PHYS4038/MLiS and AS1/MPAGS

Scientific Programming in python™

mpags-python.github.io

Steven Bamford

University of Nottingham
UK | CHINA | MALAYSIA

An introduction to scientific programming with

python™

Session 10:

Robust, fast & friendly code

Outline

- Testing for robust code
- Optimising your code
- Squeezing out extra speed
- Graphical interfaces

Writing robust code

Tests

- **Unit tests**
  - test individual units of code
  - specific units
    - e.g. a single function or interaction between functions
  - tested as generally as possible
- **Functional tests**
  - test the whole programme under a variety of inputs
- **Regression tests**
  - check for inconsistent behaviour between consecutive versions
  - detect new bugs, ensure old bugs do not reoccur

Tests

- Main testing frameworks
  - **unittest** is the main Python module
  - **doctest** enables tests in documentation strings
  - **pytest** is the most popular third-party module
    - `conda install pytest`
    - nicely automates testing, and preferred by astropy
    - interoperable with other frameworks
    - basically just name any tests `test_*`
      - files, functions, methods, classes (Test…)
- astropy has detailed testing guidelines:
  - http://docs.astropy.org/en/stable/development/testguide.html

Tests

mycode.py

```
def func(x):
    """Add two to the argument."""
    return x + 1

def test_answer():
    """Check the return value of func() for an example argument."""
    assert func(3) == 5
```

Tests

```
$ pytest mycode.py
============ test session starts ===========================
platform darwin -- Python 3.7.4, pytest-5.3.1
rootdir: /Users/spb/Work/teaching/mpags_python/test_demo
collected 1 item
mycode.py F
   [100%]
============ FAILURES ======================================
____________ test_answer __________________________________
    def test_answer():
        """Check the return value of func() for an example
   argument."""
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

mycode.py:7: AssertionError
============ 1 failed in 0.04s ==============================
```

Tests

- Online testing (continuous integration) services
    - GitHub Actions
- Also, CircleCI, Jenkins, Travis CI, Azure Pipelines
- Test coverage reports
    - Coveralls: https://coveralls.io

Optimising your code

Testing performance

timeit – use in interpreter, script or command line

```
python -m timeit [-n N] [-r N] [-s S] [statement ...]
```

Options:

-s S, --setup=S

statement to be executed once initially (default pass)

-n N, --number=N

how many times to execute 'statement' (default: take ~0.2 sec total)

-r N, --repeat=N

how many times to repeat the timer (default 3)

IPython/Jupyter magic version

```
%timeit    # one line
%%timeit   # whole notebook cell
```

Testing performance

```
# fastest way to calculate x**5?

$ python -m timeit -s 'from math import pow; x = 1.23' 'x*x*x*x*x'
10000000 loops, best of 3: 0.161 usec per loop

$ python -m timeit -s 'from math import pow; x = 1.23' 'x**5'
10000000 loops, best of 3: 0.111 usec per loop

$ python -m timeit -s 'from math import pow; x = 1.23' 'pow(x, 5)'
1000000 loops, best of 3: 0.184 usec per loop
```

Profiling

- Understand which parts of your code limit its execution time
  - print summary to screen, or save file for detailed analysis

From shell

```
$ python -m cProfile -o program.prof my_program.py
```

From IPython/Jupyter

```
%prun -D program.prof my_function()

%%prun  # profile an entire notebook cell
```

Lots of functionality… see docs

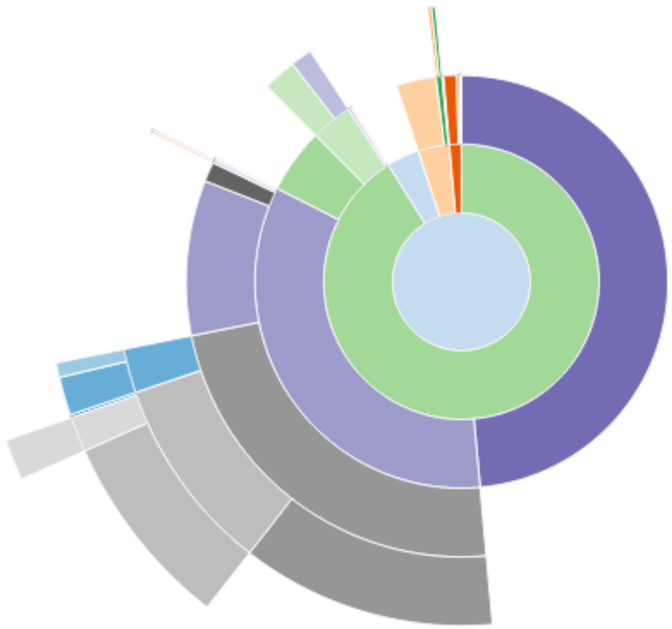
Profiling

Nice visualisation with **snakeviz** – http://jiffyclub.github.io/snakeviz/

```
$ conda install snakeviz

OR

$ pip install snakeviz
```

