Introduction to numerical computing with Python

Jonathan Boyle, Louise Lever
(& Mike Croucher)
IT Services
its-research@manchester.ac.uk

Before we start….

- This talk is intended to introduce Python
  - And assumes you want to do ‘MATLAB like stuff’
- It is not an exhaustive description of anything
  - But aims to take you to the point you can
    - Write Python programs
    - Do interactive calculations at the prompt
    - Use the IPython notebook

Disclaimer

- This is an introduction to Python
  - It does not cover everything
  - There is far too much for a 1 day course
- By the end of the day you should understand
  - The basics of writing simple Python programs
    - e.g. we don’t cover **object oriented programming** today
  - How to use Python interactively
  - Some useful topics for further reading
- After this course you should also
  - Read a good book
    - e.g. Langtangen’s *“A Primer on Scientific Programming with Python*”
  - Browse the Python documentation
  - Practice programming regularly
  - Keep up to date with developments in Python

Some reasons to use Python

- Easy to learn and get going
  - Includes Python shell (i.e. interactive command line prompt)
  - The language is designed with readability in mind
- A widely used general purpose language
  - Not just for numerical/research work
  - May aid employability
- Free & open source
  - Allows code to be freely shared and used
- Includes good numerical & plotting support
  - Including vector and matrix operations
- Includes tools (e.g. Cython) to translate code to C
  - Python is interpreted (via intermediate representation)
  - Converting to C helps create faster executables
    - There are other methods to get speed up too

Finding out more

- Where to find how to do things in Python?
  - Internet searches
    - There is a vast amount of material
  - Books
    - e.g. *"A Primer on Scientific Programming with Python"* was used to write this course
      - eBook available at link.springer.com/
      - Free for UoM members
  - The Python documentation
    - www.python.org/
- Python is evolving
  - You need to keep up to date
    - Especially with respect to packages

Official Python Documentation

- Lots of info & links at www.python.org
- Tutorials
  - e.g. for Python v2 docs.python.org/2/tutorial
- The Python Language Reference
  - docs.python.org/2/reference/index.html
  - Describes syntax and core semantics
- The Python Standard Library
  - docs.python.org/2/library/index.html
  - Non-essential built-in object types
  - Built-in functions and modules
    - e.g. maths, cmaths
  - Some optional components

Installing Python

Which version to use?

- Python has versions 2 and 3
  - These are not compatible!
    - But 2.7 is close to 3
  - No more major releases for v2
    - v3 is expected to be the future of Python
  - Some ‘add on’ packages not yet ported to v3
    - Also a lot of legacy code is written in v2
  - Can write code in v2.7, port to v3 using tools
    - e.g. *2to3*
  - Or write code that works with v2 & v3
    - e.g. use *six* python compatibility package

Python or IPython

- Python provides an interactive shell
- IPython has enhanced interactive Python shells
  - Additional functionality
- IPython also provides additional functionality
  - e.g. a web based notebook
- We'll look at IPython in more detail this afternoon

Which Python Packages?

- Install to add functionality to core Python
- Python Package Index (PyPI)
  - pypi.python.org/pypi
  - 26839 packages (Jan 2013)
  - 40297 packages (Feb 2014)
- Preferred package installer is called **`pip`**
- Probably only need few packages to start e.g.
  - NumPy, SciPy, matplotlib, IPython
- Some not available for Python 3
- Or some available only as source code
  - Usually easy to download and compile using `pip`

Which Python Packages?

IP[y]: IPython
Interactive Computing

matplotlib

scikits-image
image processing in python

NumPy

python™

SymPy

SciPy

pandas

scikits learn

Considerations when installing

- What do I install?
  - Precompiled binaries or compile the source code?
    - Probably easiest to start with precompiled binaries
    - But which precompiled version?
- Which add-on packages?
- How to test an install
  - Essential if compiling source code
  - What if tests fail?
- Install questions?
  - Ask your local research support teams for help
    - i.e. applicationsupport-eps@manchester.ac.uk

Python Distributions

- A single install to get Python & many packages ☺
- Free and paid distributions exist
- Some free science/engineering distributions are:
  - Anaconda
    - Linux, Mac OSX, Windows
    - Available from Continuum Analytics
  - Enthought Canopy (currently no Python 3)
    - Linux, Mac OSX, Windows
    - From Enthought
- More info on UoM Applications Website
  - www.applications.itservices.manchester.ac.uk

Python Distributions

- We recommend Anaconda Python
- Available on many University systems
  - Available for University Windows 7 managed desktop
    - Installed on various teaching clusters
  - The EPS Linux Image & Condor pool
  - The CSF

- At Manchester, we recommend Anaconda Python. Installed on many campus facilities.

