Speeding up Python

Liam Pattinson

Python for Scientific Computing

Python is one of the most popular programming languages for use in scientific computing, and for good reasons:

- Easy to learn the basics, and extremely powerful for veteran users.
- Designed with readability in mind.
- Very well-developed and interconnected scientific ecosystem (NumPy, SciPy, Matplotlib, etc.)
- Dynamic type system allows you to write expressive and generic code relatively easily.

It almost seems too good to be true...

The Problem with Python

Python is *SLOW*... and there are multiple reasons for this[1]:

- Python is an *interpreted* language.
- Python's *dynamic typing* has a significant performance overhead.
- Python can't use multithreading due to the *Global Interpreter Lock*.

[1] There are other performance issues, such as the use of a garbage-collection system, but we won't cover these here.

Why Python is so slow: interpreted language

- Languages like C and Fortran are *compiled* to machine code, which can be directly understood by your hardware.
- Python is instead *interpreted*, meaning it is run one command at a time by the CPython runtime (or some other interpreter)[2].

[2] Technically speaking, it's compiled to *bytecode*, which is itself interpreted.

Why Python is so slow: interpreted language

Use the `dis` built-in module to 'disassemble' Python into a human-readable representation of bytecode:

```
>>> from dis import dis
>>> def myfunc(x):
...     return x + 1
>>> dis(myfunc)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (1)
              4 BINARY_ADD
              6 RETURN_VALUE
```

There are separate **runtime** function calls to place `x` on the stack, place `1` on the stack, run the `BINARY_ADD` function, and to return the result!

Why Python is so slow: dynamic typing

- In a language like C++, almost everything about an object must be known at compile time: type, size, member data, functions, etc. The compiler can use this info to optimise the resulting machine code.
- Python instead allows us to use 'duck typing':
  - *"If it looks like a duck and quacks like a duck, then it must be a duck"*
- The downside is that the interpreter can't make any assumptions about objects, and every operation requires repetitive runtime checks.

Why Python is so slow: the GIL

The 'Global Interpreter Lock' (GIL) is a feature of Python that ensures thread safety. However, it does so by effectively limiting Python code to single-threaded mode only.

- Python uses *reference counting* to determine when objects should be removed from memory.
- Problem: if you could have multiple threads interpreting Python code at once, there could be race conditions where two threads try to modify a reference count at once.
- Putting a mutex/lock on every reference count would impose a significant performance penalty to single-threaded code (and could lead to deadlocks).
- The easy solution: place a lock on the *Python interpreter itself*, so only one thread can interpret Python at a time.

Why Python is so slow: the solutions

So how can we optimise our way around these problems?

- Minimise the number of instructions to achieve a given task.
- Delegate the performance-critical parts of our code to a compiled library (e.g. NumPy)
- Write our own compiled code using C/C++/Fortran/Rust/etc and create Python bindings. There's also Cython, which is purpose designed for the task.
- Use a 'Just-in-Time' (JIT) compiler to compile our code at runtime.

Is it worth optimising?

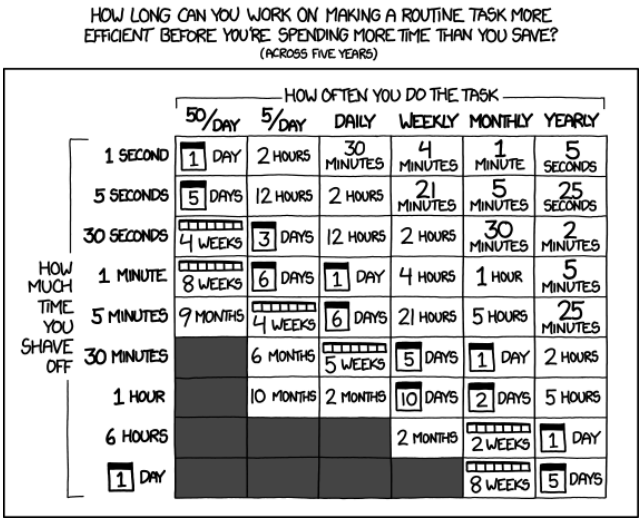


Figure: xkcd 1205, (CC BY-NC 2.5)

Is it worth optimising?

