PHYS4038/MLiS and AS1/MPAGS

Scientific Programming in python™

mpags-python.github.io

Steven Bamford

University of Nottingham
UK | CHINA | MALAYSIA

AS1/MPAGS

Course Introduction

python powered

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
    - During sessions use desktop PCs in this room or your own laptop
    - Anaconda recommended – see course webpage

- Ideally you should also have:
  - Some current or upcoming need of a scripting language
  - A piece of real or toy analysis on which you can try out using Python

python powered

Course aims

- To give you…
  - experience of using a modern scripting language
  - introduction to all essential Python syntax
  - practical 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 (but I will try to help!)
  - cover every aspect of Python

Course structure

- Ten sessions, every Monday this term
  - 13:00 – 14:00 — lecture / workshop
    - in George Green A13
    - or watch video (live or in your own time) via Echo360
    - mix of PowerPoint and Jupyter notebooks
    - have Python running and try things out as I talk
  - 14:00 – 15:00 — examples class
    - local students in George Green A13
    - remote students work together or individually, as preferred
    - work on exercises and examples
    - ask any questions
    - make progress on coursework
    - help with debugging, etc.

Questions

- Talk to me:
  - During teaching sessions *(preferred)*
    - Specific questions, clarifications – just ask
    - Bigger issues – wait until end of lecture / start of examples class
    - Remote students connect via skype group:
      - https://join.skype.com/KpW5oCLNNiJt
      - text during lecture, video during examples class
  - Via email: steven.bamford@nottingham.ac.uk
  - Arrange a meeting
    - email me
    - office: CAPT A112b
    - or in skype group

Provisional outline

- **Session 1**: Introduction to Python
  - Why Python is (mostly) awesome
  - Writing and running Python
  - Language basics
- **Session 2**: Introduction to Python, continued
  - More language basics
  - Good programming practice
- **Session 3**: Staying organised
  - Managing your environment with conda and pip
  - Version control with GitHub
- **Session 4**: Numerical Python
  - Numpy
  - Using arrays wisely
- **Session 5**: Plotting with Python
  - Matplotlib (and others)
- **Session 6**: Scientific Python overview
  - Scipy and other tools

Provisional outline

- **Session 7**: Scientific Python examples
  - Filtering, interpolation, optimisation
- **Session 8**: Data handling
  - Efficiently storing and processing large amounts of data
    - PyTables, Pandas, Dask
    - Multiprocessing
- **Session 9**: Robust, fast & friendly code
  - Testing and timing
  - Wrapping external libraries and creating the fastest code
    - cython, numba, etc.
  - Web applications
- **Session 10**: Python for specialists
  - Python for astronomers
    - Astropy
  - Python for theorists
    - Symbolic algebra
  - Bayesian inference and Deep Learning in Python
    - MCMC with emcee
    - ANNs with keras

Assessment

For those taking this module for MPAGS credits

- Assessed by development of a Python program relevant to your interests
  - put course material into practice
  - opportunity to become familiar with Python
  - get feedback on your coding
- Your code should…
  - be written as an executable module (.py file) or Jupyter notebook (.ipynb)
  - do something meaningful: analyse real data or perform a simulation
  - define at least two user functions (but typically more)
  - make use of appropriate specialist modules
  - produce at least one informative plot
  - comprise >~ 50 lines of actual code
    - excluding comments, imports and other 'boilerplate'
  - contain no more than 1000 lines in total
    - if you have written more, please isolate an individual element

Code development

- Three stages (first two optional for MPAGS students)

  1. hand-in by **1st November** (optional for feedback)
     - README describing what you intend your code to do
     - Rough outline of the code (classes, functions, snippets, comments, pseudocode)

  2. hand-in by **15th November** (optional for feedback)
     - Rough version of your code, may be incomplete, have bugs, although try to make it reasonable and easy to understand!

  3. hand-in by **13th December** (required for MPAGS credits)
     - Complete working version of your code

Deadlines are 3pm on Fridays.

An introduction to scientific programming with

Session 1:
Introduction to Python

Why use a high-level language?

- Modern high-level languages:
  - Python, R, JS, Julia, Ruby, IDL, Perl, …
- Interactive interpreter
- Ease of use
- Speed of development
- Readability

- Writing code ('scripting') better than a 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, high-level language 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
- Comprehensive scientific and data analysis tools
- Fast, efficient
- Interpreter, introspection
- Runs everywhere, completely free
- Large community

Why learn Python?

