Optimising Python code

HWSA 2021

Simon Mutch

ASTRO 3D Postdoc

Senior Research Data Specialist: Melbourne Data Analytics Platform

The University of Melbourne

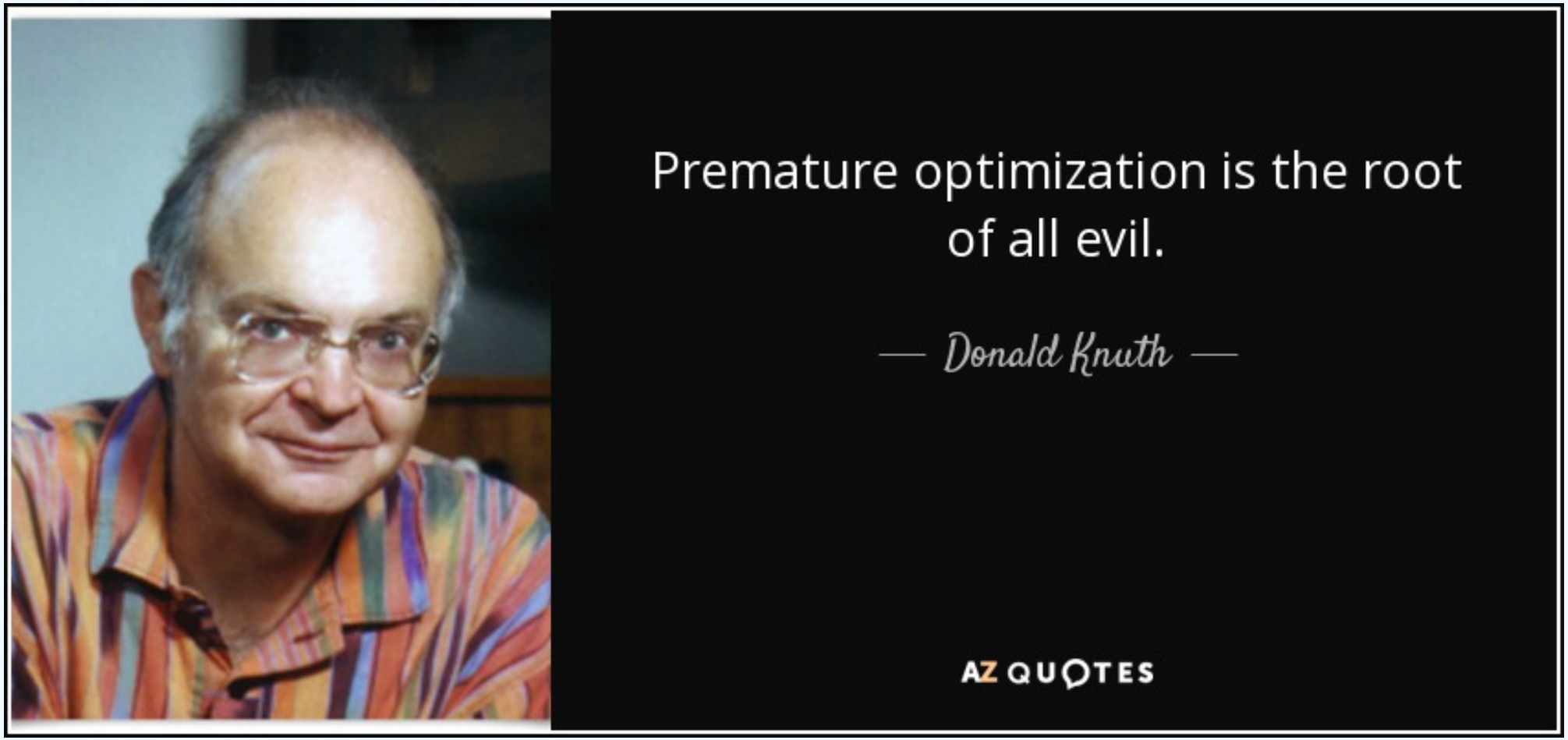
https://github.com/smutch/code_prac_hwsa2021

