Python lab 3: 2D arrays and plotting

Dr Ben Dudson

Department of Physics, University of York

11th February 2011

http://www-users.york.ac.uk/∼bd512/teaching.shtml

From last time...

- Last time started using NumPy and Matplotlib to create arrays and plot data
- Arrays could be created using functions like `linspace`, `arange` and `zeros`
- Once created, arrays can be used much like other variables, so `x = x ** 2` squares every number in an array `x`
- Matplotlib can be used to plot data, and even simple animations

This time, we'll look at some more things we can do with arrays and Matplotlib

Indexing arrays

Last time we used array operations to calculate values for every number (**element**) in an array:

```
y = sin(x)
```

- This is an efficient way to do calculations in Python, but sometimes we need to do something more complicated on each element separately.

Indexing arrays

Last time we used array operations to calculate values for every number (**element**) in an array:

```
y = sin(x)
```

- This is an efficient way to do calculations in Python, but sometimes we need to do something more complicated on each element separately.
- The main reason is if elements in the array depend on each other. If we do an array operation then each number in the array is treated separately.

In this case we can use square brackets to refer to individual numbers in the array

```
y[0] = 10
```

Indexing arrays

NumPy is designed to handle large arrays of data efficiently, so to achieve this it tries to minimise copying data. This leads to some quirks which you should watch out for.

Indexing arrays

NumPy is designed to handle large arrays of data efficiently, so to achieve this it tries to minimise copying data. This leads to some quirks which you should watch out for.

What would you expect this to do?

```
a = linspace(0, 1, 11)
b = a
b = b + 1
print a
print b
```

Indexing arrays

NumPy is designed to handle large arrays of data efficiently, so to achieve this it tries to minimise copying data. This leads to some quirks which you should watch out for.
What would you expect this to do?

```
a = linspace(0, 1, 11)
b = a
b = b + 1
print a
print b
```

```
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
[ 1.   1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2. ]
```

so far so good...

Indexing arrays

What about

```
a = linspace(0, 1, 11)
b = a
a[1] = 5.0
print a
print b
```

Indexing arrays

What about

```
a = linspace(0, 1, 11)
b = a
a[1] = 5.0
print a
print b
```

```
[ 0.   5.   0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

a has changed as expected

Indexing arrays

What about

```
a = linspace(0, 1, 11)
b = a
a[1] = 5.0
print a
print b
```

```
[ 0.   5.   0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

a has changed as expected

```
[ 0.   5.   0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

and so has b!

Copying arrays

To avoid this problem, use copy to make a copy of the array

```
a = linspace(0, 1, 11)
b = copy(a)
a[1] = 5.0
print a
print b
```

Copying arrays

To avoid this problem, use copy to make a copy of the array

```
a = linspace(0, 1, 11)
b = copy(a)
a[1] = 5.0
print a
print b
```

```
[ 0.    5.    0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
[ 0.    0.1   0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

Now behaves as expected

If you're changing individual numbers in an array, make sure you use copy() to avoid nasty side-effects

Multi-dimensional arrays

- So far we've just used one-dimensional arrays, i.e. arrays with just one index x[i]
- Often we will want to handle arrays which depend on more than one dimension
- This is not much more complicated in Python than one-dimensional arrays, and the same ideas apply to both

Creating 2D arrays

For 1D arrays, we could use:

```
x = zeros(5)
```

creates a 1D array containing 5 zeros:

```
[ 0.  0.  0.  0.  0.]
```

Creating 2D arrays

For 1D arrays, we could use:

```
x = zeros(5)
```

creates a 1D array containing 5 zeros:

```
[ 0.  0.  0.  0.  0.]
```

To create 2D arrays we can use

```
x = zeros( (4,3) )
```

creates a 2D array

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Note the double brackets in the zeros function

Using 2D arrays

Once you've created 2D arrays, they can be used like 1D arrays

```
x = zeros( (4,3) )
x = x + 1
print x
```

Using 2D arrays

Once you've created 2D arrays, they can be used like 1D arrays

```
x = zeros( (4,3) )
x = x + 1
print x
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
```

Indexing 2D arrays

With 1D arrays, we could use or modify individual numbers (elements) in the array using square brackets

```
x = linspace(-1,1,5)
print x
```

```
[-1.  -0.5  0.   0.5  1. ]
```

Indexing 2D arrays

With 1D arrays, we could use or modify individual numbers (elements) in the array using square brackets

```
x = linspace(-1,1,5)
print x
```

```
[-1.  -0.5  0.   0.5  1. ]
```

```
print x[3]
```

```
0.5
```