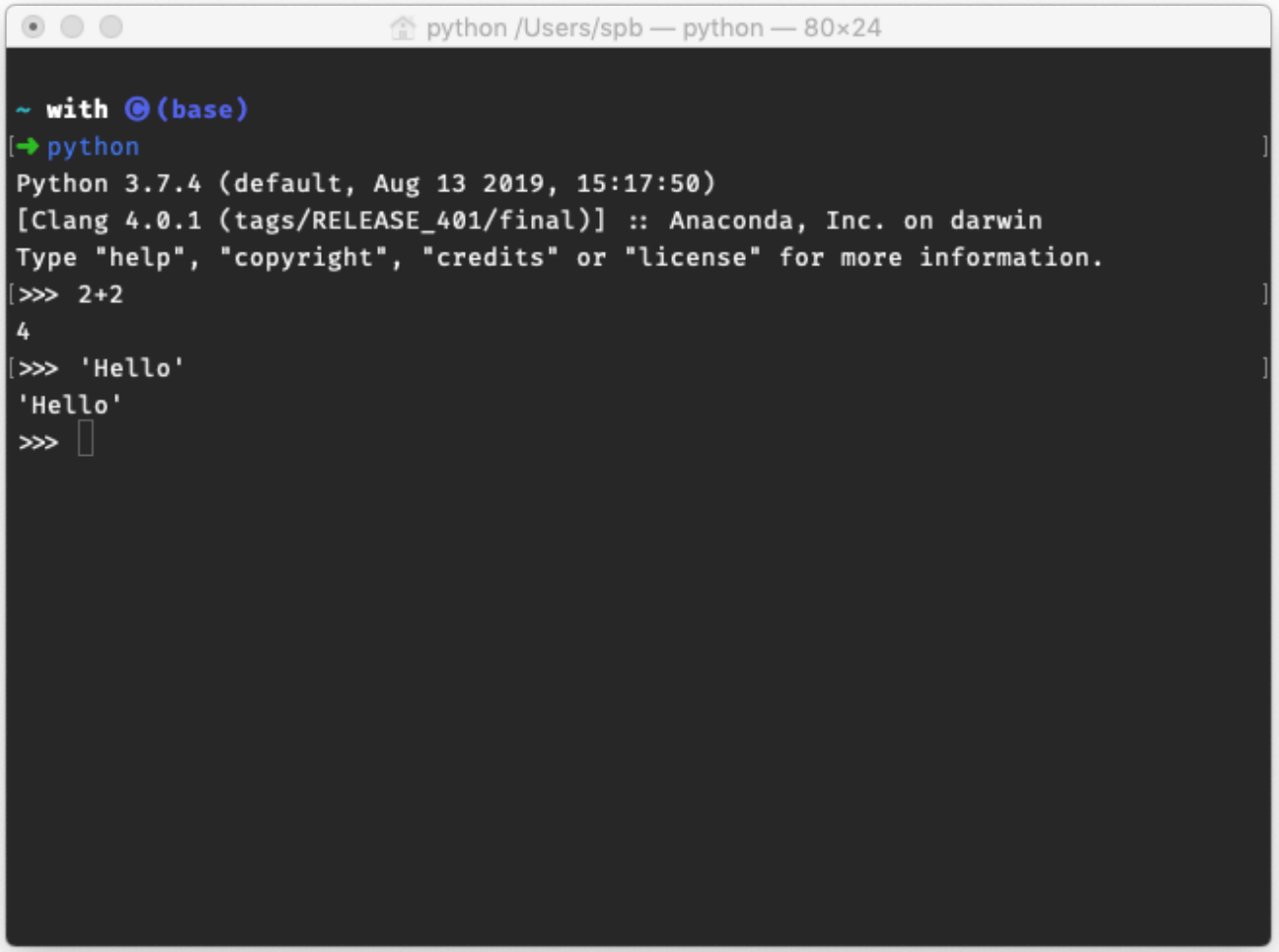
- Get more science done with less stress
- Widely used throughout academia and industry
  - NASA, AstraZeneca, Google, Industrial Light & Magic, Philips,…
  - web services, engineering, science, air traffic control, quantitative finance, games, education, data management, …
- Python programmers in demand
- Easy introduction to general programming concepts

Why not?