Session outline

- Presentation on optimisation
  - Profiling
  - Testing
  - Speeding up Python code
- Practical
  - Open ended session with a short code to try some of these techniques out on.
  - 🏆 **Competition** 🏆: Who can achieve the greatest speed-up? ⏱️

There is more content in these slides than we'll cover today, but hoepfully they will be a useful reference for the future.

https://github.com/smutch/code_prac_hwsa2021



https://github.com/smutch/code_prac_hwsa2021

The optimisation cycle

1. Profile
2. Develop regression tests
3. Baseline timing

4. Optimise identified bottlenecks
5. Time
6. Test
7. Profile then go back to 4
   or call it a day and bask in glory

https://github.com/smutch/code_prac_hwsa2021

cProfile

`cProfile` is part of the Python standard library *(you don't need to manually install it)*. It gives statistics for function calls and is often the best place to start when profiling Python code.

```
import cProfile
import pstats

profile = cProfile.Profile()

profile.enable()

dn_dlogm_list = []
for z in z_list:
    dn_dlogm_list.append(press_schecter(M_list, z, A_std_func()))

profile.disable()

ps = pstats.Stats(profile).strip_dirs().sort_stats(pstats.SortKey.TIME)
ps.print_stats(10)
```

https://github.com/smutch/code_prac_hwsa2021

cProfile

`cProfile` is part of the Python standard library *(you don't need to manually install it)*. It gives statistics for function calls and is often the best place to start when profiling Python code.

```
import cProfile
import pstats

profile = cProfile.Profile()

profile.enable()

dn_dlogm_list = []
for z in z_list:
    dn_dlogm_list.append(press_schecter(M_list, z, A_std_func()))

profile.disable()

ps = pstats.Stats(profile).strip_dirs().sort_stats(pstats.SortKey.TIME)
ps.print_stats(10)
```

https://github.com/smutch/code_prac_hwsa2021

cProfile

`cProfile` is part of the Python standard library *(you don't need to manually install it)*.
It gives statistics for function calls and is often the best place to start when profiling Python code.

```
import cProfile
import pstats

profile = cProfile.Profile()

profile.enable()

dn_dlogm_list = []
for z in z_list:
    dn_dlogm_list.append(press_schecter(M_list, z, A_std_func()))

profile.disable()

ps = pstats.Stats(profile).strip_dirs().sort_stats(pstats.SortKey.TIME)
ps.print_stats(10)
```

https://github.com/smutch/code_prac_hwsa2021

cProfile

```
         3326986 function calls in 8.092 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1094475    4.079    0.000    4.079    0.000 press_schecter.py:22(tophat_ft)
  1094475    2.482    0.000    2.482    0.000 press_schecter.py:27(power)
  1094100    0.894    0.000    7.454    0.000 press_schecter.py:38(<lambda>)
     4005    0.536    0.000    7.992    0.002 {built-in method scipy.integrate._quadpack._qagie}
     1000    0.018    0.000    6.116    0.006 common.py:75(derivative)
     1000    0.012    0.000    0.012    0.000 {method 'reduce' of 'numpy.ufunc' objects}
        5    0.012    0.002    8.089    1.618 press_schecter.py:69(press_schecter)
```

- **tottime**: time spent inside this function
- **cumtime**: time spent inside this function and all functions it calls
- **percall**: cumulative time per call

https://github.com/smutch/code_prac_hwsa2021

line_profiler

What if we want to go deeper though?

Where in the function are we spending all of our time? Calculating logarithms, multiplying numbers?

```
@profile
def press_schechter(...):
    ...
```

```
> kernprof -l press_schecter.py
> python -m line_profiler press_schechter.py.lprof
```

