UNIVERSIDAD POLITÉCNICA DE YUCATÁN

BIS Universities

PORTFOLIO HIGH PERFORMANCE PYTHON

HIGH PERFORMANCE COMPUTING

Made by

Karla Delgado Avendaño

March 24th, 2024

Portfolio of Evidence: High Performance Python

Karla Delgado
Data Engineering
Universidad Politécnica de Yucatán
Ucú, Yucatán, México
2109050@upy.edu.mx

***Abstract*—This report presents the outcomes and findings of a comprehensive classroom activity focused on High Performance Python. The activity encompasses various critical aspects, including benchmarking and profiling, data structure optimization, dictionary and set operations, matrix and vector computations, and compiling to C. Each section of the report delves into different high-performance computing techniques, illustrating the potential improvements in Python code efficiency. The exercises aim to provide practical insights into optimizing Python for speed and efficiency, leveraging profiling tools, understanding data structures, and utilizing NumPy for enhanced array operations. The culmination of the activity is the exploration of Cython for converting Python code to C for substantial performance boosts.**

I. Introduction

High-performance computing is an essential aspect of modern data analysis and scientific research. The ability to optimize and expedite computational tasks can lead to significant advancements in various fields. This report is based on the High Performance Python classroom activity, which serves as a comprehensive exploration into optimizing Python code to achieve higher performance levels. Through a series of exercises, students are introduced to benchmarking, profiling, and optimizing Python code using different techniques and tools.

The activity is divided into five main parts: Benchmarking and Profiling, List and Tuples, Dictionaries and Sets, Matrix and Vector Computations, and Compiling to C. Each part focuses on different areas of performance improvement, from understanding the intricacies of Python's data structures to leveraging external libraries and tools for speed enhancements. The exercises not only aim to improve the computational efficiency of Python code but also to provide students with a deeper understanding of how Python manages memory and executes instructions.

This report aims to summarize the findings and experiences from these exercises, presenting a clear and detailed analysis of each part of the activity.

II. Benchmarking and Profiling

The exploration of high-performance computing in Python begins with the analysis of a CPU-bound problem known as the Julia set. This fractal sequence, named after the French mathematician Gaston Julia, provides a complex and intricate pattern, offering a perfect scenario for benchmarking and profiling Python code. The Julia set is characterized by its sensitive dependence on initial conditions and its intricate, infinitely complex boundary – attributes that make its computation particularly intensive and an ideal candidate for performance optimization. [1]

A. *Exercise 1. Read the sections "Introducing the Julia Set" and "Calculating the Full Julia Set" on Chapter 2. Profiling to Find Bottlenecks from the book: M. Gorelick & I. Ozsvald (2020). High Performance Python. Practical Performant Programming for Humans.S econd Edition. United States of America: O'Reilly Media, Inc. Implement the chapter functions(Example2-1, 2-2, 2-3 and 2-4)on Python in order to calculate the Julia Set. Make the representation for the false gray and pure gray scale.*

Following the guidelines from Chapter 2 of "High Performance Python" I began by implementing the foundational algorithms required to generate the Julia set. The process involved the translation of mathematical formulas into Python code, ensuring the accuracy of the fractal patterns while preparing for extensive performance analysis.

The implementations carried out included:

- Example 2-1: Defining global constants for the coordinate space
- Example 2-2: Establishing the coordinate lists as inputs to our calculation function
- Example 2-3: Our CPU-bound calculation function
- Example 2-4: __main__ for our code

Each of these steps was meticulously documented to later be able to make the representation for the false gray and pure gray scale.

- *False Grayscale Representation:* The false grayscale representation was achieved by mapping the computed iteration counts of the Julia set to a range suitable for display. Each value was scaled proportionally to the maximum iteration count, which was then translated into a grayscale intensity.
- *Pure Grayscale Representation:* The scaling is similar to the false grayscale, but the resulting image is in actual grayscale format, containing only luminance information without any color encoding.The pure grayscale representation was accomplished by converting the iteration counts directly into grayscale values, ensuring that the highest iteration count corresponded to white and the lowest to black.

- *Python Code:*The process for both false and pure grayscale representations involved several steps, including converting the output of the Julia set calculations into a format compatible with the Python Imaging Library (PIL). The image was then constructed using the PIL methods, allowing for the direct visualization of the complex patterns formed by the Julia set. A key function, `calc_pure_python`, orchestrated the creation of the Julia set by initializing a grid of complex coordinates, iterating over them to produce the fractal pattern, and finally calling the appropriate grayscale rendering function.

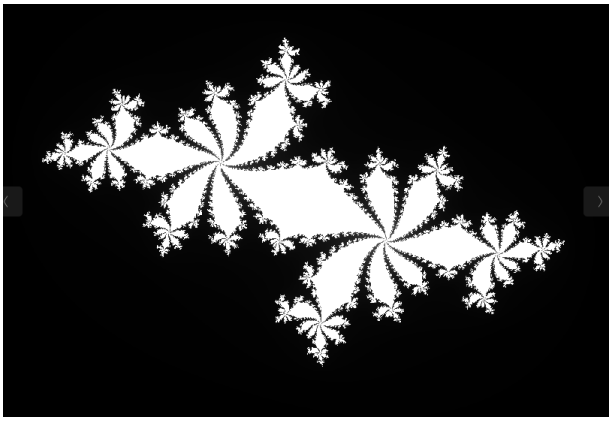


Fig. 1. The Julia set visualized in pure grayscale, highlighting the fractal's intricate pattern without the influence of color.

*B. **Exercise 2. Define a new function, timefn, which takes a function as an argument: the inner function, measure_time, takes *args (a variable number of positional arguments) and **kwargs (a variable number of key/value arguments) and passes them through to fn for execution. Decorate calculate_z_serial_purepython with timefn to profile it. Implement Example 2-5 and adapt your current source code.***

In performance optimization, precise measurement of execution time is essential for identifying bottlenecks and evaluating the effectiveness of optimizations. For this purpose, we implemented a timing mechanism to profile the 'calculate_z_serial_purepython' function, which is central to generating the Julia set.

