Python for scientific calculations

Ben Dudson, 16th June 2017

Why python?

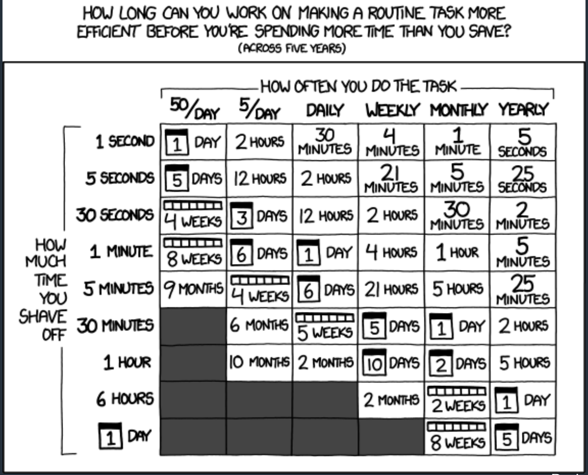
- Python is an expressive and flexible language
- Relatively easy for new users to learn, without limiting advanced users
- Can be run interactively, making debugging and experimentation easier
- A huge and easily accessible collection of libraries (pip install . . .)
  - **SciPy** `https://docs.scipy.org`
  - **Matplotlib** `https://matplotlib.org/`
  - **Sympy** `http://www.sympy.org`
  - **Scikits** `https://scikits.appspot.com/scikits`
  - **Pandas** `http://pandas.pydata.org/`
- Generally less code to write. Quicker and generally fewer bugs

Why python for scientific work?

1. Data analysis and interactive exploration
2. Many research problems need only moderate resources
   - A modern processor (e.g. Intel i7) has a peak performance of nearly 100 GFlops
   - Compare to Cray-2 (1985) with 1.9 GFlops
3. For larger problems it's often not clear what algorithm to use
   - Try out different things and fail quickly

Motivation : Time

- Implementation time vs execution time
- Absolute speed is not important. What matters is acceptable speed



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK: 50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS				2 MONTHS	2 WEEKS	1 DAY
1 DAY					8 WEEKS	5 DAYS

Motivation : Cost

- A PDRA-day : £50k / 260 days = £200
- Amazon EC2 compute nodes : 4p per core-hour
- Archer (notional) cost : 20p for 24 core-hours

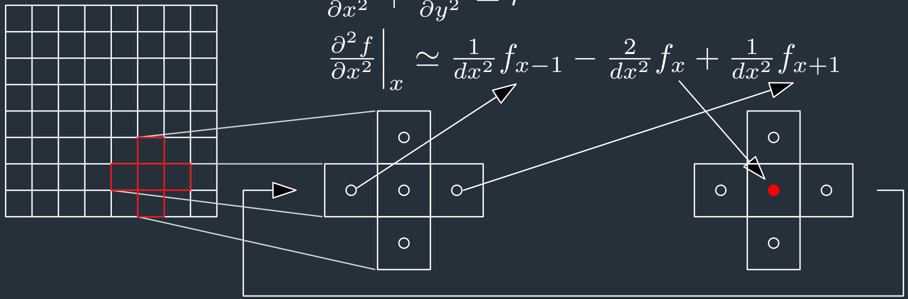
So every day spent optimising a code needs to save over 5,000 core-hours (200 days) on EC2 to be worthwhile, or 24,000 core-hours on Archer.

Example: Solving Laplace equation

2D solution to Laplace's equation using Jacobi iteration

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = r$$

$$\left.\frac{\partial^2 f}{\partial x^2}\right|_x \simeq \frac{1}{dx^2} f_{x-1} - \frac{2}{dx^2} f_x + \frac{1}{dx^2} f_{x+1}$$



Implementation in Python

Solving a Laplacian in 2D (x,y)

```
def solve(rhs, dx, dy, tol=1e-3):
    result = np.zeros(rhs.shape)

    while True:
        last = result
        result = jacobi_iteration(last, rhs, dx, dy)

        change = np.amax(np.abs(result - last))
        if change < tol:
            return result
```

