Physics PhD Python Course

Durham CDT/Astro PhD Python crash course

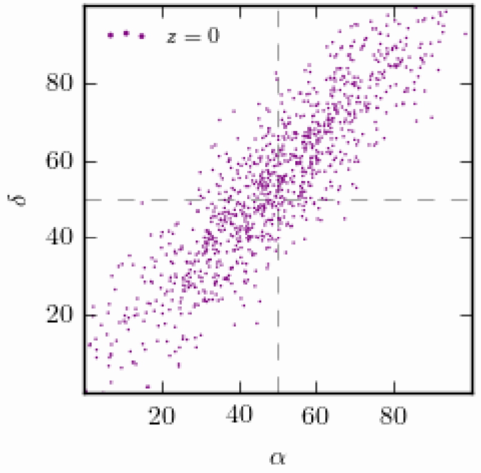
Organizer: Dr. Andrew Cooper, ICC (OCW 223) `a.p.cooper` at `durham.ac.uk`

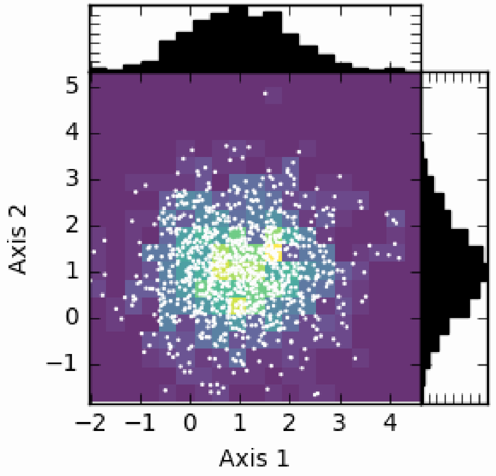
Aims

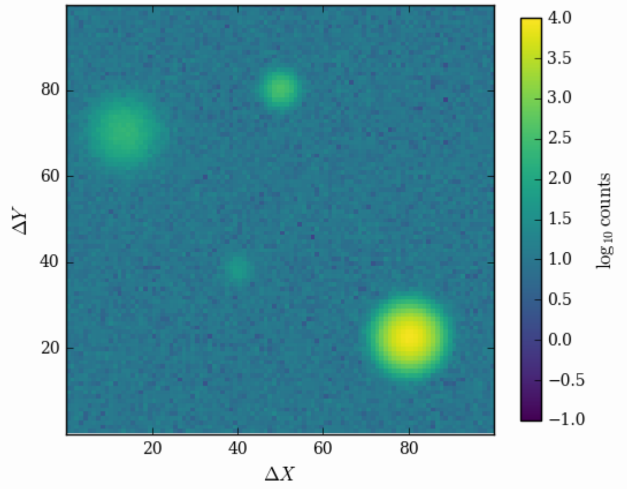
- Short introduction/background.
- Make sure everyone has an environment in which they can write and run Python.
- Introduce fundamentals of the language.
- Introduce tactics and resources for figuring things out on your own.

We'll learn how to:

- Write a basic program using the common elements of standard Python.
- Understand the structure of more complex Python programs, including classes and objects.
- Manipulate files and directories with Python code.
- Generate simple random data, read/write files and make some plots.







and understand structure of realistic Python code:

```
# Import statements
import os
import random
import numpy as np
from   numpy.random import normal
import datetime

# Variables defined at the module level.
# Captial letters for these are just my personal style.
THIS_YEAR     = datetime.datetime.now().year
BEAR_FILE     = 'my_bear_names.txt'

# Simple one-line exception definition -- just a class like any other,
# but inherits from Exception
class BearNamesError(Exception): pass

# A normal Class definition
class Bear:
    # A special function within the class definition
    def __init__(self,name,birth_year):
        """
        Args:
            name: name of bear
            age: age of bear in years
        """
        # Variables can be glued to objects using the . notation
        # (variables glued to objects are called attributes)
        self.name       = name
        self.birth_year = birth_year

        self.milestones = [('first year',      lambda x: x == 0),
                           ('decade',          lambda x: x%10 == 0),
                           ('can vote',        lambda x: x == 18),
                           ('midlife crisis', lambda x: x == 35),
```

How this works

- Some **interactive** tutorials/exercises.
- Some **non-interactive** notes with more background and links.

The interactive part uses a set of **Jupyter notebooks**. Jupyter notebooks are interactive environments for running Python code step-by-step in a web browser. Each notebook in the tutorial covers a different topic.

The aim of these sessions is to work through those notebooks at your own pace and ask questions when you get stuck.

Bored (and know Python) already?

These classes are aimed at those with little prior experience of Python. If you already know all this, please spend your time productively...

- Help others, find all the bugs/confusing statements/missing topics in the notebooks
- Complete the two challenges (even if trivial, calibration/example solutions help).

Then maybe:

- Learn something new in Python (there's always something to learn)?
- Work on your own code?
- Contribute to an open-source project (e.g. Astropy)?
- Learn Haskell?

Where stuff is

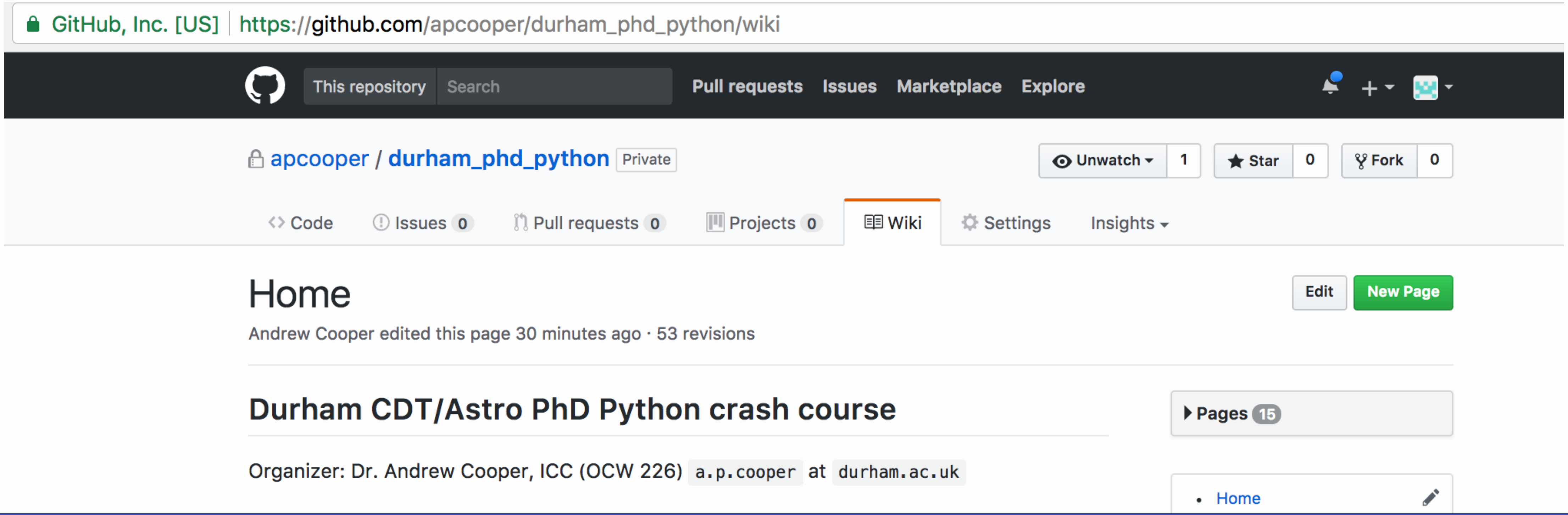
`https://github.com/apcooper/durham_phd_python`

- Instructions for getting started + all the course content.