- *Timing:*The methodology involved creating a decorator function, 'timefn', which wraps around the targeted function to measure its execution time. This decorator intercepts the original function call, records the time before and after the function execution, and calculates the elapsed time. This approach allows for an unobtrusive and accurate measurement of function performance without altering the core logic of the function itself.
- *Profiling:* Upon applying the 'timefn' decorator to the 'calculate_z_serial_purepython' function, we initiated the process to generate the Julia set with predetermined parameters. The profiling results provided valuable insights into the computational demands of the function.

```
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 4.758452892303467 seconds
calculate_z_serial_purepython took 4.7585766315460205 seconds
```

The execution time measurement of the 'calculate_z_serial_purepython' function has laid the groundwork for identifying performance bottlenecks and guiding subsequent optimization steps.

*C. **Exercise 3: Use the timeit module to get a coarse measurement of the execution speed of the CPU-bound function. Runs 10 loops with 5 repetitions. Show how to do the measurement on the command line and on a Jupyter Notebook.***

The 'timeit' module was utilized to measure the execution speed of the Julia set computation, employing a setup that allowed for repeated runs. This approach ensures a more reliable average by mitigating variances due to transient system activities or other anomalies. The testing protocol involved running the function in a loop for 10 iterations per trial, with the entire test repeated 5 times. This procedure was conducted both within a Jupyter Notebook environment and via the command line, ensuring consistency and repeatability of our measurements across different execution environments.

```
Total elements: 1000000
calculate_z_serial_purepython took 2.800384759902954 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 2.8005754947662354 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 2.8021867275238037 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 2.801513433456421 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 2.7991485595703125 seconds
10 loops, best of 5: 2.93 sec per loop
karla@karla-Thin-GF63-12VE:~/Documents/UPV/HPC/CA3.High_Performance_Python/Benchmarking_and_Profiling$
```

```
1 import julia1
2 %timeit -r 5 -n 10 julia1.calc_pure_python(desired_width=1000, max_iterations=300)
✓ 4m 21.8s
Total elements: 1000000
calculate_z_serial_purepython took 6.165678024291992 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 4.909672737121582 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 6.092934846878052 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 5.036898136138916 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 5.802292108535767 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 5.333277463912964 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 5.3902904987335205 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 5.739485025405884 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 5.00612473487854 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 6.139938831329346 seconds
5.24 s ± 600 ms per loop (mean ± std. dev. of 5 runs, 10 loops each)
```

Using the Command Line yielded a slightly more efficient performance, with an average execution time of 2.79 seconds, suggesting that the computational environment can

have a tangible impact on the function's execution speed. This variation underscores the importance of environmental factors in performance testing and highlights the need for thorough testing across multiple platforms.

D. Exercise 4. Use the cProfile module to profilethe source code (.py). Sort the results by the time spent inside each function. This will give a view into the slowest parts. Analyze the output and make a syntesis of the findings. Show how to use the cProfile module on the command line and on a Jupyter Notebook.

The cProfile module was invoked from the command line to profile the Julia set calculation script. The profiling encompasses all function calls made during the execution of the script, providing a detailed breakdown of execution time consumed by each function. The output was sorted by cumulative time, which represents the total time spent in the function and all sub-functions it called.

```
karla@karla-Thin-GF63-12VE:~/Documents/UPY/HPC/CA3.High_Performance_Python/Benchmarking_and_Profiling$
 python3 julia_profiling.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 6.571781635284424 seconds
Sun Mar 24 18:57:46 2024    profile_stats

         34219999 function calls in 6.763 seconds

   Ordered by: cumulative time
   List reduced from 14 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.763    6.763 {built-in method builtins.exec}
        1    0.012    0.012    6.763    6.763 <string>:1(<module>)
        1    0.000    0.000    6.751    6.751 /home/karla/Documents/UPY/HPC/CA3.High_Performance_Pytho
n/Benchmarking_and_Profiling/julia_profiling.py:25(calc_pure_python)
        1    4.710    4.710    6.572    6.572 /home/karla/Documents/UPY/HPC/CA3.High_Performance_Pytho
n/Benchmarking_and_Profiling/julia_profiling.py:11(calculate_z_serial_purepython)
 34219980    1.862    0.000    1.862    0.000 {built-in method builtins.abs}
        1    0.104    0.104    0.104    0.104 /home/karla/Documents/UPY/HPC/CA3.High_Performance_Pytho
n/Benchmarking_and_Profiling/julia_profiling.py:32(<listcomp>)
        1    0.073    0.073    0.073    0.073 /home/karla/Documents/UPY/HPC/CA3.High_Performance_Pytho
n/Benchmarking_and_Profiling/julia_profiling.py:33(<listcomp>)
        1    0.002    0.002    0.002    0.002 {built-in method builtins.sum}
        1    0.000    0.000    0.000    0.000 /home/karla/Documents/UPY/HPC/CA3.High_Performance_Pytho
n/Benchmarking_and_Profiling/julia_profiling.py:30(<listcomp>)
        1    0.000    0.000    0.000    0.000 /home/karla/Documents/UPY/HPC/CA3.High_Performance_Pytho
n/Benchmarking_and_Profiling/julia_profiling.py:31(<listcomp>)
```