https://github.com/smutch/code_prac_hwsa2021

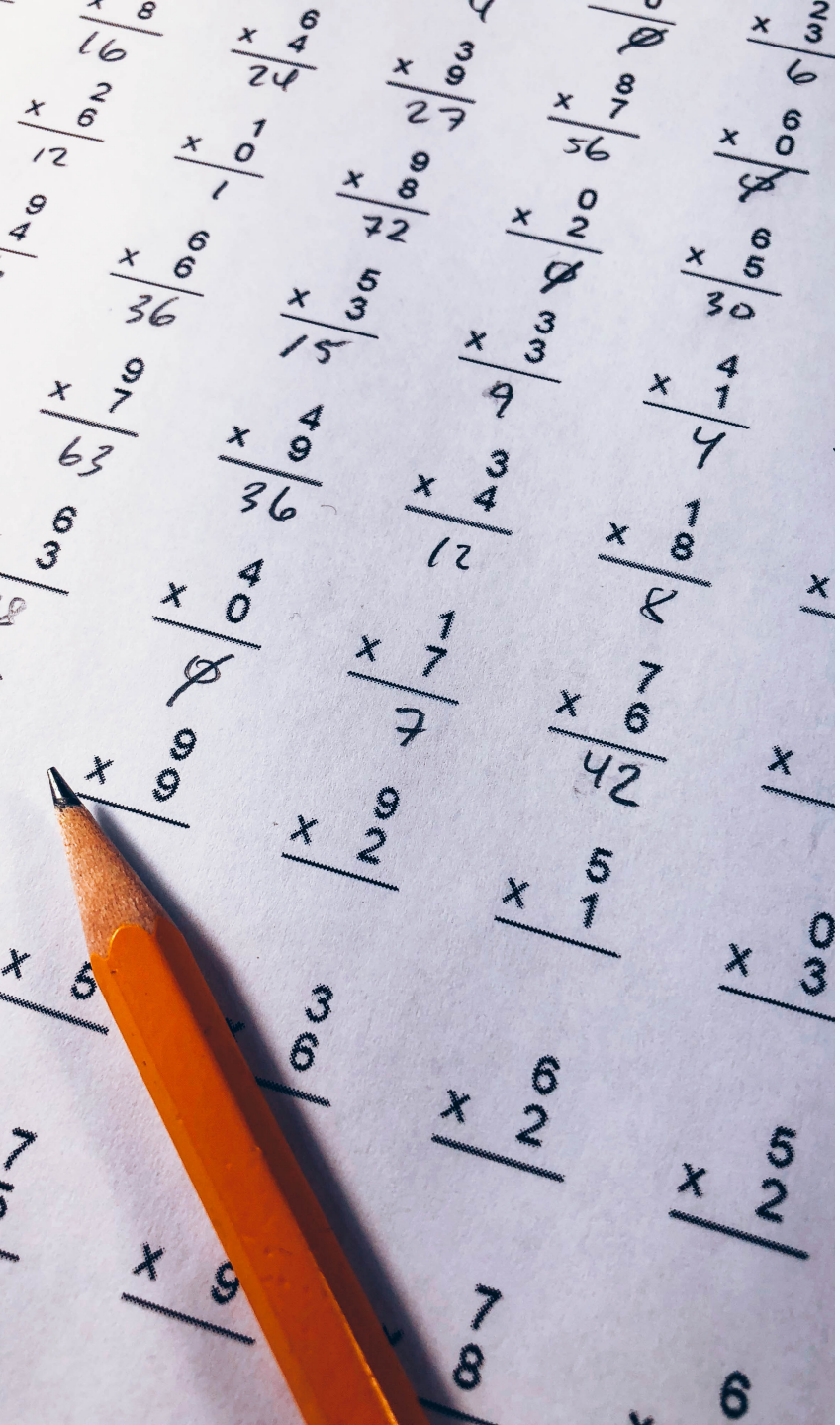
line_profiler

```
Function: press_schecter at line 66

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    66                                           @profile
    67                                           def press_schecter(M_list, z, A_std):
    68         5        115.0     23.0      0.0      R_list = (2 * (M_list) * G / (h0 ** 2 * wm)) ** (1 / 3)
    69         5         18.0      3.6      0.0      rho = rho_m(z=0)
    70
    71         5        764.0    152.8      0.0      d_c = d_crit(z)
    72
    73         5          4.0      0.8      0.0      dn_dlogm = []
    74      1005       1436.0      1.4      0.0      for r in R_list:
    75      1000    3266597.0   3266.6     24.3          std = stdev(r, A=A_std)
    76      1000       2010.0      2.0      0.0          deriv = (
    77      1000   10167353.0  10167.4     75.6              2 * G / (3 * h0 ** 2 * wm * r ** 2) * \
                                                            derivative(lambda rr: np.log(stdev(rr, A=A_std)), x0=r, dx=r * 0.5)
    78                                                    )
    79      1000       1078.0      1.1      0.0          nu = d_c / std
    80      1000       9737.0      9.7      0.1          dn_dlogm.append( \
                                                            np.sqrt(2 / np.pi) * rho * -deriv * nu * np.exp(-(nu ** 2) / 2) \
                                                        )
    81         5          3.0      0.6      0.0      return dn_dlogm
```

