DD2358 – Using `line_profiler` for Line-by-Line Measurements

Stefano Markidis

KTH Royal Institute of Technology

Intended Learning Outcomes

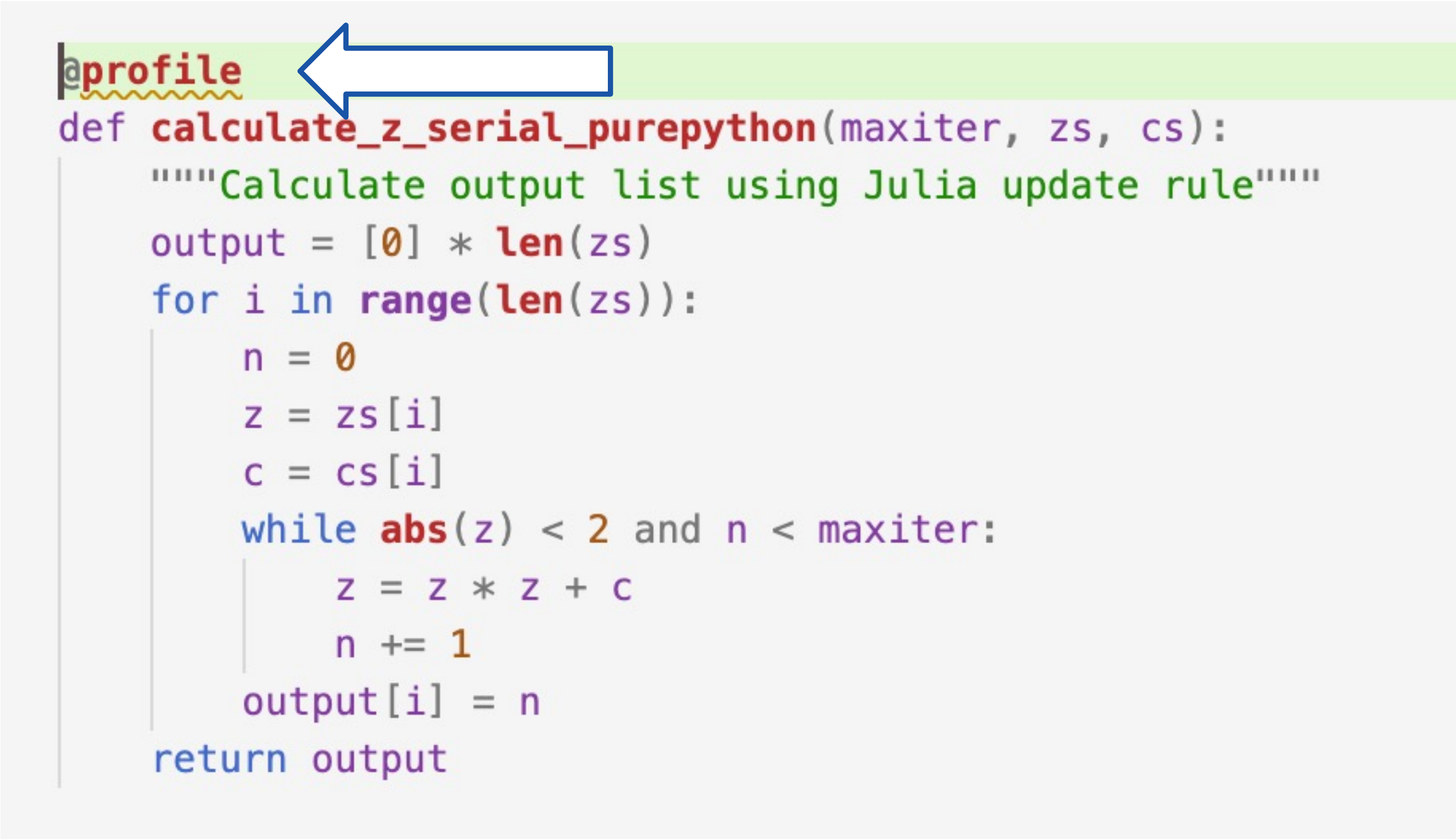
- To deploy the line_profiler module in your code to profile it line-by-line
- Analyze the line_profiler statistic output and identify fine-grained computational bottle-necks

line_profiler

- line_profiler works by profiling individual functions on a line-by-line basis
- When profiling, we should start with cProfile and use the high-level view to guide which functions to profile with line_profiler.
- To use line_profiler