In IPython/Jupyter:

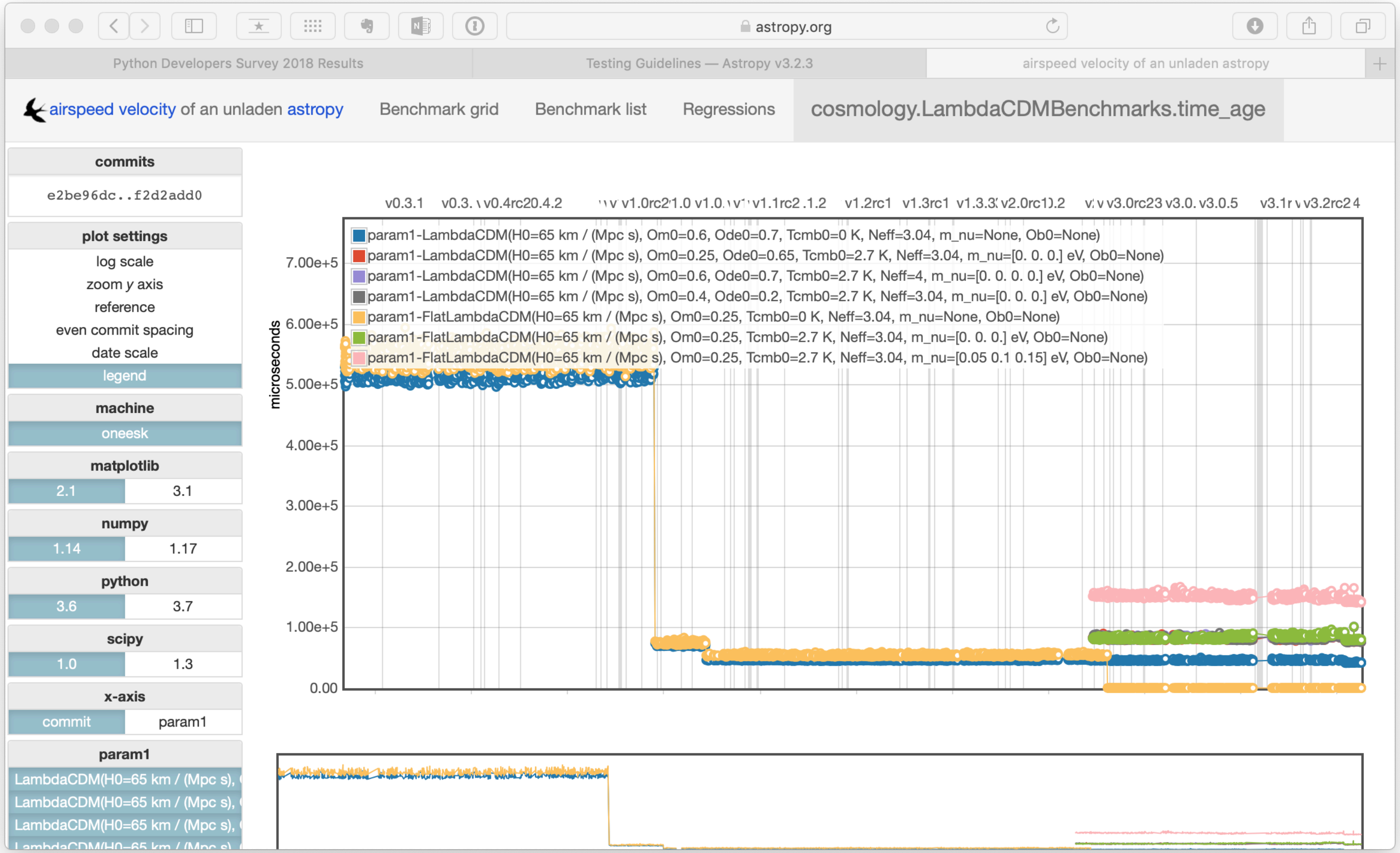
```
%load_ext snakeviz

%snakeviz my_function()

%%snakeviz   # profile entire cell
```



