# On the Peformance of the Python Language

Flávio Silva
pg57539@uminho.pt
University of Minho
Braga, Portugal

José Pacheco
pg55972@uminho.pt
University of Minho
Braga, Portugal

Sérgio Costa
pg54232@uminho.pt
University of Minho
Braga, Portugal

## Abstract

This document intends to study the performance of the Python language in terms of speed and energy efficiency.

We developed an object of research in order to find answers to the following questions: Which interpreter best handles recursion, iteration, memory management, and data manipulation throughout its processing and querying? Also, we wanted to investigate the unofficial compilers that are being developed asides the official Python organization, and to what extent are they beneficial, if any, compared with the best versions of the official interpreters.

Section 1 resides on an introduction to the aspects of this tool called "Python", following a methodology on section 3 that defines on how we purpose to find the answers to our object of research.

Section 4 and 5 shares the chosen versions trough which our investigates will take upon. In section 6 we share our test suitcase, with its subsequent results being shown and debated during section 7.

Our conclusions are presented in section 10, with its validity threats and desired future work expressed in sections 8 and 9, respectively.

## CCS Concepts

• **Software and its engineering → Runtime environments**.

## Keywords

Python, Energy Efficiency, Performance Benchmarking, Runtime Analysis, Speed Up, Green Up, Power Up

**ACM Reference Format:**
Flávio Silva, José Pacheco, and Sérgio Costa. 2025. On the Peformance of the Python Language. In *Proceedings of Universidae do Minho, Mestrado em Engenharia Informática, Experimentação em Engenharia de Software (EES)*. ACM, New York, NY, USA, 11 pages. https://doi.org/UMINHO.MEI.EES

## 1 Introduction

Although Python scripting language being one of the most poorly in terms of performance [6], it is one of the most used among all the programming languages [5]. Moreover, it is transverse within many fields of job departments, gaining its market throughout the spreadsheet world, within Microsoft Excel featuring a built-in python interpreter nowadays [3]. One of the main reasons of Python's vitality is certainly (1) the commitment from the development team to its users, as (2) the frameworks whom its community kept developing around the scripting language. Today, Python owns its vast areas of interest around the world of computer science and software, such as frameworks developed for (a) data science [4], (b) machine learning, (c) artificial intelligence, (d) object oriented programming, (e) object relational modelling, (f) web development, and so on.

With that in mind, we were challenged to conduct a study in order to keep track of Python interpreters energy performances throughout its versions lifespan. At first glance, the main goal was to conduct the study toward multiple official interpreted languages as so as unofficial compiled ones. Moreover, the original test suitcase was intended to be a couple of the main Python benchmarks to known [1]. Our goals lead us to build our own test suitcase[2], one that could benchmark the triviality of the ordinary Python user. Our benchmark runs different algorithms in the range of the known time complexities, evaluating different aspects of the Python scripting language, and giving its results individually and sectionalized within the different algorithm execution tasks.

Nonetheless, the focus of our research still in reporting the evolution of the Python interpreters along it's versions, as energy efficiency's concern. During the investigation process we found a couple of Python compilers whom results are worth to share.

## 2 Object of Research

With our investigation we want to indulge a conclusion in the following topics: memory management, recursion, iteration, data querying and processing, as long as the best Python compiler.

*RQ1:* Which interpreter best handles recursion?

*RQ2:* Which interpreter best handles iteration?

*RQ3:* Which interpreter handles memory management best?

*RQ4:* Which interpreter handles data querying and processing best?

*RQ5:* What's the most green compiler, and to which extent?

## 3 Methodology

The following section tries to elucidate on how did we conducted our research and its subsequent results.

In order to measure the time and energy consumed by any following program, we'll use the tool powered by Intel, the RAPL [1]. It estimates the total amount of energy spent by a program within the use of Intel power consumption counters that are built-in within the manufacturer chips [2]. The usage of the RAPL tool it is integrated

[1]https://benchmarksgame-team.pages.debian.net/benchmarksgame/index
[2]https://github.com/passas/on-python-energy-performance


*EES, Grupo 3*

ACM ISBN UM-MEI-EES-GRUPO-3/2025/05
https://doi.org/UMINHO.MEI.EES

in a C[3] pipeline, integrated in a bash shell one, via Python script. The main characteristic of this pipeline it is within the process before start, where we able to configure the interval temperature in which we want the measure process to start upon – with our experiments starting in the range between [25.0°C, 30.0°C]. This particular configuration enables the integration of the pipeline into a series of batch executions, *p.e.*, as it helps to control the current energy in form of heat within the circuit at the beginning of each program execution, avoiding the accumulation of previous amounts of energy from the previous execution. Furthermore, it helps to control the average work allowed from the operative system at the beginning of each execution.

An important note to share, is that RAPL also have a power capping functionality, so that we can limit the amount of CPU power within a program execution, enabling variations to the time and energy spent, accordingly.

We'll not cap the power of the executions, and so not pushing our investigations forward.

To conclude on how do we'll conduct our research, in matters of the RAPL pipeline, we'll make a batch of 10 sequential executions, with each one starting its performances between [25.0°C, 30.0°C] of CPU core temperatures. Moreover, the operative system is on idle, with no internet connection. The environment on which the tests will run are an Intel CPU, feeded at 2400MHz from a 8GB dual channel memory, within a 2,4000Mb/s ROM reach.

The producing results were made from the average of the 10 batch execution, excluding the first top and bottom execution value.