Where stuff is

`https://github.com/apcooper/durham_phd_python`

- Similar content but more detail than these slides on the wiki (`/wiki`)



Home

Andrew Cooper edited this page 30 minutes ago · 53 revisions

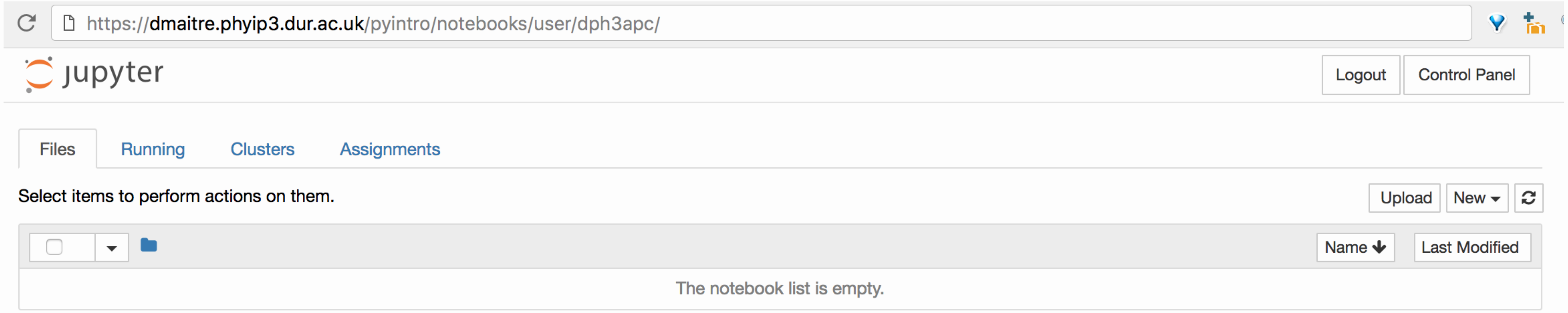
Durham CDT/Astro PhD Python crash course

Organizer: Dr. Andrew Cooper, ICC (OCW 226) `a.p.cooper` at `durham.ac.uk`

Getting to the notebooks

We'll use Daniel Maitre's local JupyterHub server to work with these notebooks to minimize setup time and so everyone sees more-or-less the same thing.

`https://dmaitre.phyip3.dur.ac.uk/pyintro/notebooks/`



Select items to perform actions on them.

The notebook list is empty.

The Course

Outline

Astronomy Course

This second part of the course introduces some Python packages for scientific computing and astronomy specifically. Content is subject to change...

1. Astropy
2. Large files with HDF5
3. Python on (Durham) clusters
4. Distributed computing with Dask
5. Other packages worth investigating

Philosophy

Teaching yourself python is easy enough once you get started - you're expected to do most of the work on your own, using resources your find for yourself online.

Being 'fluent' in a language like Python means being able to write concise code quickly and in such a way that you can build on it easily, find and fix errors quickly, and explain how it works to other people (and yourself in 6 months time).

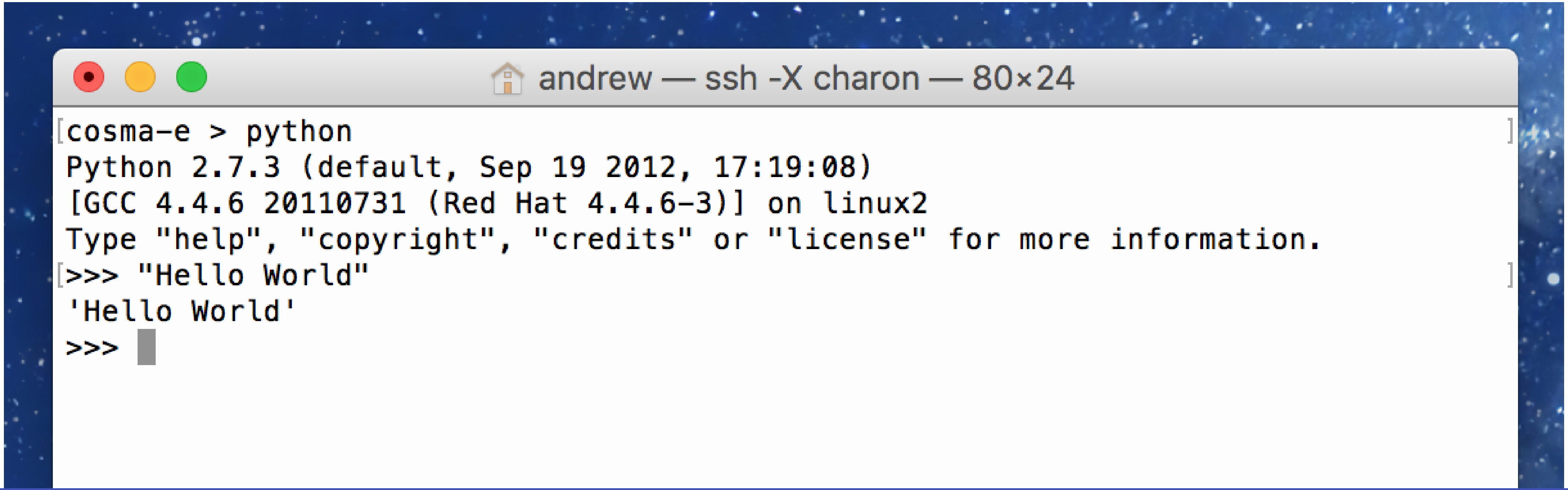
- Following tutorials is a good way to get started
- The only reliable way to improve is through trial and error (blood, sweat, tears etc.)
- Try to do some small jobs with Python
- Look for fast, short and elegant solutions
- Read books and other people's code
- Get into the habit of writing neat, self-explanatory code

Outline

We'll go through the first three topics in these slides, then it's on to the notebooks.

Quick start

- Open a terminal.
- Run the command `python`.
- At the `>>>` prompt, type `"Hello World!"` and press return.



```
cosma-e > python
Python 2.7.3 (default, Sep 19 2012, 17:19:08)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> "Hello World"
'Hello World'
>>> 
```

1. What is Python?

- Python in a nutshell
- An example Python program
- The zoo of languages
- What Python is good at (and not so good at)

Python in a nutshell

Wikipedia:

> Python is a high-level programming language used for general-purpose programming and originally created by Guido van Rossum in 1991. Python has a design philosophy which **emphasizes code readability**, and a syntax which allows programmers to **express concepts in fewer lines of code** than possible in languages such as C++ or Java. The language provides constructs intended to enable writing **clear programs on both a small and large scale**.
>
> > Python features a **dynamic type system** and **automatic memory management** and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a **large and comprehensive standard library**.