Benchmarking

Regular timing tests to check for performance regression

- **pytest-benchmark**
- **airspeed velocity**



Squeezing out extra speed

Numba

- JIT: just in time compilation of Python functions
- Compilation for both CPU and GPU hardware

nprimes.py

```
from numba import jit

@jit
def primes(kmax):
    # same code as original pure python version
    ...
    return p
```

```
$ python -m timeit -s 'import nprimes as p' 'p.primes(100)'
1000 loops, best of 3: 44.2 usec per loop
```

30x speedup

Numba

- Easy, automatic parallelization

```
from numba import vectorize

@jit(parallel=True)
def sum(a, b):
    return a + b
```

- Create your own optimised numpy 'ufuncs'

```
from numba import vectorize, float32

@vectorize('float32(float32, float32)'], target='parallel')
def sum(a, b):
    return a + b

@vectorize('float32(float32, float32)'], target='gpu')
def sum(a, b):
    return a + b
```

Mixing Python and C – fast and flexible

Cython is used for compiling Python-like code to machine-code

- supports a big subset of the Python language
- conditions and loops run 2-8x faster, overall 30% faster for plain Python code
- add types for speedups (hundreds of times)
- easily use native libraries (C/C++/Fortran) directly

- Cython code is turned into C code
    - uses the CPython API and runtime

- Coding in Cython is like coding in Python and C at the same time!

Some material borrowed from Dag Sverre Seljebotn (University of Oslo) EuroSciPy 2010 presentation

Cython

Use cases:

- Performance-critical code
  - which does not translate to array-based approach (numpy / pytables)
  - existing Python code → optimise critical parts
- Wrapping existing C/C++ libraries
  - particularly higher-level Pythonised wrapper
  - for one-to-one wrapping other tools might be better suited

Cython

Cython code must be compiled (but this can be automated)

Two stages:

- A .pyx file is compiled by Cython to a .c file, containing the code of a Python extension module
- The .c file is compiled by a C compiler
  - Generated C code can be built without Cython installed
  - Cython is a developer dependency, not a build-time dependency
  - The result is a .so file (or .pyd on Windows) which can be imported directly into a Python session

Cython

Ways of building Cython code:

- Run cython command-line utility and compile the resulting C file
  - use favourite build tool
  - for cross-system operation you need to query Python for the C build options to use
- Use pyximport to importing Cython .pyx files as if they were .py files; building on the fly (recommended to start).
  - things get complicated if you must link to native libraries
  - larger projects tend to need a build phase anyway
- Write a distutils setup.py
  - standard way of distributing, building and installing Python modules

Cython

- Cython supports most of normal Python
- Most standard Python code can be used directly with Cython
  - typical speedups of (very roughly) a factor of two
  - should not ever slow down code – safe to try
  - name file .pyx or use pyimport = True

```
>>> import pyximport
>>> pyximport.install()
>>> import mypyxmodule  # converts and compiles on the fly

>>> pyximport.install(pyimport=True)
>>> import mypymodule  # converts and compiles on the fly
                       # should fall back to Python if fails
```

Cython

- Big speedup from defining types of key variables
- Use native C-types (int, double, char *, etc.)
- Use Python C-types (Py_int_t, Py_float_t, etc.)
- Use `cdef` to declare variable types
- Also use `cdef` to declare C-only functions (with return type)
  - can also use `cpdef` to declare functions which are automatically treated as C or Python depending on usage
- Don't forget function arguments (but note `cdef` not used here)

Cython – primes example

- Efficient algorithm to find first *N* prime numbers

primes.py

```
def primes(kmax):
    p = []
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            k = k + 1
            p.append(n)
        n = n + 1
    return p
```

```
$ python -m timeit -s 'import primes as p' 'p.primes(100)'
1000 loops, best of 3: 1.35 msec per loop
```

Cython – primes example

cprimes.pyx

```
def primes(kmax):
    p = []
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            k = k + 1
            p.append(n)
        n = n + 1
    return p
```

```
$ python -m timeit -s 'import pyximport;
  pyximport.install(); import cprimes as p' 'p.primes(100)'

1000 loops, best of 3: 731 usec per loop
```

1.8x speedup

Cython – primes example

xprimes.pyx

```
def primes(int kmax):   # declare types of parameters
    cdef int n, k, i    # declare types of variables
    cdef int p[1000]    # including arrays
    result = []         # can still use normal Python types
    if kmax > 1000:     # in this case need to hardcode limit
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result       # return Python object
```

contains only C-code

40.8 usec per loop

33x speedup

Cython and Numpy

- Cython provides a way to quickly access Numpy arrays with specified types and dimensionality

  → for implementing fast specific algorithms

- Can be useful, but often using functions provided by numpy, scipy, numexpr or pytables will be easier and faster

