Important info

- Class test in Week 2 lab class (Wed 10am / Fri 11am).

- The test will appear on home page:
  http://sam-dolan.staff.shef.ac.uk/mas212

- Submission deadline Sun midnight (13th Oct):
  https://somas-uploads.shef.ac.uk/mas212
Today’s lecture

- Scientific computing modules:
  - numpy
  - matplotlib
  - scipy

- Arrays: data types, creation, slicing, views, vectorization, ufuncs, broadcasting, etc.

- Linear algebra

- Functions

- Objects and classes

- Standard modules: math, cmath, random, os, datetime
Modules for scientific computing

- **numpy**: efficient arrays, broadcasting and linear algebra.  

- **matplotlib**: 2D plotting and animation  
  [http://matplotlib.org/](http://matplotlib.org/)

- **scipy**: for common tasks in scientific computing.  
- How to create new arrays
- $N$-dimensional arrays
- Array manipulation: slicing, views, etc.
- Universal functions, **vectorization** and **broadcasting**
- Linear algebra (matrix multiplication etc)
- Other capabilities: random numbers, Fourier transforms
Convention:
Import the module as follows:

```python
>>> import numpy as np
```
Arrays in numpy

What is an array?

In essence, an array consists of:

- a block of memory – the raw data
- an indexing scheme – how to locate elements
- a data type – how to interpret the raw data

**ndarray**: the N-dimensional array in numpy

An ndarray is a (usually fixed-size) multidimensional container of items of the same type and size. The number of dimensions and items in an array is defined by its shape, which is a tuple of N positive integers that specify the sizes of each dimension. The type of items in the array is specified by a separate data-type object (dtype), one of which is associated with each ndarray.

http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html
There are many ways to make a new 1D array:

- from a list: `np.array([1, 2, 3])`
- `arange`: evenly-spaced with step size
- `linspace`: evenly-spaced with number of points
- `zeros`: array set to zeros
- `ones`: array set to 1s
- `empty`: an uninitialized array
```python
>>> np.array([1,2,3])
array([1, 2, 3])
>>> np.arange(0, 3, 1)
array([0, 1, 2])
>>> np.arange(0, 3, 0.5)
array([0.0, 0.5, 1.0, 1.5, 2.0, 2.5])
>>> np.linspace(0, 3, 5)
array([0.0, 0.75, 1.5, 2.25, 3.0])
>>> np.zeros(6)
array([0., 0., 0., 0., 0., 0.])
>>> np.ones(6)
array([1., 1., 1., 1., 1., 1.])
>>> 2*np.ones(6)  # array of twos
array([2., 2., 2., 2., 2., 2.])
>>> np.empty(2)
array([6.91667204e-310, 6.91667204e-310])
```
Arrays are **homogeneous**: All elements in an array must be of the same data type (dtype).

- Each element is of a fixed size in memory.
- numpy supports a range of data types, including:
  - **int8** A byte (-128 to 127)
  - **uint8** An unsigned integer (0 to 255)
  - **int64** A 64-bit integer (-9223372036854775808 to 9223372036854775807)
  - **float64** A double precision float: sign bit, 11 bits exponent, 52 bits mantissa.
  - **complex128** A complex number, represented by two 64-bit floats (real and imaginary components).
Examples:

```python
>>> a = np.linspace(0, 5, 11)
>>> b = np.arange(0, 11)
>>> print(a, b)
[ 0.  0.5  1.  1.5  2.  2.5  3.  3.5  4.  4.5  5. ]
[ 0 1 2 3 4 5 6 7 8 9 10]
>>> a.dtype  # data type of array
dtype('float64')
>>> b.dtype
dtype('int64')
>>> a.shape
(11,)
>>> a.ndim  # number of dimensions
1
>>> a.size
11
```
**Special values: nan and inf**

- **nan** = not a number
- **inf** = infinity.

```python
>>> a = np.arange(4)
>>> a / 0
array([ nan,  inf,  inf,  inf])
```
Making a 2D array