Things to consider:

- Would the time saved in the long run make up for the time spent optimising?
- Would optimising the code reduce its readability/maintainability?
- Is there an external library we could leverage instead of optimising by hand?
- Is the problem actually one of algorithmic complexity?

Always write tests before optimising! There's no point in getting the wrong answer quickly.

Profiling

- Premature optimisation is the root of all evil! It's import to know where the major bottlenecks exists so we can focus our attention where it really matters.
- Python has a built-in tool `cProfile` that can tell us how much time we spend in each function:

```
$ python -m cProfile -s cumulative my_script.py
```

- `[-s cumulative]` instructs `cProfile` to sort the output results with the most expensive operations first.

Profiling

Let's profile this simple script that makes two lists of random numbers and adds them together:

```
from random import random

def make_list() -> list[float]:
    return [random() for _ in range(10000000)]

def add_lists(a: list[float], b: list[float]) -> list[float]:
    return [x + y for x, y in zip(a, b)]

a = make_list()
b = make_list()
c = add_lists(a, b)
```

Profiling

The output of `cProfile` can be a little hard to read:

```
python -m cProfile -s cumulative my_script.py
         20000833 function calls (20000806 primitive calls) in 2.640 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      3/1    0.000    0.000    2.640    2.640 {built-in method builtins.exec}
        1    0.000    0.000    2.640    2.640 my_script.py:1(<module>)
        2    0.000    0.000    2.245    1.123 my_script.py:3(make_list)
        2    1.476    0.738    2.245    1.123 my_script.py:4(<listcomp>)
 20000000    0.770    0.000    0.770    0.000 {method 'random' of '_random.Random' objects}
        1    0.000    0.000    0.394    0.394 my_script.py:6(add_lists)
        1    0.394    0.394    0.394    0.394 my_script.py:7(<listcomp>)
      6/1    0.000    0.000    0.001    0.001 <frozen importlib._bootstrap>:1022(_find_and_load)
...
```

After staring at it for a while, you'll see that generating the random lists takes about six times as long as adding them!

Other profiling tools

- pProfile is a popular alternative:

```
$ python -m pprofile  my_script.py
Command line: my_script.py
Total duration: 42.0315s
File: my_script.py
File duration: 42.0314s (100.00%)
Line #|      Hits|         Time| Time per hit|      %|Source code
------+----------+-------------+-------------+-------+-----------
     3|         3|  8.82149e-06|   2.9405e-06|  0.00%|def make_list():
     4|  20000006|      27.7651|  1.38825e-06| 66.06%|    return [random() for _ in range(10000000)]
(call)|         2|       27.765|      13.8825| 66.06%|# my_script.py:4 <listcomp>
     5|         0|           0|           0|  0.00%|
     6|         2|  9.77516e-06|  4.88758e-06|  0.00%|def add_lists(a, b):
     7|  10000003|      14.2663|  1.42663e-06| 33.94%|    return [x + y for x, y in zip(a, b)]
(call)|         1|      14.2662|      14.2662| 33.94%|# my_script.py:7 <listcomp>
     8|         0|           0|           0|  0.00%|
     9|         1|  1.64509e-05|  1.64509e-05|  0.00%|a = make_list()
(call)|         1|      13.7482|      13.7482| 32.71%|# my_script.py:3 make_list
    10|         1|  2.21729e-05|  2.21729e-05|  0.00%|b = make_list()
(call)|         1|      14.0168|      14.0168| 33.35%|# my_script.py:3 make_list
    11|         1|  2.67029e-05|  2.67029e-05|  0.00%|c = add_lists(a, b)
(call)|         1|      14.2663|      14.2663| 33.94%|# my_script.py:6 add_lists
```

Timing our code

Python has built-in tools for timing our code in the `timeit` module.

```
>>> def add_lists(a: list[float], b: list[float]) -> list[float]:
        return [x + y for x, y in zip(a, b)]

>>> from timeit import timeit
>>> from random import random

>>> a = [random() for _ in range(100000)]
>>> b = [random() for _ in range(100000)]
>>> timeit(lambda: add_lists(a, b), number=100) / 100
0.003945145839825272
```

- It gives the total time over `number` repetitions. The higher this is, the more reliable your results will be.