**Table 1: Environment Inspects**

| Component | Rate | Capacity |
|---|---|---|
| Intel® Core™ i3-9100F | 3.60 GHz to 4.20 GHz | – |
| RAM | DDR4 @ 2400 MHz | 2 x 8 GB |
| ROM | Read @ 2,400 MB/s | 250 GB |
| Operative System | | |
| Ubuntu 24.04.2 LTS w/ Linux 6.11.0-19-generic x86_64 | | |

## 4 Python Interpreted versions

For the purpose of this research, we majorly conduct our experiments on the 8 most recent python (3.x) releases – by date – after all, those were the alive versions [4] at the moment of this research. Nonetheless, in some cases we included python (2.x), in order to conclude in a broader sense.

**Table 2: Python interpreted versions**

| Version | Released Date | Maintenance Status |
|---|---|---|
| 3.13.2 | Feb. 4, 2025 | Bugfix |
| 3.13.1 | Dec. 3, 2024 | Bugfix |
| 3.12.9 | Feb. 4, 2025 | Bugfix |
| 3.12.8 | Dec. 3, 2024 | Bugfix |
| 3.12.3 | April 9, 2024 | Bugfix |
| 3.11.11 | Dec. 3, 2024 | Security |
| 3.10.16 | Dec. 3, 2024 | Security |
| 3.9.21 | Dec. 3, 2024 | Security |
| 3.0.1 | Feb. 13, 2009 | End-of-life |
| 2.7.18 | April 20, 2020 | End-of-life |
| 2.0.1 | June 22, 2001 | End-of-life |

The versions were downloaded from the official Python site.

## 5 Python Compilers

We eared rumours[5] of some efforts being taken on the development of independent Python compilers, which in some cases took the algorithm's performance near the performance of the C programming language [7].

In order to test those compilers, was carried some adaptations throughout the traditional Python's scripts (once the definitions behave as a compiled language usually do: declarations first, call appearance after).

**Table 3: Python compilers**

| Compiler | Version |
|---|---|
| Codon | 0.18.2 |
| Nuitka | 0.4.1 |

## 6 Test Suitcase

In order to answer our research questions, we developed our own programs[6]. The programs were developed under the following premises: (1) coverage representativeness, (2) under trivial functionalities, (3) ranging a vast time complexity scale, and (4) aim our the object of research. Furthermore, in some cases, we broke the tests into them different computational stages. This helps to comprehend different structures of the overall work whom being benchmarked.

The first category is related to known demand algorithms and its different time complexity versions, such as recursive Fibonacci and calculate the first $N$ primes via the Sieve of Eratosthenes algorithm.

[3]The C Programming Language.
[4]https://devguide.python.org/versions/#versions
[5]https://www.reddit.com/r/Python/comments/13cbemn/list_of_python_compilers/?rdt=38151
[6]https://github.com/passas/on-python-energy-performance/tree/main

**Table 4: Mainstream Test Suitcase**

| Name | Time Complexity | Brief |
|---|---|---|
| Fibonacci | $O(2^N)$ | Recursive |
| Sieve of Eratosthenes | $O(N * log(log(N)))$ | Prime table |

The second category is related to the sorting algorithms, where's meant to sort an array of $N$ integers.

**Table 5: Sorting Algorithms Test Suitcase**

| Algorithm | Best | Worst | Average |
|---|---|---|---|
| Bubble Sort | $O(N)$ | $O(N^2)$ | $O(N^2)$ |
| Insertion Sort | $O(N)$ | $O(N^2)$ | $O(N^2)$ |
| Shell Sort | $O(N * log^2(N))$ | $O(2^N)$ | $O(N * log^2(N))$ |
| Quick Sort | $O(N * log^2(N))$ | $O(2^N)$ | $O(N * log^2(N))$ |
| Merge Sort | $O(N * log^2(N))$ | $O(N * log^2(N))$ | $O(N * log^2(N))$ |
| Tim Sort | $O(N)$ | $O(N * log^2(N))$ | $O(N * log^2(N))$ |

The third category is related to the data analysis field. Here we do a couple of queries where first we have to load 3 files, each one representing a SQL table, and validate the semantic of the fields in it. Totalling an ingestion of `1.38 GB` of information, where there's 1 000 000 users to validate, 100 000 riders, and 10 000 000 rides, totalling in a validation of 107 900 026 words with an average of `13 bytes` per word.

In order to get each query answered, it's necessary to structure the loaded fields. Furthermore, a process of relating it's also necessary as a SQL query where we relate each entry table by some field (usually obeying to a constraint of a foreign key, as it's the case).

**Table 6: Querying Test Suitcase**

| Time Complexity | User Total Spent | Driver Score Rank |
|---|---|---|
| Best Case | $O(W) + O(1)$ | $O(W) + O(N) + k$ |
| Average Case | $O(W) + k$ | $O(W) + O(N) + k$ |
| Worst Case | $O(W) + O(N)$ | $O(W) + O(N) + k$ |

Where $W$ means the total words in validation, which are 107 900 026 total, according to our dataset, with a 12 average length. The $k$ is a constant that represents an $O(1)$ access, multiple – $k$ – times. The N represents the total of riders in the dataset which are about 100 000.

## 7 Results

In order to answer to our object of research, we adopted the following methodology:

- Allow the theoretical slower version to perform a `60 second` approximately computation; pick that work as the first round for all versions;
- If more than one version did the work under `30 seconds`, pick the shown slower and apply the `60 second` work criteria; proceed to the second round;
- And so forth.

Our work can be recognized on https://github.com/passas/on-python-energy-performance.

### 7.1 Fibonacci

This work leads the calculus of the Fibonacci sequence by applying the algorithm throughout a recursive fashion.

This benchmark revealed the Python's 3.11.11, and 3.12.8, as the most energy efficient interpreted versions. This may indicate that this versions stands out on recursive computational work. Such interpretation is made during the fact that this one is more feature equipped than previous ones, in order to argue that more features doesn't mean more slowness.

As compiled versions concerned, they are better, with special attention to Codon compiler, with results near the C compiled code.