- **Hits**: Number of times this line is called.
- **Total**: Cumulative time spent on this line.
- **Per Hit**: Time spent on line each time it's called.
- **% Time**: Percentage of time spent on this line relative to whole function.

https://github.com/smutch/code_prac_hwsa2021



Testing

- **Unit tests**: Ideally test the smallest logical units of your code (e.g. individual functions).
  - Should (ideally) test correctness, edge cases and unexpected input.
- **Integration tests**: Tests larger units and how they interact, or the whole code.
  - Should (ideally) test correctness and be used in conjunction with unit tests.
- **Regression tests**: Tests larger units or the whole code to make sure you recover the same answer after changes.
  - Sometimes you don't want these (e.g. when changing physical models), but they are **very important** for optimisation.

Make testing part of your routine.
It's a lot easier to write tests as you go along, rather than coming back after-the-fact.

https://github.com/smutch/code_prac_hwsa2021

Pytest

Covering pytest and all it has to offer would take a whole session itself!
There are a lot of great examples and articles online.

Some useful topics to check out include:

- Fixtures
- Parameterizing tests
- Setting up continuous integration (CI) for automated testing
- Coverage measurements

For regression testing, the regtest plugin is popular[1]...

[1] See also https://github.com/astropy/pytest-arraydiff
https://github.com/smutch/code_prac_hwsa2021

Regression tests with pytest

Naming of directories, files and functions all matter with pytest. You can configure this, but standard practice is:

- Place all of your tests in a directory called `tests`.
- You need to have a `__init__.py` file in `tests` (to make them part of a module) but it can be empty.
- Each file should be called `test_*.py`.
- Each test function should be called `test_*`.

```
tests
├── __init__.py
└── test_regressions.py
```

```
def test_regressions(...): ...
```

https://github.com/smutch/code_prac_hwsa2021

Regression tests with pytest

```
from pytest_regtest import regtest

from mycode import myfunc

def test_mycode(regtest):

    result = myfunc(...)

    with regtest:
        print(result)
```

We need to initialise and store the expected output the first time around.

```
> pytest --regtest-reset
```

The results will go in `tests/_regtest_outputs/` **which should be committed to version control**.

https://github.com/smutch/code_prac_hwsa2021

Regression tests with pytest

Now, after trying to optimise the code, we can check we still get the right result using:

```
> pytest

============================================================================= test session starts ==============================================================================
platform darwin -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /blaa/blaa/mycode
plugins: regtest-1.4.6
collected 1 item

tests/test_regressions.py .                                                                                                                                      [100%]

============================================================================== 1 passed in 9.22s ===============================================================================
```

https://github.com/smutch/code_prac_hwsa2021

Regression tests

If the output of the `press_schechter` function changes:

```
> pytest

============================================================================= test session starts ==============================================================================
platform darwin -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /blaa/blaa/mycode
plugins: regtest-1.4.6
collected 1 item

tests/test_regressions.py F                                                                                                                                      [100%]

================================================================================== FAILURES ===================================================================================
_____________________________________________________________________________ test_press_schechter ______________________________________________________________________________

regression test output differences for tests/test_regressions.py::test_mycode:

>   --- current
>   +++ tobe
>   @@ -1,2 +1,2 @@
>   -[[3943684.4663377525, 3433036.716787263, 2988666.7237358466, 2601939.030501016, 2265406.0334823187, 1972476.6100945876, 1717547.9139849006, 1495624.5560116607, 1302452.0154

...

=========================================================================== short test summary info ============================================================================
FAILED tests/test_regressions.py::test_mycode
============================================================================== 1 failed in 10.03s ==============================================================================
```

https://github.com/smutch/code_prac_hwsa2021

Let's go fast!