Pure Python

- A lot of common Python patterns can be rephrased to cut down on the number of instructions we run.
- However, there's only so much we can do to improve the speed of Python without invoking external libraries.

Pure Python: reducing instructions

We'll use a simple problem for our example: finding the second derivative of a function using finite differences:

$$\frac{\partial^2 f_i}{\partial x^2} \approx \frac{f_{i-1} - 2f_i + f_{i+1}}{\Delta x^2} \tag{1}$$

A common approach to solving this sort of problem is to create a new list and append to it:

```
second_deriv = []
for i in range(1, len(f)-1):
   second_deriv.append((f[i-1] - 2*f[i] + f[i+1]) / dx**2)
```

Pure Python: reducing instructions

```
second_deriv = []
for i in range(1, len(f)-1):
   second_deriv.append((f[i-1] - 2*f[i] + f[i+1]) / dx**2)
```

What are the problems here?

- Recalculating `dx**2` every iteration.
- Indexing the lists rather than iterating over them directly.
- Many calls to the append function. Would be faster to use a *list comprehension*.

Pure Python: reducing instructions

```
# Calculate common factor in advance
inv_dx2 = 1.0 / dx**2
# Use list comprehension
second_deriv = [
    (f1 - 2*f2 + f3) * inv_dx2
    for f1, f2, f3 in zip(f[:-2], f[1:-1], f[2:]) # Iterate directly over lists
]
```

- Original version: 2.634s
- New version: 1.926s

We could do even better if we used some iterator magic to avoid taking slices of the lists.

Pure Python: Multiprocessing

- Due to the existence of the Global Interpreter Lock (GIL), we can't use multiple threads to execute our Python code in parallel[3].
- However, for 'embarrassingly parallel' problems, we can use *multiprocessing*. This avoids the GIL because each process gets its own set of data.
- The inputs and outputs of multiprocessing functions must be 'pickleable'.
- Alternatives: the library `tqdm` offers a nicer interface, and gives you command line progress bars!

[3] There is a built-in `threading` module, but it is only really of use for performing IO alongside computation, as this can sidestep the GIL. It works similarly to `multiprocessing`, and may be of use for IO-bound applications, such as those relying heavily on web services.

Pure Python: Multiprocessing

As an example, lets say we're trying to read a bunch of `.csv` files and we're taking the mean over some variable:

```
filenames = [f"my_data_{x}.csv" for x in range(1000)]
def get_mean(csv_filename):
    ...

# Serial
mean_values = [get_mean(filename) for filename in filenames]

# Multiprocessing
from multiprocessing import Pool

with Pool as p:
    mean_values = p.map(get_mean, filenames)
```

NumPy

- NumPy is the basis of much of the scientific Python ecosystem. Many other libraries use it as a backend, or can interface with it directly:
  - SciPy
  - Pandas
  - scikit-learn
  - xarray
- To get the most out of NumPy, it's important to do as little work in Python as possible!