Implementation in Python

Solving a Laplacian in 2D (x,y)

```
def jacobi_iteration(last, rhs, dx, dy):
    out = last.copy()

    nx,ny = last.shape
    for x in range(1,nx-1):
        for y in range(1, ny-1):
            out[x,y] = (  (last[x+1,y] + last[x-1,y])/dx**2
                        + (last[x,y+1] + last[x,y-1])/dy**2
                        - rhs[x,y] )
                       / (2./dx**2 + 2./dy**2)
    return out
```

Profiling

Before optimising this, we first need to:

1. Add tests, to make sure it's correct and we don't break it
2. Measure it : Is it fast enough, and if not then why?

Some tools to do the measuring:

- **timeit** `https://docs.python.org/3/library/timeit.html`
- **cProfile** `https://docs.python.org/3/library/profile.html`
- **line profiler** `https://github.com/rkern/line_profiler`
- **pProfile** `https://github.com/vpelletier/pprofile`

Timing using timeit

```python
import timeit

def fun():
    result = solve(rhs, dx, dy)

niter = 10
time = timeit.Timer(fun, 'gc.enable()').timeit(number=niter)/niter
```

Note:

- Timer can either be given a string or a function without arguments
- By default the garbage collector is turned off
- The time returned by timeit() is for all iterations

The Python interpreter is slow

Timing in seconds, comparing against a C implementation compiled with -O3

	10x10	100x100	1000x1000	10000x10000
C code	2.47e-06	1.08e-04	4.07e-03	1.13
Python	2.00e-03	7.62e-02	1.58	159.5
Ratio	810	706	388	141

cProfile gives function timings

```python
import cProfile
cProfile.run('fun()')
```

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.009    0.009   11.082   11.082 01-python.py:17(solve)
     1   11.059   11.059   11.071   11.071 01-python.py:3(jacobi_iteration
     1    0.000    0.000   11.082   11.082 01-python.py:44(fun)
     1    0.000    0.000   11.082   11.082 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 _methods.py:15(_amax)
     1    0.000    0.000    0.001    0.001 fromnumeric.py:2048(amax)
     1    0.005    0.005    0.005    0.005 {method 'copy' of 'numpy.ndarra
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.P
     1    0.001    0.001    0.001    0.001 {method 'reduce' of 'numpy.ufun
     1    0.002    0.002    0.002    0.002 {numpy.core.multiarray.zeros}
   999    0.007    0.000    0.007    0.000 {range}
```

line profiler gives individual line timings

```
@profile
def jacobi_iteration(last, rhs, dx, dy):
...

$ kernprof -l -v 01-python.py

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def jacobi_iteration(last, r
     8         1         1607   1607.0      0.0      result = last.copy()
    10         1            4      4.0      0.0      nx,ny = last.shape
    11       999          705      0.7      0.0      for x in range(1,nx-1):
    12    997002       865450      0.9      5.9          for y in range(1, ny
    14    996004     13769696     13.8     94.1              result[x,y] = (
    15         1            1      1.0      0.0      return result
```

Why is python so slow?

Many of its nice features (for humans) lead to poor performance:

- **Types** : Python has a very flexible dynamic type system, only known at run time
- **Flexibility** : Python allows objects to be modified in many ways, which means lots of checks
- **No threading** : Reference counting and thread locking remove performance benefit of threads

Inside the Python Virtual Machine:
`http://leanpub.com/insidethepythonvirtualmachine`

Disassembling Python bytecode

```
def square(x):
  return x*x

from dis import dis
dis(square)
```

```
      0 LOAD_FAST                0 (x)
      3 LOAD_FAST                0 (x)
      6 BINARY_MULTIPLY
      7 RETURN_VALUE
```

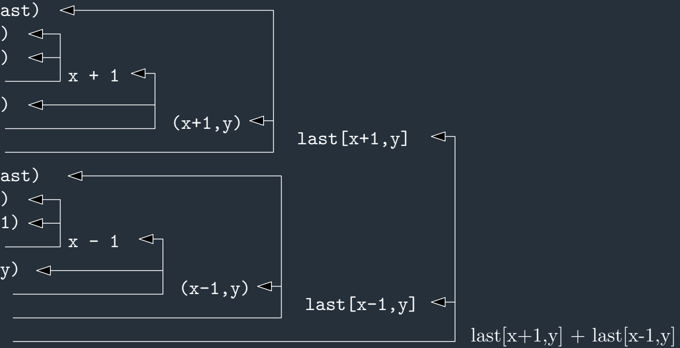
Python uses a stack system, loading variables onto a stack and applying operators

Note: The same bytecode is used for all types of `x`

Disassembling Python bytecode