1. You need to install line_profiler with pip install line_profiler
2. A decorator (@profile) is used to **mark the chosen function**.
3. The **kernprof script** is used to execute our code
   - the CPU time and other statistics for each line of the chosen function are recorded.

Add @profile Decorator before Function



```
@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
```

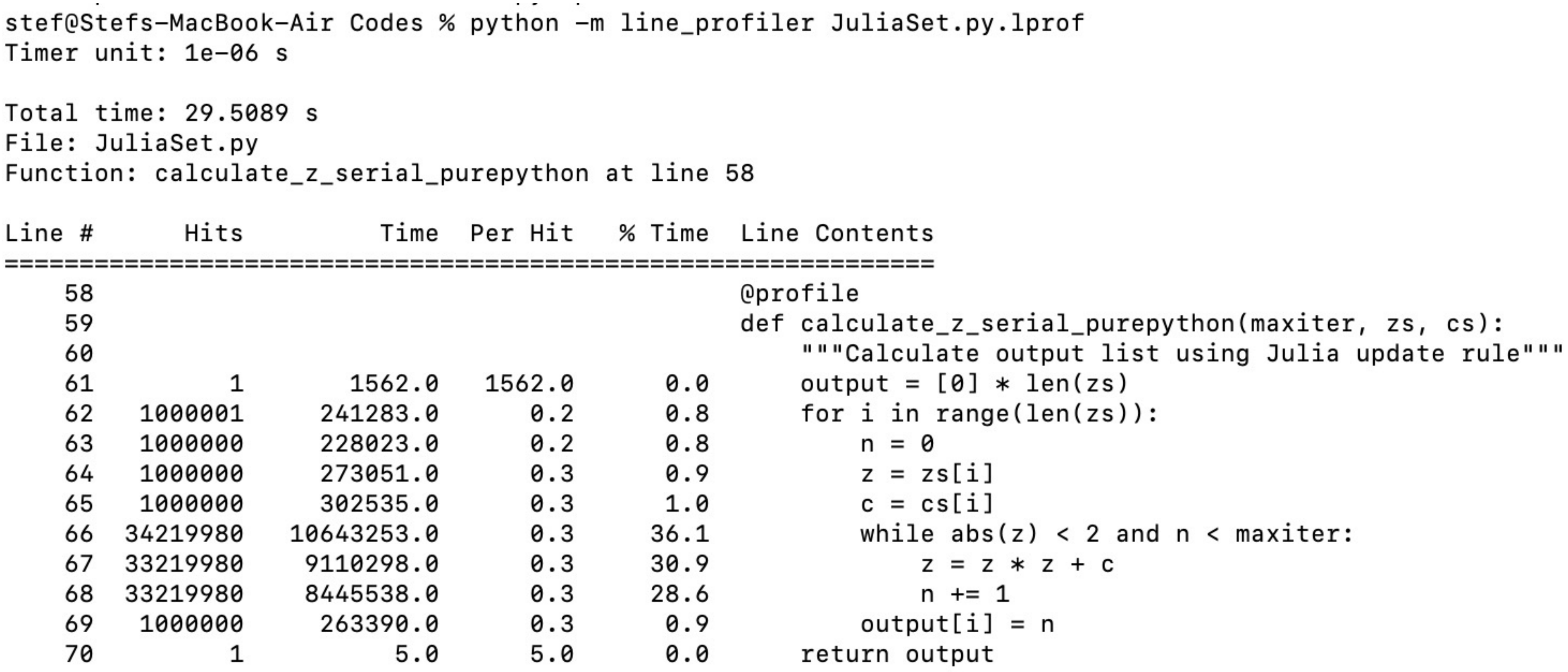
Run `kernprof` for Obtaining Profiling

```
python -m kernprof -l JuliaSet.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 54.3584349155426 seconds
Wrote profile results to JuliaSet.py.lprof
```

Note the timing!