NumPy: Vectorisation

- NumPy's `ndarray` should be used in place of lists for numerical calculations, but your performance likely won't improve if you're still using Python `for` loops:

```
>>> import numpy as np
>>> from timeit import timeit
>>> a, b = np.random.random(10000000), np.random.random(10000000)
>>> list_a, list_b = list(a), list(b)
>>> timeit(lambda: [x+y for x, y in zip(list_a, list_b)], number=10) / 10
0.6331382742966525
>>> timeit(lambda: [x+y for x, y in zip(a,b)], number=10)) / 10
1.0967841257923283
```

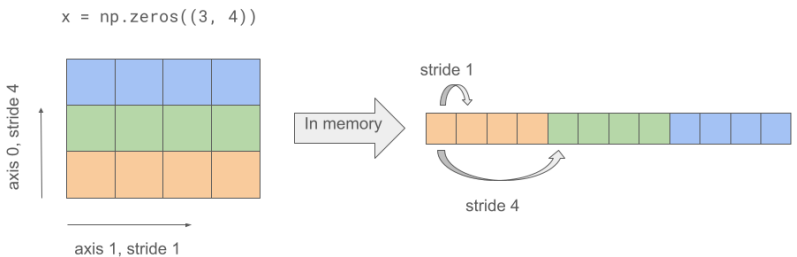
- You need to perform operations over whole arrays to get the full performance benefits. This is known as *vectorisation*[4].

```
>>> timeit(lambda: a+b, number=10) / 10
0.0733434968977235
```

[4] Not to be confused with the SIMD concept, although NumPy is probably offering that too!

NumPy: Strided arrays

- Internally, NumPy represents N-dimensional arrays via *strided* arrays.
- To increment in the Nth dimension, you jump forward in memory by the Nth stride:



- Many of NumPy's key features are powered by stride tricks under the hood.

NumPy: Slicing

- List slicing creates a shallow copy of the original list. NumPy instead creates a *view* of the original, which is much more efficient:

```
>>> a = np.random.random(100000000)
>>> list_a = list(a)
>>> timeit(lambda: list_a[1:-1], number=10) / 10
1.1540974205941893
>>> timeit(lambda: a[1:-1], number=10) / 10
9.231967851519585e-07
```

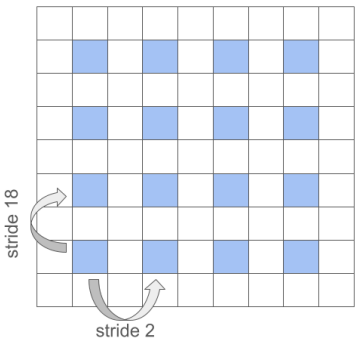
- A view can have a different shape and stride to the original array, while still referencing the same data:

```
>>> a[1:-1].shape
(99999998,)
>>> a[1:-1].strides # Multiplied by 8, as floats are 8 bytes long
(8,)
>>> a[::-1].strides # View of original array in reverse
(-8,)
```

NumPy: Slicing

By playing with strides, NumPy can create more complex views.

```
>>> x = np.zeros((9, 9))
>>> x.shape
(9, 9)
>>> x.strides
(72, 8)
>>> x[1:-1:2,1:-1:2].shape
(4, 4)
>>> x[1:-1:2,1:-1:2].strides
(144, 16)
```



NumPy: Transposes and Reshapes

- NumPy can transform arrays in various ways without incurring expensive copies by adjusting strides. For example, transposes and reshapes:

```
>>> x = np.zeros((3, 4))
>>> x.strides
(32, 8)
>>> x.T.strides
(8, 32)
>>> x.reshape((3, 2, 2)).strides
(32, 16, 8)
>>> x.ravel().strides  # Get underlying 1D array
(8,)
```

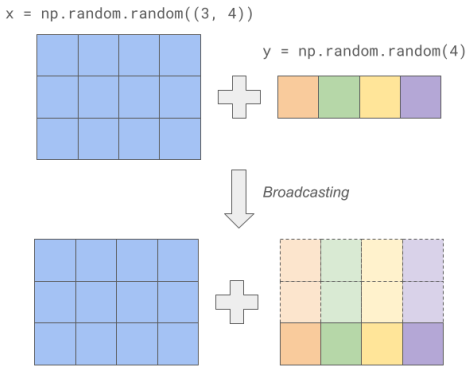
- However, if a transformation can't be achieved by adjusting strides, NumPy will quietly perform a copy – be careful of this!

```
>>> x.ravel().base is x  # Is x.ravel() just a view to x?
True
>>> x[::2, :].ravel().base is x
False
```

NumPy: Broadcasting

If you try to perform a binary operation over two arrays, they must have the same shape, or be *broadcastable* to the same shape.