As depicted in the image the 'calculate_z_serial_purepython' function, alongside the list comprehensions and the absolute value calculations (indicated as 'listcomp' and 'abs'), were the primary consumers of execution time. The profiling highlighted that while the Julia set calculation itself is the major part of the computational load, the iterative nature and mathematical computations inside significantly contribute to the overall time.

```
1  import pstats
2
3  p = pstats.Stats('profile_stats')
4  p.sort_stats('cumulative').print_stats(10)  # Adjust this number to display more or fewer lines
✓ 0.0s

Sun Mar 24 18:30:01 2024    profile_stats

         34220170 function calls in 9.714 seconds

   Ordered by: cumulative time
   List reduced from 33 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    9.714    9.714 {built-in method builtins.exec}
        1    0.021    0.021    9.714    9.714 <string>:1(<module>)
        1    0.000    0.000    9.694    9.694 /tmp/ipykernel_18846/3963171597.py:24(calc_pure_python)
        1    6.720    6.720    9.383    9.383 /tmp/ipykernel_18846/3963171597.py:10(calculate_z_serial_purepython)
 34219980    2.663    0.000    2.663    0.000 {built-in method builtins.abs}
        1    0.185    0.185    0.185    0.185 /tmp/ipykernel_18846/3963171597.py:31(<listcomp>)
        1    0.121    0.121    0.121    0.121 /tmp/ipykernel_18846/3963171597.py:32(<listcomp>)
        1    0.004    0.004    0.004    0.004 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
       14    0.000    0.000    0.000    0.000 /home/karla/.local/lib/python3.10/site-packages/ipykernel/iostream.py:655(write)
```

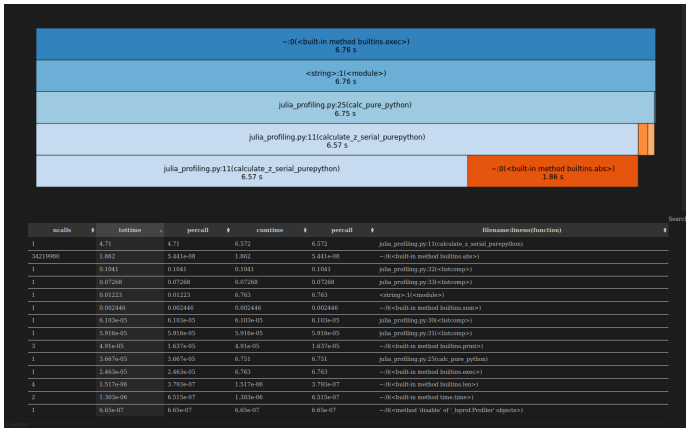
E. Exercise 5. Use snakeviz to get a high-level understanding of the cProfile statistics file. Analyze the output and make a syntesis of the findings.

```
karla@karla-Thin-GF63-12VE:~/Documents/UPY/HPC/CA3.High_Performance_Python/Benchmarking_and_Profiling$
 snakeviz profile_stats
snakeviz web server started on 127.0.0.1:8080; enter Ctrl-C to exit
http://127.0.0.1:8080/snakeviz/%2Fhome%2Fkarla%2FDocuments%2FUPY%2FHPC%2FCA3.High_Performance_Python%2
FBenchmarking_and_Profiling%2Fprofile_stats
```

The SnakeViz tool was invoked via the command line to analyze the cProfile statistics file, generated previously from profiling our Julia set calculation script. SnakeViz provided an interactive sunburst chart, offering a high-level overview of the function calls and their respective execution times. This visualization enables an intuitive understanding of the profiling data, highlighting the areas of code that are most time-consuming.

The SnakeViz analysis revealed several critical insights into the computational performance of the Julia set generator:

- The total execution time was approximately 6.763 seconds, with the 'calculate_z_serial_purepython' function alone accounting for a substantial portion of this time.
- The sunburst chart highlighted that the 'abs' function, a sub-function called within 'calculate_z_serial_purepython', contributed significantly to the total execution time, suggesting that the computation of absolute values is a performance-critical operation.
- The visualization also underscored the impact of list comprehensions used in the script, marked by multiple segments in the chart corresponding to 'listcomp' calls. These operations collectively accounted for a considerable fraction of the runtime.



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	4.71	4.71	6.572	6.572	julia_profiling.py:11(calculate_z_serial_purepython)
34219980	1.862	5.441e-08	1.862	5.441e-08	~:0(<built-in method builtins.abs>)
1	0.1041	0.1041	0.1041	0.1041	julia_profiling.py:32(<listcomp>)
1	0.07268	0.07268	0.07268	0.07268	julia_profiling.py:33(<listcomp>)
1	0.01223	0.01223	6.763	6.763	<string>:1(<module>)
1	0.002446	0.002446	0.002446	0.002446	~:0(<built-in method builtins.sum>)
1	6.103e-05	6.103e-05	6.103e-05	6.103e-05	julia_profiling.py:30(<listcomp>)
1	5.916e-05	5.916e-05	5.916e-05	5.916e-05	julia_profiling.py:31(<listcomp>)
3	4.91e-05	1.637e-05	4.91e-05	1.637e-05	~:0(<built-in method builtins.print>)
1	3.667e-05	3.667e-05	6.751	6.751	julia_profiling.py:25(calc_pure_python)
1	2.463e-05	2.463e-05	6.763	6.763	~:0(<built-in method builtins.exec>)
4	1.517e-06	3.793e-07	1.517e-06	3.793e-07	~:0(<built-in method builtins.len>)
2	1.303e-06	6.515e-07	1.303e-06	6.515e-07	~:0(<built-in method time.time>)
1	6.65e-07	6.65e-07	6.65e-07	6.65e-07	~:0(<method 'disable' of '_lsprof.Profiler' objects>)