Python code looks like this

```python
# Another function at the module level
def generate_bears(n_bears=100,names=None):
    """
    n_bears (int): number of bears to generate.
    names (list of str): list of bear names.

    Returns:
        list of Bear objects
    """
    if names is None:
        names = ['Unknown Bear']

    # Uses an imported function and variable from the module scope
    random_birth_years = 1970 + np.minimum(THIS_YEAR-1970,
                                           normal(loc=0,scale=30,size=n_bears))

    random_names = [random.choice(names) for i in range(0,n_bears)]

    # Build a list of Bear objects
    bears = list()
    for n, y in zip(random_names,random_birth_years):
        bears.append(Bear(n,y))
    return bears

# And one more function at the module level
def report_random_bear_events(n_bears=100):
    """
    Makes up some random bears and prints a report on those
    celebrating important birthday milestone this year.
    """
    names = get_names(False)
    bears = generate_bears(n_bears,names=names)

    ages  = np.array([THIS_YEAR - bear.birth_year for bear in bears])
    names = [bear.name for bear in bears]
```

Indentation

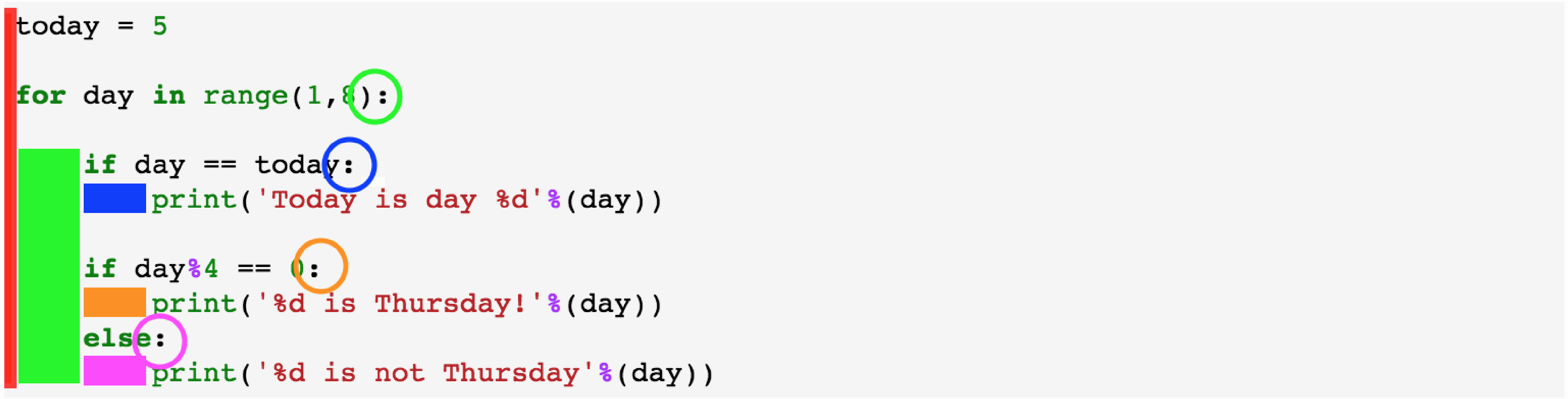
Python statements are grouped into **blocks** (functions, classes, loops, conditional expressions etc.) using **indentation**. The start of a block is usually indicated by a `:`.

```
today = 5

for day in range(1,8):

    if day == today:
        print('Today is day %d'%(day))

    if day%4 == 0:
        print('%d is Thursday!'%(day))
    else:
        print('%d is not Thursday'%(day))
```

By convention these indentations are 4 spaces, ***not*** `TAB` characters. Other languages have different ways of indicating blocks (e.g. C uses `{}`).

Indentation

Python statements are grouped into **blocks** (functions, classes, loops, conditional expressions etc.) using **indentation**. The start of a block is usually indicated by a `:`.



```python
today = 5

for day in range(1,8):

    if day == today:
        print('Today is day %d'%(day))

    if day%4 == 0:
        print('%d is Thursday!'%(day))
    else:
        print('%d is not Thursday'%(day))
```

By convention these indentations are 4 spaces, ***not*** `TAB` characters. Other languages have different ways of indicating blocks (e.g. C uses `{}`).

The zoo of languages

Compared to Python:

- `C/C++` is harder for humans to read and write, harder to learn, but fast and all-powerful (e.g. the standard Python interpreter is written in C).
- `Fortran` is easier to read and write, fast, but more restricted than `C`.
- `Python` is easy to read, easy to write, comprehensive and flexible, but slower.

These are only 3 of a gazillion programming languages.

Facts of life

There are always realistic alternatives, but:

- If you want a job in astronomy, you *should* learn Python. The more you know the better for your career, but you will get by with basic knowledge.

Facts of life

There are always realistic alternatives, but:

- If you want a job in astronomy, you *should* learn Python. The more you know the better for your career, but you will get by with basic knowledge.
- If you want a job in industrial Data Science you almost certainly *must* learn Python, and *should* be proactive about constantly improving your skills. You also need a sound working knowledge of other major languages.

What Python is good at

Python is popular with (most) scientists in large part because:

- They prefer solutions that minimize person-time at the cost of algorithmic elegance, computing time and code length (they know it's bad to over-optimize).
- They don't like reinventing the wheel and prefer to re-use code (with caution about understanding and trusting it). Python has the best reservoir of good third party code for science of any language.
- They work on multiple things at once, and prefer to use the minimum number of tools.
- They like that fact that Python can inter-operate with C and Fortran in a reasonably straightforward way.

What are the downsides?

Python is naturally slow if you don't think about how to make it fast.

Python is rather poor at running multiple processes at once (concurrency) and hence in making use of machines more powerful than your laptop.

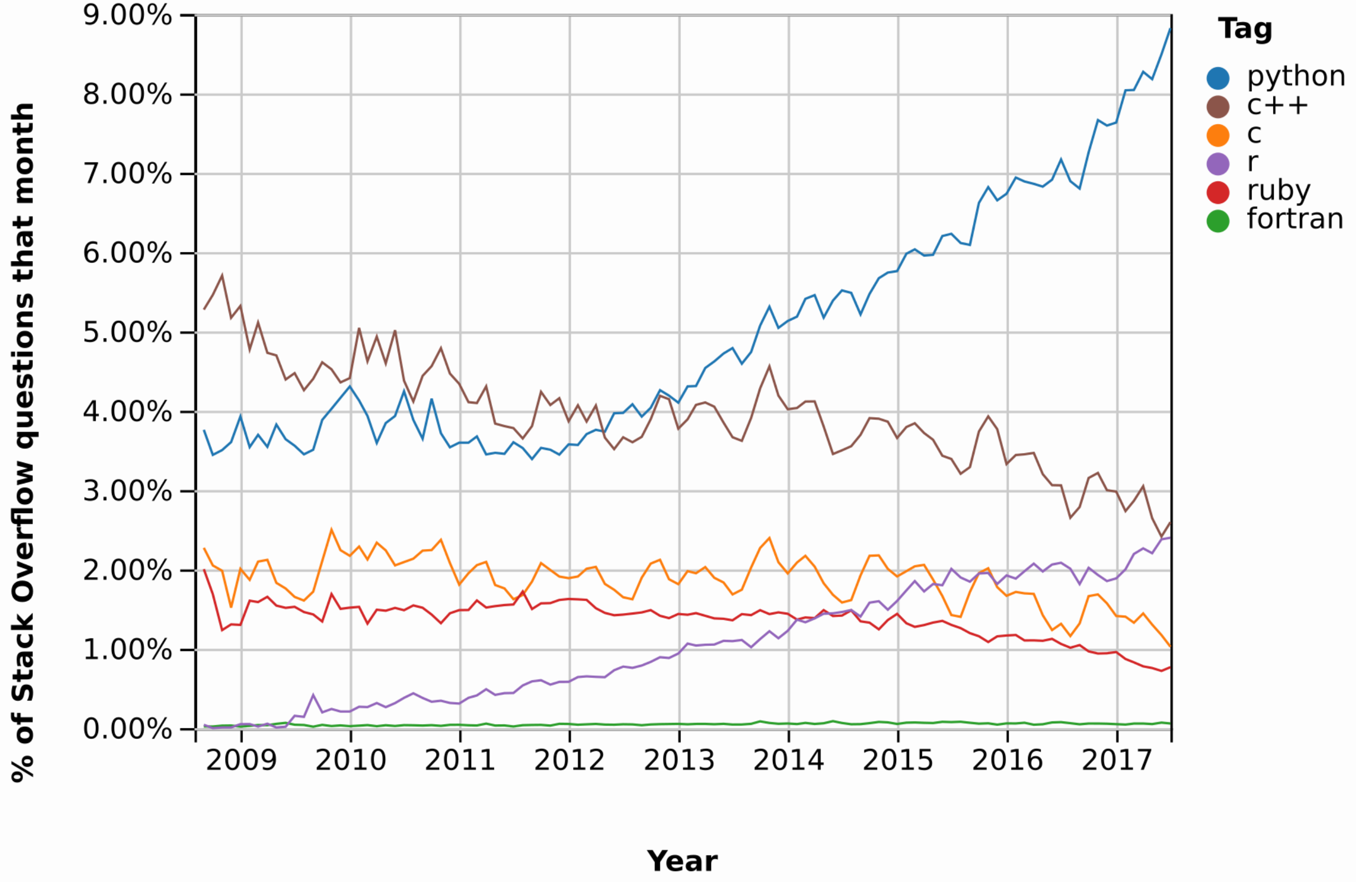
This *can* be done in a practical and realistic way (we'll see how to get started), but it's not a first-class part of the language.