https://github.com/smutch/code_prac_hwsa2021

Things to know about Python and speed

Python is an interpreted language and (almost) everything is an object

This means there is a cost with looking up and accessing values.

```
a = 8.0
print(a)
```

Translated to C this is actually something more like *(but in reality way more complicated!)*:

```
void *a = (void*)malloc(sizeof(double));
*(double*)a = 8.0;
printf("{:g}\n", *((double*)a));
```

https://github.com/smutch/code_prac_hwsa2021

The most important scientific python optimisation rule

For loops should be avoided if possible. Take advantage of Numpy's ufuncs (which will vectorize it and do it more efficiently).

```
arr = np.arange(100_000)

# BAD: 165 ms ± 4.78 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
for ii, val in enumerate(arr):
    arr[ii] = np.sqrt(val)

# GOOD: 147 µs ± 2.51 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
arr = np.sqrt(arr)
```

147 µs = 0.147 ms

∴ x1,112 speed up !!! 😮

https://github.com/smutch/code_prac_hwsa2021

An aside...

A common misconception:

Numpy `vectorize` does not "vectorize" your code.

From the docs:

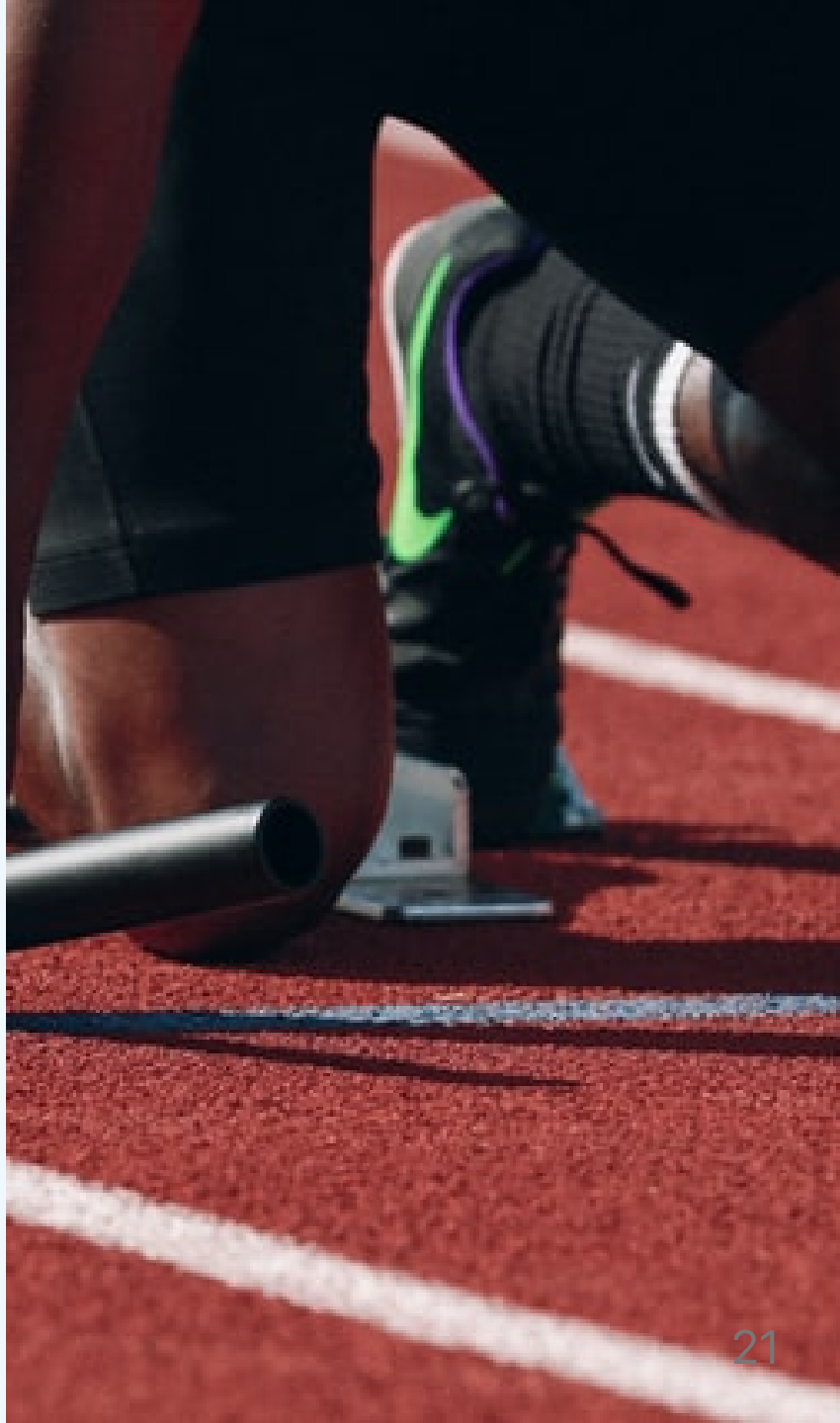
> "The vectorize function is provided primarily for convenience, not for performance. The implementation is essentially a for loop."
>
> -- *Numpy docs*