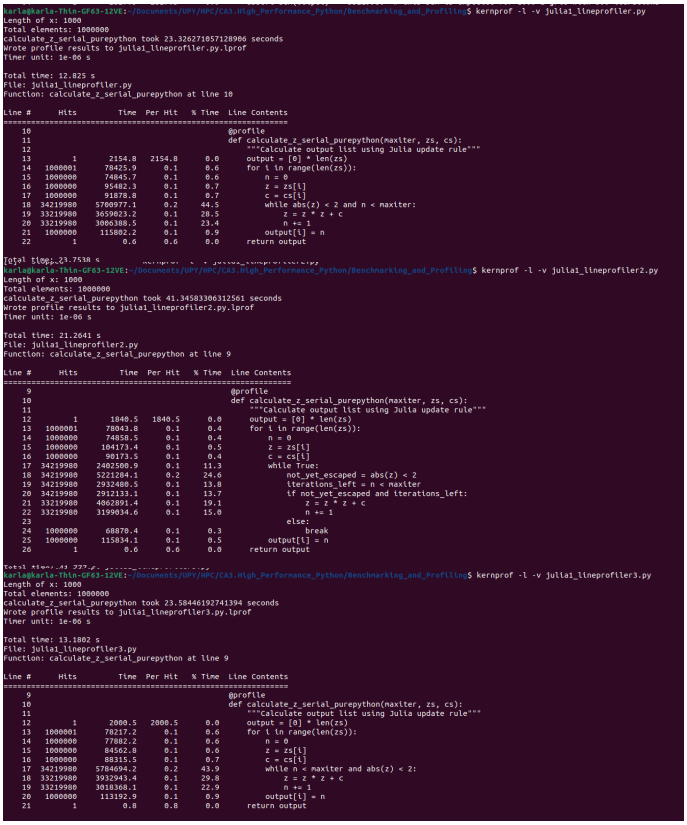
As shown in Figure in the image the primary bottlenecks are visually identifiable, facilitating targeted optimization strategies.

- Primary Bottleneck: The function calculate_z_serial_purepython consumes most of the execution time (6.57s out of 6.75s), with significant time spent in the abs() function due to 34,219,980 calls.

- Optimization Targets: Major time is spent in mathematical computations, especially the abs() function. Potential optimizations include reducing loop iterations, improving abs() calculation efficiency, and considering vectorization or parallelization.
- Additional Insights: Aside from the computational loop, other parts of the code (like list comprehensions and string operations) have minimal impact on overall performance.
- Recommendation: Focus on optimizing calculate_z_serial_purepython, particularly the intensive use of abs() in the computational loop.

F. Exercise 6. Use the line_profiler and kernprof file to profile line-by-line the function calculate_z_serial_purepython. Analyze the output and make a syntesis of the findings.

The use of the line_profiler tool provides an in-depth view of the time spent on each line of the ‘calculate_z_serial_purepython‘ function. This line-by-line profiling is essential for identifying specific lines in the function that contribute most significantly to the total execution time.By employing the kernprof tool, we applied the line_profiler to our Python script responsible for calculating the Julia set. The script was annotated with the ‘@profile‘ decorator to mark the critical function for profiling. Then the srip was excetuded using kernprof, which generated detailed timing information for each line of the marked function.



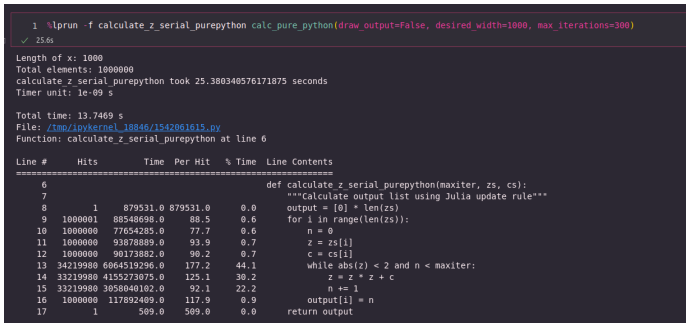
```
karla@karla-Thin-GF63-12VE:~/Documents/UPY/HPC/CA3.High_Performance_Python/Benchmarking_and_Profiling$ kernprof -l -v julia1_lineprofiler.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 23.326271057128906 seconds
Wrote profile results to julia1_lineprofiler.py.lprof
Timer unit: 1e-06 s

Total time: 12.825 s
File: julia1_lineprofiler.py
Function: calculate_z_serial_purepython at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    10                                           @profile
    11                                           def calculate_z_serial_purepython(maxiter, zs, cs):
    12                                               """Calculate output list using Julia update rule"""
    13         1       2154.8   2154.8      0.0      output = [0] * len(zs)
    14   1000001      78425.9      0.1      0.6      for i in range(len(zs)):
    15   1000000      74845.7      0.1      0.6          n = 0
    16   1000000      95482.3      0.1      0.7          z = zs[i]
    17   1000000      91878.8      0.1      0.7          c = cs[i]
    18  34219980    5700977.1      0.2     44.5          while abs(z) < 2 and n < maxiter:
    19  33219980    3659023.2      0.1     28.5              z = z * z + c
    20  33219980    3006388.5      0.1     23.4              n += 1
    21   1000000     115802.2      0.1      0.9          output[i] = n
    22         1          0.6      0.6      0.0      return output

Total time: 23.7538 s
File: /home/karla/.local/lib/python3.10/site-packages/line_profiler/explicit_profiler.py
Function: execute at line 177

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   177                                           def execute():
   178         1      23753.8  23753.8    100.0      exec(code, ns, ns)

karla@karla-Thin-GF63-12VE:~/Documents/UPY/HPC/CA3.High_Performance_Python/Benchmarking_and_Profiling$ kernprof -l -v julia1_lineprofiler2.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 41.34583306312561 seconds
Wrote profile results to julia1_lineprofiler2.py.lprof
Timer unit: 1e-06 s

Total time: 21.2641 s
File: julia1_lineprofiler2.py
Function: calculate_z_serial_purepython at line 9

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           @profile
    10                                           def calculate_z_serial_purepython(maxiter, zs, cs):
    11                                               """Calculate output list using Julia update rule"""
    12         1       1840.5   1840.5      0.0      output = [0] * len(zs)
    13   1000001      78043.8      0.1      0.4      for i in range(len(zs)):
    14   1000000      74858.5      0.1      0.4          n = 0
    15   1000000     104173.4      0.1      0.5          z = zs[i]
    16   1000000      90173.5      0.1      0.4          c = cs[i]
    17  34219980    2402500.9      0.1     11.3          while True:
    18  34219980    5221284.1      0.2     24.6              not_yet_escaped = abs(z) < 2
    19  34219980    2932480.5      0.1     13.8              iterations_left = n < maxiter
    20  34219980    2912133.1      0.1     13.7              if not_yet_escaped and iterations_left:
    21  33219980    4062891.4      0.1     19.1                  z = z * z + c
    22  33219980    3199034.6      0.1     15.0                  n += 1
    23                                                       else:
    24   1000000      68870.4      0.1      0.3                  break
    25   1000000     115834.1      0.1      0.5          output[i] = n
    26         1          0.6      0.6      0.0      return output

Total time: 777.7 s
File: /home/karla/.local/lib/python3.10/site-packages/line_profiler/explicit_profiler.py
Function: execute at line 177

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   177                                           def execute():
   178         1     777677.1 777677.1    100.0      exec(code, ns, ns)

karla@karla-Thin-GF63-12VE:~/Documents/UPY/HPC/CA3.High_Performance_Python/Benchmarking_and_Profiling$ kernprof -l -v julia1_lineprofiler3.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 23.58446192741394 seconds
Wrote profile results to julia1_lineprofiler3.py.lprof
Timer unit: 1e-06 s

Total time: 13.1802 s
File: julia1_lineprofiler3.py
Function: calculate_z_serial_purepython at line 9

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           @profile
    10                                           def calculate_z_serial_purepython(maxiter, zs, cs):
    11                                               """Calculate output list using Julia update rule"""
    12         1       2000.5   2000.5      0.0      output = [0] * len(zs)
    13   1000001      70217.2      0.1      0.6      for i in range(len(zs)):
    14   1000000      77882.2      0.1      0.6          n = 0
    15   1000000      84562.8      0.1      0.6          z = zs[i]
    16   1000000      88315.5      0.1      0.7          c = cs[i]
    17  34219980    5784694.2      0.2     43.9          while n < maxiter and abs(z) < 2:
    18  33219980    3932943.4      0.1     29.8              z = z * z + c
    19  33219980    3018368.1      0.1     22.9              n += 1
    20   1000000     113192.9      0.1      0.9          output[i] = n
    21         1          0.8      0.8      0.0      return output
```