There are realistic alternatives to Python plus C/Fortran:

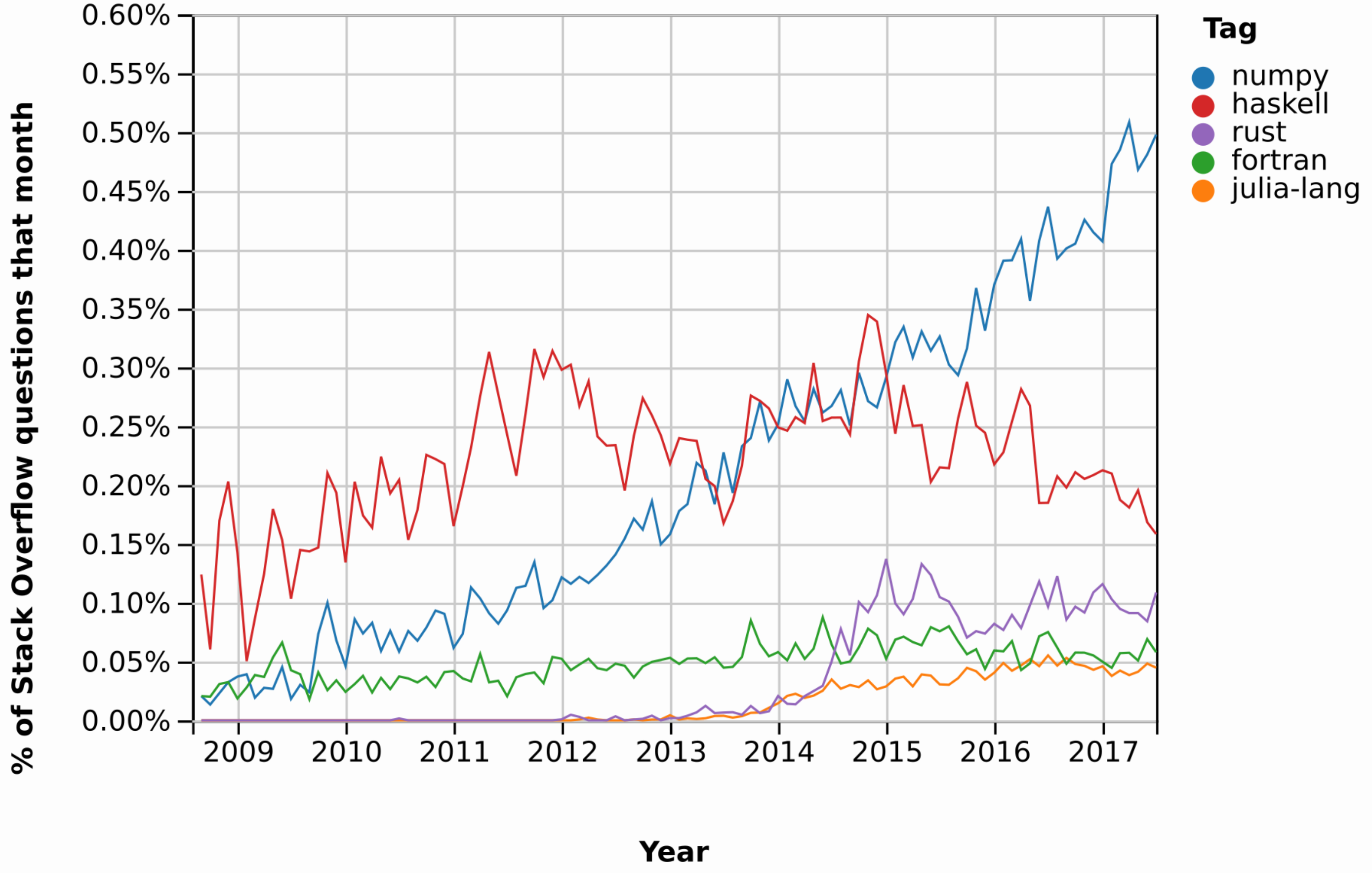
- `IDL`: what astronomers used before they discovered Python. Mostly awful (and closed source).
- `R`: slightly nicer equivalent to IDL prioritising statistics, *still* slowly growing in popularity.
- `Rust`: an up-and-coming alternative to C++.
- `Julia`: an aggressively trendy language designed for scientists. Some would say *too* trendy.
- `Haskell`: a powerful language using the 'functional' style, with a brutal learning curve.

Of course there are many others. Some of the reason for this diversity is in the 'style' of the languages, not what they are capable of doing or how fast they can do it.

Here are two plots showing the fraction of questions on the StackOverflow website each month that are about the languages mentioned above. First the major ones:



Then some minor ones (`numpy` is a python package for numerical computing):



Outline

2. Basic Computing Skills

To get started with Python for everyday scientific computing, you need to have some understanding of:

- how to use the command line on a UNIX-like system ('the shell').
- how to install stuff you find on the web and configure your system to run it.

As a minimum:

- Understand at least one common shell, like `bash`, `csh` or `tcsh`, and appreciate the differences between them.

As a minimum:

- Understand at least one common shell, like `bash`, `csh` or `tcsh`, and appreciate the differences between them.
- Know commands like `ls`, `cd`, `cat`, `echo`, `mkdir`, `rm` etc.

As a minimum:

- Understand at least one common shell, like `bash`, `csh` or `tcsh`, and appreciate the differences between them.
- Know commands like `ls`, `cd`, `cat`, `echo`, `mkdir`, `rm` etc.
- Know about command line tools like less, `grep`, `tar`, `gzip` etc.