**Table 7: Recursive Fibonacci – Energy X Time**

| N | 41 | | 43 | |
|---|---|---|---|---|
| Version | Energy (J) | Time (s) | Energy (J) | Time (s) |
| 2.0.1 | 1 730 | 74 | – | – |
| 2.7.18 | – | – | – | – |
| 3.0.1 | – | – | – | – |
| 3.9.21 | 1 036 | 49 | – | – |
| 3.10.16 | 1 154 | 54 | – | – |
| 3.11.11 | 531 | 26 | 1 418 | 67 |
| 3.12.3 | 551 | 26 | 1 479 | 69 |
| 3.12.8 | 536 | 25 | 1 423 | 66 |
| 3.12.9 | 536 | 25 | 1 430 | 66 |
| 3.13.1 | 573 | 29 | 1 577 | 76 |
| 3.13.2 | 591 | 29 | 1 586 | 76 |
| Nuitka | – | – | 975 | 46 |
| Codon | – | – | 64 | 0.003 |

**Table 8: Recursive Fibonacci – Power, Speed and Green Up**

| N | 43 | | |
|---|---|---|---|
| Version | Power Up | Speed Up | Green Up |
| 2.0.1 | – | – | – |
| 2.7.18 | – | – | – |
| 3.0.1 | – | – | – |
| 3.9.21 | – | – | – |
| 3.10.16 | – | – | – |
| 3.11.11 | 1 | 1 | 1 |
| 3.12.3 | 1.01 | 0.97 | 0.96 |
| 3.12.8 | 1.02 | 1.02 | 1.00 |
| 3.12.9 | 1.02 | 1.01 | 0.99 |
| 3.13.1 | 0.98 | 0.88 | 0.90 |
| 3.13.2 | 0.98 | 0.88 | 0.89 |
| Nuitka | 1.00 | 1.46 | 1.45 |
| Codon | 0.97 | 21.30 | 22.01 |

Table 8 supports the affirmation on the most green versions throughout the `Green Up` coefficient.

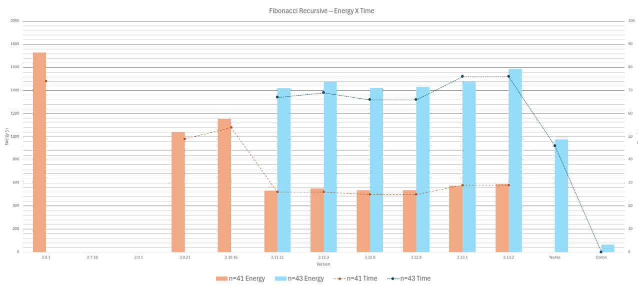


**Figure 1: Recursive Fibonacci – Energy X Time**

## 7.2 Sieve of Eratosthenes

This computational work consists on a for-cycle ranging $[0, N]$, each one comprehending a cycle between $[3, i[$ in order to test the remainder of $(i \div [3, i[)$.

Energetically speaking, version 3.12.8 of the Python interpreters were the most efficient one – proven by its `Green Up` as shown in table 10, arguing that is the best fitted in terms of iteration throughout a Python integer's list.

Once more, Codon were flawless on its energy performance.

**Table 9: Primes – Energy X Time**

| Range | 2000 000 000 | | 300 000 000 | |
|---|---|---|---|---|
| Version | Energy (J) | Time (s) | Energy (J) | Time (s) |
| 2.0.1 | – | – | – | – |
| 2.7.18 | – | – | – | – |
| 3.0.1 | 1 460 | 69 | – | – |
| 3.9.21 | 1 407 | 67 | – | – |
| 3.10.16 | 1 488 | 71 | – | – |
| 3.11.11 | 1 145 | 56 | – | – |
| 3.12.3 | 913 | 44 | 1 203 | 57 |
| 3.12.8 | 882 | 42 | 1 122 | 43 |
| 3.12.9 | 899 | 43 | 1 148 | 54 |
| 3.13.1 | 898 | 43 | 1 138 | 55 |
| 3.13.2 | 909 | 44 | 1 169 | 56 |
| Nuitka | – | – | 1 415 | 67 |
| Codon | – | – | 33 | 2 |

**Table 10: Primes – Power, Speed and Green Up**

| Range | 300 000 000 | | |
|---|---|---|---|
| Version | Power Up | Speed Up | Green Up |
| 2.0.1 | – | – | – |
| 2.7.18 | – | – | – |
| 3.0.1 | – | – | – |
| 3.9.21 | – | – | – |
| 3.10.16 | – | – | – |
| 3.11.11 | – | – | – |
| 3.12.3 | 1 | 1 | 1 |
| 3.12.8 | 1.0 | 1.08 | 1.07 |
| 3.12.9 | 1.01 | 1.06 | 1.05 |
| 3.13.1 | 0.99 | 1.05 | 1.06 |
| 3.13.2 | 0.99 | 1.02 | 1.03 |
| Nuitka | 1.01 | 0.86 | 0.85 |
| Codon | 1.02 | 37.16 | 36.60 |

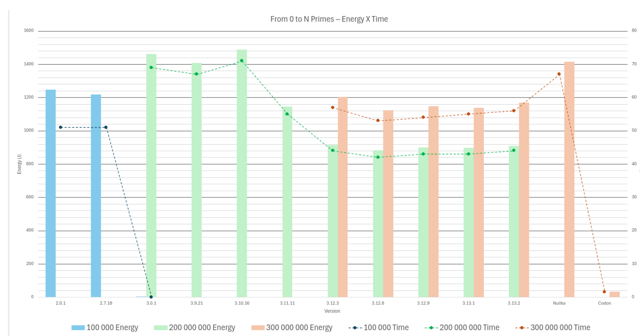


**Figure 2: Primes – Energy X Time**

## 7.3 Bubble Sort

This work intends to demonstrate a time complexity of a $\theta(N^2)$ with in a comparison and a swap operation between two consecutive elements within an array, with memory management as the object of research.