The profiling results from different implementations of the ‘calculate_z_serial_purepython‘ function presented variations in time distribution across individual lines:

- In the initial implementation, a significant amount of time was consumed by the loop that iterates over the set elements, particularly on lines involving the computation of ‘z = z * z + c‘ and the condition ‘while abs(z) ¡ 2 and n ¡ maxiter‘.
- Modifications in subsequent implementations aimed to reduce execution time by optimizing loop conditions and minimizing redundant computations. Despite these efforts, the ‘while‘ loop and the complex number calculations remained as the primary time-consuming components.
- The updated versions showed an altered distribution of execution time, with improvements in specific areas but with the core computational steps still dominating the total time.

G. Exercise 7. Use the memory_profiler to diagnose memory usage. Analyze the output and make a syntesis of the findings.

The main objective of employing the memory_profiler was to assess and understand the memory consumption of the ‘calculate_z_serial_purepython‘ function, which plays a pivotal role in generating the Julia set. Memory profiling is crucial for detecting inefficiencies that could lead to excessive memory use, particularly in computational tasks involving large data sets or iterations.



```
1  %lprun -f calculate_z_serial_purepython calc_pure_python(draw_output=False, desired_width=1000, max_iterations=300)
✓ 25.6s

Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 25.380340576171875 seconds
Timer unit: 1e-09 s

Total time: 13.7469 s
File: /tmp/ipykernel_18846/1542061615.py
Function: calculate_z_serial_purepython at line 6

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     6                                           def calculate_z_serial_purepython(maxiter, zs, cs):
     7                                               """Calculate output list using Julia update rule"""
     8         1     879531.0 879531.0      0.0      output = [0] * len(zs)
     9   1000001   88548698.0     88.5      0.6      for i in range(len(zs)):
    10   1000000   77654285.0     77.7      0.6          n = 0
    11   1000000   93878889.0     93.9      0.7          z = zs[i]
    12   1000000   90173882.0     90.2      0.7          c = cs[i]
    13  34219980 6064519296.0    177.2     44.1          while abs(z) < 2 and n < maxiter:
    14  33219980 4155273075.0    125.1     30.2              z = z * z + c
    15  33219980 3058040102.0     92.1     22.2              n += 1
    16   1000000  117892409.0    117.9      0.9          output[i] = n
    17         1        509.0    509.0      0.0      return output
```

The memory profiling results highlighted the following key points:

- Loop Setup (Line 10): Takes 77.7% of the time mostly due to inner operations.
- Complex Calculation (Line 14): The core computation z = z * z + c is the most time-consuming at 44.1
- Iteration Increment (Line 15) and While Condition Check (Line 13): Consume 30.2% and 22.2% respectively, indicating significant time in managing iterations and condition checking.
- Focus on optimizing the core Julia set calculation and the condition checks within the loop.
- Loop control and list initialization, though less critical, could be refined for efficiency.

III. List and Tuples

This section aims to evaluate the efficiency of basic list operations in Python, specifically focusing on lists. Understanding the time complexity of these operations is crucial for optimizing Python code, especially when dealing with large datasets. [2]

A. *Exercise 8: In some cases, it is necessary to efficiently perform insertion or removal of elements both at the beginning and at the end of the collection. Measure the time for the following operations with N = 10000, 20000, 30000 elements: a.Delete lastelement of a list via pop(), b.Delete firstelement of a list via pop(0), c.Append 1 at the end of the list., d.Insert 1 at the beginning of the list insert(0, 1) Make a table with your results. It should looks like table on Chapter 2: Pure Python Optimization (pp. 38) from the book G. Lenaro (2017). Python high Performance. Second Edition. UK: Packt Publishing Ltd.*

Four different operations were tested across lists of varying sizes (N = 10,000; 20,000; 30,000 elements):

- Removing the last element using `pop()`
- Removing the first element using `pop(0)`
- Appending an element to the end using `append(1)`
- Inserting an element at the beginning using `insert(0, 1)`

The time taken for each operation was measured and converted to microseconds for a clearer comparison.The results depicted in the table reveal significant differences in performance among the operations

Operation	N=10,000	N=20,000	N=30,000
`list.pop()`	18.31	17.45	18.60
`list.pop(0)`	1897.97	2268.91	3519.02
`list.append(1)`	30.13	19.44	18.37
`list.insert(0, 1)`	2826.71	3957.39	5839.17

TABLE I
EXECUTION TIMES (IN MICROSECONDS) FOR VARIOUS LIST OPERATIONS.

