

MPAGS ASI

An introduction to scientific computing with



http://stevenbamford.com/python_mpags_2013/

Steven Bamford



The University of
Nottingham

Course structure

- Five sessions of ≤ 2 hours each:
 - Thursday 3:00 – 5:00 pm
 - ~1 hour lecture
 - 30 minute working on exercises
 - 15 min reviewing solutions
 - Lecture notes will go online just before each session
- Questions:
 - During exercises
 - Come to my office or skype `srbamford`
 - Email to arrange a meeting (steven.bamford@nottingham.ac.uk)
- Formative assessment by coursework
 - Development of a working Python program related to your studies
 - Submission by 13th December (end of term)

Course prerequisites

- To make the most of this course, you should have:
 - Some programming experience (in any language)
 - Access to a computer with Python installed
 - Preferably a laptop so you can try things out during the sessions
 - You will need (at least) numpy, matplotlib and scipy modules installed
 - See your sys. admin. and the course webpage for installation help
- Ideally you should also have:
 - Some current or upcoming need of a scripting language
 - A piece of real or toy analysis you can try out using Python for

Course aims

- To give you...
 - experience of using a modern scripting language (specifically Python)
 - advice about scientific programming
 - knowledge of the main scientific modules for Python
 - the ability to do basic data analysis tasks in Python
(e.g. data manipulation, plotting, ...)
 - knowledge of some specific tools for scientific computing
(e.g. signal processing, optimisation, ...)
 - an overview of Python's full capabilities
- Not to...
 - teach programming in general or detailed Python usage

Outline

- **Session 1:** Introduction to Python
 - Why (and why not) to use a modern, high-level, scripting language
 - Why Python is awesome
 - Introduction to the language:
 - start-up, syntax, constructs, functions, classes, getting help
 - Good programming practice versus 'thinking aloud'
 - Python 2.x versus 3.x
- **Session 2:** Numerical Python and plotting
 - Numpy
 - Using arrays wisely
 - Plotting:
 - matplotlib and others
- **Session 3:** Scientific Python
 - Scipy
 - optimisation, interpolation, statistics, filtering, integration, ...
 - Other tools
 - GNU Scientific Library, R, Theano, scikit, ...

Outline

- **Session 4: Python for specialists**
 - Python for observers
 - Handling FITS files
 - PyRAF – scripting IRAF with Python
 - Astropy
 - Python for theorists
 - Symbolic algebra (Sage and sympy)
- **Session 5: Extreme Python**
 - Efficiently storing and processing huge amounts of data
 - PyTables
 - Numexpr
 - Multiprocessing
 - Wrapping external libraries and creating the fastest code
 - Cython
 - Web applications
 - Django and others

An introduction to scientific programming with



Session I:
Introduction to Python

Why use a scripting language?

- Modern scripting languages:
 - Python, IDL, R, Perl, Ruby, ...
 - High-level
 - Interactive interpreter
- Ease of use
- Speed of development
- Encourages scripting, rather than one-off analysis
- Permanent record
- Repeatability

Why not?

- If you want fastest possible performance
 - at the expense of everything else
 - You need *highly* parallel code
 - Need low-level control
-
- Unless you are working on a supercomputer or developing operating systems components, these probably don't apply to you
 - Even then, Python could be useful in places (*glue, tests, etc.*)

Why Python is awesome

- Designed to be easy to learn and use – clear syntax
- Well documented
- Powerful, flexible, fully-featured programming language
- Multi-paradigm
- 'Batteries included'
- Comprehensive scientific tools
- Fast, efficient
- Interpreter, introspection
- Runs everywhere, completely free
- You probably already have it


Why learn Python?

- Less stress
- Get more science done
- Widely used and growing popularity
- Throughout academia and industry
 - NASA, AstraZeneca, Google, Industrial Light & Magic, Philips,...
 - Web services, engineering, science, air traffic control, quantitative finance, games, education, data management, ...
- Python programmers in demand
- Easy introduction to general programming concepts

Why not?

- Existing code for your project in another language, but still...