Table 12 suggests an improvement at memory management on version 3.11.11, which takes the recognition for being the most green efficient version of all the interpreted versions.

**Table 11: Bubble Sort – Energy X Time**

| Integers Version | 25 000 Energy (J) | Time (s) | 30 000 Energy (J) | Time (s) |
|---|---|---|---|---|
| 2.0.1 | 1 361 | 58 | – | – |
| 2.7.18 | 815 | 36 | 1 167 | 52 |
| 3.0.1 | 1 197 | 56 | – | – |
| 3.9.21 | 1 073 | 50 | – | – |
| 3.10.16 | 1 116 | 53 | – | – |
| 3.11.11 | 565 | 26 | 818 | 37 |
| 3.12.3 | 604 | 28 | 881 | 40 |
| 3.12.8 | 612 | 28 | 905 | 41 |
| 3.12.9 | 617 | 29 | 908 | 41 |
| 3.13.1 | 596 | 28 | 855 | 39 |
| 3.13.2 | 590 | 27 | 863 | 40 |
| Nuitka | – | – | 1 073 | 49 |
| Codon | – | – | 57 | 3 |

**Table 12: Bubble Sort – Power, Speed and Green Up**

| Integers Version | Power Up | 30 000 Speed Up | Green Up |
|---|---|---|---|
| 2.0.1 | – | – | – |
| 2.7.18 | 1 | 1 | 1 |
| 3.0.1 | – | – | – |
| 3.9.21 | – | – | – |
| 3.10.16 | – | – | – |
| 3.11.11 | 0.98 | 1.39 | 1.43 |
| 3.12.3 | 0.97 | 1.28 | 1.32 |
| 3.12.8 | 0.97 | 1.25 | 1.29 |
| 3.12.9 | 0.97 | 1.25 | 1.29 |
| 3.13.1 | 0.96 | 1.32 | 1.36 |
| 3.13.2 | 0.96 | 1.30 | 1.35 |
| Nuitka | 0.96 | 1.05 | 1.09 |
| Codon | 0.87 | 17.98 | 20.59 |

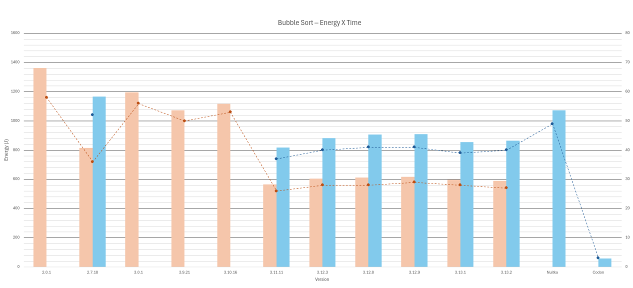


**Figure 3: Bubble Sort – Energy X Time**

## 7.4 Insertion Sort

This computational work intends to benchmark the memory management between far spatial array elements. With table 14 corroborating the advantage carried by version 3.11.11, with 1 . 39 of Green Up.

**Table 13: Insertion Sort – Energy X Time**

| Version | 40 000 Energy (J) | Time (s) | 50 000 Energy (J) | Time (s) |
|---|---|---|---|---|
| 2.0.1 | 1 194 | 53 | – | – |
| 2.7.18 | 678 | 33 | 1 090 | 51 |
| 3.0.1 | 1 171 | 58 | – | – |
| 3.9.21 | 964 | 48 | – | – |
| 3.10.16 | 974 | 48 | – | – |
| 3.11.11 | 498 | 24 | 784 | 38 |
| 3.12.3 | 575 | 28 | 927 | 45 |
| 3.12.8 | 573 | 28 | 916 | 44 |
| 3.12.9 | 573 | 28 | 916 | 44 |
| 3.13.1 | 645 | 34 | 1 017 | 52 |
| 3.13.2 | 644 | 33 | 1 019 | 52 |
| Nuitka | – | – | 883 | 41 |
| Codon | – | – | 47 | 2 |

**Table 14: Insertion Sort – Power, Speed and Green Up**

| Integers Version | Power Up | 50 000 Speed Up | Green Up |
|---|---|---|---|
| 2.0.1 | – | – | – |
| 2.7.18 | 1 | 1 | 1 |
| 3.0.1 | – | – | – |
| 3.9.21 | – | – | – |
| 3.10.16 | – | – | – |
| 3.11.11 | 0.97 | 1.35 | 1.39 |
| 3.12.3 | 0.97 | 1.14 | 1.18 |
| 3.12.8 | 0.98 | 1.16 | 1.19 |
| 3.12.9 | 0.98 | 1.16 | 1.19 |
| 3.13.1 | 0.92 | 0.99 | 1.07 |
| 3.13.2 | 0.93 | 0.99 | 1.07 |
| Nuitka | 1.02 | 1.26 | 1.23 |
| Codon | 0.99 | 22.74 | 23.07 |