- From a list-of-lists.
- By reshaping 1D arrays.
- By broadcasting 1D arrays.

```python
>>> np.array([[1, 2], [3, 4]])  # from a list-of-lists
array([[1, 2],
       [3, 4]])

>>> np.arange(4).reshape(2, 2)  # by reshaping
array([[0, 1],
       [2, 3]])

>>> np.array([[1, 2]]) * np.array([[3], [4]])  # by broadcasting
array([[3, 6],
       [4, 8]])
```
Re-shaping an array

```python
>>> a = np.arange(12)

>>> a
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

>>> a.reshape(3,4)  # 3 rows, 4 columns
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])

>>> a.reshape(2,2,3)  # a 3D array
array([[[ 0, 1, 2],
        [ 3, 4, 5]],
       [[[ 6, 7, 8],
         [ 9, 10, 11]]])
```
Indexing and Slicing

It is straightforward to access a single element of the array:

```python
>>> a = np.arange(6).reshape(2,3)  # create a 2x3 array
>>> print(a)
[[0 1 2]
 [3 4 5]]
>>> a[1,2]  # get element in the 2nd row, 3rd column
5
>>> a[1,2] = -1  # modify this element
>>> print(a)
[[ 0  1  2]
 [ 3  4 -1]]
```
**Indexing and Slicing**

Using **slicing**, one can access subarrays:

```python
>>> a = np.arange(12).reshape(3,4)  # create a 3x4 array
>>> print(a)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> print(a[0,:])  # the 1st row
[ 0  1  2  3]
>>> print(a[:, 1])  # the 2nd column
[ 1  5  9]
>>> print(a[::-1, :])  # reverse the order of the rows
[[ 8  9 10 11]
 [ 4  5  6  7]
 [ 0  1  2  3]]
```
Views

- Slices of arrays are **not** new arrays. They are **views** on the original array.

**Example:**

```python
>>> a = np.arange(12).reshape(3,4)
>>> b = a[0, :]  # the first row
>>> b[2] = 99    # By changing an element of b ...
>>> print(a)     # ... we are changing the data in a
[[ 0  1  99  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```
Universal functions (ufuncs)

- A ufunc is a function that operates on whole arrays in an element-by-element fashion.

Examples:

```python
>>> a = np.linspace(0, 2*np.pi, 7)
>>> np.sin(a)  # sin is an example of a ufunc
array([ 0.00000000e+00, 8.66025404e-01, 8.66025404e-01,
        1.22464680e-16, -8.66025404e-01, -8.66025404e-01,
        -2.44929360e-16])

>>> # multiplication, addition and raising-to-the power are also ufuncs
>>> 3*a + a**2
array([ 0.        , 4.23821536, 10.66967615, 19.29438236,
        30.11233399, 43.12353105, 58.32797353])
```
Universal functions (ufuncs)

- Standard ufuncs are typically implemented in compiled C code
- ⇒ they are fast
- Example: multiplying by 2

```python
>>> # Using list comprehension:
>>> a = range(1000)
>>> %timeit [2*n for n in a]
10000 loops, best of 3: 65.3 μs per loop
>>> # Using an array with multiplication ufunc
>>> a = np.arange(1000)
>>> %timeit 2*a
1000000 loops, best of 3: 1.59 μs per loop
```

- i.e. the latter is approximately 40 times faster
Vectorization

- **Vectorization**: the practice of replacing for loops with array expressions ...
- ... so that batch operations are implemented efficiently.
Exercise

- Generate 1000 random numbers from the standard uniform distribution \([0, 1)\)
- Compute the variance of your sample using:
  1. for loops and lists
  2. arrays and vectorization
- Compare the efficiencies of the implementations.

\[
\text{var}(x) = \langle x^2 \rangle - \langle x \rangle^2
\]
Implementation #1 (with loops)

```python
import random as rnd

def attempt1(n=1000):
    l = []
    for i in range(n):  # Build a list of random numbers
        l.append(rnd.random())

    xsum = 0; x2sum = 0;
    for i in range(n):
        xsum += l[i]
        x2sum += l[i]**2

    xmean = xsum / n
    x2mean = x2sum / n

    return x2mean - xmean**2
```
Implementation #2 (vectorized)

```python
import numpy as np

def attempt2(n=1000):
    r = np.random.random(n)
    xmean = r.sum() / n
    x2mean = (r**2).sum() / n
    return x2mean - xmean**2
```
Comparing efficiencies

```python
>>> %timeit attempt1()
1000 loops, best of 3: 372 us per loop
>>> %timeit attempt2()
100000 loops, best of 3: 17.8 us per loop
```

- The vectorized version is $\sim 20$ times faster.
Broadcasting

What is broadcasting?

- An efficient way of combining two arrays of different shapes during arithmetic operations.
- The smaller array is ‘broadcast’ across the larger array.
- The loops are carried out in C rather than Python ⇒ **fast!**
Broadcasting

Example: Arrays of same size

- If the arrays are the same size, they are combined element-by-element:

```python
>>> a = np.array([1,2,3])
>>> b = np.array([4,5,6])
>>> a * b
array([4, 10, 18])

>>> a + b
array([5, 7, 9])

>>> a - b
array([-3, -3, -3])

>>> a**b
array([1, 32, 729])
```
Broadcasting

Example: Scalar × vector

```python
>>> a = np.array([1,2,3])
>>> b = 2
>>> a*b
array([2, 4, 6])
```
Broadcasting

A more interesting example

```python
>>> x = np.arange(4)
>>> xx = x.reshape(4, 1)
>>> y = np.arange(5)
>>> print(x.shape, xx.shape, y.shape)
(4,) (4, 1) (5,)
>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4,)
>>> xx + y
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```
Broadcasting

**Broadcasting rules** for combining two arrays:

- Their shapes are compared, starting with last dimension
- Two dimensions are compatible iff:
  - They are the same, or
  - One or both of them is 1.
- If dimensions are not compatible, an error is thrown
- **Note:** the arrays do not need to have same number of dimensions
- Example:
  
  A (4d array): 8 x 1 x 6 x 1  
  B (3d array): 7 x 1 x 5  
  A*B (4d array): 8 x 7 x 6 x 5

- More info: [http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html](http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)
Warning!

Broadcasting is not the same as matrix multiplication! For example,

```python
>>> A = np.arange(4).reshape(2,2)
>>> b = np.arange(1,3).reshape(2,1)
>>> print(A)
[[0 1]
 [2 3]]
>>> print(b)
[[1]
 [2]]
>>> A * b  # Warning : not matrix multiplication
array([[0, 1],
        [4, 6]])
```
Matrix multiplication

For matrix multiplication, use `np.dot()` or first convert arrays to matrices:

```python
>>> np.dot(A, b)
array([[2],
       [8]])

>>> np.mat(A) * np.mat(b)
matrix([[2],
        [8]])
```
Linear algebra with `numpy.linalg`

http://docs.scipy.org/doc/numpy/reference/routines.linalg.html

- Matrix & vector multiplication with `np.dot` function
- The dot product $a \cdot b$:

```python
>>> a = np.array([1, 2])
>>> b = np.array([3, 4])
>>> np.dot(a, b)  # dot product
11
```

- Matrix multiplication:

```python
>>> A = np.array([[1, 2], [3, 4]])
>>> b = np.array([5, 6])
>>> np.dot(A, b)  # matrix-by-vector
array([17, 39])
>>> np.dot(A, A)  # matrix-by-matrix
array([[ 7, 10],
       [15, 22]])
```
Linear algebra with `numpy.linalg`

- **Matrix determinant & inverse**

  ```python
  >>> import numpy.linalg as la
  >>> A = np.array([[1,2], [3,4]])
  >>> la.det(A)
  -2.0000000000000004
  >>> la.inv(A)
  array([[-2. , 1. ],
         [ 1.5, -0.5]])
  >>> np.dot(A, la.inv(A))
  array([[ 1.00000000e+00, 0.00000000e+00],
         [ 8.88178420e-16, 1.00000000e+00]])
  ```

**Exercise**

Find out how to:

- Find eigenvalues and eigenvectors
- Solve a set of linear equations.
Challenge: the Mandelbrot set

- Use `numpy` arrays, with universal functions/vectorization and broadcasting, to make an image of the Mandelbrot set.

- The Mandelbrot set is the set of complex numbers $c$ for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$. 
# Python code to plot the Mandelbrot set, using
# numpy arrays, broadcasting, vectorization and universal functions.

import numpy as np
import matplotlib.pyplot as plt

# Part (a).
xmin = -1.5; xmax = 0.5; ymin = -1; ymax = 1.0; # the domain
npts = 501; # number of points on each axis
xs = np.linspace(xmin,xmax,npts) # real parts
ys = np.linspace(ymin,ymax,npts, dtype=np.complex128)*1j # imag parts

# Now make a 2D array by broadcasting two 1D arrays
Cs = xs.reshape((1,npts)) + ys.reshape((npts,1))
# Part (b)
zs = np.zeros((npts,npts), dtype=np.complex128)