- apptools 4.2.1
- argcomplete 0.6.7
- astropy 0.3.0
- atom 0.3.7
- beautiful-soup 4.3.1
- binstar 0.4.4
- biopython 1.63
- bitarray 0.8.1
- blaze 0.4.2
- blz 0.6.1
- bokeh 0.4.1
- boto 2.25.0
- cairo 1.12.2 *L*
- casuarius 1.1
- cdecimal 2.3
- chaco 4.4.1
- colorama 0.2.7
- conda 3.4.1
- conda-build 1.3.1
- configobj 4.7.2
- cubes 0.10.2
- curl 7.30.0 *LM*
- cython 0.20.1
- datashape 0.1.1
- dateutil 2.1
- disco 0.4.4 *L*
- docutils 0.11
- dynd-python 0.6.1
- enable 4.3.0
- enaml 0.9.1
- envisage 4.4.0
- erlang R15B01 *L*
- flask 0.10.1
- freetype 2.4.10
- future 0.11.2
- gevent 1.0
- gevent-websocket 0.9.2
- gevent_zeromq 0.2.5
- greenlet 0.4.2
- grin 1.2.1
- h5py 2.2.1
- hdf5 1.8.9
- ipython 1.1.0
- itsdangerous 0.23
- jinja2 2.7.2
- keyring 3.3
- kiwisolver 0.1.2
- launcher 0.1.2
- libnetcdf 4.2.1.1 *LM*
- libpng 1.5.13 *LM*
- libsodium 0.4.5 *L*
- libtiff 4.0.2 *LM*
- libxml2 2.9.0 *LM*
- libxslt 1.1.28 *LM*
- llvm 3.3
- llvmpy 0.12.3
- lxml 3.3.1
- markupsafe 0.18
- matplotlib 1.3.1
- mayavi 4.3.1
- mdp 3.3
- menuinst 1.0.3 *W*
- mingw 4.7 *W*
- mock 1.0.1
- mpi4py 1.3 *L*
- mpich2 1.4.1p1 *L*
- netcdf4 1.0.8 *LM*
- networkx 1.8.1
- nltk 2.0.4
- nose 1.3.0
- numba 0.12.1
- numexpr 2.3.1
- numpy 1.8.0
- opencv 2.4.6 *L*
- openpyxl 1.8.2
- openssl 1.0.1g *LM*
- pandas 0.13.1
- patsy 0.2.1
- pep8 1.4.6
- pil 1.1.7
- pip 1.5.2
- ply 3.4
- psutil 1.2.1
- py 1.4.20
- py2cairo 1.10.0 *L*
- pyaudio 0.2.7 *M*
- pycosat 0.6.0
- pycparser 2.10
- pycrypto 2.6.1
- pycurl 7.19.0 *LM*
- pyface 4.4.0
- pyflakes 0.7.3
- pygments 1.6
- pykit 0.2.0
- pyparsing 2.0.1
- pyreadline 2.0 *W*
- pysal 1.6.0
- pysam 0.6 *LM*
- pyside 1.2.1
- pytables 3.1.0
- pytest 2.5.2
- python 2.7.6
- pytz 2013b
- pywin32 218.4 *W*
- pyyaml 3.10
- pyzmq 2.2.0.1
- qt 4.8.5
- redis 2.6.9 *LM*
- redis-py 2.9.1 *LM*
- requests 2.2.1
- rope 0.9.4
- scikit-image 0.9.3
- scikit-learn 0.14.1
- scipy 0.13.3
- setuptools 2.2
- six 1.5.2
- sphinx 1.2.1
- spyder 2.2.5
- sqlalchemy 0.9.2
- ssl_match_hostname 3.4.0.2
- statsmodels 0.5.0
- sympy 0.7.4.1
- theano 0.6.0 *L*
- tk 8.5.13 *LM*
- tornado 3.2.0
- traits 4.4.0
- traitsui 4.4.0
- ujson 1.33
- vtk 5.10.1
- werkzeug 0.9.4
- xlrd 0.9.2
- xlsxwriter 0.5.2
- xlwt 0.7.5
- yaml 0.1.4 *LM*
- zeromq 2.2.0 *LM*
- zlib 1.2.7

Course Outline

- Introduction to Python Programming
  - Flow control
  - Functions
- Scientific Python
  - NumPy and SciPy
  - Plotting
- Interactive Python
  - IPython
  - IPython Notebook