- A dimension of size 1 can be broadcasted to an arbitrary size, so an array of shape `(N,1)` can be added to one of size `(N,M)`.
- Dimensions may be automatically added in the first position, so a `(N,)` array can be considered `(1,N)`, `(1,1,N)`, etc.
- If you want to add new dimensions in any other position, you need to add them manually:




```
>>> a = np.random.random((10, 5))
>>> b = np.random.random(10)
>>> c = a + b[:, np.newaxis]
```

NumPy: Performance

- Slicing, broadcasting, reshapes, transposes, etc allows us to perform complex calculations without copying data or using Python for loops.
- Optimal memory access patterns are achieved when all arrays have a stride of `1*sizeof(dtype)` in the rightmost dimension (or `0`, if we're broadcasting).

```
>>> a = np.random.random((10000, 10000))
>>> b = np.random.random((10000, 10000))
>>> from timeit import timeit
>>> timeit(lambda: a + b, number=10) / 10
0.29923874668311328
>>> timeit(lambda: a + b.T, number=10) / 10
0.487061994895339
```

- If you're using a reshaped/transposed array in many calculations, it might be more efficient to make a copy with 'corrected' strides (use `np.ascontiguousarray`).

NumPy: Avoiding new allocations

Most NumPy mathematical operations allow you to optionally specify an `out` parameter. This lets you write to an existing array rather than building a new one for each calculation.

```
>>> a = np.random.random(20000000)
>>> b = np.random.random(20000000)
>>> result = np.empty(20000000)
>>> timeit(lambda: a + b, number=10) / 10
0.0539267607033252
>>> timeit(lambda: np.add(a, b, out=result), number=10) / 10
0.0362531559076160
```

NumPy: Case Study

- Let's use NumPy for a slightly harder problem: a 2D scalar Laplacian operator:

$$\nabla^2 f_{i,j} = \frac{\partial^2 f_{i,j}}{\partial x^2} + \frac{\partial^2 f_{i,j}}{\partial y^2} \approx \frac{f_{i-1,j} - 2f_{i,j} + f_{i+1,j}}{\Delta x^2} + \frac{f_{i,j-1} - 2f_{i,j} + f_{i,j+1}}{\Delta y^2} \tag{2}$$

- The naive Python implementation isn't the prettiest:

```
>>> def laplacian_python(f, dx, dy):
        out = []
        for i in range(1, len(f)-1):
            out.append([])
            for j in range(1, len(f[0])-1):
                out[i-1].append(
                    (f[i-1][j] - 2 * f[i][j] + f[i+1][j]) / dx**2
                    + (f[i][j-1] - 2 * f[i][j] + f[i][j+1]) / dy**2
                )
        return out
```

NumPy: Case Study

- The optimised version is worse:

```
def stencil_i(it):
    im, i, ip = tee(it, 3)
    next(i), next(ip), next(ip)
    return zip(im, i, ip)

def stencil_ij(stencil_i):
    im, i, ip = stencil_i
    imj = iter(im)
    ijm, ij, ijp = tee(iter(i), 3)
    ipj = iter(ip)
    next(imj), next(ij), next(ipj)
    next(ijp), next(ijp)
    return zip(imj, ijm, ij, ijp, ipj)
```

```
def laplacian_python_optimised(f, dx, dy):
    invdx2 = 1.0 / dx**2
    invdy2 = 1.0 / dy**2
    inv2dx2dy2 = 2 * (invdx2 + invdy2)
    return [
        [
            (imj + ipj) * invdx2
            + (ijm + ijp) * invdy2
            - ij * inv2dx2dy2
            for imj, ijm, ij, ijp, ipj in (
                stencil_ij(stencil)
            )
        ] for stencil in stencil_i(f)
    ]
```