Starting the interpreter

A terminal window with a black background and white text. The prompt is 'steven@ip-10-49-91-86:~\$'. The command 'python' has been entered. The output shows 'Python 2.7.2+ (default, Oct 4 2011, 20:06:09)', '[GCC 4.6.1] on linux2', and a message to type 'help', 'copyright', 'credits' or 'license' for more information. The prompt '>>>' is shown with a white cursor.

```
steven@ip-10-49-91-86:~$ python
Python 2.7.2+ (default, Oct 4 2011, 20:06:09)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

- Also...
 - **IPython**
 - IDLE
 - Pylab
 - ...

Basics

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2.0 # and a comment on the same line as code
4.0
>>> (50-5*6)/4
5
>>> width = 20 # assignment, no type declaration
>>> height = 5*9
>>> width * height
900
>>> x = y = z = 0 # zero x, y and z
>>> y
0
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Scripts

```
2+2
# This is a comment
2+2
2+2.0 # and a comment on the same line as code
(50-5*6)/4
width = 20 # assignment, no type declaration
height = 5*9
width * height
x = y = z = 0 # zero x, y and z
print(y)
```

- Write code in a text editor (ideally one Python aware!)
- Copy and paste into interpreter
- Save in a file and execute from command line:
\$ python test.py
- Save and use interactively in future sessions (import t)
- Create executable files

Strings

```
>>> 'spam and eggs'
'spam and eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> hello = 'Greetings!'
>>> hello
'Greetings!'
>>> print(hello)
Greetings!
>>> print(hello + ' How do you do?')
Greetings! How do you do?
>>> print(hello, 'How do you do?')
('Greetings!', 'How do you do?')
>>> howdo = 'How do you do?'
>>> print(hello+' '+howdo)
Greetings! How do you do?
```

Numbers

```
>>> 10 + 3
13
>>> 10 - 3
7
>>> 10 * 3
30
>>> 10 / 3
3
>>> 10 // 3
3
>>> 10 % 3
1
>>> 10**3
1000
>>> 10 + 3 * 5 # *,/ then +,-
25
>>> (10 + 3) * 5
65
>>> -1**2 # -(1**2)
-1
```

```
>>> 10.0 + 3.0
13.0
>>> 10.0 - 3.0
7.0
>>> 10.0 * 3
30.0
>>> 10.0 / 3
3.3333333333333335
>>> 10.0 // 3
3.0
>>> 10.0 % 3.0
1.0
>>> 10.0**3
1000.0

>>> 4.2 + 3.14
7.339999999999999
>>> 4.2 * 3.14
13.188000000000001
```


Numbers

Augmented assignment:

```
>>> a = 20
>>> a += 8
>>> a
28
>>> a /= 8.0
>>> a
3.5
```

Functions:

```
>>> abs(-5.2)
5.2
>>> sqrt(25)
5.0
```

Comparisons:

```
>>> 5 * 2 == 4 + 6
True
>>> 0.12 * 2 == 0.1 + 0.14
False
>>> a = 0.12 * 2; b = 0.1 + 0.14
>>> eps = 0.0001
>>> a - eps < b < a + eps
True
```

Python 2.x versus 3.x

- New 3.x branch is intentionally backwards incompatible
- Various improvements, removal of obsolete code, but annoying!
- `print a` → `print(a)`
- `1/3 == 1//3 == 0` → `1/3 == 1.0/3.0 == 1.333...`, `1//3 == 0`
- ... various others ...
- Versions ≥ 2.6 contains most backward compatible changes, and can warn of usage (`-3` switch) which would be an error in 3.x
- 2to3 (semi-)automated code translator
- A few useful modules still not compatible with 3.x
- Use either, but 2.7 makes life easiest...

Containers

Lists:

```
>>> a = [1, 2, 4, 8, 16] # list of ints
>>> c = [4, 'candles', 4.0, 'handles'] # can mix types
>>> c[1]
'candles'
>>> c[2] = 'knife'
>>> c[-1] # negative indices count from end
'handles'
>>> c[1:3] # slicing
['candles', 'knife']
>>> c[2:] # omitting defaults to start or end
['knife', 'handles']
>>> c[0:4:2] # variable stride (could just write c[::2])
[4, 'knife']
>>> a + c # concatenate
[1, 2, 4, 8, 16, 4, 'candles', 'knife', 'handles']
>>> len(a)
5
```

Containers

Tuples:

```
>>> q = (1, 2, 4, 8, 16) # tuple of ints
>>> r = (4, 'candles', 4.0, 'handles') # can mix types
>>> s = ('lonely',) # singleton
>>> t = () # empty
>>> r[1]
'candles'
>>> r[2] = 'knife' # cannot change tuples
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> u = 3, 2, 1 # parentheses not necessary

>>> v, w = 'this', 'that'
>>> v
'this'
>>> w
'that'
```

Containers

Dictionaries:

```
>>> a = {'eyecolour': 'blue', 'height': 152.0,
         42: 'the answer'}
>>> a['age'] = 28
>>> a
{42: 'the answer', 'age': 28, 'eyecolour': 'blue', 'height': 152.0}

>>> del(a['height'])
>>> a
{42: 'the answer', 'age': 28, 'eyecolour': 'blue'}

>>> b = {}
>>> b['hello'] = 'Hi!'