Profiling information
Stored in .lprof

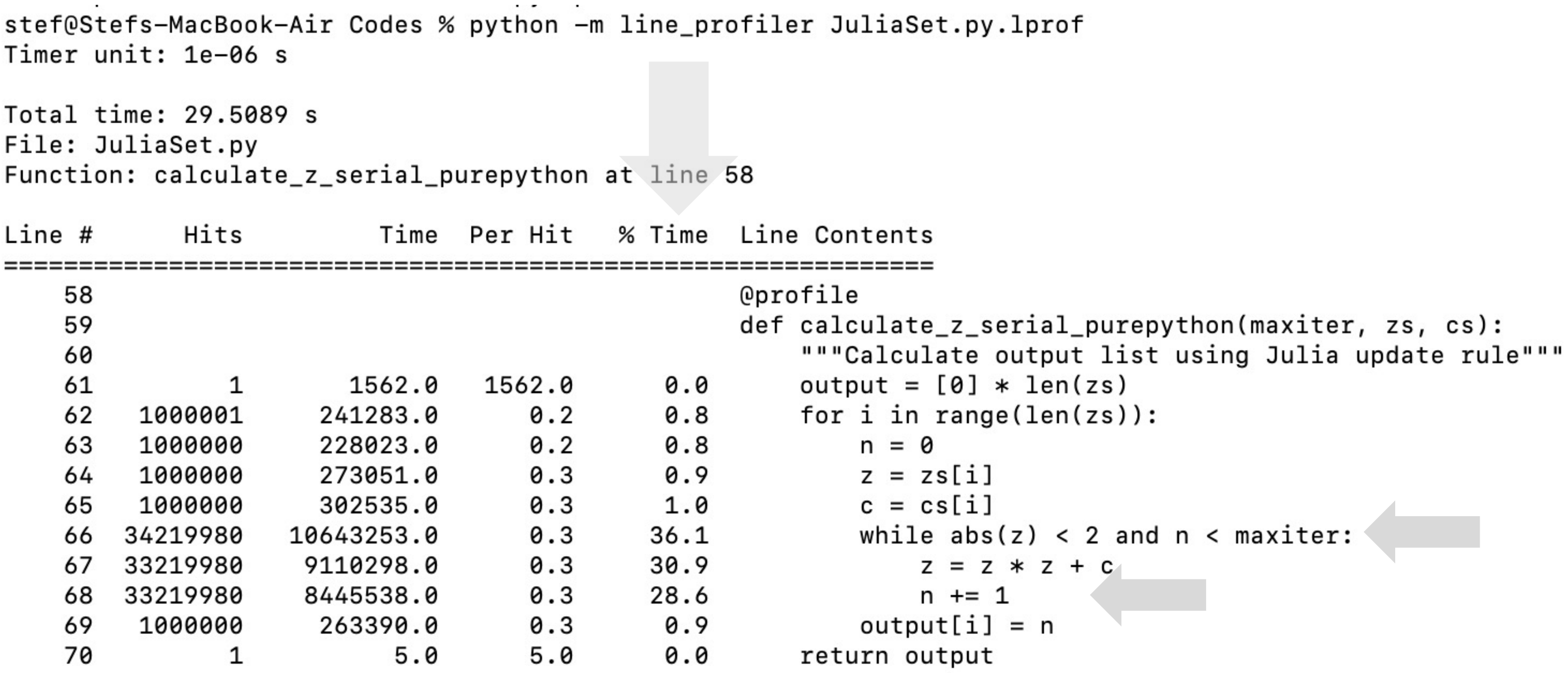
Print profile: `python -m line_profiler JuliaSet.py.lprof`



```
stef@Stefs-MacBook-Air Codes % python -m line_profiler JuliaSet.py.lprof
Timer unit: 1e-06 s

Total time: 29.5089 s
File: JuliaSet.py
Function: calculate_z_serial_purepython at line 58

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    58                                           @profile
    59                                           def calculate_z_serial_purepython(maxiter, zs, cs):
    60                                               """Calculate output list using Julia update rule"""
    61         1       1562.0   1562.0      0.0      output = [0] * len(zs)
    62   1000001     241283.0      0.2      0.8      for i in range(len(zs)):
    63   1000000     228023.0      0.2      0.8          n = 0
    64   1000000     273051.0      0.3      0.9          z = zs[i]
    65   1000000     302535.0      0.3      1.0          c = cs[i]
    66  34219980   10643253.0      0.3     36.1          while abs(z) < 2 and n < maxiter:
    67  33219980    9110298.0      0.3     30.9              z = z * z + c
    68  33219980    8445538.0      0.3     28.6              n += 1
    69   1000000     263390.0      0.3      0.9          output[i] = n
    70         1          5.0      5.0      0.0      return output
```