The Python programming language

keywords

- To get a list of the python keywords

```
>>> help('keywords')

Here is a list of the Python keywords.  Enter any keyword to get more
help.

and                 elif                if                  print
as                  else                import              raise
assert              except              in                  return
break               exec                is                  try
class               finally             lambda              while
continue            for                 not                 with
def                 from                or                  yield
del                 global              pass
```

Variables

- Variable names
  - Can contain:
    - Any upper and lower case letter
    - Numbers 0 to 9
    - Underscore _
  - Can't start with a number
  - Are case sensitive
- Assign using `=`

```
>>> my_variable = 3.14
>>> my_variable
3.14
```

- Remove using `del`

```
>>> del my_variable
```

A warning

- You can overwrite built in names
  - e.g. in Python 2
    - Note: this example doesn’t work in v3

```
>>> True == False
False
>>> True = False
>>> True == False
True
```

float & int

- Python automatically assigns numeric types
- Integers (at least 32 bits) have no decimal point

```
>>> type(1)
<class 'int'>
```

- Floating point (double precision)

```
>>> type(1.0)
<class 'float'>
```

A warning about division

- In Python 3

```
>>> type(1/2)
<class 'float'>
>>> 1/2
0.5
```

- And in Python 2

```
>>> type(1/2)
<class 'int'>
>>> 1/2
0
```

Complex numbers

```
>>> cn1 = 10 + 4j
>>> type(cn1)
<type 'complex'>
>>> cn1
(10+4j)
>>> cn1.real
10.0
>>> cn1.imag
4.0
>>> cn1.conjugate()
(10-4j)
```

Type casting

```
>>> int(1.9)
1
>>> int(1.1)
1
>>> float(2)
2.0
>>> long(10.5)
10L
>>> type(10L)
<type 'long'>
```

Practical session

Simple exercises: Part 1

Modules
&
Namespaces

Modules

- Modules add to the core Python functionality
- Some are built in
  - The standard modules library
    - See the Python Library Reference for more info
- Some come in packages
  - e.g. NumPy, SciPy
- And you can write your own
  - Good practice for modular code

Importing modules

```
>>> sqrt(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined

>>> import math
>>> math.sqrt(4)
2.0

>>> del math          # to remove module
```

Importing module functions

```
>>> from math import sqrt
>>> sqrt(4)     # now don't need math.
2.0

>>> from math import sqrt, exp, log, sin
>>> from math import *   # imports everything
```

Importing & renaming

```
>>> import math as m
>>> m.sqrt(4)
2.0
>>> from math import sin as s, cos as c
>>> s(0)
0.0
>>> c(0)
1.0
```

Complex maths

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error

>>> import cmath
>>> cmath.sqrt(-1)
1j
>>> cmath.sqrt(1)
(1+0j)
```

Name Spaces

- `>>> import math, cmath`
- This accesses the math and cmath namespaces
  - math objects start with `math.`
    - e.g. `math.sqrt()`
  - cmath objects start with `cmath.`
    - e.g. `cmath.sqrt()`
- Name spaces allows the safe reuse of names in different modules ☺

A reminder

```
>>> from math import sqrt
>>> sqrt(4)
2.0

>>> sqrt=int
>>> sqrt(4)
4

>>> del sqrt
```

Navigating directories

- In Python

```
>>> import os
>>> os.getcwd()
'/home/user'
>>> os.chdir("/tmp/")
>>> os.getcwd()
'/tmp'
```

- In IPython (covered later) it is much easier
  - We can use `pwd`, `cd`, etc

Some useful commands

- `locals()`
  - Returns local variables including modules
- `dir()`
  - Lists names currently defined
    - i.e. variables, **modules**, functions, etc
  - Can list names in a module, e.g. cmath
    - `dir(cmath)`
- `help()`

help()

- `help()`
  - Starts Python help
- Includes help on modules, e.g. for NumPy package
  - `help('numpy')`
- Help on commands e.g.
  - `help('help')`
- Sometimes odd behaviour, e.g. v2

```
>>> help('print')
no documentation found for 'print'
```

  - Print is a ‘special statement’ in v2

Practical session

Simple exercises: Part 2

Control flow

Boolean values

```
>>> False == 0
True
>>> True == 1
True
>>> type(True)
<class 'bool'>
>>> True + True
2
>>> type(True + True)
```

Relational Operators

- == tests for equality
- != for inequality
- Also >, < , <=, >=