I've removed all the loops I can, but I still need more speed!!!

There are a number of techniques and tools at our disposal. Which ones will work best is heavily dependent on the problem. e.g.:

- Use 3rd-party libraries which are already optimised!
  (I won't cover this, but it should always be your first stop.)
- Algorithmic changes (e.g. identifying redundant calculations).

  ⬇️ **This is what we'll look at today.** ⬇️
- Memoisation (caching of results for reuse later)
- Dropping down to a lower level using Numba, Cython, etc.
- Parallelisation



https://github.com/smutch/code_prac_hwsa2021

Memoisation

The idea

Sometimes we have a function that may be called many times with the same input. In these situations it may be faster to store the results and reuse them rather than recalculating them each time.

Works well when

- You have an expensive function being called a number of times with the same input.
- You have any non-trivial function being called many, many times with the same *small* input.

https://github.com/smutch/code_prac_hwsa2021

Memoisation

A (very contrived) example[1]

```
import numpy as np

points = np.random.rand(10_000, 2)

def myfunc(x, y):
    return (np.log10(np.sqrt(x**2 + y**2) * 5) / 0.2)**6

result = 0.0
for _ in range(100):
    result += np.array([myfunc(x, y) for x, y in points])
```

```
5.02 s ± 242 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[1] Don't do this!!!

https://github.com/smutch/code_prac_hwsa2021

Memoisation

A (very contrived) example[1]

```
from functools import cache

import numpy as np

points = np.random.rand(10_000, 2)

@cache
def myfunc(x, y):
    return (np.log10(np.sqrt(x**2 + y**2) * 5) / 0.2)**6

result = 0.0
for _ in range(100):
    result += np.array([myfunc(x, y) for x, y in points])
```

```
1.28 s ± 423 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

A factor of ~4 speed-up by adding 2 lines... Not bad!

[1] Don't do this!!!

https://github.com/smutch/code_prac_hwsa2021

Memoisation

However...

`functools.cache` stores the result in RAM and needs to compare each input argument against the cached results.
This works well when arguments and output are small e.g `float`s `int`s etc.

joblib.Memory

These limitations can be overcome in some cases with joblib.Memory.
It caches to disk and removes much of the overhead with large input and output arrays.

Going lower level...

Sometimes the best way to improve the speed of our code is to use another language!... 😱

But fear not! This does not mean having to re-write your code in C/C++/Rust etc.

Numba Does **J**ust **I**n **T**ime (JIT) compilation of Python code. It can be incredibly easy to use and can provide big speed gains. It can also be used for GPU programming.

Cython A python-superset language that can be used to create efficient C extensions. As well as providing similar speed-ups to Numba (with a bit more effort), it can be used in more complex situations and also provides a great and flexible way to interface C libraries with Python.

https://github.com/smutch/code_prac_hwsa2021

Numba

Using Numba can be incredibly easy:

```
import numba
import numpy as np

@numba.njit
def myfunc(arr, n_loops):
    result = np.zeros(arr.shape[0])
    for _ in range(n_loops):
        result += (np.log10(np.sqrt(arr[:, 0]**2 + arr[:, 1]**2) * 5) / 0.2)**6
    return result

points = np.random.rand(10_000, 2)
result = myfunc(points, 100)
```

- `njit` means compile in "no-python" mode. This requires the function to be simple and use only support functions and types (most of Numpy) but not Python classes or dicts.
- Loops are fine! They will be optimised by LLVM (the compiler) for your CPU.
- `15.7 ms` with decorator vs. `42.5 ms` without **= ~2.5x speed-up**

https://github.com/smutch/code_prac_hwsa2021

Where Numba might not be enough (WARNING: very subjective!)