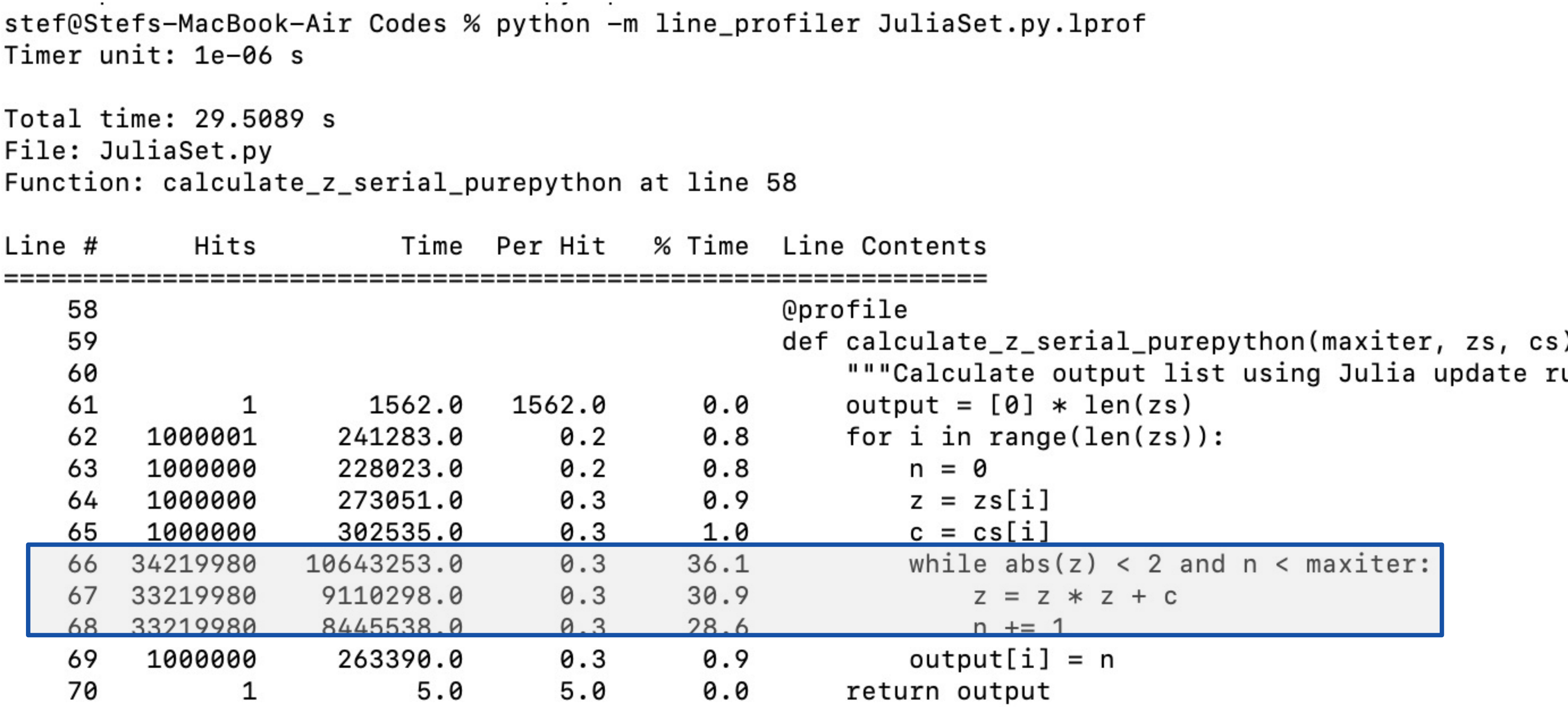
```
stef@Stefs-MacBook-Air Codes % python -m line_profiler JuliaSet.py.lprof
Timer unit: 1e-06 s

Total time: 29.5089 s
File: JuliaSet.py
Function: calculate_z_serial_purepython at line 58

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    58                                           @profile
    59                                           def calculate_z_serial_purepython(maxiter, zs, cs):
    60                                               """Calculate output list using Julia update rule"""
    61         1       1562.0   1562.0      0.0      output = [0] * len(zs)
    62   1000001     241283.0      0.2      0.8      for i in range(len(zs)):
    63   1000000     228023.0      0.2      0.8          n = 0
    64   1000000     273051.0      0.3      0.9          z = zs[i]
    65   1000000     302535.0      0.3      1.0          c = cs[i]
    66  34219980   10643253.0      0.3     36.1          while abs(z) < 2 and n < maxiter:
    67  33219980    9110298.0      0.3     30.9              z = z * z + c
    68  33219980    8445538.0      0.3     28.6              n += 1
    69   1000000     263390.0      0.3      0.9          output[i] = n
    70         1          5.0      5.0      0.0      return output
```