```
>>> 1>10
False
>>> False == True
False
>>> 1 >= 10 > 1
False
```

Boolean operators

- `and`, `or`, `not`

```
>>> True and False
False
>>> True or False
True
>>> not True
False
```

Indenting code blocks

- Indenting is **very important**
  - Code blocks **must** be indented
  - Code blocks **must** have the same indent
    - Tab and space are considered different
  - **This is not optional, blocks must be indented**
  - Blocks can be nested
  - e.g.
    - while loops
    - for loops
    - Functions
    - if, elif, else constructs

Branching

- `if,elif,else`
  - Remember: indenting is important
  - Colons mark the start of a block

```
x = int(raw_input("Please enter an integer: "))

if x < 0:
  print "Value is negative"
elif x == 0:
  print "Value is zero"
elif x == 1:
  print "value is 1"
else:
  print "Value is positive"
```

`while` loops

- Repeat a code block as long as some expression returns true
  - Start with the `while` keyword followed by expression and a colon
    - The colon marks the start of the loop block
  - All code in the loop must be indented with the same indent
- e.g. to convert Celsius to Fahrenheight

```
C = -20
dC = 5
while C <= 40:
    F = (9.0/5)*C + 32
    print C, F
    C = C + dC
```

- Note: this is Python 2 code
  - Use `print(C, F)` for Python 3

`for` loops using lists

- For loops can iterate through Python lists
  - e.g. print all values from a list called `my_list`

```
for i in my_list:
    print i
```

- Note:
  - The body of loop is indented
  - There is a loop variable `i`
  - `i` takes a copy of each value in the list in turn

Lists

- Lists are general purpose collection of objects
  - Each entry is accessed via an integer index
  - First element has index `0`
- Create using comma separated values in square brackets
  - Can be nested [[…] , […] , […]]

```
>>> list2 = [1.0, "hello", 1]
>>> list2[1]
'hello'
>>> list2[1] = 4+423j
>>> list2
[1.0, (4+423j), 1]
```

Efficiency

- Some list operations are efficient
  - e.g. insertion, deletion, appending values, etc
- Lists don’t support vectorised operations
  - Use NumPy arrays instead
- Lists require more memory
  - Compared to the equivalent NumPy array

List slices

- `list2[i:]`
  - Elements from index `i` to last value in the list
- `list2[i:j]`
  - Elements from index `i` to **`j-1`**
  - e.g.

```
>>> list2 = [1.0, "hello", 1]
>>> list2[0:3]
[1.0, 'hello', 1]
```

Negative indices

- Negative indices count from the end of the list
  - `-1` is the last element of a list
  - `-2` is second last
  - etc
- List slices
  - `[i:-j]` gives elements from `i` to `-j-1`
  - e.g. for all elements except first and last
    - `list2[1:-1]`

Copying lists

- Be careful
  - To copy values into a new list
    - `list1 = list2[:]`
    - list1 contains a copy of the values in list2
  - To assign an additional name to an existing list
    - `list1 = list2`
    - list1 and list2 point to the same list

Some list functionality…

- Insert data by extending the list
  - `list1.extend()` or `list3 = list1 + [4,5]`
    - Adds values at the end
  - `list1.insert(…)`
    - Inserts data before a specific location
- Get the list length
  - `len(list3)`
- Delete element 2
  - `del list1[2]`
- Create an empty list
  - `list4 = []`
- Return index of first item matching `"hello"`
  - `list2.index("hello")`

Preallocating lists

- May improve performance

```
>>> somelist = [0]*10
>>> somelist
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> somelist = [0.]*5
>>> somelist
[0.0, 0.0, 0.0, 0.0, 0.0]
```

`range()`

- Generate a list of integers e.g.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Can define start, end and step values

```
>>> range(4, 100, 2) # 4 to 98 in steps of 2
```

- range is useful with for loops

```
for i in range(len(list2)):
    list2[i] += 2
```

`range()` or `xrange()`?

- Python 2
  - `range()` returns a list
  - `xrange()` returns an xrange object
    - Returns same values as corresponding list
    - But doesn’t store values **so uses less memory**
- Python 3
  - `range()` behaves as per xrange in Python 2
    - i.e. returns a range object
  - There is no `xrange()` function

`for` loops with `zip` & `enunerate`

- To operate on multiple lists simultaneously
  - e.g. get `e1` from `list1` & `e2` from `list2`

```
for e1, e2 in zip(list1, list2):
```

- Get loop counter and values simultaneously

