Lecture 2.2

Functions which return lists

If a function is supposed to create a list, then create a list **inside** the function:

```
stuff = [] # DON'T DO THIS IF YOU INTEND TO RETURN stuff
def f(N):
    for n in range(1,N+1):
        stuff.append(n)
    return stuff # DON'T RETURN A LIST CREATED OUTSIDE

a = f(2)
b = f(4)
b # << this now contains [1,2,1,2,3,4] which is probably not what was intended
```

```
[1, 2, 1, 2, 3, 4]
```

`is` vs `==`

```
a = [1,2,3]; b = a
a == b, a is b
```

```
(True, True)
```

```
a = [1,2,3]; b = [1,2,3]
a == b, a is b
```

```
(True, False)
```

If `a is b` then when I make a change to `a` I have also made a change to `b`. Not so if they are different things which just so happen to be equal. (No distinction for immutable types such as integers.)

Try the above using Python Visualizer.

Assignment 1: Part 3

`trajectory` illustrates a worked example of where we should get to at the end of the course i.e. numerical solution to a differential equation, wrapped up in a function which gives you the stuff you want (points in time, position, velocity) which result from a simulation with given initial conditions.

It returns these as 5 separate 1D `numpy.array`.

The questions are designed to test the following:

- using the code in a straightforward way
  - assign a specific value to `theta` e.g. 1 radian or so
  - run the simulation
  - plot the resulting (x, y)
- using the code to build something using a function we've met `newton`
  - make a function which returns the difference between 1000 and the position where the trajectory ends, for a given `theta`
  - use `newton` to find the value of `theta` which sets this function to zero
  - you have to give `newton` an initial guess (e.g. 1 radian) but then it does the hard work for you
- using the code to extract some interesting quantities for different values of `theta`
  - this is probably going to involve a for loop over different values of `theta`
  - for "last" or "final" value, use `x[-1]` or similar. Negative indices count backwards from the end.

Interpolation

Say we have some points on a curve and we want a model-independent way to approximate values in between. e.g. we might have some altitude measurements on a hillside, and we want to estimate the altitude at intermediate points. This is called *interpolation*.

Start with 1D. Imagine we have some points on a curve.

```
x = [-2. , -1.6, -1.2, -0.8, -0.4,  0. ,  0.4,  0.8,  1.2,  1.6,  2. ]
y = [ 0.018,  0.077,  0.237,  0.527,  0.852,  1.   ,  0.852,  0.527,  0.237,  0.077,  0.018]
```

To estimate the underlying function from which these points are picked, we interpolate.

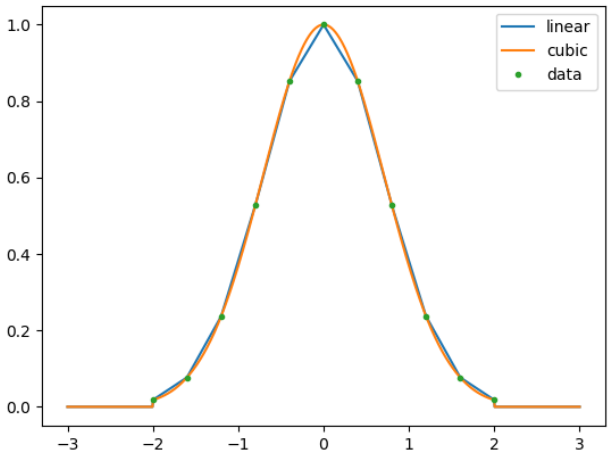
```
from scipy.interpolate import interp1d
f = interp1d(x,y) # basic form.  doesn't allow points outside.  defaults to linear interpolation

f_lin = interp1d(x,y,kind='linear', bounds_error=False, fill_value=0)
f_cub = interp1d(x,y,kind='cubic',  bounds_error=False, fill_value=0)

x_int = np.linspace(-3,3,1001)
y_lin = f_lin(x_int)
y_cub = f_cub(x_int)

plt.plot(x_int,y_lin,label='linear')
plt.plot(x_int,y_cub,label='cubic')
plt.plot(x,y,'.',label='data')

plt.legend()
plt.savefig('figs/interp1d.png')
plt.close('all')
```



2D Interpolation [BEYOND]

This is beyond the scope of the course. See e.g. `scipy.interpolate.interp2d` or `griddata`:

```
from scipy.interpolate import griddata
def f(x,y): return np.exp(-x**2-y**2)    # a simple 2D Gaussian
xp = (np.random.random(100)-0.5)*3       # pick random points on
yp = (np.random.random(100)-0.5)*3       # this Gaussian hillside
zp = [f(xi,yi) for xi,yi in zip(xp,yp)] # evaluate the height at each x,y
xs = np.linspace(-1.5,1.5,101)           # this makes a regular grid of
ys = np.linspace(-1.5,1.5,101)           # points in x and y
X,Y = np.meshgrid(xs,ys)
# Evaluate the interpolating function at all points X,Y
Z = griddata((xp,yp),zp, (X,Y), method='linear', fill_value=0)
```

Curve fitting

Given some data, and a theoretical curve with some free parameters, find the values of the parameters which minimize the sum of the squares of the residuals.