>>> a.keys()
[42, 'age', 'eyecolour']
>>> a.values()
['the answer', 28, 'blue']
```

Conditionals

```
>>> a = 4; b = 3
>>> if a > b:
...     result = 'bigger'
...     c = a - b
...
>>> print(result, c)
bigger 1

>>> a = 1; b = 3
>>> if a > b:
...     result = 'bigger'
... elif a == b:
...     result = 'same'
... else: # i.e. a < b
...     result = 'smaller'
...
>>> print(result)
smaller

>>> if a < b: print 'ok'
ok
```

- Indentation is important!
 - be consistent
 - use four spaces
 - do not use tabs
- Colon indicates the start of an indented block

Comparison operators:

==	!=
>	<
>=	<=
is	is not
in	not in

Boolean operators:

and
or
not

Conditionals

```
>>> if 'Steven' in ['Bob', 'Amy', 'Steven', 'Fred']:  
...     print 'Here!'  
...  
Here!
```

```
>>> if 'Carol' not in ['Bob', 'Amy', 'Steven', 'Fred']:  
...     print 'Away!'  
...  
Away!
```

```
>>> test = a == b  
>>> if test: print 'Equal'  
'Equal'
```

Loops

```
>>> a = b = 0
>>> while a < 10:
...     a += 3
...     print(a)
...
3
6
9
12

>>> while True:
...     b += 3
...     if b >= 10: break
...     print(b)
3
6
9
```

```
>>> for i in [2, 5, 3]:
...     print(i**2)
4
25
9

>>> for j in range(5): print(j)
0
1
2
3
4

>>> range(3, 10, 2)
[3, 5, 7, 9]
```


Loops

```
>>> d = {'this': 2, 'that': 7}
>>> for k, v in d.items():
...     print('%s is %i'%(k, v))
this is 2
that is 7
```

```
>>> numbers = ['none', 'one', 'two', 'lots']
>>> for i, j in enumerate(numbers):
...     print('%i: %s' % (i, j))
0: none
1: one
2: two
3: lots
```

Functions

```
>>> def my_func(x, y=0.0, z=1.0):  
...     a = x + y  
...     b = a * z  
...     return b  
...  
>>> my_func(1.0, 3.0, 2.0)  
8.0  
>>> my_func(1.0, 3.0)  
4.0  
>>> my_func(1.0, y=3.0)  
4.0  
>>> my_func(5.0)  
5.0  
>>> my_func(2.0, z=3.0)  
6.0  
>>> my_func(x=2.0, z=3.0)  
6.0
```

Methods

```
>>> a = [2, 5, 3, 6, 5]
```

```
>>> a.sort()
```

```
>>> print(a)
```

```
[2, 3, 5, 5, 6]
```

```
>>> a.count(5)
```

```
2
```

```
>>> a.reverse()
```

```
>>> print(a)
```

```
[6, 5, 5, 3, 2]
```

```
>>> d = {'black': 100, 'grey': 50, 'white': 0}
```

```
>>> d.values()
```

```
[0, 50, 100]
```

```
>>> s = '-'.join(('2009', '07', '07'))
```

```
>>> print(s)
```

```
2009-07-07
```

```
>>> a.__contains__(3)
```

```
True
```

```
# leading underscores indicate
```

```
# not intended for general use
```

Lecture one ended early here...

... so lecture two will start here.

List comprehensions

- A neat way of creating lists (and arrays) without writing a loop

```
my_fav_num = [3, 17, 22, 46, 71, 8]
```

```
even_squared = []  
for n in my_fav_num:  
    if n%2 == 0:  
        even_squared.append(n**2)
```

```
# in one line:  
even_better = [n**2 for n in my_fav_num if n%2 == 0]
```

```
# both produce [484, 2116, 64]
```

```
freshfruit = [' banana', ' loganberry ', 'passion fruit ']  
stripped = [weapon.strip() for weapon in freshfruit]  
print(stripped)  
['banana', 'loganberry', 'passion fruit']
```

Classes

```
>>> class MyClass:
...     def __init__(self, x, y=1.0):
...         self.my_x = x
...         self.my_y = y
...
...     def product():
...         return x*y
...
>>> m = MyClass(2, 4)
>>> m.product()
8
>>> p = MyClass(5)
>>> p.product()
5
>>> m.product()
8
```

Modules

- Modules can contain any code
- Classes, functions, definitions, immediately executed code
- Can be imported in own namespace, or into the global namespace

```
>>> import math
>>> math.cos(math.pi)
-1.0
```

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

```
>>> from math import cos, pi
>>> cos(pi)
-1.0
```

```
>>> from math import *
```

Your own modules

- Simply put code in a file with extension .py
- Import it in interpreter
- Use it!