```
for i, c in enumerate(range(10)):
    cube[i] = c**3
```

List comprehension

- Alternative for loop

```
newlist = [<expression> for x in somelist]
```

  – `<expression>` is some expression involving `x`
- e.g.
  – A list contains values 0 to 9
    - `>>> list1 = range(10)`

  – Create a list containing squares of these values

```
>>> list2 = [x**2 for x in list1]
>>> list2
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- List comprehension may be more efficient

Tuples

- `tup1 = (1,2)`
- Behaves like a list but values are constant
  - Can reassign e.g. `t1 = t1 + t1`
- Can usually drop parenthesis
  - `tup2 = 35, "hi", 12431.1`
- May be faster than lists
- Reminder: square brackets for lists []

Writing and running programs

- Use a text editor to write a script
  - Save to a file with a `.py` extension
    - e.g. `hello.py`
- Can run from operating system terminal window

```
python hello.py
```

- Can't run script directly from Python
  - Note: can run programmes directly in **IPython**
    - Use `run hello.py`
- Anything following `#` is treated as a comment
- `/` denotes a line continuation

<u>Practical session</u>

Simple exercises: Part 3

Functions

Functions

- Can insert anywhere in a program
  - Or even define at the prompt
- Function blocks must be indented
  - e.g. define a function at the prompt with:
    - 2 input arguments `x` & `y`
    - 1 output argument $x^3 + y^2$

```
>>> def f(x, y):
>>>     return x**3 + y**2

>>> f(10,10)
1100
```

Functions are first-class objects

- Use function names like any variable e.g.
  - Add functions to lists

```
>>> import numpy as np
>>> list = [np.sin, np.cos]
>>> list[1](0)
1.0
>>> list[0](0)
0.0
```

  - Pass functions to other functions
    - Via input arguments

Local and global variables

- Variables created outside functions are global
  - Can be accessed within functions
- Variables created in functions are local
  - Unless explicitly set to be global
  - Can only be used in that function
  - Are deleted when the function ends
  - Local variables hide globals with the same name

Default input arguments

- Can define default values
  - i.e. optional arguments
- For example

```
def func(x=1, y=2, z=3):
```

- Non-default arguments **must not** follow defaults
  - This is allowed

```
def func(x, y=2, z=3):
```

  - This is **not** allowed

```
def func(x=1, y=2, z):
```

Keyword arguments

- Can call arguments by name
  - e.g. `val = f(x=1, y=10)`
  - Equivalent to `val = f(y=10, x=1)`
- Non-keyword arguments **must not** follow keyword arguments
  - This is **not** allowed

    ```
    f(x=2,100)
    ```

  - This is allowed

    ```
    f(100,y=2)
    ```

Output arguments

- Can have multiple return values
  - e.g. `return f, g`
    - Returns a tuple
- `return` not required
  - Can pass back no arguments
  - Equivalent to `return None`
  - `None` is a Python object
    - Used to represent nothing

Lambda functions

- Simple one line functions
  - e.g.

```
>>> f = lambda x : x**3 + 1
>>> f(10)
1001
>>> f = lambda x,y : x**3 + y
>>> f = lambda x,y=1: x**3 + y
```

Create your own modules

- Save your functions to a file
  - Give the file a name with a .py extension
    - This name (without the extension) is the module name
- Import the module as described earlier

```
>>> import MyModule
```

- File can contain executable statements
  - e.g. to initialize the module
    - Executed the first time the module is imported
- More info
  - docs.python.org/2/tutorial/modules.html

Debugging

Debuggers

- Set stop points
  - Execution halts at the stop point
  - Can examine variables
  - Stop points can be conditional
- Step through code line by line
  - Step into function calls
  - Or step over functions
- Move through the stack trace
  - e.g. to analyse variables in nested functions calls

Starting the debugger

- It is usually easiest to debug via an IDE e.g.
  - Spyder
  - Visual Studio
    - First install PythonTools For Visual Studio
- IDE = integrated development environment
- Alternatively use from the shell
  - Python shell

    ```
    python -m pdb myscript.py
    ```

  - Ipython shell

    ```
    In [1]: run -d myscript.py
    ```

Debugging from the shell

- To see the current line
  - `l` or `list`
- To print a stack trace
  - `w` or `where`
- To move up or down the stack trace
  - `d` or `down`
  - `u` or `up`

Break points

- To set a break at a function called ‘myfunction’

```
break myfunction
```

- To set a break at line number x

```
break x
```

- Each break point is given a number to identify it
- To disable break point 2

```
disable 2
```