The function is called `curve_fit` and it is available from `scipy.optimize`.

We need to tell it

- the function to fit
- the data x and y
- an initial guess at the fit parameters

It gives us

- the best fit parameters
- an estimate of the uncertainty *as a covariance matrix*

```
from scipy.optimize import curve_fit

def f(x,a,b):
    return a*x + b # function to fit

guess = [1,2] # initial guess
xdata = [1,2,3]
ydata = [2.9,5.1,6.9]
popt,pcov = curve_fit(f, xdata, ydata, p0=guess)
popt
```

```
[ 2.          0.96666667]
```

`popt` is the best-fit parameters, which we can feed into function `f` by unpacking (more in next section).

Covariance matrix (aside)

`pcov` is the covariance matrix. "Variance" is a name for the square of the uncertainties. The diagonal elements are the variances of the fit parameters. However, if your estimate of the gradient is wrong then *this affects your estimate of the offset*. This inter-dependence is captured by the concept of "covariance" i.e. the off-diagonals in the covariance matrix.

This is usually important but usually ignored. For a superb book, see Data Analysis (A Bayesian Tutorial) by Sivian and Skilling or Measurements And Their Uncertainties: A practical guide to modern error analysis by Ifan Huges.

For now, it's enough to note that the square root of the diagonal elements of the covariance matrix is the uncertainties. i.e.

```
np.sqrt(pcov.diagonal())
```

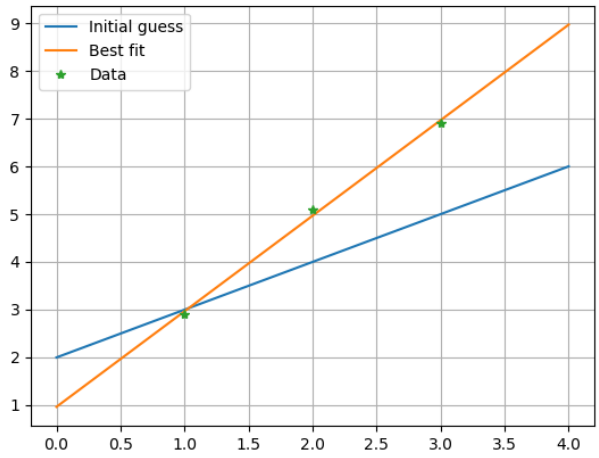
```
[ 0.11547005  0.24944382]
```

`thing.diagonal()` extracts the diagonal elements of a square 2D array into a 1D array

`np.sqrt` calculates the square root of each element of the array passed to it.

Let's plot this to see if it's sensible:

```
x = np.linspace(0,4,1001) # raster the x axis at a finer resolution
ybest  = f(x,*popt) # see "Unpacking" section, below
yguess = f(x,*guess)
plt.plot(x,yguess,label='Initial guess')
plt.plot(x,ybest, label='Best fit')
plt.plot(xdata,ydata,'*',label='Data')
plt.legend(); plt.grid(True)
plt.savefig('figs/curve_fit.png'); plt.close('all')
```



Unpacking

As demonstrated when calling the function with best-fit parameters, we can **unpack** an iterable (e.g a list or a tuple) by using a star `*`. This passes each element from the iterable to the function, one by one. e.g.

```
def f(x,a,b):
    return a*x + b

parameters = (2,5)
x = np.linspace(0,4,1001)
y = f(x,*parameters) # the same as y = f(x, parameters[0], parameters[1])
```

You can also unpack a dictionary to fill in named arguments i.e. those with a default value. See e.g. here.

Arrays and matrices

Numpy arrays can be multi-dimensional, and you can refer to elements by index

```
A = np.array([[1,0,0],[0,1,0],[0,0,1]])
A
```

```
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

```
A[2,1] = 6
```

Refer to an entire row or column like so:

```
A[:,0]
```

This is **slicing**. Syntax is `A[start:stop:step]` for each dimension; if a value is omitted then the following defaults are used: `{start: 0, stop: [last element], step: 1}`. More advanced indexing is possible (but beyond the scope of this course). For further reading, I recommend Scipy Lectures.