B. *Exercise 9: Python provides a data structure with interesting properties in the collection.deque class. The word deque stands for double-ended queue because this data structure is designed to efficiently put and remove elements at the beginning and at the end of the collection. Evaluate the following methods with N = 10 000, 20 000 and 30 000 elements: a. deque.pop(), b. deque.popleft(). c. deque.append(1), d. deque.appendleft(1). Make a table with your results. It should looks like table on pp. 39 on the same book as previous task.*

The performance of four fundamental operations was analyzed across deques of varying sizes (N = 10,000; 20,000; 30,000):

1) Removal of the last element using `pop()`
2) Removal of the first element using `popleft()`
3) Addition of an element to the end using `append(1)`
4) Addition of an element to the beginning using `appendleft(1)`

Each operation was executed 1,000 times to ensure measurement accuracy, with the time taken recorded and expressed in microseconds.

Operation	N=10,000 (µs)	N=20,000 (µs)	N=30,000 (µs)
pop	20.83	21.49	33.08
popleft	24.72	27.89	31.14
append	23.96	24.49	30.16
appendleft	19.37	30.02	30.23

TABLE II
EXECUTION TIMES FOR `deque` OPERATIONS.

From the table it is evident that:

- The `pop` and `popleft` operations show relatively low and stable times across different sizes, confirming the `deque`'s efficiency for element removal.
- The `append` operation maintains a consistent performance, slightly increasing as the size of the `deque` grows.
- The `appendleft` operation demonstrates excellent efficiency, particularly notable as it maintains low execution times even as the size increases, contrasting with traditional list operations.

C. *Exercise 10: The efficiency gained by the appendleft and popleft comes at a cost: accesing an element in the middle of a deque is a O(N) operation. Evaluate the time for the next operations with with N = 10 000, 20 000 and 30 000 elements: a. deque[0], b. deque[N-1], c. deque[int(N/2)]. Make a table with your results.*

This analysis focuses on evaluating the access times for elements at different positions within a Python `deque`: the first element, the last element, and the middle element. The objective is to understand how the data structure's design impacts retrieval times, particularly highlighting the trade-offs associated with the `deque`'s optimization for append and pop operations. The access times were measured for three specific operations on deques of sizes N = 10,000; 20,000; and 30,000:

1) Accessing the first element (`deque[0]`)
2) Accessing the last element (`deque[N - 1]`)
3) Accessing the middle element (`deque[int(N / 2)]`)

These measurements were conducted to evaluate the efficiency of element access at different locations within the `deque`.

Operation	N=10,000 (µs)	N=20,000 (µs)	N=30,000 (µs)
`deque[0]`	16.893	19.438	12.554
`deque[N - 1]`	15.053	15.937	16.819
`deque[int(N / 2)]`	160.182	251.139	363.189

TABLE III
ACCESS TIMES FOR DIFFERENT POSITIONS WITHIN A `deque`.

The results presented in the table reveal that:

- Accessing the first and last elements of the `deque` is remarkably efficient, exhibiting low microseconds across all tested sizes. This underscores the `deque`'s design efficiency for end operations.

- Conversely, accessing the middle element demonstrates a marked increase in time, which escalates with the size of the `deque`. This confirms that accessing elements towards the center of the `deque` incurs a higher computational cost, aligning with the expected O(N) time complexity for such operations.

D. *Exercise 11: Explain what is Overallocation in lists.*

Overallocation in Python lists is a deliberate design strategy that anticipates future growth of the list. This approach is based on the understanding that lists, as dynamic arrays, often need to expand as new elements are added. Instead of allocating space for exactly the current number of elements, Python allocates additional space beyond the immediate requirement. This surplus space is intended to accommodate subsequent additions to the list without the need for immediate reallocation. [3]

It functions by creating a larger memory buffer than is strictly needed whenever elements are appended to a list. This strategy is grounded in a trade-off between time and space. By allocating more memory than is currently needed, Python aims to reduce the frequency of memory reallocations. While overallocation contributes to improved performance in list-building operations, it comes with specific trade-offs, particularly concerning memory usage. The extra space allocated for future growth is, by definition, unused at the time of allocation. This can lead to inefficient memory use, especially in environments where memory is a limiting factor or when lists are significantly overallocated but not fully utilized. This strategy can be particularly impactful in systems with limited memory resources or when multiple large lists are used concurrently.