```
>>> import mymod
>>> mymod.dostuff()
>>> reload(mymod)           # if you change the module code
```

- Can also collect files together in folders

```
myphdmodule/
  __init__.py           contains code run upon import
  backgroundreader.py
  datareducer.py
  resultsinterpreter.py
  thesiswriter/         submodule
    introwriter.py
    bulkadder.py
    concluder.py
```


Executable scripts

```
#!/usr/bin/env python

def countsheep(n):
    sheep = 0
    for i in range(n):
        sheep += 1
    return sheep

if __name__ == "__main__":
    import sys
    if len(sys.argv) > 1:
        n = int(sys.argv[1])
    else:
        n = 100
    s = countsheep(n)
    print(s)
```

Executable scripts (better)

```
#!/usr/bin/env python

def countsheep(n):
    sheep = 0
    for i in range(n):
        sheep += 1
    return sheep

def main():
    if len(sys.argv) > 1:
        n = int(sys.argv[1])
    else:
        n = 100
    s = countsheep(n)
    print(s)

if __name__ == "__main__":
    import sys
    sys.exit(main())
```

Documentation and tests

```
# My totally wicked function
def my_func(x, y=0.0, z=1.0):
    """This does some stuff.
```

For example:

```
>>> my_func(1.0, 3.0, 2.0)
8.0
```

Yes, it really is that good!

```
    """
    a = (x + y) * z
    return a
```

- Comments before function, class, etc. are used to generate help
- “Docstrings”
 - preferred way of documenting code
 - can contain examples, which are automatically turned into tests!
 - See doctest module

File I/O

```
>>> fname = 'myfile.dat'

>>> f = file(fname)
>>> lines = f.readlines()
>>> f.close()

>>> f = file(fname)
>>> firstline = f.readline()
>>> secondline = f.readline()

>>> f = file(fname)
>>> for l in f:
>>>     print l.split()[1]
>>> f.close()

>>> outfname = 'myoutput'
>>> outf = file(outfname, 'w') # second argument denotes writable
>>> outf.write('My very own file\n')
>>> outf.close()
```

String formatting for output

```
>>> sigma = 6.76/2.354
>>> print('sigma is %5.3f metres'%sigma)
sigma is 2.872 metres
>>> d = {'bob': 1.87, 'fred': 1.768}
>>> for name, height in d.items():
...     print('%s is %.2f metres tall'%(name.capitalize(), height))
...
Bob is 1.87 metres tall
Fred is 1.77 metres tall

>>> nsweets = range(100)
>>> calories = [i * 2.345 for i in nsweets]
>>> fout = file('sweetinfo.txt', 'w')
>>> for i in range(nsweets):
...     fout.write('%5i %8.3f\n'%(nsweets[i], calories[i]))
...
>>> fout.close()
```

- Also a template system

Help

- Powerful help tools
- Most objects, functions, modules, ... can be inspected

```
>>> help(math)

>>> help(math.cos)

>>> a = [1, 2, 3]
>>> help(a)
```

(ignore things starting with _ _)

- If in doubt, hit 'tab'
- If impatient, hit 'tab'

Lots of support online

- python.org/doc
 - Language documentation
 - Library documentation
 - Tutorials
- google.com
- stackoverflow.com
- etc. ...

Good programming practice versus 'thinking aloud'

- Compromise between:
 - producing results quickly
versus
 - easy reusability and adaptation of code
 - code that can be quickly understood by others
- Comment clearly
- Use functions
- Use modules
- Consider 'refactoring' before code gets too messy
- Look into version control (git is best!)
- **Science, not software development**

Version control – do it!



*free private and public code hosting
academic accounts unlimited*



free public code hosting



GUI for Mac OSX



GUI for Windows, Linux, OSX

Exercises I

- 1) Start your python interpreter and check the version.
- 2) Use python as a calculator (use variables and the math module).
- 3) Look at help for the math module.
- 4) Create a list of your five favourite numbers on the range (0, 10) and check the identity $\cosh^2(x) - \sinh^2(x) = 1$ holds true for them, using (a) a for loop, and (b) a list comprehension.
- 5) Write a function $f(x, n)$, where x is a list of numbers and n is a single number, which returns a list of the indices of x where the value is exactly divisible by n . Check it works using your list of favourite numbers.
- 6) Put the above function in a script, with documentation. Import and test it.
- 7) Adapt your previous solution to take a filename instead of a list, and read the numbers to be tested from that file, considering one line at a time.
- 8) Adapt your solution to use a class with an attribute n and a method that returns which elements of a provided list are divisible by n .