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

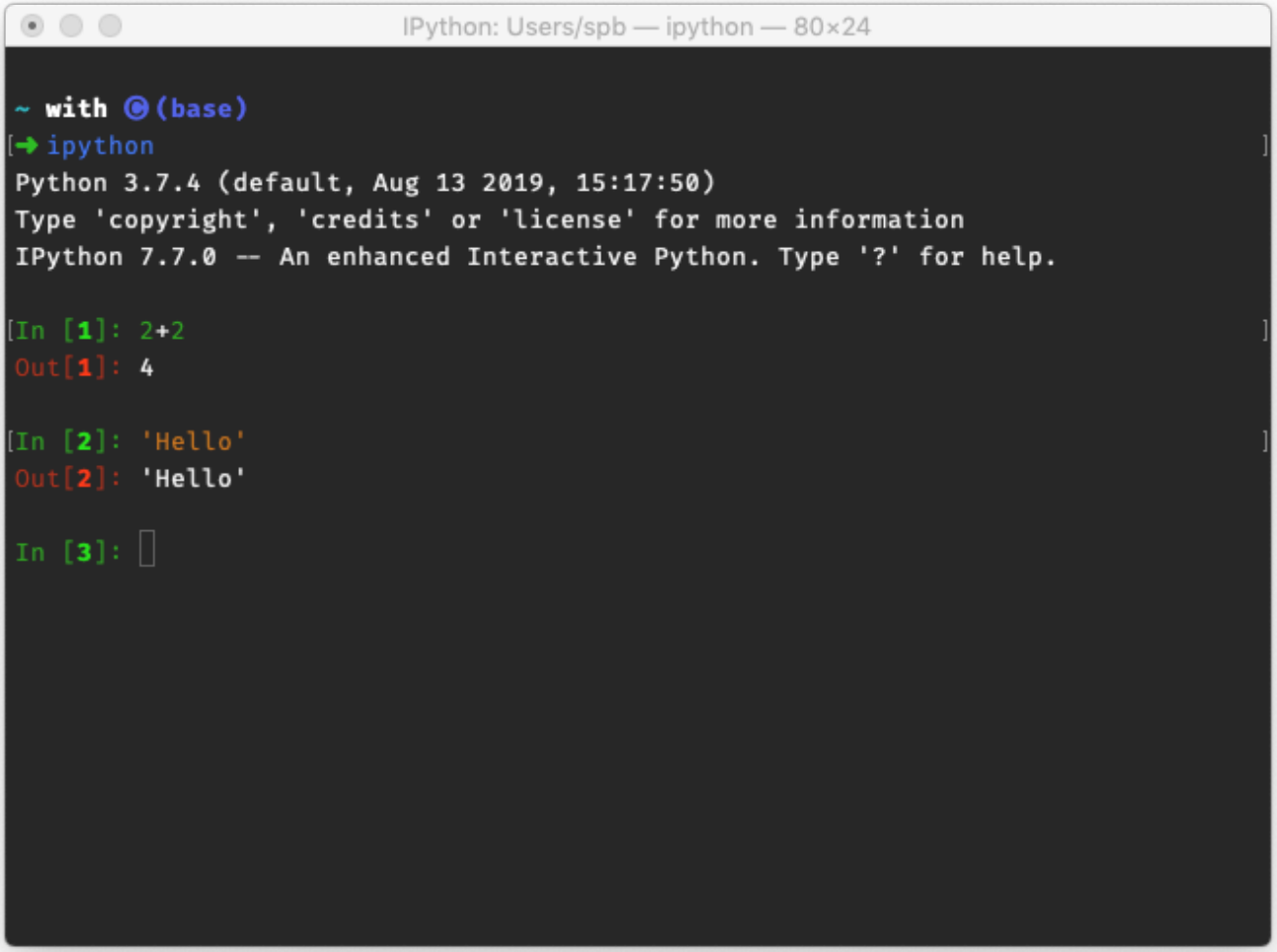
Running Python

- **Command line**
  - Basic Python interpreter
  - Terminal / Anaconda prompt
  - Just type `python`
  - To exit:
    - `Ctrl-D`
    - `exit()`



Running Python

- **Command line**
  - IPython – enhanced Interactive Python
  - Terminal / Anaconda prompt : just type `ipython`
  - Or use launcher
  - To exit:
    - `Ctrl-D`
    - `exit()`