Graphical interfaces

GUIs

- Give your scientific code a friendly face!
  - easy configuration
  - monitor progress
  - particularly for public code, cloud computing, HPC

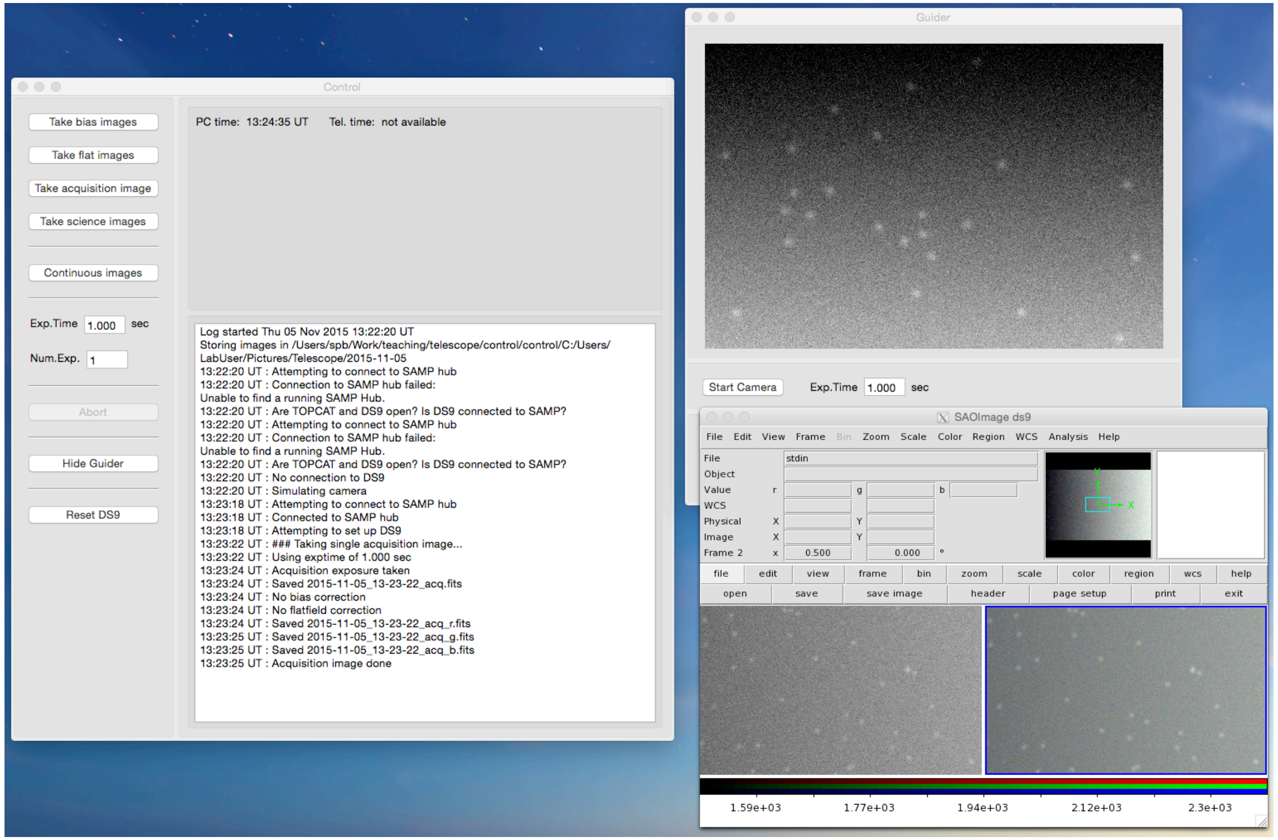
- Many modules to construct GUIs in Python…
- Tkinter – built-in
- Qt – C++
- wx – C++
- Remi – browser based
- PySimpleGUI – one interface, multiple GUI frameworls
- Kivy – modern and cross-platform

GUIs

Example using **wxpython**

www.wxpython.org

https://github.com/bamford/control/



GUIs

For simple GUI, especially if output is a plot…

matplotlib **widgets** are very useful

- layout controls on a figure canvas
- functionality implemented using *callback* functions:
  - every time a control is activated it will call the function
  - function then examines the event and takes action

In Jupyter notebooks…

IPython **widgets** provide a quick graphical interface

Python web frameworks

Most popular…

Flask

light and flexible, more explicit, good for smaller projects

Django

full-featured, automated, good for getting big projects going quickly

but also…

Pyramid, web2py, …

- An (unscientific) example

An introduction to scientific programming with

python™

The End