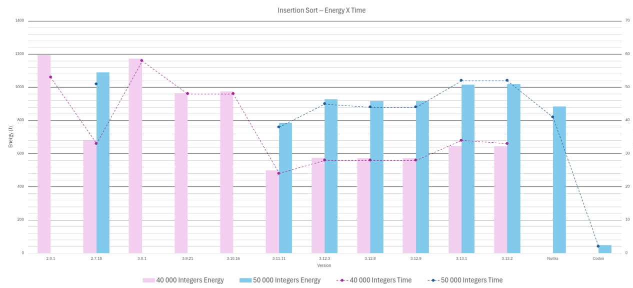


**Figure 4: Insertion Sort – Energy X Time**

**Table 16: Shell Sort – Power, Speed and Green Up**

| Integers Version | Power Up | 4 000 000 Speed Up | Green Up |
|---|---|---|---|
| 2.0.1 | – | – | – |
| 2.7.18 | – | – | – |
| 3.0.1 | – | – | – |
| 3.9.21 | – | – | – |
| 3.10.16 | – | – | – |
| 3.11.11 | 1 | 1 | 1 |
| 3.12.3 | 0.98 | 0.89 | 0.90 |
| 3.12.8 | 1.00 | 0.89 | 0.89 |
| 3.12.9 | 0.99 | 0.88 | 0.90 |
| 3.13.1 | 0.98 | 0.88 | 0.90 |
| 3.13.2 | 0.98 | 0.88 | 0.89 |
| Nuitka | 0.97 | 0.86 | 0.88 |
| Codon | 0.99 | 17.86 | 18.03 |

## 7.5 Shell Sort

This computational work helps conclude on memory management object of study, within the work of swap operation throughout the elements of the array. With table 16 concluding the advantage on that matter of field to version 3.11.11, with its unsurpassed Green Up value.

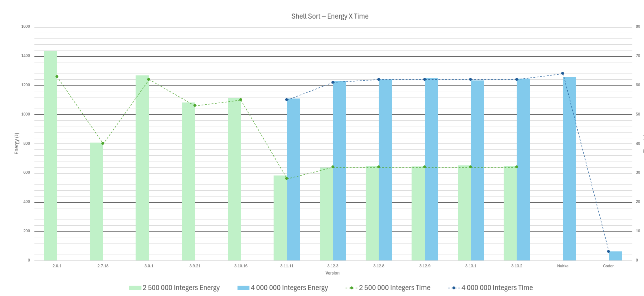


**Figure 5: Shell Sort – Energy X Time**

**Table 15: Shell Sort – Energy X Time**

| | 2 500 000 | | 4 000 000 | |
|---|---|---|---|---|
| Version | Energy (J) | Time (s) | Energy (J) | Time (s) |
| 2.0.1 | 1 434 | 63 | – | – |
| 2.7.18 | 806 | 40 | – | – |
| 3.0.1 | 1 268 | 62 | – | – |
| 3.9.21 | 1 082 | 53 | – | – |
| 3.10.16 | 1 113 | 55 | – | – |
| 3.11.11 | 582 | 28 | 1 110 | 55 |
| 3.12.3 | 636 | 32 | 1 228 | 61 |
| 3.12.8 | 646 | 32 | 1 240 | 62 |
| 3.12.9 | 645 | 32 | 1 248 | 62 |
| 3.13.1 | 652 | 32 | 1 233 | 62 |
| 3.13.2 | 646 | 32 | 1 245 | 62 |
| Nuitka | – | – | 1 255 | 64 |
| Codon | – | – | 62 | 3 |

## 7.6 Merge Sort

The jobs between a work like merge sort resides on two main factors: (1) memory management, and (2) context switching – abruptly [7], throughout its divide and conquer phases.

Once more, the victory resides on version 3.11.11, accordingly to the data on table 18.

[7] Cache being almost entirely replaced, within contribution to low spatial, and time location.

**Table 17: Merge Sort – Energy X Time**

| | 6 000 000 | | 9 000 000 | |
|---|---|---|---|---|
| Version | Energy (J) | Time (s) | Energy (J) | Time (s) |
| 2.0.1 | 1 423 | 63 | – | – |
| 2.7.18 | 801 | 40 | 1 252 | 62 |
| 3.0.1 | 1 171 | 58 | – | – |
| 3.9.21 | 1 012 | 49 | – | – |
| 3.10.16 | 994 | 49 | – | – |
| 3.11.11 | 563 | 28 | 845 | 44 |
| 3.12.3 | 615 | 30 | 926 | 47 |
| 3.12.8 | 618 | 30 | 923 | 47 |
| 3.12.9 | 616 | 30 | 921 | 47 |
| 3.13.1 | 627 | 32 | 944 | 51 |
| 3.13.2 | 636 | 33 | 960 | 51 |
| Nuitka | – | – | 1 028 | 52 |
| Codon | – | – | 118 | 6 |

**Table 18: Merge Sort – Power, Speed and Green Up**

| Integers | | 9 000 000 | |
|---|---|---|---|
| Version | Power Up | Speed Up | Green Up |
| 2.0.1 | – | – | – |
| 2.7.18 | 1 | 1 | 1 |
| 3.0.1 | – | – | – |
| 3.9.21 | – | – | – |
| 3.10.16 | – | – | – |
| 3.11.11 | 0.95 | 1.40 | 1.48 |
| 3.12.3 | 0.96 | 1.30 | 1.35 |
| 3.12.8 | 0.96 | 1.30 | 1.36 |
| 3.12.9 | 0.97 | 1.32 | 1.36 |
| 3.13.1 | 0.92 | 1.22 | 1.33 |
| 3.13.2 | 0.92 | 1.20 | 1.30 |
| Nuitka | 0.97 | 1.18 | 1.22 |
| Codon | 1.06 | 11.20 | 10.60 |

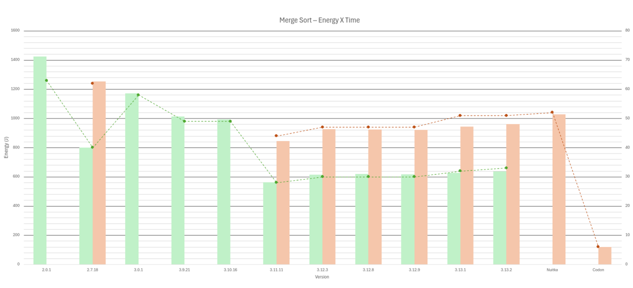


**Figure 6: Merge Sort – Energy X Time**

## 7.7 Quick Sort

Quick sort is an algorithm of "Divide & Conquer" type, although, its context switching does not require much effort compared to merge sort. Here, we're benchmarking memory management aspect only, with version 3.11.11 being the greener one (Table 20).