# Part (c).
nits = 100
for i in range(nits):
    zs = zs**2 + cs   # using a ufunc

# Part (d).
maxval = 100.0
mandelbrot = np.abs(zs) < maxval

# Part (e)
fig = plt.imshow(mandelbrot, cmap='Purples', origin='lower')
plt.axis('off')
plt.savefig("mandelbrot.png")
Functions

- A function is like a ‘black box’ that takes one or more inputs (arguments or parameters) and produces one output.
- (Since the output may be a container type (e.g. list), it can actually produce several outputs).
- New functions are defined with the `def` and `return` keywords. Example:

```python
>>> def square(i):
...     return i**2
...

>>> square(7)  # try the function
49
>>> square(7.0)
49.0
```

- Try passing a list or `str` data type to `square` – what happens?
- More info: https://docs.python.org/release/1.5.1pl/tut/functions.html
An example function: Fibonacci sequence

Let’s try a function to compute the Fibonacci sequence from the recurrence relation $f_{k+1} = f_k + f_{k-1}$:

```python
>>> def fibonacci(n=10):
...     """Computes a list of the first n Fibonacci numbers."""
...     l = [0, 1]
...     for i in range(n-1):
...         l.append(l[-1] + l[-2])
...     return l
...
```

Example output

```python
>>> fibonacci(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```
An example function: Fibonacci sequence

- The ratio of successive terms should tend towards the Golden Ratio \((\sqrt{5} + 1)/2\). Let's check this:

```python
>>> l = fibonacci(100)
>>> l[-1] / float(l[-2])  # an approximation to the Golden Ratio
1.618033988749895

>>> (5**0.5 + 1)/2.0  # the true Golden Ratio
1.618033988749895
```
Functions: optional parameters

- Functions can have *optional* named parameters. Example:

```python
>>> def raisepower(a, power=2):
...    # Note that 'power' is assigned a default value of 2
...    return a**power
...
>>> raisepower(3)
9
>>> raisepower(2, power=3)
8
>>> raisepower(2, 0.5)
1.4142135623730951
```

- The function may be called *without* specifying optional parameters, or,

- optional parameters may be set by name, or in order.
Functions: docstrings

At the start of a function, you may write a (multiline) docstring to explain what the function does:

```python
>>> def raisepower(a, power=2):
...     '''This is a docstring.
...     This function raises the first parameter to
...     the power of the second parameter.'''
...     return a**power
... 
>>> help(raisepower)
```

Now the docstring should appear in the 'help' for the function.

In ipython, there is enhanced help. Try entering ?raisepower.
In Python, functions are allowed to modify their parameters. Example:

```python
>>> def add_animal(a):
...     a.append("aardvark")
...     
...     
...     l = ["horse"]
>>> add_animal(l)
>>> l
['horse', 'aardvark']
```
Q. Are Python arguments passed by value or by reference?

A. By value, but the value is a reference to an object …

… and so if the object is mutable, it may be changed by the function.
Functions: scope

- A function can define, use and modify local variables . . .
- . . . and can use global variables defined elsewhere.
- When the function ends, all local variables fall out of scope.

- If there are local and global variables with the same name, Python will use the local variable.

**LEGB**: Python checks Local then Enclosing (nested) then Global then Built-in scope.

```python
>>> def fn():
...     a = 4
>>> a = "hello"  # global variable with the same name
>>> fn()
>>> a  # the global variable was not changed
'hello'
```
Functions: scope

- To allow a function to modify a global variable, use the `global` keyword:

```python
def fn1():
    x += 1

def fn2():
    global x
    x += 1

>>> x = 3
>>> fn1()
UnboundLocalError: local variable 'x' referenced before assignment
>>> fn2()
>>> x
4
```

- (Or, pass the global variable as an argument).
Simple file Input/Output

- Open a file for writing:

  ```
  >>> f = open('dickens.txt', 'w')
  ```

- `f` is a an object with attributes:

  ```
  >>> f.name
  'dickens.txt'
  >>> f.closed
  False
  ```
Simple file I/O

- **Write a line of text:**

  ```python
  >>> s = ""
  ... It was the best of times,
  ... it was the worst of times.
  ... ""
  >>> f.write(s)
  ```

- **Close the file**

  ```python
  >>> f.close()
  >>> f.closed
  True
  ```
Open the file to read the contents:

```python
>>> f = open('dickens.txt', 'r')
>>> s = f.read()  # Read into a string
>>> f.close()
>>> print(s)

It was the best of times,
it was the worst of times.
```

https://docs.python.org/3/tutorial/inputoutput.html
os module

Example:

```python
>>> import os
>>> os.getcwd()  # current directory
'/Code/ipython_notebooks'
>>> os.mkdir('temp')  # Make a new directory
>>> os.chdir('temp')  # Move to that directory
>>> os.chdir('..')  # Move back again
>>> os.rmdir('temp')  # Remove the directory
```

https://docs.python.org/3/library/os.html
Objects

- Python is object-orientated
- An **object** is created (‘instantiated’) from a blueprint called a **class**.
- The class defines the object’s **attributes** and **methods**.
- Methods are functions defined by the class
- Syntax: `<object>..<method>(<arguments>)`
- Everything in Python 3 is an object (numbers, lists, arrays, etc.)
Objects: an example

```python
>>> z = 1 + 2j  # Python instantiates a new 'complex' object
>>> type(z)
<class 'complex'>
>>> z.imag     # This is an attribute
2.0
>>> z.conjugate() # This is a method
(1-2j)
```
Standard modules: math

- The math module provides basic mathematical constants and functions.
- https://docs.python.org/3/library/math.html#module-math

Constants

- \( \pi = 3.141 \ldots \)
- \( e = 2.718 \ldots \)

Basic functions

- \( \exp(x) = e^x \), \( \log(x) = \ln(x) \), \( \sqrt{x} = \sqrt{x} \)

Trigonometric & hyperbolic functions

- Functions: \( \cos \), \( \sin \), \( \tan \), \( \cosh \), \( \sinh \), \( \tanh \)
- Inverses: \( \arccos \), \( \arcsin \), \( \arctan \), \( \arccosh \), \( \arcsinh \), \( \arctanh \)
Namespace dangers

- In Python it is very easy to accidently supercede functions or variables.
- **Example:** suppose we wished to sum the numbers 1 to 10

```python
>>> sum = 0  # set a counter variable to zero
>>> for i in range(11):
...    sum += i
...
>>> print(sum)
55
```

- This looks fine, **but** ... by choosing the variable name `sum`, we have accidently **superceded** the in-built function called `sum` (!)
Namespace dangers

- If we try using `sum` as a function after the previous code:

```python
>>> sum(range(11))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

- A confusing error message indeed!
- This is a quirk of Python we have to live with

**Good practice**

Avoid `from module import *` statements.

- **Good:** `import math` then later `print(math.pi)`
- **Bad:** `from math import *` then later `print(pi)`
The `cmath` module

- The `cmath` module provides functions for complex arithmetic.
- Use `cmath` instead of `math` module, if appropriate
- [https://docs.python.org/3/library/cmath.html#module-cmath](https://docs.python.org/3/library/cmath.html#module-cmath)
- `1j` is unit imaginary

**Example:** check that

\[
\exp \left( \ln(2) + i\pi/2 \right) = e^{\ln 2} e^{i\pi/2} = 2i
\]

```python
>>> import cmath
>>> z = cmath.log(2.0) + 1j * cmath.pi / 2.0
>>> cmath.exp(z)
(1.2246467991473532e-16+2j)
```
random module

Random number generation

- `seed()` : initialize the random number generator
- `random()` : float in range 0 to 1
- `randint(1, 10)` : integer in range 1...10
- `shuffle(l)` : in-place shuffle of a list `l`
- `gauss(m, s)` : random float drawn from a Gaussian distribution with mean `m` and standard deviation `s`.
random module

Example: Generating and shuffling a random string of letters

```python
>>> import random
>>> letters = [chr(random.randint(65,90)) for k in range(10)]
>>> for k in range(5):
...     random.shuffle(letters)
...     print(''.join(letters))

EHUMMFWSDJ
DSMUHJFJWM
HMGJEDWUF
SFMJWUHMDE
MUMJWFSDDE
```
datetime module

```python
>>> import datetime
>>> today = datetime.datetime.today()  # get time and date now
>>> today
datetime.datetime(2014, 10, 7, 14, 4, 10, 47725)
>>> today.weekday()  # 0 = Monday, 6 = Sunday
1
```

https://docs.python.org/3/library/datetime.html