Some debugging commands

- To continue execution (until next break)
  - `c` or `continue`
- To execute next line (stepping into functions)
  - `s` or `step`
- To execute next line (bypassing function code)
  - `n` or `next`
- To examine variables
  - `p` or `print`, e.g. `print a,b,c`
- To check type of variables
  - `whatis`, e.g. `whatis a`
- See the manual for the full debugger functionality

Practical session

Simple exercises: Part 4

NumPy & SciPy

NumPy & SciPy packages

- NumPy
  - http://www.numpy.org
  - General-purpose array-processing package
  - Some basic numerical routines
    - Linear algebra, Fourier transforms, random numbers
- SciPy
  - http://www.scipy.org
  - Includes various modules for scientific computing
    - Statistics, optimization, integration, linear algebra, Fourier transforms, signal processing, image processing, ODE solvers, etc
  - Uses NumPy arrays
- Both included in Anaconda & Enthought distributions

Using NumPy & SciPy modules

- First ensure package is installed
- Then import as described earlier e.g.

```
>>> import numpy
>>> numpy.lib.scimath.sqrt(-1)
1j
>>> numpy.lib.scimath.sqrt(1)
1.0
>> from numpy.lib.scimath import sqrt
```

Efficiency

- Vectorised operations & functions
  - These act on multiple values simultaneously
    - i.e. no need to loop over scalar values
  - These are often far more efficient
    - i.e. faster
  - Not part of base Python
  - Hence use NumPy arrays
- All elements of an array must be same type
- In general access elements as per lists

Some array constructors

- Convert list (or tuple) to an array

```
>>> import numpy as np
>>> array1 = np.array(list2)
```

- Create array of length `n` containing zeros
  - Of type float

```
>>> array2 = np.zeros(n)
```

  - Of type int

```
>>> array4 = np.zeros(10,int)
```

- Create `n` x `n` array of zeros

```
>>> array5 = np.zeros((n,n))
```

- Create `n` values between `first` and `last`

```
>>> array6 = np.linspace(first, last, n)
```

Copying an array

- Correct method: copies values from `x` into a new array

```
>>> a = x.copy()
```

- Incorrect: `a` and `b` point to the same array

```
>>> b = a[:]
```

- e.g.

```
>>> a=np.linspace(1,10,4)
>>> a
array([  1.,    4.,    7.,   10.])
>>> b=a[:]
>>> b[-1] = 1000
>>> a
array([    1.,       4.,       7.,  1000.])
```

Whole array functions

```
>>> import numpy as np, math
>>> a = np.linspace(0, math.pi, 5)
>>> np.sin(a)
array([  0.00000000e+00,   7.07106781e-01,   1.
         7.07106781e-01,   1.22464680e-16])
```

Whole array operators

- Add scalar to all values in an array

```
>>> b = a + 1
```

- Can add two arrays, but be careful
  - To add a to b, store result in a temporary array, then assign values to a

```
>>> a = a + b
```

  - Or to add a to b and immediately assign results to a
    - i.e. more efficient

```
>>> a += b
```

NumPy for MATLAB users

- http://mathesaurus.sourceforge.net/matlab-numpy.html

Practical session

Simple exercises: Part 5

Using Python interactively

Python or IPython?

- IPython shell is designed for interactive use, e.g.
  - Ability to `run` Python programs from the prompt
  - Tab complete
  - Scroll through previous commands
  - Some useful Linux commands are built in
    - `pwd`, `date`, `ls`, `mkdir` etc
  - Can run OS commands
    - Proceed command with `!`
- Debugging
- IPython notebook
- And much more …….
- **But IPython features are not compatible with Python**
  - **Don't include them in Python programs**

Some reasons to use IPython

- Includes a much nicer command line interface
- Has a browser **notebook** for documents containing
  - Code
  - Text
  - Mathematical expressions
  - Plots
  - etc
- Designed to be language-agnostic
  - Supports R, Octave, etc
- See documentation for full functionality
  - ipython.org/

To start IPython

- Use the Start Menu or
  - To run Ipython in a terminal window enter
    `ipython`
  - For IPython GUI
    `ipython qtconsole`
  - For IPython notebook
    `ipython notebook`
- Note different Python & IPython prompts

```
In [1]:                    # IPython prompt
>>>                        # Python prompt
```

Using the IPython notebook

- Runs in a web browser
- Enter code / text into cells
- Then execute the cell
  - Enter Shift-Enter or press "Play" button