**Table 19: Quick Sort – Energy X Time**

| | 9 000 000 | | 17 000 000 | |
|---|---|---|---|---|
| Version | Energy (J) | Time (s) | Energy (J) | Time (s) |
| 2.0.1 | 1 330 | 60 | – | – |
| 2.7.18 | 797 | 39 | – | – |
| 3.0.1 | 1 120 | 53 | – | – |
| 3.9.21 | 899 | 43 | – | – |
| 3.10.16 | 897 | 43 | – | – |
| 3.11.11 | 513 | 27 | 1 034 | 54 |
| 3.12.3 | 572 | 29 | 1 166 | 60 |
| 3.12.8 | 586 | 30 | 1 200 | 61 |
| 3.12.9 | 565 | 29 | 1 157 | 59 |
| 3.13.1 | 572 | 30 | 1 156 | 60 |
| 3.13.2 | 584 | 30 | 1 178 | 61 |
| Nuitka | – | – | 1 755 | 87 |
| Codon | – | – | 91 | 5 |

**Table 20: Quick Sort – Power, Speed and Green Up**

| Integers | | 17 000 000 | |
|---|---|---|---|
| Version | Power Up | Speed Up | Green Up |
| 2.0.1 | – | – | – |
| 2.7.18 | – | – | – |
| 3.0.1 | – | – | – |
| 3.9.21 | – | – | – |
| 3.10.16 | – | – | – |
| 3.11.11 | 1 | 1 | 1 |
| 3.12.3 | 1.03 | 0.91 | 0.89 |
| 3.12.8 | 1.04 | 0.90 | 0.86 |
| 3.12.9 | 1.02 | 0.92 | 0.89 |
| 3.13.1 | 1.01 | 0.91 | 0.89 |
| 3.13.2 | 1.02 | 0.90 | 0.88 |
| Nuitka | 1.06 | 0.62 | 0.59 |
| Codon | 1.01 | 11.45 | 11.33 |

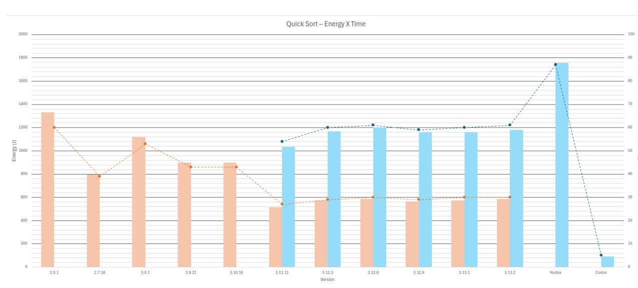


**Figure 7: Quick Sort – Energy X Time**

### 7.8 Tim Sort

Much like Merge sort algorithm, Tim sort changes abruptly its context – during the phases of divide and conquer. With this particular algorithm we manage to study the memory management of each version, as long as its context switching apparel.

Version 3.11.11 takes advantage on this benchmark too. Table 22 display the fact.

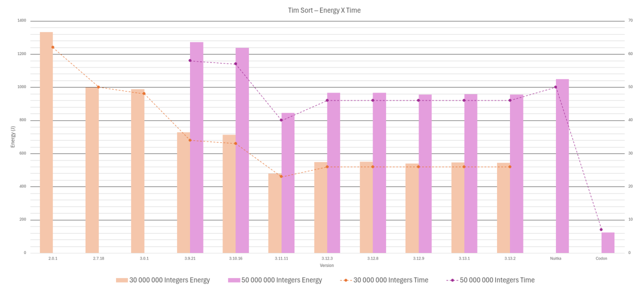


**Figure 8: Tim Sort – Energy X Time**

**Table 21: Tim Sort – Energy X Time**

| | 30 000 000 | | 50 000 000 | |
|---|---|---|---|---|
| Version | Energy (J) | Time (s) | Energy (J) | Time (s) |
| 2.0.1 | 1 333 | 62 | – | – |
| 2.7.18 | 999 | 50 | – | – |
| 3.0.1 | 987 | 48 | – | – |
| 3.9.21 | 727 | 34 | 1 271 | 58 |
| 3.10.16 | 712 | 33 | 1 238 | 57 |
| 3.11.11 | 478 | 23 | 844 | 40 |
| 3.12.3 | 549 | 26 | 966 | 46 |
| 3.12.8 | 550 | 26 | 967 | 46 |
| 3.12.9 | 540 | 26 | 955 | 46 |
| 3.13.1 | 547 | 26 | 957 | 46 |
| 3.13.2 | 544 | 26 | 955 | 46 |
| Nuitka | – | – | 1 049 | 50 |
| Codon | – | – | 124 | 7 |

### 7.9 User Total Spent

With this particular work we are investigating the behaviour of: (1) data ingestion, (2) data processing, (3) data structuring, and (4) data querying. The main underlaying principles are (a) memory management, (b) parsing and (c) context switching.

Here we observe that the most recent versions of Python interpreters take the advantage, with version 3.13.1 being the greener and faster (speed up) one – Table 24

Here we insight about a liability on Codon compiler, which is context switching, on table 24 we can see the close execution time and energy efficiency when compared to the best Python interpreted version.