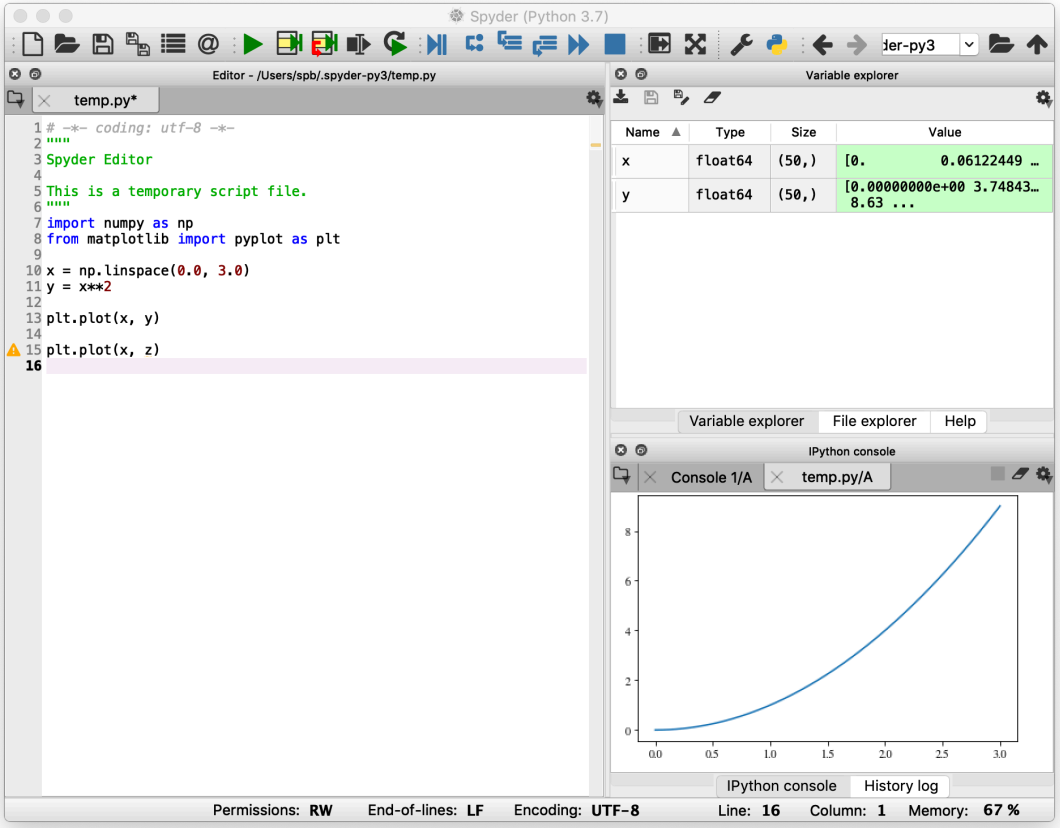
Writing Python

- **Editors**
  - Choose wisely
    - you will use it a lot
    - it will save you a lot of time in the long run
    - worth putting in some effort to learn features and shortcuts
    - cross-platform is an advantage
  - Old-school:
    - Emacs, Vim
  - New-school:
    - Atom, TextMate, Sublime Text, …
    - tend to be extensible, lots of functionality, customisable
  - But perhaps better to use…

Writing and running Python

- **Integrated Development Environment (IDEs)**
  - Editor, interpreter, inspector, graphical output viewer all-in-one
  - Tools for organizing, debugging, inline documentation, etc.

- **Spyder**
- Python-only
- Included with Anaconda
- Terminal / Anaconda prompt:
  - just type `spyder`
- Or use launcher



Writing and running Python

- **Integrated Development Environments (IDEs)**
  - Editor, interpreter, inspector, graphical output viewer all-in-one
  - Tools for organizing, debugging, inline documentation, etc.

  - **PyCharm**
    - Python-specific, but similar versions for other languages
    - Professional version free for academic use
    - https://www.jetbrains.com/pycharm/
    - https://www.jetbrains.com/education/

