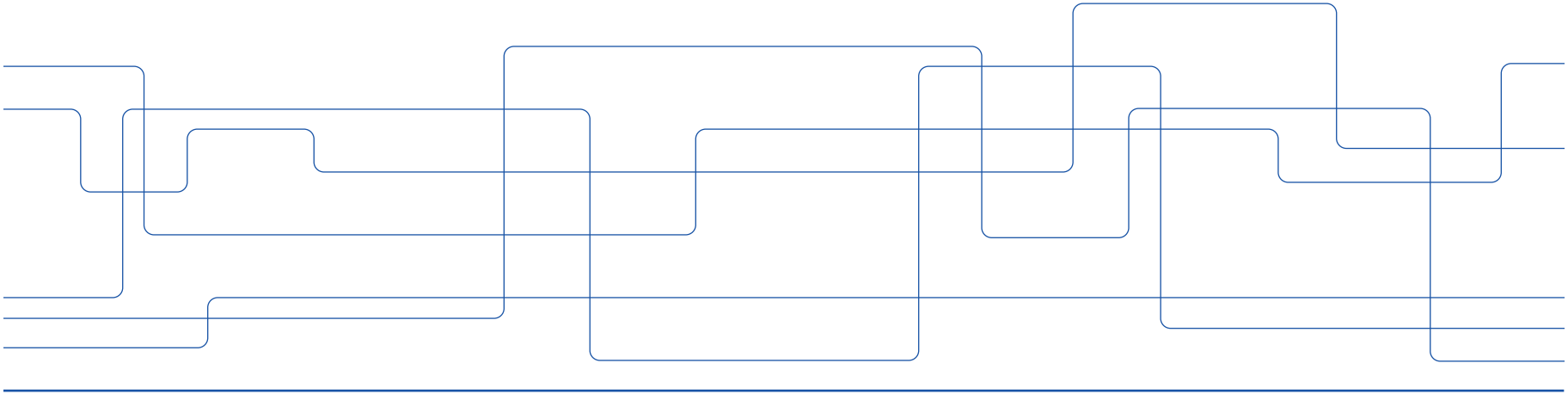
DD2358 – Using `memory_profiler` to Diagnose Memory Usage

Stefano Markidis

KTH Royal Institute of Technology



`memory_profiler` module

- The memory_profiler module by Fabian Pedregosa and Philippe Gervais measures memory usage on a line-by-line basis. Understanding the memory usage characteristics of your code allows you to ask yourself two questions:
  - Could we use *less* RAM by rewriting this function to work more efficiently?
  - Could we use *more* RAM and save CPU cycles by caching?

`memory_profiler` Installation

- memory_profiler operates in a very similar way to line_profiler but runs far more slowly.
  - If we install the `psutil` package (optional but recommended), memory_profiler will run faster.
    - > *Psutil provides hardware information*
- Memory profiling may easily make your code run **10 to 100 times slower**!
  - In practice, we will use `memory_profiler` occasionally and `line_profiler` (for CPU profiling) more frequently.
- Install memory_profiler with the command `pip install memory_profiler` (and optionally with `pip install psutil`).

Beware of the Profiling Time

- It may therefore make sense to run your tests on a smaller problem that completes in a useful amount of time.
- Overnight runs might be sensible for validation, but you need quick and reasonable iterations to diagnose problems and hypothesize solutions. The code used the full 1,000 × 1,000 grid, and the statistics took about one hour on my laptop!
- **Reduce the problem size!**
  - Instead of 1,000 × 1,000 grid → 100 × 100 grid

Profiling Memory Usage

- When dealing with memory allocation, you must be aware that the situation is not as clear-cut as it is with CPU usage.
- Generally, it is more efficient to overallocate memory in a process that can be used at leisure, as memory allocation operations are relatively expensive.
- Furthermore, garbage collection is not instantaneous, so objects may be unavailable but still in the garbage collection pool for some time.