As a minimum:

- Understand at least one common shell, like `bash`, `csh` or `tcsh`, and appreciate the differences between them.
- Know commands like `ls`, `cd`, `cat`, `echo`, `mkdir`, `rm` etc.
- Know about command line tools like less, `grep`, `tar`, `gzip` etc.
- Know how to use `man` to get help on commands.

As a minimum:

- Be able to use at least one common text editor that works in a terminal, like `emacs` or `vim` (and make it use 4 spaces to represent tabs!).

As a minimum:

- Be able to use at least one common text editor that works in a terminal, like `emacs` or `vim` (and make it use 4 spaces to represent tabs!).
- Understand the role of environment variables, particularly `$PATH`, and how to show/set/modify them.

As a minimum:

- Be able to use at least one common text editor that works in a terminal, like `emacs` or `vim` (and make it use 4 spaces to represent tabs!).
- Understand the role of environment variables, particularly `$PATH`, and how to show/set/modify them.
- Understand the *basics* of configuration files like `.bashrc` and `.profile`.

As a minimum:

- Be able to use at least one common text editor that works in a terminal, like `emacs` or `vim` (and make it use 4 spaces to represent tabs!).
- Understand the role of environment variables, particularly `$PATH`, and how to show/set/modify them.
- Understand the *basics* of configuration files like `.bashrc` and `.profile`.
- Understand the basics of shell instructions to redirect output ('redirects' and 'pipes')

As a minimum:

- Be able to use at least one common text editor that works in a terminal, like `emacs` or `vim` (and make it use 4 spaces to represent tabs!).
- Understand the role of environment variables, particularly `$PATH`, and how to show/set/modify them.
- Understand the *basics* of configuration files like `.bashrc` and `.profile`.
- Understand the basics of shell instructions to redirect output ('redirects' and 'pipes')
- Understand how to use a common program like `top` to check the resources you're using and how to send `STOP` and `KILL` signals to rogue processes.

Outline

3. Getting Started with Python

- The Python environment
- Python versions
- Writing and running simple Python scripts
- Running Python with IPython and Jupyter
- How to use the interactive help system
- Where else to get help

We won't cover these, but see the wiki:

- Installing packages with `pip`
- Managing Python environments with Conda
- Tools for editing Python code

The Python environment

From a practical point of view, 'Python' is:

- a program (the `python` interpreter) that takes some instructions you write (your Python code) and runs some computations on your machine.
- some 'modules', individual files that contain functions which you or others have written to do various specific jobs, which you can include in new code to avoid having to write everything from scratch each time.
- a bunch of 'packages' that group together modules according to some logical function (including Python's own **Standard Library** packages).
- a means of telling the `python` interpreter where to find all the packages that you want to use (the 'python environment').

Python versions

- Python has two versions: 2 and 3
- In future, Python3 will be the only version.
- Obvious differences are not huge.
- Most major packages relevant to scientists now fully support Python3.
- Python3 assumed here, recommend you use that for your own work.

You can check what version of Python you're using in a terminal:

```
python --version
```

Writing and running simple Python scripts

Our first task is to be able to run Python programs.

For the rest of the tutorial we'll be using Jupyter notebooks, but first we'll cover the more fundamental way to run Python, by calling the interpreter from the command line.

Python is an interpreted language -- you pass commands to the Python interpreter (using the `python` command) rather than using a compiler to make self-contained executable binary files. This makes Python programs very short.

```python
print('Hello World')
```

We can write this in a file `hello.py` and run it with Python:

```
> python hello.py
Hello World!
```

Compare with C:

```c
#include<stdio.h> // Need this for output

main() // Need a 'main' function
{
    printf("Hello World"); // Need a ; to end lines.
} // code in the main function is marked with {}
```

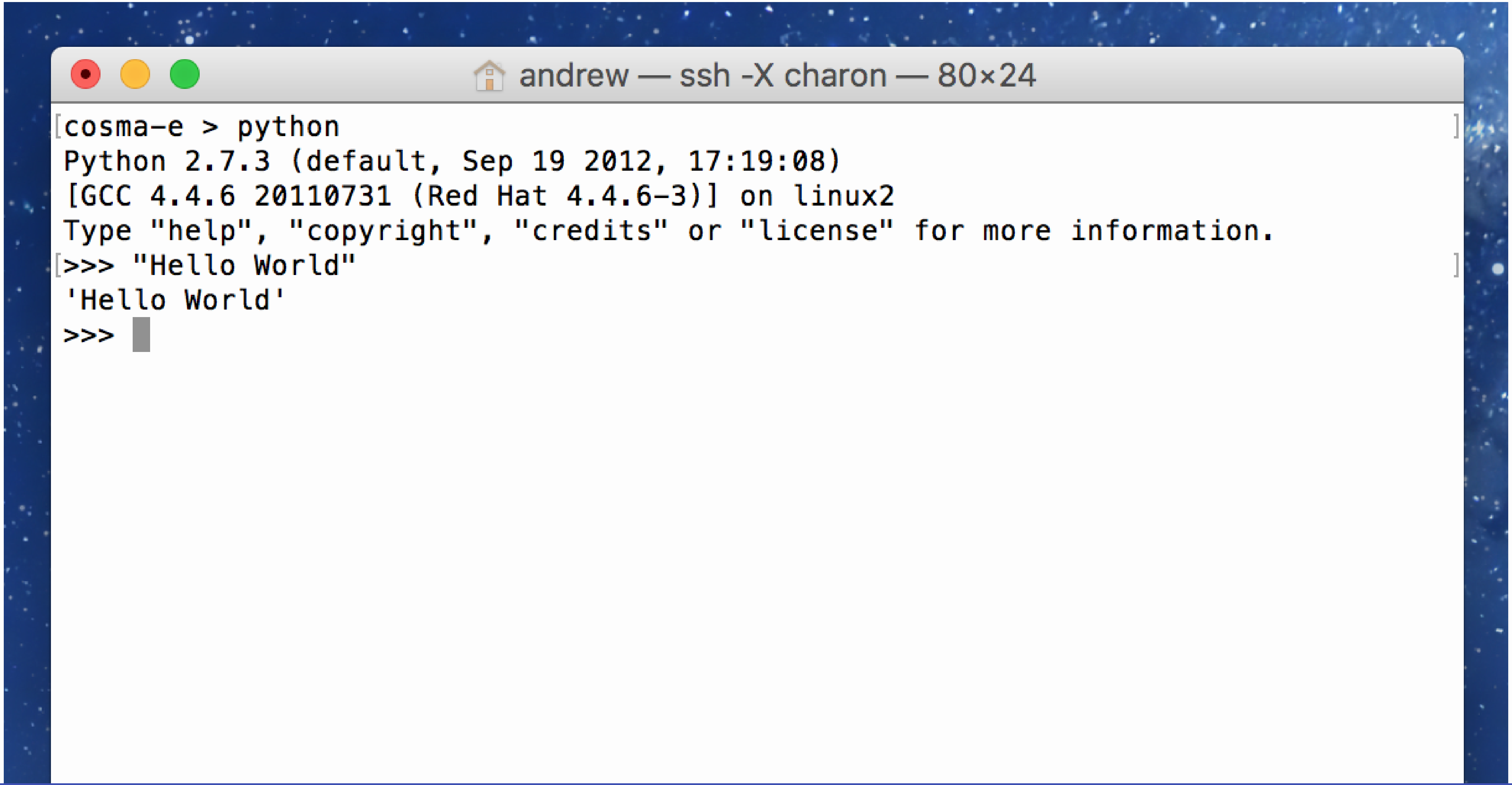
We would need to compile this and *then* run the executable. Python programs are *not* compiled.