Demo

Python command line
& IPython

Practical session

IPython notebook

Plotting

matplotlib

- For plotting
- Backends defines output type
  - Some backends write to file only
    - e.g. AGG backend can only write to file
- To change backend
  - `matplotlib.use('QTAgg')  # Agg and QT`
  - Or set backend parameter in your matplotlibrc file

matplotlib.pylab

- Combines matplotlib.pyplot and NumPy
  - In a single namespace

```
import matplotlib
matplotlib.use('Qt4Agg')

from matplotlib.pylab import *
x = linspace(0, 2*pi, 100)
y = sin(x)
plot(x, y)
savefig('sinefig1.png')
show()
```

Annotating plots

```
x = linspace(0, 4*pi, 100)
y = sin(x)
plot(x, y)
xlabel('x')
ylabel('sin(x)')
axis([0., 4*pi, -1, 1])
title('Plot of sin function')
savefig('sinefig2.png')
show()
```

Multiple lines

```
y1=sin(x)
y2=cos(x)
plot(x,y1,'r-')
hold('on')
plot(x,y2,'b-')
legend(['sin','cos'])
show()
```

PyPlot

```
import numpy as np
x = np.linspace(0, 4*np.pi, 100)
y1 = np.sin(x); y2 = np.cos(x)

import matplotlib.pyplot as plt
plt.subplot(2, 1, 1)
plt.plot(x, y1,'rx')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.subplot(2, 1, 2)
plt.plot(x, y2,'rd')
plt.xlabel('x')
plt.ylabel('cos(x)')
plt.show()
```

<u>Demo</u>

Plotting

Practical session

Feedback via

wiki.rac.manchester.ac.uk/community/Courses/feedback

IPython exercises

Performance

Demo

Performance issues

Profiling

Profiling

- Find where time is spent when your code runs
- Typical techniques
  - Manually insert Python functions to measure time
  - Use Python profilers
    - These identify expensive functions
      - `cProfile` is currently recommended
  - Use `line_profiler` package
    - Identifies expensive lines

Time module

- Useful to time code execution
  - The following will give elapsed time in seconds

```
import time

time1 = time.clock()

time2 = time.clock()
total_time = time2-time1
```

Python's profilers

- Several are built into Python
  - `cProfile` is recommended
  - See documentation
    - e.g. docs.python.org/2/library/profile.html
- Can run from within Python
  - First `import cProfile`
  - Then use `cProfile.run(…)`
  - Saves profile data to file for subsequent analysis
- Or can run from command line e.g.

```
python -m cProfile -o prof.dat myscript.py
```

Viewing profile data

- Can use pstats.Stats from pstats module

```
import pstats
prof = pstats.Stats('prof.dat')
```

- To generate list of top 20 functions sorted by time or cumulative time

```
prof.sort_stats('time').print_stats(20)
prof.sort_stats('cumtime').print_stats(20)
```

- Can plot caller information

```
prof.sort_stats('time').print_callers(…)
```

- And can do far more
  - See the documentation for details

cProfile data

```
Thu May 08 15:25:07 2014    prof.dat

         352602178 function calls (352452575 primitive calls) in 149.718 seconds

   Ordered by: internal time
   List reduced from 1220 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1   39.556   39.556   62.477   62.477 runme.py:417(data_to_flux)
    44214   33.918    0.001   55.397    0.001 C:\Anaconda\lib\site-packages\scipy\sparse\lil.py:365(tocsr)
29497/29488   23.023    0.001   24.287    0.001 C:\Anaconda\lib\site-packages\scipy\sparse\lil.py:85(__init__)
  2000770   10.639    0.000   10.639    0.000 {numpy.core.multiarray.array}
216902108    7.018    0.000    7.018    0.000 {method 'extend' of 'list' objects}
   471712    3.928    0.000    3.928    0.000 {method 'reduce' of 'numpy.ufunc' objects}
112721404/112720781    3.628    0.000    3.629    0.000 {len}
    29477    2.837    0.000   69.164    0.002 C:\Anaconda\lib\site-
packages\scipy\sparse\construct.py:457(bmat)
    14/12    2.342    0.167   36.986    3.082 runme.py:530(easy_lp)
        1    1.501    1.501    1.501    1.501 {_libsbml.SBMLReader_readSBML}
   179148    1.207    0.000    3.295    0.000 C:\Anaconda\lib\site-
packages\numpy\lib\stride_tricks.py:35(broadcast_arrays)
   663297    1.096    0.000    1.497    0.000 C:\Anaconda\lib\site-packages\scipy\sparse\coo.py:195(getnnz)
   147385    1.039    0.000    5.838    0.000 C:\Anaconda\lib\site-packages\scipy\sparse\coo.py:206(_check)
   179148    0.971    0.000    1.614    0.000 C:\Anaconda\lib\site-
packages\numpy\lib\stride_tricks.py:22(as_strided)
147385/103174    0.697    0.000   63.524    0.001 C:\Anaconda\lib\site-
packages\scipy\sparse\coo.py:115(__init__)
    47429    0.672    0.000    0.672    0.000 {method 'index' of 'list' objects}
    89574    0.643    0.000    7.795    0.000 C:\Anaconda\lib\site-
packages\scipy\sparse\lil.py:280(__setitem__)
        3    0.642    0.214   45.176   15.059 runme.py:646(optimize_cobra_model)
  1423521    0.595    0.000   10.766    0.000 C:\Anaconda\lib\site-packages\numpy\core\numeric.py:392(asarray)
   191726    0.592    0.000    0.592    0.000 {numpy.core.multiarray.empty}
```