Decorator for the function we Monitor

```
@profile
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while abs(z) < 2 and n < maxiter:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

python -m memory_profiler JuliaSet.py

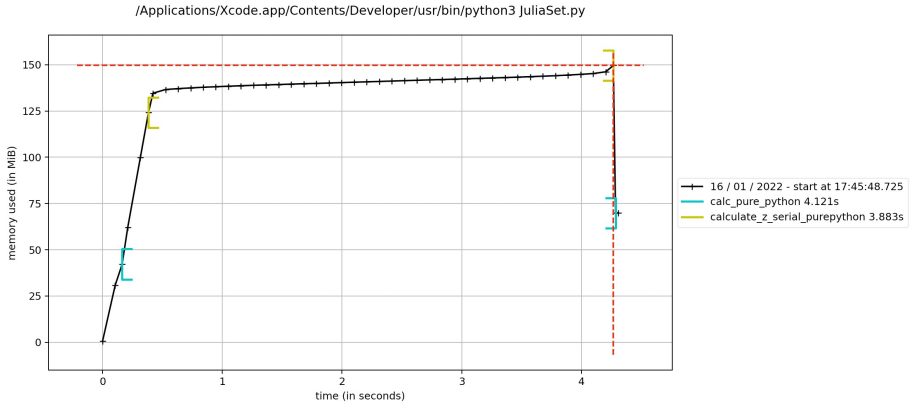
```
stef@Stefs-MacBook-Air Codes % python -m memory_profiler JuliaSet.py
Length of x: 100
Total elements: 10000
calculate_z_serial_purepython took 13.444536209106445 seconds
Filename: JuliaSet.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    60   42.391 MiB   42.391 MiB           1   @profile
    61                                         def calculate_z_serial_purepython(maxiter, zs, cs):
    62                                             """Calculate output list using Julia update rule"""
    63   42.391 MiB    0.000 MiB           1       output = [0] * len(zs)
    64   42.422 MiB    0.000 MiB       10001       for i in range(len(zs)):
    65   42.422 MiB    0.000 MiB       10000           n = 0
    66   42.422 MiB    0.000 MiB       10000           z = zs[i]
    67   42.422 MiB    0.000 MiB       10000           c = cs[i]
    68   42.422 MiB    0.000 MiB      344236           while abs(z) < 2 and n < maxiter:
    69   42.422 MiB    0.000 MiB      334236               z = z * z + c
    70   42.422 MiB    0.031 MiB      334236               n += 1
    71   42.422 MiB    0.000 MiB       10000           output[i] = n
    72   42.422 MiB    0.000 MiB           1       return output
```

Track memory usage

How to Use `mprof`

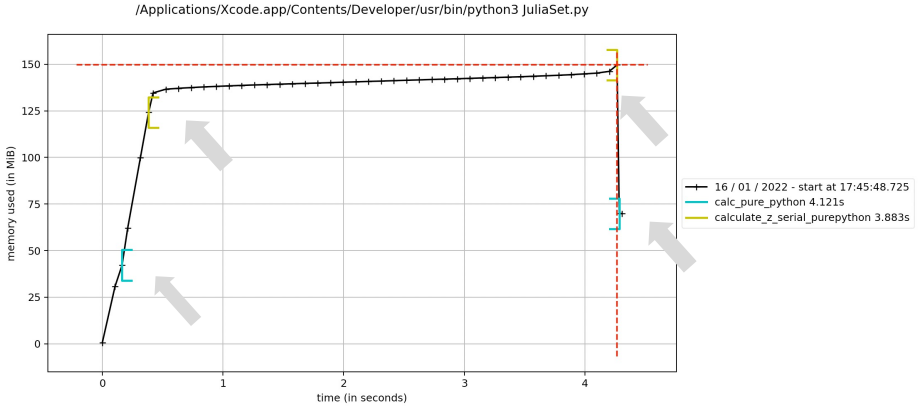
- memory_profiler has a utility called mprof, used once to sample the memory usage and a second time to visualize the samples.
  - **It samples by time and not by line**, so it barely impacts the runtime of the code.
- Collect memory statistics `python -m` **`mprof run`** `JuliaSet.py`
- Generate a memoprofile .dat file
- mprof to visualize: `python -m mprof plot mprofile_20220116173811.dat`

```
stef@Stefs-MacBook-Air Codes % python -m mprof run JuliaSet.py
mprof.py: Sampling memory every 0.1s
running new process
running as a Python program...
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 3.88464093208313 seconds
stef@Stefs-MacBook-Air Codes % ls
JuliaSet.py                    JuliaSet.py.lprof              line_profiler
stef@Stefs-MacBook-Air Codes % python -m mprof plot mprofile_20220116173811.dat
```



Analyzing `mprof` Plot

- Our two functions are bracketed
  - This shows where in time they are entered, and we can see the growth in RAM as they run.
- Inside `calculate_z_serial_purepython`, we can see the steady increase in RAM usage throughout the execution of the function
  - this is caused by all the small objects (int and float types) that are created.



Add Labels Using a Context Manager

- We can see the create_output_list label: it appears momentarily at around 1.5 seconds after `calculate_z_serial_purepython` and results in the process being allocated more RAM.
- We then pause for a second; `time.sleep(1)` is an artificial addition to make the graph easier to understand.

```
@profile
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    with profile.timestamp("create_output_list"):
        output = [0] * len(zs)
    time.sleep(1)
    with profile.timestamp("calculate_output"):
        for i in range(len(zs)):
            n = 0
            z = zs[i]
            c = cs[i]
            while abs(z) < 2 and n < maxiter:
                z = z * z + c
                n += 1
            output[i] = n
    return output
```