**Table 22: Tim Sort – Power, Speed and Green Up**

| Integers | | 50 000 000 | |
|---|---|---|---|
| Version | Power Up | Speed Up | Green Up |
| 2.0.1 | – | – | – |
| 2.7.18 | – | – | – |
| 3.0.1 | – | – | – |
| 3.9.21 | 1 | 1 | 1 |
| 3.10.16 | 0.99 | 1.02 | 1.03 |
| 3.11.11 | 0.96 | 1.44 | 1.51 |
| 3.12.3 | 0.96 | 1.27 | 1.31 |
| 3.12.8 | 0.96 | 1.27 | 1.31 |
| 3.12.9 | 0.96 | 1.28 | 1.33 |
| 3.13.1 | 0.95 | 1.27 | 1.33 |
| 3.13.2 | 0.95 | 1.27 | 1.33 |
| Nuitka | 0.97 | 1.18 | 1.21 |
| Codon | 0.84 | 8.63 | 10.25 |

**Table 23: User Total Spent – Energy X Time**

| Accesses | 10 | | 30 | |
|---|---|---|---|---|
| Version | Energy (J) | Time (s) | Energy (J) | Time (s) |
| 2.0.1 | – | – | – | – |
| 2.7.18 | – | – | – | – |
| 3.0.1 | – | – | – | – |
| 3.9.21 | 1 196 | 60 | 1 224 | 59 |
| 3.10.16 | 1 203 | 59 | 1 225 | 59 |
| 3.11.11 | 1 041 | 52 | 1 066 | 52 |
| 3.12.3 | 1 063 | 53 | 1 084 | 53 |
| 3.12.8 | 1 050 | 53 | 1 081 | 53 |
| 3.12.9 | 1 061 | 53 | 1 086 | 53 |
| 3.13.1 | 918 | 46 | 932 | 46 |
| 3.13.2 | 926 | 46 | 942 | 46 |
| Nuitka | – | – | | |
| Codon | 833 | 37 | 835 | 37 |

**Table 24: User Total Spent – Power, Speed and Green Up**

| Accesses | | 33 | |
|---|---|---|---|
| Version | Power Up | Speed Up | Green Up |
| 2.0.1 | – | – | – |
| 2.7.18 | – | – | – |
| 3.0.1 | – | – | – |
| 3.9.21 | 1 | 1 | 1 |
| 3.10.16 | 1.01 | 1.01 | 1.00 |
| 3.11.11 | 1.00 | 1.15 | 1.15 |
| 3.12.3 | 1.00 | 1.13 | 1.13 |
| 3.12.8 | 1.00 | 1.13 | 1.13 |
| 3.12.9 | 1.00 | 1.13 | 1.13 |
| 3.13.1 | 0.98 | 1.29 | 1.31 |
| 3.13.2 | 0.99 | 1.28 | 1.30 |
| Nuitka | – | – | – |
| Codon | 1.11 | 1.62 | 1.47 |

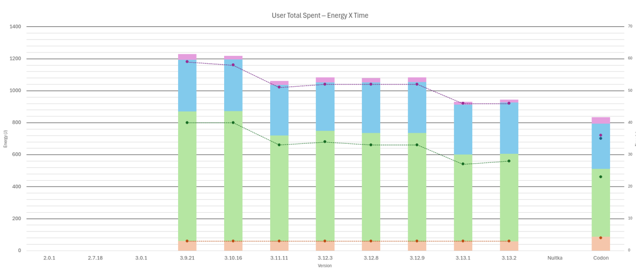


**Figure 9: User Total Spent – Energy X Time**

## 7.10 Drivers Score Rank

With this particular work, like in the "User Total Spent" benchmark, we're investigating the behaviour of: (1) data ingestion, (2) data processing, (3) data structuring, and (4) data querying. The main underlaying principles are (a) memory management, (b) parsing and (c) context switching.

Here we observe that the most recent versions of Python interpreters takes the advantage. Version 3.13.1 demonstrates to being the greener and faster (speed up) of the interpreted versions – Table 26

Here we, once more, insight about a liability on Codon compiler, which is context switching, on table 26 we can see the close execution time and energy efficiency when compared to the best Python interpreted version.

**Table 25: Drivers Score Rank – Energy X Time**

| Top | 10 | | 100 000 | |
|---|---|---|---|---|
| Version | Energy (J) | Time (s) | Energy (J) | Time (s) |
| 2.0.1 | – | – | – | – |
| 2.7.18 | – | – | – | – |
| 3.0.1 | – | – | – | – |
| 3.9.21 | 1 228 | 59 | 1 273 | 59 |
| 3.10.16 | 1 219 | 59 | 1 273 | 59 |
| 3.11.11 | 1 031 | 50 | 1 064 | 50 |
| 3.12.3 | 1 058 | 51 | 1 107 | 52 |
| 3.12.8 | 1 059 | 51 | 1 095 | 51 |
| 3.12.9 | 1 061 | 52 | 1 103 | 52 |
| 3.13.1 | 915 | 45 | 945 | 45 |
| 3.13.2 | 918 | 45 | 945 | 45 |
| Nuitka | – | – | | |
| Codon | 833 | 37 | 833 | 37 |

**Table 26: Drivers Score Rank – Power, Speed and Green Up**

| Top | | 100 000 | |
|---|---|---|---|
| Version | Power Up | Speed Up | Green Up |
| 2.0.1 | – | – | – |
| 2.7.18 | – | – | – |
| 3.0.1 | – | – | – |
| 3.9.21 | 1 | 1 | 1 |
| 3.10.16 | 1.00 | 1.00 | 1.00 |
| 3.11.11 | 0.99 | 1.18 | 1.20 |
| 3.12.3 | 1.00 | 1.14 | 1.15 |
| 3.12.8 | 0.99 | 1.15 | 1.16 |
| 3.12.9 | 1.00 | 1.15 | 1.15 |
| 3.13.1 | 0.98 | 1.32 | 1.35 |
| 3.13.2 | 0.98 | 1.30 | 1.33 |
| Nuitka | – | – | – |
| Codon | 1.02 | 1.50 | 1.47 |