- It only supports a subset of Python + Numpy (although a very large one).
- It can be hard to debug if things don't work as expected.
- There isn't much scope for workarounds. If it doesn't "just work" then often it won't work.

Despite these, I definitely recommend Numba as the first-stop for speeding up numerical code!

https://github.com/smutch/code_prac_hwsa2021

Parallelisation

Almost all computers have multiple cores. In some cases we can take advantage of this to speed up our calculations. Numpy & Scipy actually do this for some functions without you necessarily realising.

This technique works well when:

- you have an expensive function
- being called multiple times with different arguments
- independently of each other.

https://github.com/smutch/code_prac_hwsa2021

Extra

Native parallelisation in Python is expensive

The GIL

The Global Interpreter Lock (GIL) is a mutex which ensures only one thread can use the interpreter at any one time. It's a feature that makes Python simple and easy to use.
But it's a hurdle to efficient parallelisation.

To get around this natively we can use the multiprocessing standard library.
It uses multiple processes, each running their own Python interpreter, with their own private memory space.
But this is expensive, so we need to have enough work for each process to make it worthwhile.

Parallelisation with Numba

As it turns out, Numba (does) and Cython (can) release the GIL! 🥳

Numba can parallelise code automatically in some cases (see the docs), but we can also manually request it:

```
import numba
import numpy as np

@numba.njit(parallel=True)
def myfunc(arr, n_loops):
    result = np.zeros(arr.shape[0])
    for _ in numba.prange(n_loops):
        result += (np.log10(np.sqrt(arr[:, 0]**2 + arr[:, 1]**2) * 5) / 0.2)**6
    return result

points = np.random.rand(10_000, 2)
result = myfunc(points, 10_000)
```

Even with Numba, the best performance gains only come when the units of work are large enough.

For more information and a good, flexible library for multiple different parallelisation backends, see joblib.

https://github.com/smutch/code_prac_hwsa2021

Practical time!

https://github.com/smutch/code_prac_hwsa2021

Practical

Main suggestion

- Optimize `code_prac/press_schechter.py` and see how fast you can make it[1].
  - Time your attempts using
    `python -m timeit -s 'import code_prac' 'code_prac.press_schechter.main()'`
- If you followed the setup instructions in the repo then you should have all of the libraries and packages you may need installed.
- **Compete for the title of 🏆🏆 "Optimizer Prime" 🏆🏆!**
  - There are no rules, except that we must be able to call `code_prac.press_schechter.mass_function` with the same arguments as the starting code and get the same result.

[1] I got a 300x speed-up. Can you beat me? 😉
https://github.com/smutch/code_prac_hwsa2021

Practical

and/or

Take advantage of the knowledge of your peers and practice your skills with the `press_schechter` code.
Some ideas include:

- Add inline documentation and build docs using Sphinx
  - Tip: use the Napolean extension and Numpy or Googledoc style docstrings.
  - Bonus: Fork of the `code_prac_hwsa2021` repo on Github and serve the docs online using Github Pages.
- Develop unit tests for the code
  - Bonus: Use coverage.py tool to measure your unit test coverage and aim for >90% coverage.
  - Bonus bonus: Fork the `code_prac_hwsa2021` repo and use Github Actions to automatically test the code on every push.
- Add type annotations to the code then use mypy to check these.
- If you manage to speed up the `press_schechter` function enough (or ask me for a faster version), try making an interactive tool for exploring the PS mass function using Jupyter Notebooks and ipywidgets (or a tool of your choice).

https://github.com/smutch/code_prac_hwsa2021

Remember

1. Profile
2. Develop regression tests
3. Baseline timing

4. Optimise identified bottlenecks
5. Time
6. Test
7. Profile then go back to 4
   or call it a day

https://github.com/smutch/code_prac_hwsa2021