The REPL way of working

The fact that Python is run through an interpreter makes it natural to work with interactively (i.e. by writing one line at a time, pressing return to evaluate that line, and looking at the result that gets printed).

The built-in interactive interpreter (Python shell) that you get by running `python` with no file argument is the most basic version of this so-called 'REPL' (read-evaluate-print-loop) way of working with Python.

Every line you type is passed as a command to the Python interpreter, and then prints whatever the Python interpreter sends back as the 'result' of the command. The results of previous lines stay in memory as if they were all being executed one after the other.



```
cosma-e > python
Python 2.7.3 (default, Sep 19 2012, 17:19:08)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> "Hello World"
'Hello World'
>>>
```

Practical Exercises

- What happens if you type the following at the interactive python prompt?
  - `"Hello World!"`
  - `print("Hello World!")`
  - `Hello` (note, no `""`)
  - `1+1`
  - `print(1+1)`
  - `x=1`
  - `print(x)`
- Figure out how to exit the interactive interpreter session you started above.
- Check the version of Python that's running.

Practical Exercises

For any serious program you need to write all the commands in one or more files and run them from the command line with `python`.

- Create a file `hello.py` that contains a Python program that prints `Hello World!` (or the text of your choice). Run it from the command line (i.e. without starting the interactive interpreter).
- Create a file `oneplusone.py` that prints the result of 1+1 when run from the command line.

Running Python with IPython and Jupyter

The REPL way of working with Python is **extremely** useful for two things:

1. Developing more sophisticated codes, by experimenting with different ways of doing things, one step at a time.
2. Using python to explore data, by doing calculations and making plots interactively.

The built-in Python shell is very basic and pretty much useless for either task.

IPython (interactive Python) is a more advanced interactive prompt, and the most common way of working interactively with Python. Many useful features, including:

- colourful syntax highlighting
- a command history
- interactive help
- interactive plotting
- a way to inspect the state of variables in programs after they cause errors, to find out what happened (debugging)

There are other more fancy 'development environments' that are mostly GUI applications built on top of IPython.

Juptyer notebooks are a way of working with IPython in a 'less linear' way by adding an extra layer of javascript that shows a fancy version of an IPython session in a web browser.

This makes it easy to organize and document your interactive work so you can share it with others.

- Great for interactive tutorials!
- Great as a notebook for exploring data and developing code.
- Complexity makes it somewhat fragile.
- Fiddly if you want to work remotely, especially behind a firewall.
- Ultimately limited to short snippets, debugging clumsier than IPython.

Demo of how to use Jupyter

Where to get help

Python has a built-in help system that can give you some minimum information about a module, function or variable, if you already know its name. For example, anywhere you can execute Python code, you can use the help() function:

```
help(str.upper)
```

In IPython (and hence in Juptyer notebooks), you can also type a `?` before or after the method name and press `RETURN`.

```
> ?str.upper
```

```
> str.upper?
```

In Jupyter notebooks you can also put the cursor at the end of the method name and press `SHIFT+TAB`. This pops up a floating window with help in it. You can click the buttons in the top right of that widow to read more.

There are many more tricks for interacting more productively with Jupyter notebooks and IPython sessions.

Other sources of help

```
pydoc str.upper
```

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

http://stackoverflow.com/

Now the work starts!

Outline

This is only a suggested order -- you might prefer to skip to the `matplotlib` introduction before tackling classes/objects.

BasicPythonReview.ipynb

This notebook introduces the most basic features of Python.

First a bit more orientation:

- Get familiar with Juptyer and practice basic maths at the Python interactive prompt.
- See how to write comments (and be warned that this is super-important).
- See what happens when your Python code has errors.
- Understand the `print()` command a bit more.

BasicPythonReview.ipynb

Then the serious stuff:

- How variables work
- How to manipulate text ('strings')
- What the special value `None` is (this is important)
- How boolean logic works
- How to write conditional expressions (`if...else`)
- How to write `while` loops.

These notebooks assume you have done some programming before and are familiar with ideas like integer and floating point representations of numbers.

CollectionsAndLoops.ipynb

Collections are extremely important. Collections are objects that group other objects together. The three most important types of collection are `list`, `tuple` and `dict`. Manipulating these three basic collections is fundamental to everyday programming in Python.

Usually you want to do something to each element of a list. This involves iterating (looping) over the elements, so this notebook also shows how `for` loops work.

DataInDataOut.ipynb

Compared to some other languages, Python makes interacting with data and the file system very easy. It's possible to replace a lot of what used to be done with confusing and complicated shell scripts with more elegant Python programs.

This notebook covers:

- opening files and reading lines of text
- creating files and writing lines of text
- processing text files line by line
- storing complex python objects
- other useful formats for large data files
- working with filesystem paths
- finding files with `glob`

FunctionsAndClasses.ipynb

This notebook covers:

- The elements of structure in Python: functions and classes
- Defining functions (including the idea of variable scope, nested functions, lambdas and recursion)
- Defining classes and creating objects

LongerPrograms.ipynb

This notebook covers:

- Modules and namespaces
- `import` statements and `PYTHONPATH`
- Organizing your code by making your own modules and packages
- Command line arguments

Outline

`numpy` is by far the most important single Python package for data-intensive/scientific calculations of any kind.

Numpy101.ipynb

This notebook covers:

- why the `numpy` package and its `array` object are very important
- creating arrays filled with specific values or random numbers
- extracting subsections of arrays with indexing and slicing operations
- finding specific values in arrays using `where`
- reading and writing arrays from disk
- using a handy function from `numpy` to read CSV tables into arrays

See the wiki for links to more information about `numpy`.

Numpy102.ipynb

Building on Numpy101, this notebook introduces more things that, based on my experience, you will end up using all the time. There is a huge amount of stuff in `numpy` and this only scratches the surface.

- Differences between copies of arrays and references to arrays
- Sorting, concatenating, stacking and splitting arrays
- Creating arrays with `repeat`, `reshape`
- Record arrays (recarrays)
- Logical masks and crazy floating point values (`nan` and `inf`)
- `numpy.where` with 2d arrays
- Finding unique values
- Testing if values from one array are in another array, using in1d
- Finding locations of specific values in arrays

matplotlib101.ipynb

`matplotlib` is a comprehensive package for making plots with Python. This notebook shows the basic steps to make a plot, including figure size, axis labels, font sizes and legends.

matplotlib102.ipynb

This notebook introduces a few more common techniques that are useful for making plots for astronomy papers. See the wiki for links to more information about `matplotlib`.

- Histograms
- Legends
- Grids of plots
- Filling between lines

ProductivityPerformanceProfiling.ipynb

This notebook covers tools that can be used to monitor your code, catch errors and fix them:

- Simple timing with the `time` module
- More detailed timing with `cProfile`
- A brief introductions to exceptions
- An even more brief introduction to the built-in debugger in IPython/Juptyer

Challenges

These are a couple of longer exercises to help you calibrate your Python abilities. Experts can jump to these directly.