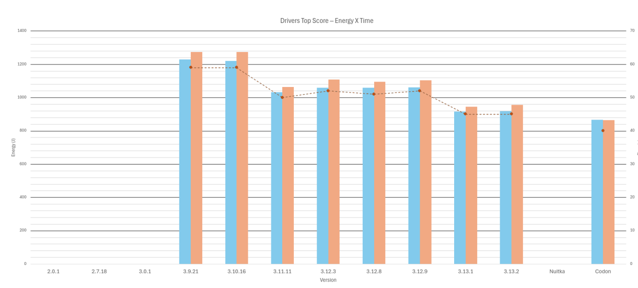


**Figure 10: Drivers Score Rank – Energy X Time**

## 8 Threats to Validity

This work have some threats to validity as we're only estimating the energy of an Intel CPU, which means that can not be representative to a more general way of sense like ARM architecture energy consumptions and so on.

The second validity threat of our work it's the lack of representativeness. Despite fulfilment of our object of research, in terms of Python work, we only estimate its consumptions during Fibonacci work, Primes work, Sorting Algorithms work and Data Analysis work.

The third validity threat of this work, it's the not pursuit-ness of power capping during computational work time, this can make an execution greener, as long as faster or slower.

## 9 Future Work

For our future work we would like to establish some benchmark according to Object Oriented Programming, as long as different data structures behaviour along the Python interpreters.

Furthermore, we would like to conduct more intensive memory management works, in order to fully understand the weaknesses and strengths of each versions during its specific tasks. Moreover, we would like to conduct a further study on context switching to fully understand the capabilities and liabilities in which we are inclined to believe that Codon compiler weakness reside.

## 10 Conclusions

This study evaluated the energy efficiency and runtime performance of Python across multiple interpreter versions and compilers, revealing critical insights into the language's evolution and optimization potential, with our findings leading us to believe that the tool is evolving toward a best data science fit.

In matter of our object of research we feel to give the following answers:

*RQ1:* Which interpreter best handles recursion? Python 3.12.8 – 7.1.

*RQ2:* Which interpreter best handles iteration? Python 3.12.8 – 7.2.

*RQ3:* Which interpreter handles memory management best? Python 3.11.11 – 7.3, 7.4, 7.5, 7.6, 7.7, and 7.8.

*RQ4:* Which interpreter handles data querying and processing best? Python 3.13.1 – 7.9, and 7.10.

*RQ5:* What's the most green compiler, and to which extent? Codon, with an average of `1.00` Power Up, `13.09` Speed Up, and `13.27` Green Up than the best Python interpreter version in each individual computational work category.

Furthermore, we would like to share a couple insights from the collaboration with the Codon compiler. Its main liabilities resides upon the lack of information within its compile error messages, and within the lack of feature support accordingly to the native language evolution, which can lead to a more slow development – may use on sideways.

## References


[1] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3, 2, Article 9 (March 2018), 26 pages. doi:10.1145/3177754

[2] Mateusz. August, 2024. *What is RAPL?* Green Compute UK. Retrieved April 12, 2025 from https://greencompute.uk/Measurement/RAPL

[3] Microsoft. 2023. *Get started with Python in Excel.* Microsoft. Retrieved April 12, 2025 from https://support.microsoft.com/en-us/office/get-started-with-python-in-excel-a33fbcbe-065b-41d3-82cf-23d05397f53d

[4] Felix Nahrstedt, Mehdi Karmouche, Karolina Bargieł, Pouyeh Banijamali, Apoorva Nalini Pradeep Kumar, and Ivano Malavolta. 2024. An Empirical Study on the Energy Usage and Performance of Pandas and Polars Data Analysis Python Libraries. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 58–68. doi:10.1145/3661167.3661203

[5] Stack Overflow. May, 2024. *Stack Overflow Annual Developer Survey.* Stack Overflow. Retrieved April 12, 2025 from https://survey.stackoverflow.co/2024/technology#most-popular-technologies

[6] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (Vancouver, BC, Canada) *(SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 256–267. doi:10.1145/3136014.3136031

[7] Unk. 2024. *What is Codon?* Exaloop. Retrieved April 12, 2025 from https://github.com/exaloop/codon?tab=readme-ov-file#examples


## A Codon – Green, Speed and Power Up

Here we share the results of the Codon compiler gains, by category.

### A.1 Overall

**Table 27: Overall Gains**

| Version | Power Up | Speed Up | Green Up |
|---|---|---|---|
| Top Tiers | 1 | 1 | 1 |
| Codon | 1.00 | 13.09 | 13.27 |

### A.2 Data Handling

**Table 28: Data Handling Gains**

| Version | Power Up | Speed Up | Green Up |
|---|---|---|---|
| 3.13.1 | 1 | 1 | 1 |
| Codon | 1.10 | 1.20 | 1.09 |

### A.3 Memory Management

**Table 29: Memory Management Gains**

| Version | Power Up | Speed Up | Green Up |
|---|---|---|---|
| 3.11.11 | 1 | 1 | 1 |
| Codon | 0.98 | 12.17 | 12.39 |

## A.4 Recursion

**Table 30: Recursion Gains**

| Version | Power Up | Speed Up | Green Up |
|---|---|---|---|
| 3.12.8 | 1 | 1 | 1 |
| Codon | 0.95 | 20.99 | 22.08 |

## A.5 Iteration

**Table 31: Iteration Gains**

| Version | Power Up | Speed Up | Green Up |
|---|---|---|---|
| 3.12.8 | 1 | 1 | 1 |
| Codon | 1.01 | 34.51 | 34.14 |