```
dis(jacobi_iteration)
```

```
79 LOAD_FAST            0 (last)
82 LOAD_FAST            7 (x)
85 LOAD_CONST           1 (1)
88 BINARY_ADD
89 LOAD_FAST            8 (y)
92 BUILD_TUPLE          2
95 BINARY_SUBSCR
96 LOAD_FAST            0 (last)
99 LOAD_FAST            7 (x)
102 LOAD_CONST           1 (1)
105 BINARY_SUBTRACT
106 LOAD_FAST            8 (y)
109 BUILD_TUPLE          2
112 BINARY_SUBSCR
113 BINARY_ADD
```



Switch statement handles bytecodes in loop

https://github.com/python/cpython/blob/master/Python/ceval.c#L1158

```
TARGET(LOAD_FAST) {
    PyObject *value = GETLOCAL(oparg);
    if (value == NULL) {
        format_exc_check_arg(PyExc_UnboundLocalError,
                             UNBOUNDLOCAL_ERROR_MSG,
                             PyTuple_GetItem(co->co_varnames, oparg));
        goto error;
    }
    Py_INCREF(value);
    PUSH(value);
    FAST_DISPATCH();
}

TARGET(LOAD_CONST) {
    ...
```

Switch statement handles bytecodes in loop

https://github.com/python/cpython/blob/master/Python/ceval.c#L1158

```
TARGET(BINARY_MULTIPLY) {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *res = PyNumber_Multiply(left, right);
    Py_DECREF(left);
    Py_DECREF(right);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

Lots of type checking and indirection

https://github.com/python/cpython/blob/master/Objects/abstract.c#L954

```
PyNumber_Multiply(PyObject *v, PyObject *w) {
    PyObject *result = binary_op1(v, w, NB_SLOT(nb_multiply));
    if (result == Py_NotImplemented) {
        PySequenceMethods *mv = v->ob_type->tp_as_sequence;
        PySequenceMethods *mw = w->ob_type->tp_as_sequence;
        Py_DECREF(result);
        if  (mv && mv->sq_repeat) {
            return sequence_repeat(mv->sq_repeat, v, w);
        } else if (mw && mw->sq_repeat) {
            return sequence_repeat(mw->sq_repeat, w, v);
        }
        result = binop_type_error(v, w, "*");
    }
    return result;
}
```

CPython does not optimise

The Python compiler does not do a lot of optimisation e.g common factors:

```
inv_dx2 = 1./dx**2
inv_dy2 = 1./dy**2
inv_diag = 1. / (2./dx**2 + 2./dy**2)
for x in range(1,nx-1):
    for y in range(1, ny-1):
        out[x,y] = inv_diag * (  inv_dx2*(last[x+1,y] + last[x-1,y])
                               + inv_dy2*(last[x,y+1] + last[x,y-1])
                               - rhs[x,y] )
```

	10x10	100x100	1000x1000	10000x10000
Previous	810	706	388	141
New	482	389	215	79

Better ways : Don't use Python!

Trying to optimise the Python interpreter is not a productive way forward...

Use Python as "glue" to organise calls to C/Fortran code:

- NumPy, SciPy : `http://www.scipy-lectures.org/`
- Numexpr : `https://github.com/pydata/numexpr`
- Numba (JIT compiler) : `http://numba.pydata.org/`
- PyPy : `http://pypy.org/` (was Psyco)

Note: You can install many packages as user

```
pip install --user numexpr
```

The easiest way: NumPy

```
def jacobi_iteration(last, rhs, dx, dy):
    out = last.copy()

    out[1:-1,1:-1] = (  (last[2:,1:-1] + last[:-2,1:-1])/dx**2
                      + (last[1:-1,2:] + last[1:-1,:-2])/dy**2
                      - rhs[1:-1,1:-1] )
                     / (2./dx**2 + 2./dy**2)

    return out
```