line_profiler

- Gives line by line profile information
  - Essential to identify expensive lines of code
- First ensure line_profiler is installed
- Then define the functions to be profiled
  - e.g. add `@profile` on lines before relevant `def`

```
@profile
def slow_function(a, b, c):
    ...
```

- See documentation at
  - http://pythonhosted.org/line_profiler/

line_profiler

- Various methods exist to generate profile data
  - One is to use kernprof.py
    - e.g. located at C:\Anaconda\Scripts
  - Run kernprof.py at the command prompt

    ```
    kernprof.py -l myscript.py
    ```
  - This creates a file called myscript.py.lprof

line_profiler

- To view profile data at the command prompt

```
python -m line_profiler myscript.py.lprof
```

- Or can view inside Python, e.g.

```
>>> import line_profiler
>>> profile = line_profiler.load_stats("myscript.py.lprof")
>>> line_profiler.show_text(profile.timings, profile.unit)
```

line_profiler data

```
File: runme.py
Function: convert_sbml_to_cobra at line 602
Total time: 10.2804 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   602                                           @profile
   603                                           def convert_sbml_to_cobra(sbml, bound=INF):
   604                                               """Get Cobra matrices from SBML model."""
   605         6           89     14.8      0.0      model = sbml.getModel()
   606         6        77809  12968.2      0.2      S = sparse.lil_matrix((model.getNumSpecies(), model.getNumReact
   607         6           51      8.5      0.0      lb, ub, c, b, rev, sIDs = [], [], [], [], [], []
   608     16008       389448     24.3      1.1      for species in model.getListOfSpecies():
   609     16002       123339      7.7      0.4          sIDs.append(species.getId())
   610     16002        60812      3.8      0.2          b.append(0.)
   611     16008       442204     27.6      1.3      sIDs = [species.getId() for species in model.getListOfSpecies(
   612     12372       377182     30.5      1.1      for j, reaction in enumerate(model.getListOfReactions()):
   613     35244      1033574     29.3      3.0          for reactant in reaction.getListOfReactants():
   614     22878       196021      8.6      0.6              sID = reactant.getSpecies()
   615     22878       162850      7.1      0.5              s = reactant.getStoichiometry()
   616     22878      1118177     48.9      3.3              if not model.getSpecies(sID).getBoundaryCondition():
   617     22854      1116951     48.9      3.3                  i = sIDs.index(sID)
   618     22854     12040246    526.8     35.4                  S[i, j] = S[i, j] - s
   619     36006      1121865     31.2      3.3          for product in reaction.getListOfProducts():
   620     23640       203308      8.6      0.6              sID = product.getSpecies()
   621     23640       171198      7.2      0.5              s = product.getStoichiometry()
   622     23640      1164428     49.3      3.4              if not model.getSpecies(sID).getBoundaryCondition():
   623     22518      1052664     46.7      3.1                  i = sIDs.index(sID)
   624     22518     11820392    524.9     34.7                  S[i, j] = S[i, j] + s
```

What next?

- Spyder
  - **S**cientific **PY**thon **D**evelopment **E**nvi**R**onment
    - Including editor & debugger
- Cython
  - For optimisation
- Anaconda Python distribution
  - Installs many useful packages
    - Linux, Mac OSX & Windows
  - http://docs.continuum.io/anaconda/index.html

Questions?

Practical session

Do some of the advanced exercises, or write your own program