- Linear Algebra
- Basic Stats
- Basic Random Numbers
- Nearly always import as **np**

```
>>> import numpy as np
```

So far I've just plotted standard python lists of number. We don't get far in science without more dimensions, and for that we use numpy which adds the power of multi-dimensional arrays to python. It also includes methods for linear algebra, statistics, and random numbers.

Basics

CHICAS centre for health informatics, computing, and statistics

```
>>> a = np.array([5,4,3,2,1])
>>> a
array([5, 4, 3, 2, 1])
>>> type(a)
<class 'numpy.ndarray'>

>>> np.arange(1, 4, .3)
array([1. , 1.3, 1.6, 1.9, 2.2, 2.5, 2.8, 3.1, 3.4, 3.7])
>>> np.linspace(1, 4, 11)
array([1. , 1.3, 1.6, 1.9, 2.2, 2.5, 2.8, 3.1, 3.4, 3.7, 4. ])

>>> np.zeros(12).reshape(3,4)
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
>>> a = np.zeros(12).reshape(3,4)
>>> a.shape
(3, 4)
```

The basic data type is the array. Here's some one-dimensional arrays constructed from plain python lists or from ranges. You can see these are of type "ndarray" for N-Dimensional. To make 2 dimensional arrays, start with a one-dimensional array and then call the "reshape" method with the desired dimension. Get the "shape" element to see the dimension.

"Reference" semantics

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

a is a *reference* to that array

```
>>> aa = a
```

aa is now another *reference* to that array…

```
>>> aa[1] = 999
>>> a
array([  0, 999,   2,   3,   4,   5,   6,   7,   8,   9])
```

changing **aa** changes **a** because they are references

Note that python uses "reference" semantics for most data structures. So here "a" doesn't "store" those values, it "points to" them, stored off in memory somewhere. If you then do "aa = a" you've not copied the data into "aa", you've copied the reference. There's still only one copy of the data. That means if you change the value of anything in "a", you'll change it in "aa" as well. Watch out, because R doesn't work like this, it does copy the data.

Make Copies

CHICAS centre for health informatics, computing, and statistics

If you do need a copy, call the `copy` method:

```
>>> a = np.arange(10)
>>> aa = a.copy()
>>> aa[1]=999
>>> aa
array([  0, 999,   2,   3,   4,   5,   6,   7,   8,   9])
>>> a
array([0, 1, 2, 3, 4, 5, 6,   8, 9])
```

Copy all the data and point `aa` to it

Changing `aa` doesn't affect `a` now.

There's a "copy" method for when you do need to do this.

Arithmetic

```
>>> a = np.array([1,2,3,4]).reshape(2,2)
>>> a
array([[1, 2],
       [3, 4]])
>>> a + 100
array([[101, 102],
       [103, 104]])
>>> a * 100
array([[100, 200],
       [300, 400]])
>>> a > 2
array([[False, False],
       [ True,  True]])
>>> a @ a
array([[ 7, 10],
       [15, 22]])
```

Python R

```
> a = matrix(1:4,2,2)
> a
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> a+100
     [,1] [,2]
[1,]  101  103
[2,]  102  104
> a * a
     [,1] [,2]
[1,]    1    9
[2,]    4   16
> a %*% a
     [,1] [,2]
[1,]    7   15
[2,]   10   22
```

Standard arithmetic on numpy arrays works as expected, and a bit like R, except R's matrices are constructed in the other order. Add, multiply, test etc. The matrix multiplication operator is different - in numpy its an "@" sign compared to R's %*% operator. But it does the same thing.

Like R, not like R

CHICAS centre for health informatics, computing, and statistics

R will repeat small vectors:

```
> a = 1:10
> b = 1:2
> a+b
 [1]  2  4  4  6  6  8  8 10 10 12
```

numpy refuses:

```
>>> a = np.arange(10)
>>> b = np.arange(2)
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (10,) (2,)
```

So its a bit like R, except numpy won't repeat a short array to the size of a large array when needed. For example adding (0,1) to (0,1,2,3,4,5,6,7,8,9) will fail.

"Broadcasting"

CHICAS centre for health informatics, computing, and statistics

Compatible dimensions can be "broadcast":

```
>>> e = np.arange(12).reshape(3,4)
>>> e
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Set up a 3x4 array

```
>>> 100*np.arange(4)
array([  0, 100, 200, 300])
```

Want to add these four values to the four columns....

```
>>> e + 100*np.arange(4)
array([[  0, 101, 202, 303],
       [  4, 105, 206, 307],
       [  8, 109, 210, 311]])
```

The small array is compatible with the dimension of `e`

Numpy will repeat things across dimensions that are either equal to 1 or equal to each other. So for example a 3x4 array and a 1x4 array can be operated on. Here you can see how to add a constant to each column from another array.

"Broadcasting"

CHICAS centre for health informatics, computing, and statistics

Suppose we want to add to the rows...