1. Making a 2D image with `numpy` and `matplotlib`
2. Matching very long lists of integers

If you complete these without difficulty, you're probably not going to struggle with the coding required for your other courses. They are not super difficult and they don't require anything that isn't covered in the tutorial, but they aren't intended to be easy for beginners either.

The challenges are ***not*** coursework.

For simpler step-by-step exercises, see the Google Python course link on the wiki.

Session 3

- Updated slides: `https://github.com/apcooper/durham_phd_python` then `slides/pdf/presentation.pdf`
- New notebooks
- Challenge solutions

New notebooks

- `SciencePackages.ipynb`
- `SciencePackagesAstro.ipynb`

You'll need to open the `PyIntro2` assignment on the server.

SciencePackages.ipynb

This notebook provides less of a tutorial and more of a brief introduction to some useful tools that you can investigate for yourself:

- Function interpolation
- Integration
- Neighbour-finding in point data
- Gaussian fitting
- Binary file formats for data storage (HDF5 and Fortran-format)
- The Pandas package for data analysis (links only)
- Packages for Markov Chain Monte Carlo (links only)

Despite what it says, you shouldn't need to install any new packages yourself for this notebook.

SciencePackagesAstro.ipynb

Following on from the SciencePackages notebook, this introduces a few more packages that are most relevant to astronomers:

- Coordinates and units
- Tables (including FITS)
- World Coordinate System information in plots
- Querying astronomical databases and images
- Checking the visibility of astronomical objects
- Using HEALPIX in Python

You will need to change `./data` to `/data/pyintro` (no leading `.`) in some of the cells.

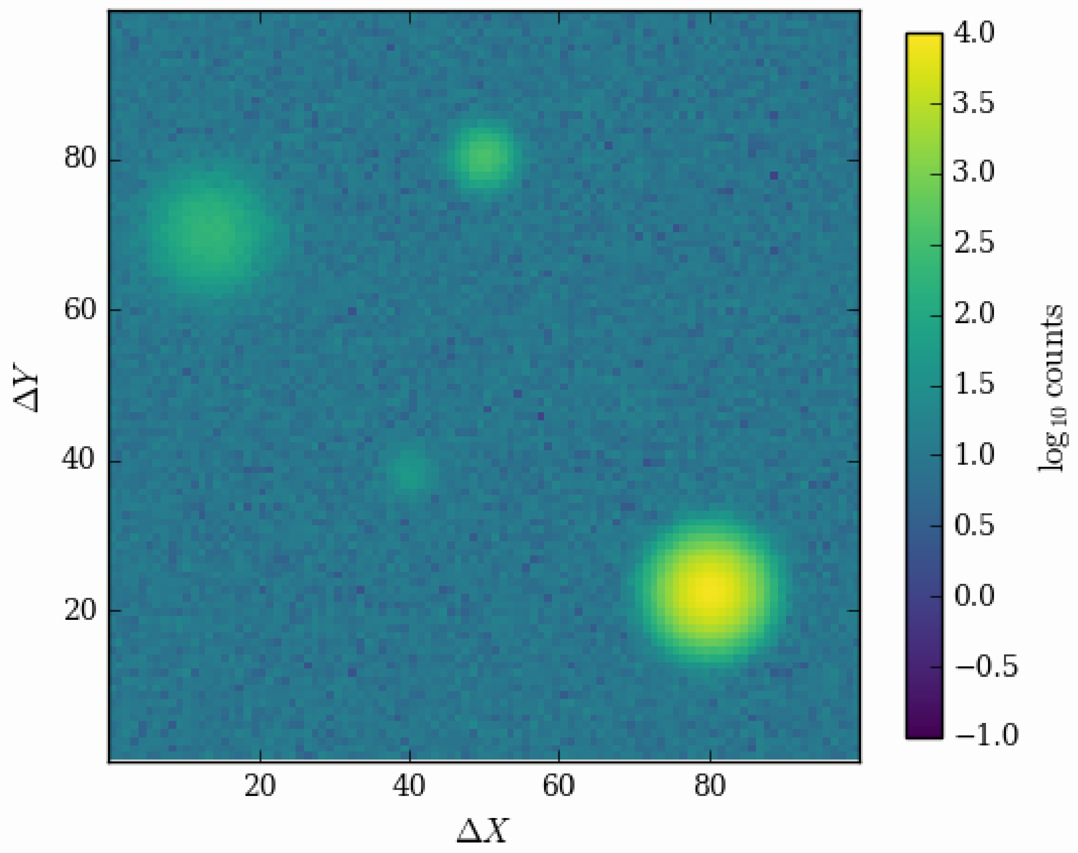
Challenge Solutions

Two challenges:

- CCD image
- Integer matching

CCD Image

https://gist.github.com/apcooper/607a52a98e1dca9169f0aaffa5af0283 (see github wiki)



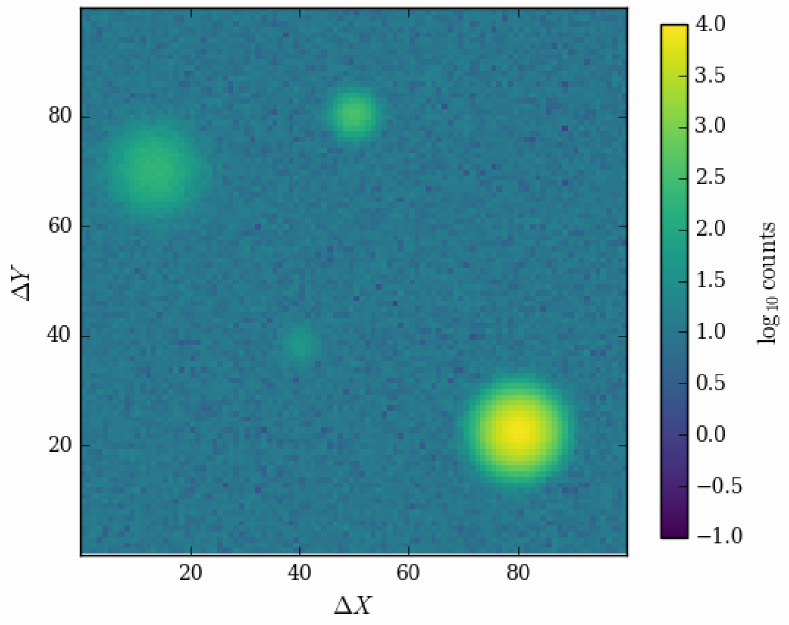
```python
def plot_challenge(input_file):
    """
    Reads data from input file and ...
    """
    # We want random but reproducible results, so
    # set the seed explictly:
    np.random.seed(42)

    # The number of CCD pixels in each dimension
    nx, ny = 100, 100

    # Read the input data from the file provided.
    # Use usecols= to skip the first column because
    # it's got strings in it.
    x,y,flux,fwhm = np.loadtxt(input_file,usecols=[1,2,3,4],
                                           unpack=True,dtype=np.float32)
```

```python
ccd_pixels = np.zeros((nx,ny),dtype=np.float64)

for i in range(0,n_sources): # Loop over sources
    n_counts = int(flux[i])  # Number of counts for this source.
    dim      = (n_counts,2)  # Dimensions of the array

    # Randomly sample points from distribution for this source
    photons    = np.random.normal(loc=(x[i],y[i]),
                                   scale=fwhm[i], size=dim)

    # Bin these points on on a grid with same dimensions as CCD.
    bins = (np.arange(0,nx+1),np.arange(0,ny+1))
    binned_counts, bx, by = np.histogram2d(photons[:,0],photons[:,1]
                                           bins=bins)
    # Add this component to the CCD image.
    ccd_pixels           = ccd_pixels + binned_counts
```

```
# Finally, add poisson noise (again taking advantage)
background_mean_counts = 10
ccd_pixels = ccd_pixels + np.random.poisson(background_mean_counts,
                                            size=(nx,ny))

# Print the total flux and mean flux
print('Total counts:  %e'%(np.sum(ccd_pixels)))
print('Maximum count: %e'%(np.max(ccd_pixels)))
```