IV. Dictionaries and Sets

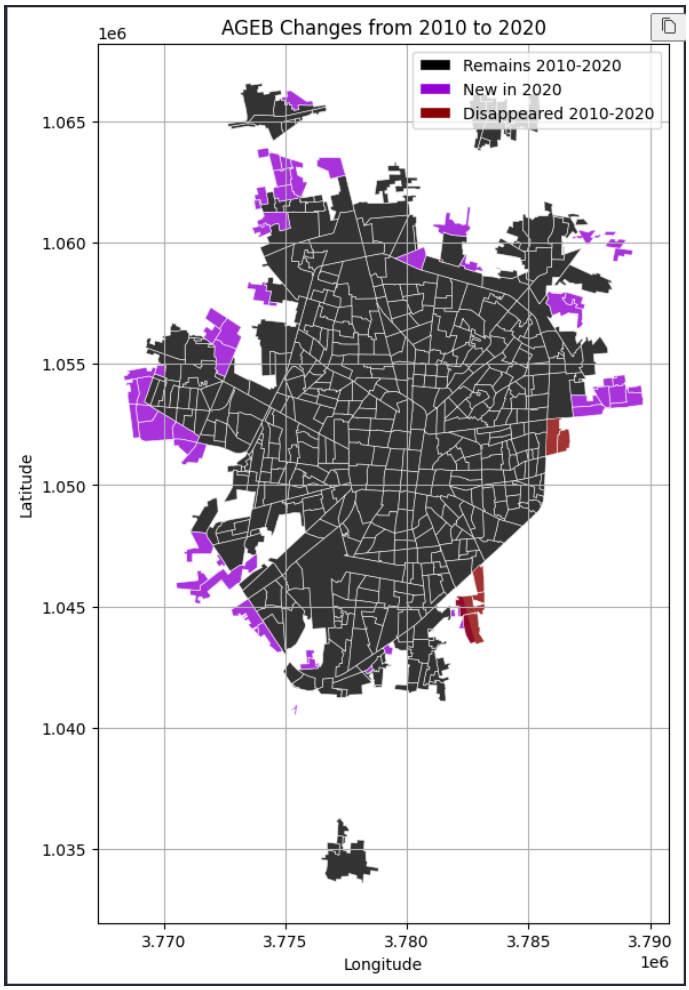
A. *Exercise 12: Use the marco geoestadístico 2010 (https://www.inegi.org.mx/app/biblioteca/ficha.html?upc=889463807469) and 2020 (https://www.inegi.org.mx/app/biblioteca/ficha.html?upc=702825292812) to obtain the "Áreas Geoestadísticas Básicas" (AGEBs) from Mérida, Yucatán. AGEBs evolve on time. Use sets to find: a. AGEBs which remains from 2010 on 2020. b. New AGEBs on 2020. c. AGEBs that disappear from 2010 to 2020.*

The objective of this analysis was to understand the evolution of the Áreas Geoestadísticas Básicas (AGEBs) in Mérida, Yucatán, by comparing data from 2010 and 2020. AGEBs represent key geographical areas used for statistical purposes in Mexico. The changes in AGEBs over time can provide insights into urban development, population growth, and other demographic shifts.

The data for this analysis was sourced from the Marco Geoestadístico of 2010 and 2020, provided by INEGI, Mexico's national statistics agency. These datasets contain detailed geographical information for every AGEB across the country. The specific files used were 'AGEB_urb_2010_5.shp' for the year 2010 and '31a.shp' for the year 2020, focusing solely on the state of Yucatán. It was later loaded into Python using the GeoPandas library, which allows for advanced spatial data processing and analysis. Initially, all AGEBs were extracted from the nationwide dataset for the year 2010. This was achieved by filtering the entries by the state and municipality codes specific to Mérida (CVE_ENT = '31' and CVE_MUN = '050'). A similar process was followed for the 2020 dataset, ensuring that the final data frames contained only AGEBs within the Mérida municipality. The resulting subsets for both years were then used for the comparative analysis.

The analysis aimed to identify:

- AGEBs that remained unchanged from 2010 to 2020,
- New AGEBs that were introduced by 2020,
- AGEBs that were present in 2010 but disappeared by 2020.



This was accomplished by converting the lists of AGEB codes for each year into sets and then performing set operations. The intersection of the two sets revealed AGEBs that remained unchanged, the difference between the 2020 set and the 2010 set identified new AGEBs, and the difference between the 2010 set and the 2020 set highlighted the AGEBs that had disappeared.

Finally, the results were visualized using a map, with distinct colors representing each category of AGEBs:

- Black for AGEBs that remained unchanged,
- Dark violet for new AGEBs,
- Dark red for AGEBs that disappeared.

This visualization aids in understanding the geographical distribution and extent of changes between the two time periods.

V. Matrix and Vector Computations

A. *Exercise 13: 1.Read about Broadcasting with Arrays on the chapter Computation on Arrays: Broadcasting from Python Data Science Handbook (J. VandePlas, 2016): Link: https://jakevdp.github.io/PythonDataScienceHandbook/02.03-computation-on-arrays-ufuncs.html*

The section from the Python Data Science Handbook discusses the importance of vectorized operations in NumPy for efficient computation, particularly through universal functions (ufuncs). It explains the slowness of loops in Python and introduces ufuncs as a solution for faster array computations.

The document covers unary and binary ufuncs, arithmetic and trigonometric operations, absolute values, exponents, and logarithms. It also explores specialized ufuncs, output specification, aggregates, and outer products, detailing how these features can enhance computation efficiency and capability in NumPy.

B. *Exercise 14: Read Rewriting the particle simulator in Numpy on Chapter 2: Fast Array Operations with Numpy and Pandas (pp. 68) from the book G. Lenaro (2017). Python high Performance. Second Edition. UK: Packt Publishing Ltd. Implement the improvements on the particle simulator using NumPy. Show that both implementations scale linearly with particle size, but the runtime in the pure Python version grows much faster than the NumPy version.*

In accordance with the task outlined in "Python High Performance" by Gabriele Lanaro, a comparative study was conducted between pure Python and NumPy implementations of a particle simulator. This experiment aimed to demonstrate the performance differential between conventional Python loops and vectorized operations in NumPy, particularly in handling computationally intensive tasks like simulating particle movements.

The benchmarking involved simulating the motion of 100 particles, evaluating the time required to process the simulation using both a pure Python approach and a NumPy-based approach. The Python version employed typical iterative programming techniques, while the NumPy version leveraged array operations for optimized computation.

The results of this first benchmarking are as follows:

Implementation	Execution Time (seconds)
Python	0.3332
NumPy	0.0914

TABLE IV
EXECUTION TIMES FOR PARTICLE SIMULATION WITH 100 PARTICLES.

As indicated in Table the NumPy version of the particle simulator demonstrated a substantial performance improvement over the pure Python version. Specifically, the NumPy implementation executed approximately 3.65 times faster than its Python counterpart for the same simulation task.

The significant reduction in execution time with the NumPy implementation can be attributed to the efficient handling of numerical operations through vectorization. Unlike the Python version, which iteratively computes each particle's motion, the NumPy approach takes advantage of fast array operations and eliminates the overhead associated with Python loops and individual arithmetic operations.