```
>>> e + 100*np.arange(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,4) (3,)

>>> 100*np.arange(3).reshape(3,1)
array([[  0],
       [100],
       [200]])

>>> e + 100*np.arange(3).reshape(3,1)
array([[  0,   1,   2,   3],
       [104, 105, 106, 107],
       [208, 209, 210, 211]])
```

Reshape into a 2d array with compatible dimensions

Can now add to rows

If you want to add to the rows. That means adding a (3,1) shape to a (3,4) shape, so we reshape our array into that using the reshape method. Now we can add them.

Like R, not like R

CHICAS centre for health informatics, computing, and statistics

How do we solve this?

```
>>> a = np.arange(10)
>>> b = np.arange(2)
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (10,) (2,)
```

So how do we do this?

Reshape and Broadcast

```
>>> a.reshape(-1,b.shape[0])
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> a.reshape(-1,b.shape[0]) + b
array([[ 0,  2],
       [ 2,  4],
       [ 4,  6],
       [ 6,  8],
       [ 8, 10]])
>>> (a.reshape(-1,b.shape[0]) + b).reshape(-1)
array([ 0,  2,  2,  4,  4,  6,  6,  8,  8, 10])
```

`a` reshaped so its second axis matches `b`'s first axis

This is now compatible with `b` for broadcasting

Flatten `a` back to one dimension.

We have to think of this as a "hidden" dimension. By reshaping "a" into two-columns (from b.shape[0]) we can then add it to our length-2 "b" array and then flatten the whole thing back to one dimension afterwards. Note that this is not an expensive operation since python doesn't move any data around for a reshape operation, it only changes its idea of what the dimensions are, which is a quick operation.

Mathematical Functions

```
>>> a = np.linspace(0, 1, 10)
>>> a
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
>>> import math
>>> math.sin(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only size-1 arrays can be converted to Python scalars
>>>
>>> np.sin(a)
array([0.        , 0.11088263, 0.22039774, 0.3271947 , 0.42995636,
       0.5274153 , 0.6183698 , 0.70169788, 0.77637192, 0.84147098])
```

Python math functions fail

Use `numpy` functions

If you want to do mathematical function on numpy arrays you have to use the functions from the package since the standard “math” module ones wont work. The numpy ones will also work on non-numpy lists of numbers so if you are using numpy in your project you don’t need the math library as well.

Functions along an axis

CHICAS centre for health informatics, computing, and statistics

```
>>> a = np.arange(9).reshape(3,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.mean(a)
4.0
>>> np.mean(a, axis=0)
array([3., 4., 5.])
>>> np.mean(a, axis=1)
array([1., 4., 7.])
```

Global mean value

Mean along any axis

Apply any function along any axis

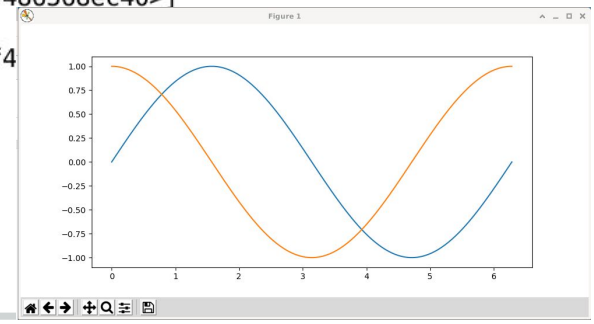
```
>>> np.apply_along_axis(np.mean,0,a)
array([3., 4., 5.])
>>> np.apply_along_axis(np.mean,1,a)
array([1., 4., 7.])
```

Standard summary-type functions are also there, for example the overall mean of all values, or the mean computed along any dimension axis. There's also a function like R's apply which lets you compute with any function along any axis.

Simple Plots

`numpy` works with `matplotlib`

```
>>> x = np.linspace(0, 2*math.pi, 360)
>>> y = np.sin(x)
>>> plt.plot(x,y)
[<matplotlib.lines.Line2D object at 0x7f486568ec40>]
>>> plt.plot(x,np.cos(x))
[<matplotlib.lines.Line2D object at 0x7f4
```



And numpy works well with matplotlib so you can send numpy arrays to plots and they all work perfectly. In fact matplotlib converts anything to numpy arrays internally.

Conclusions

CHICAS centre for health informatics, computing, and statistics

- Get familiar with:
  - jupyter lab notebook
  - `numpy`
  - `matplotlib`
- Workshop Exercise
  - Statistics
  - Plotting
- But do read more online tutorials...
  - You need experience and practice

So that's the first taste of scientific python. There's notebooks and numpy and matplotlib for starters. The workshop exercise does some numerical computation and plotting. But you should also check out other online tutorials for numpy and maplotlib and try and get plenty of experience.