Getting the orientation of the image to match the coordinates in the input file is tricky, because there are multiple conventions involved for the 'origin' of the coordinate system:

- the X,Y convention in the input file
- `numpy`'s convention for the order of columns and rows
- `matplotlib`'s convention for `imshow`



Note the transpose operator `.T` and `origin=lower`

```
    pl.imshow(np.log10(ccd_pixels).T,
              vmin=-1,vmax=4,
              cmap='viridis',
              extent=(bx[0],bx[-1],by[0],by[-1]),
              interpolation='nearest',origin='lower')
```

```python
cbar = pl.colorbar(shrink=0.75)
cbar.set_label(r'$\log_{10}\,\mathrm{counts}$',fontsize=12)

ax = pl.gca() # get the current axis

# Get rid of tick labels
pl.setp(ax.get_xticklabels()[0], visible=False) # etc.

# Set the axis labels
pl.xlabel('$\Delta X$',fontsize=12)
pl.ylabel('$\Delta Y$',fontsize=12)

# Save the plot; note 'bbox_inches' and 'pad_inches' commands.
output_path = './challenge.png'
pl.savefig(output_path,bbox_inches='tight',pad_inches=0.1)
```

```python
import numpy as np # etc..

def plot_challenge(input_file):

    # ...

    # Return the figure.
    return f

# This block is executed if run from the command line
if __name__ == '__main__':
    input_file = sys.argv[1] # Get the first command line argument
    f = plot_challenge(input_file)
    pl.close(f)
```

- The colormap is viridis. What is special about this colormap?
    - Viridis is *perceptually uniform*
    - Some colourmaps are hard to read with colour blindness or greyscale printing
    - Their high contrast in some regions creates spurious emphasis of features
    - There are other choices of perceptually uniform colourmap
    - More here: http://matplotlib.org/users/colormaps.html
- The image should look the same if you make it twice.
    - Remember to set the random seed explicitly, for reproducibility.

Integer matching

https://gist.github.com/apcooper/e7ce6e1232537c444a69b7206b42bdf0 (see github wiki)

```python
import numpy as np
import sys

def match(arr1,arr2,arr2_sorted=False):
    ...
def test_match_sorted_all_present():
    ...
def test_match_unsorted_all_present():
    ...
def test_match_unsorted_not_all_present():
    ...
def test_huge_unsorted(na,nb):
    ...
if __name__ == '__main__':
    # Run one of the tests using two integer arguments
    na,nb = int(sys.argv[1]), int(sys.argv[2])
    t = test_huge_unsorted(na,nb)
    print('Matching {} integers against {} integers took {:f}s'.format(na,nb,t))
```

```python
def match(arr1,arr2,arr2_sorted=False):
    """
    Returns an array of len(arr1) holding the index in arr2
    matching each element of arr1. For elements of arr1 that
    are not matched in arr2, elements in the returned array -1.

    Remember to check for these -1 elements when using the
    result.

    Neither arr1 nor arr2 have to be sorted first. Only arr2
    is sorted in operation.

    If arr2 is already sorted, set arr2_sorted=True to speed
    up the routine.
    Example:
        midx    = match(a,b) # Match a in b
        matched = (m >= 0)   # Mask on matched elements
        w       = np.where(matched)[0] # Select only matched

        assert(b[midx[w]] == a[w]) # True

    Credit:
        Originally based on Fortran code by John Helly.
    """
```

1. Make sure the array is sorted

```
    if arr2_sorted:
        # arr2 is already sorted
        sortidx = slice(0,len(arr2))
        tmp2    = arr2
    else:
        # arr2 not sorted, so sort it
        sortidx  = np.argsort(arr2)
        tmp2     = arr2[sortidx]
```

- We don't want to change either of the input arrays in place (no side effects).
- `tmp2` is a reference to the sorted array, `sortidx` is the indices of the array in sorted order.

2. Find where the elements of `arr1` can be inserted in `arr2`

```
idx_l = np.searchsorted(tmp2,arr1,side='left')
idx_r = np.searchsorted(tmp2,arr1,side='right')
```

This is the 'trick' at the heart of the solution. There was a strong hint given about this at the end of the `Numpy102` notebook.

3. Return -1 where no match is found.

```
    mask            = idx_r-idx_l == 0
    idx_l[mask]     = -1
```

- For elements of `arr1` that are in `tmp2`, `idx_l` holds the appropriate index of the match.
- Elements that aren't matched have `idx_r == idx_l`. Use this to set their `idx_l` entries to `-1`.

4. Put everything back in the right order.

```
    if arr2_sorted:
        matched_index = idx_l
    else:
        # The original list of indices (0,1,2...) in the order of tmp2
        original_index = np.arange(len(arr2))[sortidx]

        # Fill in the values where idx_l is positive
        matched_index = np.where(idx_l >= 0, original_index[idx_l], -1)

    return matched_index
```

- We matched against the sorted array, but we want to return the index in the original array, which might be unsorted.
- If it was we need a bit of index gymnastics to return the match indices in the input array.
- This makes use of an ability of `np.where` we haven't met before.

Write a function to time the routine

```python
def test_huge_unsorted(na,nb):
    import time
    import random # python's random, not numpy.random

    b     = np.array(random.sample(range(0,10*nb),nb))
    c     = random.sample(range(0,nb),na)
    a     = b[c]

    t0_m = time.time()
    m     = match(a,b)
    t1_m = time.time()

    return t1_m-t0_m
```

```
andrew$ python match.py 10000 1000000
Matching 10000 integers against 1000000 integers took 0.206739s
```

Changes made

The string-processing example was confusing!

This will be a challenge next time...

Questions raised:

What's going on with `_` in list comprehensions?

```
[np.log(_) for _ in range(1,100)]
```

```
andrew$ python
Python 3.6.1 | packaged by conda-forge | (default, May 23 2017, 14:31:56)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print(_)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_' is not defined
>>> x = 10
>>> print(_)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_' is not defined
>>> 10
10
>>> print(_)
10
```

Things promised but not covered

- 'Python on COSMA'.
- Multiple threads/cores.
- Data that's much larger than the available memory.

What next?

- The course materials on github will be updated to fix bugs (see github issues page).
- Eventually the github page may move to a different account.
- There is also a wiki page with 'other interesting packages'. Additions would be welcome.

- The notebook server will stay up for a while, but not indefinitely. You will still be able to download the notebooks from GitHub, but the challenge solutions might disappear after a while...

- SciencePackages: Despite what it says, you shouldn't need to install any new packages yourself for this notebook.

- SciencePackagesAstro: You will need to change `./data` to `/data/pyintro` (no leading `.`) in some of the cells.

Enjoy Python!