NumPy: Case Study

- The NumPy version is cleaner and much faster:

```
>>> def laplacian_numpy(f, dx, dy):
        return (
            (f[:-2,1:-1] + f[2:,1:-1]) * (1.0 / dx**2)
            + (f[1:-1,:-2] + f[1:-1,2:]) * (1.0 / dy**2)
            - f[1:-1,1:-1] * (2.0 / dx**2 + 2.0 / dy**2)
        )
>>> timeit(lambda: laplacian_python(f, dx, dy), number=10) / 10
59.52381650721654
>>> timeit(lambda: laplacian_python_optimised(f, dx, dy), number=10) / 10
28.17065314420033
>>> timeit(lambda: laplacian_numpy(f, dx, dy), number=10) / 10
1.975178470998071
```

- About 30 times faster than the naive version, and still 15 times faster after invasive optimisation! But we can do better...

Numba: Just-in-Time Compilation

- Numba is a library that allows you to compile a large subset of Python/NumPy at runtime and run it in parallel.
- You can usually achieve this just by 'decorating' your existing functions:

```
@njit(parallel=True)  # Only have to add this line, otherwise regular NumPy!
def laplacian_numba_parallel(f, dx, dy):
    return (
        (f[:-2,1:-1] + f[2:,1:-1]) * (1.0 / dx**2)
        + (f[1:-1,:-2] + f[1:-1,2:]) * (1.0 / dy**2)
        - f[1:-1,1:-1] * (2.0 / dx**2 + 2.0 / dy**2)
    )
```

- The first time you run a Numba-compiled function will be slower than all subsequent calls:

```
>>> timeit(lambda: laplacian_numba_parallel(f, dx, dy), number=1)
0.9609746721107513
>>> timeit(lambda: laplacian_numba_parallel(f, dx, dy), number=1)
0.5885984578635544
```

Cython

- Cython is a superset of Python which can be compiled (via C). It looks similar to pure Python, but you have to specify types.

```
# file: laplacian.pyx
import numpy as np
cimport cython
from cython.parallel import prange

@cython.boundscheck(False)
@cython.wraparound(False)
cpdef laplacian_cython(double[:, ::1] f, double dx, double dy):
    cdef Py_ssize_t i_max = f.shape[0]
    cdef Py_ssize_t j_max = f.shape[1]
    result = np.empty((i_max-2, j_max-2), dtype=np.double) # Regular NumPy array
    cdef double[:, ::1] view = result # 2D memory view with stride 1 in last dim
    cdef Py_ssize_t i, j
    for i in prange(1, i_max-1, nogil=True): # uses OpenMp parallel for
        for j in range(1, j_max-1):
            view[i-1, j-1] = (
                (f[i-1, j] - 2 * f[i, j] + f[i+1, j]) / dx**2
                + (f[i, j-1] - 2 * f[i, j] + f[i, j+1]) / dy**2
            )
    return result
```

How do they all stack up?

Version	Time (s)
Pure Python	59.523816
Pure Python (opt)	28.170653
NumPy	1.9751784
Numba (serial)	0.57726825
Numba (parallel)	0.21948093
Cython	0.22601821

- Cython and parallel Numba are roughly comparable: 300 times faster than where we started!
- This is on a 4 core machine. You may see even starker differences for bigger problem sizes and more cores.

Going even further

- CuPy works just like NumPy, but runs on the GPU via CUDA (Nvidia only). Also offers a CUDA JIT compiler.
- Dask can be used to scale up NumPy to clusters/HPC, and work on data sets larger than you can fit into memory.
- mpi4py provides bindings for the Message Passing Interface (MPI)
- Specialised optimised libraries for certain domains:
  - Deep learning: TensorFlow, PyTorch
  - Finite Element Methods: FEniCS, Firedrake