- The **% Time column** is the most helpful - we can see that 36% of the time is spent on the while testing.
  - We don’t know whether the first statement (abs(z) < 2) is more expensive than the second (n < maxiter), though.
  - Inside the loop, we see that the update to z is also fairly expensive: 30.
- **Even n += 1 is expensive**! Python’s dynamic lookup machinery is at work for every loop, even though we’re using the same types for each variable in each loop
- The creation of the outputlist and the updates on line 20 are relatively cheap compared to the cost of the while loop.

Performance Improvement Opportunities



```
stef@Stefs-MacBook-Air Codes % python -m line_profiler JuliaSet.py.lprof
Timer unit: 1e-06 s

Total time: 29.5089 s
File: JuliaSet.py
Function: calculate_z_serial_purepython at line 58

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    58                                           @profile
    59                                           def calculate_z_serial_purepython(maxiter, zs, cs)
    60                                               """Calculate output list using Julia update ru
    61         1       1562.0   1562.0      0.0      output = [0] * len(zs)
    62   1000001     241283.0      0.2      0.8      for i in range(len(zs)):
    63   1000000     228023.0      0.2      0.8          n = 0
    64   1000000     273051.0      0.3      0.9          z = zs[i]
    65   1000000     302535.0      0.3      1.0          c = cs[i]
    66  34219980   10643253.0      0.3     36.1          while abs(z) < 2 and n < maxiter:
    67  33219980    9110298.0      0.3     30.9              z = z * z + c
    68  33219980    8445538.0      0.3     28.6              n += 1
    69   1000000     263390.0      0.3      0.9          output[i] = n
    70         1          5.0      5.0      0.0      return output
```

To Summarize

- The `line_profiler` module allows us to profile individual functions on a line-by-line basis using the @profile decorator in your code.
- The analysis of the profile information (the .lsprof file) shows us code lines with potential to be optimized.