	10x10	100x100	1000x1000	10000x10000
Previous	482	389	215	79
New	98	6.0	5.5	7.7

Adding out parameter improves a little

A common pattern in NumPy code is an “out” argument, which reduces memory allocation

```
def jacobi_iteration(last, rhs, dx, dy, out=None):
    if out is None:
        out = last.copy()

    out[1:-1,1:-1] = (  (last[2:,1:-1] + last[:-2,1:-1])/dx**2
                      + (last[1:-1,2:] + last[1:-1,:-2])/dy**2
                      - rhs[1:-1,1:-1] )
                     / (2./dx**2 + 2./dy**2)

    return out
```

Inlining to remove function calls

```
def solve(rhs, dx, dy, tol=1e-3):
    result = np.zeros(rhs.shape)
    last = result.copy()

    while True:
        last, result = result, last # swap

        result[1:-1,1:-1] = (  (last[2:,1:-1] + last[:-2,1:-1])/dx**2
                             + (last[1:-1,2:] + last[1:-1,:-2])/dy**2
                             - rhs[1:-1,1:-1] )
                            / (2./dx**2 + 2./dy**2)

        change = np.amax(np.abs(result - last))
        if change < tol:
            return result
```

Numexpr is easy to try

```
rhs_middle = rhs[1:-1,1:-1]

while True:
    last, result = result, last # swap

    xm = last[:-2,1:-1] # at x-1
    xp = last[2:,1:-1]  # at x+1
    ym = last[1:-1,:-2] # at y-1
    yp = last[1:-1,2:]  # at y+1

    result[1:-1,1:-1] = ne.evaluate("(  (xm + xp)/dx**2
                                      + (ym + yp)/dy**2
                                      - rhs_middle )
                                     / (2./dx**2 + 2./dy**2)")
```

Summary of timings

Relative to C implementation (in seconds)

	10x10	100x100	1000x1000	10000x10000
C code (-O3)	2.47e-06	1.08e-04	4.07e-03	1.13
Simple python	810	706	388	141
Opt. python	482	389	215	79
NumPy	98	6.0	5.5	7.7
NumPy out	97	5.8	5.1	7.7
Numpy inline	97	5.7	5.2	7.5
Numexpr	195	7.1	3.2	2.3

Just In Time (JIT) compilers

- The first time a function is called is slow as it compiles
- More information is available about types, resulting in faster code

Numba is a package which adds JIT support to Python

```
from numba import jit

@jit    # Numba decorator compiles function when called
def jacobi_iteration(last, rhs, dx, dy):
    ... # original version with nested for loops
```

	10x10	100x100	1000x1000	10000x10000
Original	1246	697	284	174
Numba	80	3.2	3	1.5

Just In Time (JIT) compilers

- The first time a function is called is slow as it compiles
- More information is available about types, resulting in faster code

PyPy is a separate implementation of Python

- JIT compiles everything
- Compatible with standard CPython
- Many libraries but not all: NumPy but not SciPy
- Retains Global Interpreter Lock (no threading)

I found both Numba and PyPy quite hard to install (works on sausage)

Cython for optimisation and linking

http://cython.org/

1. Compiles to C. Optimises if given additional type information

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        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:
```

Cython for optimisation and linking

http://cython.org/

1. Compiles to C. Optimises if given additional type information
2. Easily links to C and Fortran code

```
cdef extern from "idamclient.h":
    bint getIdamProperty(const char *property)

def getProperty(property):
    "Get a property for client/server behavior"
    return getIdamProperty(property)
```

Parallel programming

Due to the Global Interpreter Lock, standard Python (CPython) does not do threading

- **multiprocessing** This provides a way to spawn multiple Python processes and pass data between them
- **MPI4py**
- GPU programming
  - **Theano**, **Tensorflow**
  - **PyCUDA**

Many libraries for large scientific problems:

- **Fenics** `https://fenicsproject.org/`
- **Firedrake** `http://www.firedrakeproject.org`
- **PyFR** `http://www.pyfr.org/`

Conclusions

Always treat benchmarks like this with extreme caution: Test for your problems

- Many libraries for scientific computing: Don't reinvent the wheel!
- Use NumPy (and SciPy) whenever possible
- Numexpr is simple to try, but not always faster
- Numba looks impressive but hard to install
- Cython requires some time investment, but allows optimisation and coupling to C/Fortran code
- Huge number of useful packages, including for parallel and GPU programming

"Premature optimization is the root of all evil" – D.Knuth