Indexing 2D arrays

2D arrays work the same way, so if we create a 2D array of random numbers

```
from numpy import *
a = random.random((2,4))
print a
```

```
[[ 0.10023954  0.7639587   0.79888706  0.05098369]
 [ 0.77588887  0.00608434  0.31309302  0.20368021]]
```

Indexing 2D arrays

2D arrays work the same way, so if we create a 2D array of random numbers

```
from numpy import *
a = random.random((2,4))
print a
```

```
[[ 0.10023954  0.7639587   0.79888706  0.05098369]
 [ 0.77588887  0.00608434  0.31309302  0.20368021]]
```

```
print a[1,2]
```

```
0.31309302
```

Creating 2D arrays

Another example using linspace function in 1D:

```
x = linspace(0,4,5)
```

which produces

```
[ 0.  1.  2.  3.  4.]
```

Unfortunately this doesn't work for 2D arrays, but instead there's a useful trick to use mgrid which does a similar job for 2D arrays

```
x,y = mgrid[0:5, 0:3]
```

This produces 2 arrays x and y

```
x = [[0 0 0]                              y = [[0 1 2]
     [1 1 1]                                   [0 1 2]
     [2 2 2]                                   [0 1 2]
     [3 3 3]                                   [0 1 2]
     [4 4 4]]                                  [0 1 2]]
```

Using 2D arrays

This mgrid function can be used to calculate 2D functions. For example, to calculate

$$f(x, y) = e^{-x^2} sin(y)$$

between $-2 \leq x \leq 2$ and $0 \leq y \leq 2\pi$ we could use

```
from numpy import *
x,y = mgrid[-2:2:20j, 0:(2*pi):20j]
f = exp(-x**2) * sin(y)
```

Using 2D arrays

This mgrid function can be used to calculate 2D functions. For example, to calculate

$$f(x, y) = e^{-x^2} sin(y)$$

between $-2 \leq x \leq 2$ and $0 \leq y \leq 2\pi$ we could use

```
from numpy import *
x,y = mgrid[-2:2:20j, 0:(2*pi):20j]
f = exp(-x**2) * sin(y)
```

The general format is

```
mgrid[start:end:step,...]
```

(without a 'j' at the end) which specifies the step size, or

```
mgrid[start:end:numj,...]
```

with a number followed by a 'j' to give the number of steps

Plotting 2D arrays

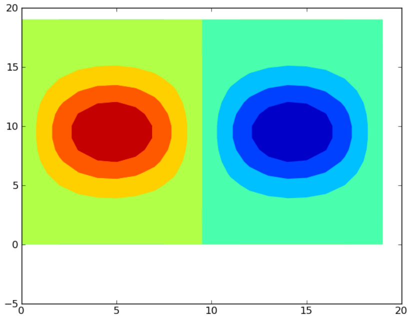
2D data can't be plotted using plt.plot() which we used for 1D data before. Instead, there are other types of plots we can use

```
from numpy import *
import matplotlib.pyplot as plt
x,y = mgrid[-2:2:20j, 0:(2*pi):20j]
f = exp(-x**2) * sin(y)
plt.contourf(f)
plt.show()
```

Plotting 2D arrays

2D data can't be plotted using plt.plot() which we used for 1D data before. Instead, there are other types of plots we can use

```
from numpy import *
import matplotlib.pyplot as plt
x,y = mgrid[-2:2:20j, 0:(2*pi):20j]
f = exp(-x**2) * sin(y)
plt.contourf(f)
plt.show()
```



Surface plots

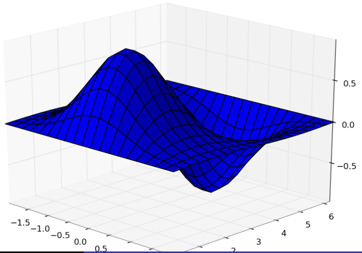
plotting 3D surfaces is a little more tricky

```
from numpy import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
x,y = mgrid[-2:2:20j, 0:(2*pi):20j]
f = exp(-x**2) * sin(y)
fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(x, y, f, rstride=1, cstride=1)
plt.show()
```

Surface plots

plotting 3D surfaces is a little more tricky

```
from numpy import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
x,y = mgrid[-2:2:20j, 0:(2*pi):20j]
f = exp(-x**2) * sin(y)
fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(x, y, f, rstride=1, cstride=1)
plt.show()
```



Summary

- NumPy can be used to create arrays with more than one dimension
- As with 1D arrays, multi-dimensional arrays can be treated as single numbers, and calculations are done for all the numbers in the array
- If we need to refer to individual numbers in the array, we use the array index which counts from zero
- If you want to make a copy of an array, use copy() to avoid strange side-effects

`http://www-users.york.ac.uk/~bd512/teaching.shtml`