Writing and running Python

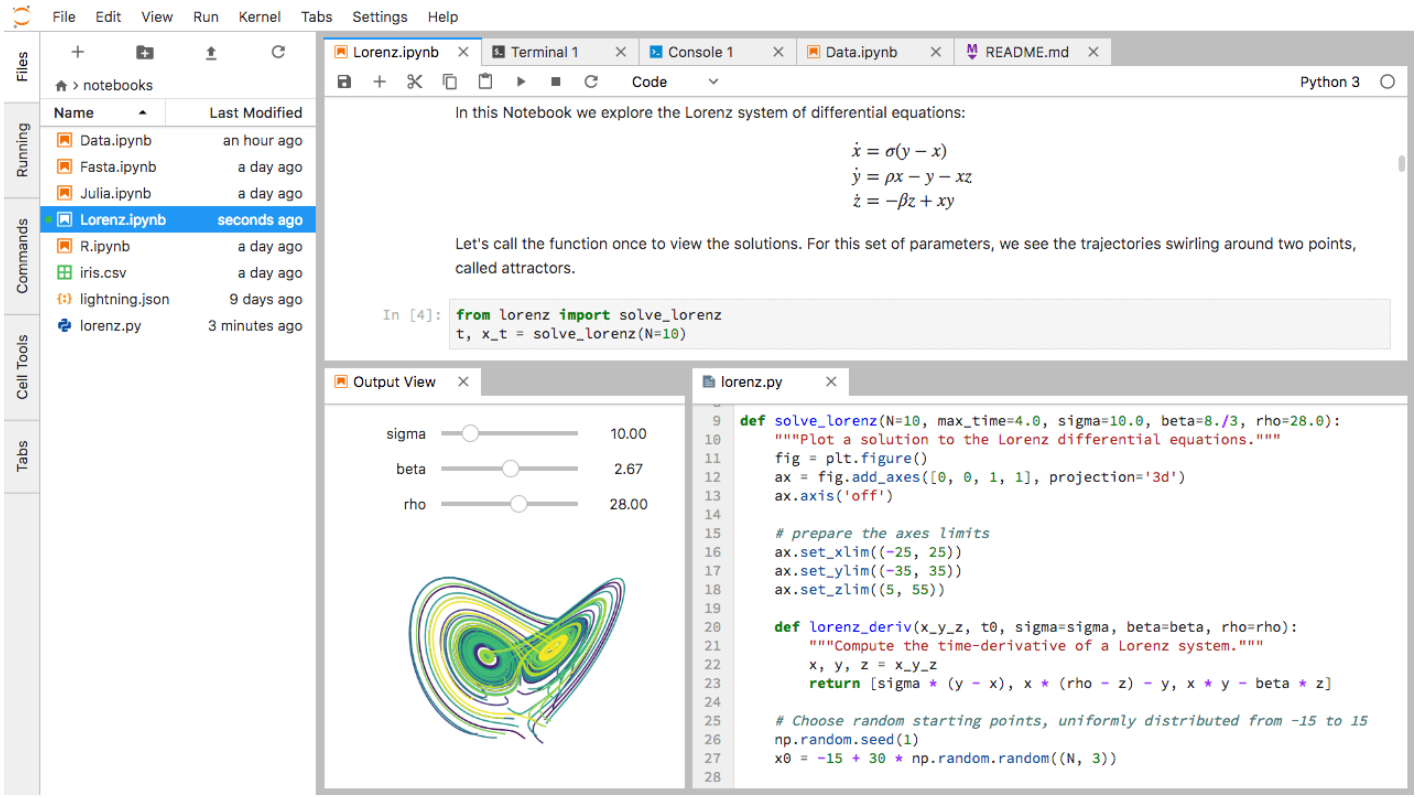
- **Jupyter**
  - Mathematica/Maple-style notebooks
  - Store code and output together in one file
  - Blend interactive prompt and scripts
  - Good for demonstrations / trying things out
  - Keep reproducible record of interactive analyses

  - To start, in terminal / Anaconda prompt: `jupyter notebook`
  - Or use launcher
  - Opens notebook interface in web browser

  - Can easily display online in GitHub or with nbviewer.ipython.org
  - Easily converted to python/html/slides, etc.