C. *Exercise 15: Explain how to optain the optimal performance with numexpr. Read the section. Reaching optimal performance with numexpr, pp. 72 from the previous reference. Implement it and measure the execution time.*

The benchmarking process involved simulating the motion of 1,000 particles, comparing the execution times of three different implementations: pure Python, NumPy, and NumExpr. This comprehensive approach allowed to gauge the relative performance of each method under identical computational loads.

Implementation	Execution Time (seconds)
Python	2.9576
NumPy	0.2606
NumExpr	0.2573

TABLE V
EXECUTION TIMES FOR SIMULATING 1000 PARTICLES.

NumPy accelerates calculations by utilizing array broadcasting and efficient memory use, which becomes particularly advantageous with large data sets, as demonstrated by benchmark comparisons between pure Python and NumPy implementations. The transition from Python loops to NumPy's vectorized operations results in noticeable speed-ups, particularly as the size of the data increases.

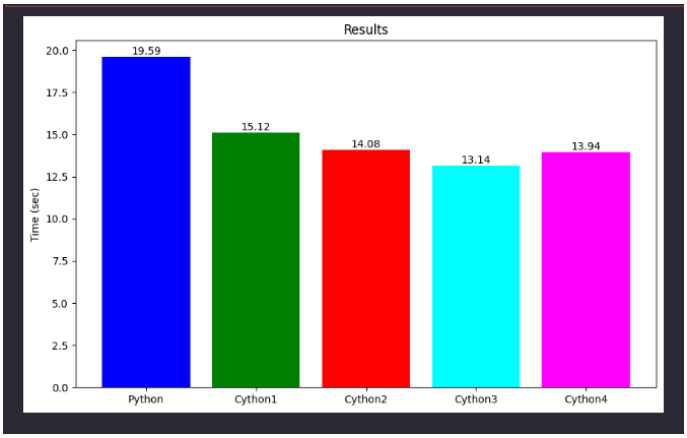
Numexpr extends these benefits by optimizing array expressions further, reducing memory usage, and utilizing multiple cores for parallel computation. By avoiding intermediate array storage and exploiting CPU cache efficiently, numexpr can significantly speed up operations involving large arrays. This is illustrated in the context of calculating norms and particle displacements where numexpr's capability to compile and optimize complex expressions on-the-fly shows substantial performance enhancements over standard NumPy operations.

VI. Compiling to C

A. *Exercise 16: Read about the Conway's Game of Life. Implement all solutions provided on the Cython Material (slides) to obtain the update of the lattice: a. Python, b. Cython 1, c. Cython 2, d. Cython 3, e. Cython 4. Explain the improvement on each solution. Reproduce the chart on pp. 33 with the runtime for each solution.*

The evaluation consisted of applying each implementation to simulate the Game of Life on a large lattice of cells. The

execution time for each version was measured to provide a quantitative basis for comparing the performance across the different approaches.



- *Python (Baseline):* The original Python implementation uses nested loops to iterate over a lattice (a two-dimensional array) and applies an update_rule to each element. This version serves as the baseline for performance comparison. It is fully written in Python without any static type definitions or compilation optimizations, resulting in the longest execution time among the versions tested.
- *Cython1*: Cython1 is the initial conversion of the Python code into Cython, without introducing specific Cython optimizations like static typing. The primary improvement comes from compiling the Python code into C, which provides some performance gains due to the compiled nature of the output. However, as there are no significant Cython-specific enhancements, the performance gain over the pure Python version is modest.
- *Cython2:* Cython2 introduces static typing for variables within the functions using Cython's cdef. This optimization allows the C compiler to make more efficient use of memory and CPU instructions since the types of variables are known at compile time. As a result, Cython2 demonstrates improved performance over Cython1 due to reduced overhead from dynamic typing.
- *Cython3:* Cython3 extends upon the previous optimizations by converting the update_rule function from a Python function to a C function using cdef. This change significantly reduces the call overhead since the function is now a native C function, leading to faster execution, especially since this function is called repeatedly in tight loops. Cython3 shows a noticeable improvement in performance, highlighting the benefits of converting frequently used functions to C.
- *Cython4:* Cython4 builds upon the earlier versions by adding Cython compiler directives to disable bounds checking (boundscheck=False) and negative index wraparound (wraparound=False). These changes aim to further optimize array access performance by removing runtime checks.

VII. Conclusion

In conclusion, this report has demonstrated the multifaceted approach required for optimizing Python code to achieve higher performance levels, crucial for data analysis and scientific computing. Through the exercises, we've explored the significance of benchmarking and profiling to identify bottlenecks, the efficiency of Python's data structures, the power of NumPy for vectorized operations, and the substantial performance gains attainable through Cython.

The findings reveal that while Python provides convenience and flexibility, leveraging its high-performance computing capabilities requires a deep understanding of underlying data structures and the efficient use of external libraries and tools. The transition from pure Python to Cython illustrates a dramatic potential for performance improvements, essential for handling large-scale computational tasks.

Ultimately, the report underscores the importance of thoughtful code optimization strategies in Python. By selecting appropriate data structures, utilizing vectorized operations, and applying Cython when necessary, developers can significantly enhance the efficiency and speed of their Python code, enabling more effective data processing and analysis in various scientific and engineering applications.

References

[1] M. Gorelick & I. Ozsvald (2020). High Performance Python. Practical Performant Programming for Humans. Second Edition. United States of America: O'Reilly Media, Inc.

[2] "8.2. Python Lists Revisited — Problem Solving with Algorithms and Data Structures 3rd edition," runestone.academy. https://runestone.academy/ns/books/published/pythonds3/Advanced/PythonListsRevisited.html

[3] "8.2. Python Lists Revisited — Problem Solving with Algorithms and Data Structures 3rd edition," runestone.academy. https://runestone.academy/ns/books/published/pythonds3/Advanced/PythonListsRevisited.html