Writing and running Python

- **Jupyter Lab**
  - All-in-one: a browser-based IDE
  - Terminal / Anaconda prompt: `jupyter lab`
  - Or use launcher



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)
```

- Better to write code in a text editor / notebook
- Save in a file and execute…

from command line:	`$ python test.py`
from the IPython prompt:	`In [1]: %run test.py`
from a Jupyter cell:	`shift / ctrl / alt + enter`
from an IDE:	Click the run icon / appropriate shortcut

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)
```

- Better to write code in a text editor / notebook
- Save and use in future sessions / code (`>>> import test`)
  - more later…
- Create executable files (`$ ./test.py`)
  - more later…

Numbers

```
>>> 10 + 3
13
>>> 10 - 3
7
>>> 10 * 3
30
>>> 10 / 3
3 OR 3.333333333333335
>>> 10 // 3
3
>>> 10 % 3
1
>>> 10**3
1000
>>> 10 + 3 * 5  # *,/ then +,-
25
>>> (10 + 3) * 5
65
>>> -1**2    # Note: -(1**2)
-1
```

```
>>> 10.0 + 3.0
13.0
>>> 10.0 - 3.0
7.0
>>> 10.0 * 3
30.0
>>> 10.0 / 3
3.333333333333335
>>> 10.0 // 3
3.0
>>> 10.0 % 3.0
1.0
>>> 10.0**3
1000.0

>>> 4.2 + 3.14
7.3399999999999999
>>> 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
```

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?
```

String formatting for output

```
>>> name = 'Steven'; day = 'Wednesday'
>>> print('Hello {}. It is {}.'.format(name, day))
Hello Steven. It is Wednesday.

>>> # Same effect:
>>> print('Hello {1}. It is {0}'.format(day, name))
>>> print('Hello {n}. It is {d}'.format(d=day, n=name))

>>> d = {'Bob': 1.87, 'Fred': 1.768}
>>> for name, height in d.items():
...     print('{who} is {height:.2f}m tall'.format(who=name,
...                                                 height=height))
...
Bob is 1.87m tall
Fred is 1.77m tall

>>> # older alternative uses '%'
>>> for name, height in d.items():
...     print('%s is %.2f metres tall'%(name, height))
```

String formatting for output

```
>>> d = {'Bob': 1.87, 'Fred': 1.768}
>>> for name, height in d.items():
...     print('{who} is {height:.2f}m tall'.format(who=name,
...                                                height=height))


>>> # f-strings (Python 3.6+) – more compact syntax
>>> for name, height in d.items():
...     print(f'{name} is {height:.2f}m tall’


>>> # older alternative uses '%'
>>> for name, height in d.items():
...     print('%s is %.2f metres tall'%(name, height))
```

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] = 'fork'
>>> c[-1]  # negative indices count from end
'handles’

>>> c[1:3]  # slicing
['candles', 'fork']
>>> c[2:]  # omitting defaults to start or end
['fork', 'handles']
>>> c[0:4:2]  # variable stride (could just write c[::2])
[4, 'fork’]

>>> len(a)
5
```

Containers

Lists:

```
>>> a + c  # concatenate
[1, 2, 4, 8, 16, 4, 'candles', 'knife', 'handles’]

>>> a.append(32)
>>> a
[1, 2, 4, 8, 16, 32]

>>> a.extend(c)
>>> a
[1, 2, 4, 8, 16, 4, 'candles', 'knife', 'handles’]
```

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 (real) tabs
  - any decent editor will handle this for you (try tab / shift-tab)
- Colon always indicates the start of an indented block
- Block closed by de-indent

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
```

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)
range(3, 10, 2)

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

Loops

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

>>> numbers = ['none', 'one', 'two', 'lots']
>>> for i, j in enumerate(numbers):
...     print(f'{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)     # leading underscores indicate
True                      # not intended for general use
```

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 _ _)

In IPython:

```
In [1]: math.cos?
In [2]: a?
```

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

Lots of support online

- python.org/doc
  - Language documentation
  - Library documentation
  - Beginner's Guide and Tutorials
- ipython.org/documentation.html
- www.codecademy.com/en/tracks/python
- google.com
- stackoverflow.com
- etc. …

That's it for today!

Next up:

- **Session 2**: Introduction to Python, continued
  - More language basics
  - Good programming practice

- **Session 3**: Staying organised
  - Managing your environment with conda and pip
  - Version control with GitHub

Questions and exercises

Any questions?

- shout and wave
- skype (spbamford)
  - https://join.skype.com/KpW5oCLNNiJt
- email steven.bamford@nottingham.ac.uk

Exercises?

Get started on them now.

I'll be around for support.

Solutions are online.

I will go through them either later in this session or in the next one.

Exercises 1

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 five numbers on the range (0, 10) and check the identity $\cosh^2(x) - \sinh^2(x) = 1$ holds true for them, using a for loop.

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!

Any questions?

- skype (spbamford): https://join.skype.com/KpW5oCLNNiJt
- email steven.bamford@nottingham.ac.uk