GreenCode: Promoting Eco-Friendly Coding Practices

Esha Dutta
TU Delft
5915996
e.dutta@student.tudelft.nl

Smruti Kshirsagar
TU Delft
5938643
s.s.kshirsagar@student.tudelft.nl

Giovanni Loureiro
TU Delft
4926854
g.fincatodeloureiro@student.tudelft.nl

ABSTRACT

This paper introduces GreenCode, a novel approach for identifying energy-efficient Python coding practices in Jupyter notebooks through the development of a Google Chrome plugin. By identifying and proposing alternative coding practices for common scenarios, GreenCode aims to reduce energy consumption in software development. In its current version, GreenCode has been built for four commonly used coding practices. This paper also presents a comparison of the energy consumption of these four coding practices and their alternatives. The effectiveness of GreenCode is validated through experiments on publicly available Python code, demonstrating significant reductions in energy usage. It is found that GreenCode gives favourable results for three of the four coding practices. Despite some limitations, GreenCode presents a promising solution for promoting eco-friendly coding practices and contributing to sustainability in the computing industry.

1 INTRODUCTION

As computational capabilities continue to advance, driven by the increasing integration of digital technologies into various aspects of society, there has been an exponential growth in the demand for computational resources. This surge in demand has not only transformed the landscape of computing but has also led to a significant increase in energy consumption, particularly within data centres and supercomputing facilities. According to a study by Katal et al., [1], the energy consumption of data centres alone is projected to rise from 200 TWh in 2016 to 2967 TWh by 2030, highlighting the substantial environmental impact of computing infrastructure. Consequently, there is a growing urgency to address the environmental implications of this energy consumption, with researchers focusing on reducing the carbon footprint of software applications [2]. By adopting responsible practices in software development, such as optimizing code to minimize power consumption, it is possible to not only promote conservation and sustainability but also lower operational costs for businesses and organizations.

Traditionally, code optimization techniques have primarily focused on improving performance and enhancing code extensibility, reusability, and testability. However, recent research has shown that such methods of code refactoring can also significantly reduce power consumption [3]. This is particularly relevant for commonly accessed code segments, which are executed frequently within applications. In data science, for example, operations such as data preprocessing, cleaning, and transformation often involve iterative or batch processing of large datasets. By optimizing commonly accessed functions to consume less energy, developers may achieve compounded energy savings, as these functions are invoked repeatedly within loops or by other functions. Even minor optimizations in such code segments have the potential to yield substantial energy savings, underscoring the importance of prioritizing energy efficiency in software development [4]. To enable developers to make informed decisions regarding energy-efficient software design, a solid knowledge base and guidance are essential for software architects, data scientists, and developers alike [5].

Python has emerged as a ubiquitous programming language, finding applications across diverse domains such as web development, data science, machine learning, and scientific computing [6]. Complementing Python's versatility, Jupyter Notebooks have gained widespread popularity for their interactive and exploratory environment, which seamlessly integrates code, text, and visualisations. Jupyter Notebook App is a server-client application that allows editing and running notebook documents,i.e., computer code and rich text elements via a web browser. The combination of Python and Jupyter Notebooks offers developers and researchers a powerful platform for prototyping, experimenting, and collaborating on projects across various disciplines. Apart from the actual applications, during the coding and debugging phase also, certain blocks of code or invoked functions are executed a large number of times by developers. Hence, there is a need to introduce energy efficiency for development in Python and Jupyter environments.

In this paper, we present a novel solution aimed at addressing energy inefficiencies in Python code through the development of a Jupyter plugin. We created a Google Chrome plugin designed to detect and suggest improvements for specific code segments identified as commonly inefficient. For the selection of these segments, we assessed different scenarios and commonly used inefficient code snippets, along with steps to improve them. The energy consumption of these codes was measured before and after the improvement. This study showed a considerable difference in energy consumption over a large number of reruns, which further highlights the need for our solution. Grounded in this empirical research, our plugin targets four specific code segments and suggests methods for refactoring them. Thus, the solution focuses on detecting and providing actionable recommendations for improved energy efficiency of the code.

The remainder of this paper is structured in the following way: The Related Work section provides a systematic review of literature pertaining to sustainable coding. The Methodology section gives insights into the proposed solution and its implementation. The following Validation section delves into the testing of the implemented plugin. The Discussion section states the contributions and effectiveness of the solution and Limitations and Future Work section discusses the drawbacks of the presented solution along with prospects of possible improvements. Finally, the Conclusion summarises the results and implications of the proposed solution.

2 RELATED WORK

Code refactoring for energy efficiency has been a subject of recent research. In this section, we briefly discuss some of the key findings and methodologies applied in studies related to sustainable software development.

The authors of [7] conducted a comparative analysis of energy consumption while using various Python functions, data initialization strategies, data access patterns, various data structures, string formatting and data visualisation. Some of the differences were significant; for example, for printing strings, the comma method was found to be most energy inefficient, as compared to f-string, format() method, and string concatenation using the + operator. The results demonstrated the importance of energy-efficient software development in the context of the increasing demand for energy and the growing use of programming.

Pinto et al [3] looked into potential opportunities and limitations of code refactoring to improve the energy efficiency of a software system, focusing mainly on mobile applications. Based on a vast literature study, they found that reduction of I/O operations, making API calls lightweight, reduction of background running features and using parallelism are some of the techniques which can help make an application greener. Limitations or challenges associated with implementing these changes were also discussed, addressing the technological dependencies, corner cases in the identification of inefficient code and accurately refactoring the code to not affect the operation of the application. The paper did not include a working model.

The empirical study in [8] investigated the impacts of applying refactorings on energy usage. 197 instances of 6 commonly used class-level refactorings such as – inline methods and introduction of parameter options, were studied. All of the refactorings were found to be statistically significant in terms of impact on the energy usage of an application. They also studied whether the effects of applying refactoring were consistent across applications and platforms by using different JVMs for the experiment.

In [2], two different Green Software practices were applied to two software applications, namely query optimization in MySQL Server and usage of "sleep" instruction in the Apache web server. This was followed by a comparison of the energy consumption at system-level and at resource-level, before and after applying the practice. It was found that both practices were effective in improving software energy efficiency, reducing consumption by up to 25

Most of the existing research was centred around finding inefficiencies and providing greener versions of certain coding practices. However, an implemented solution addressing and raising awareness about these practices was not found. Hence, we decided to implement a Proof of Concept of a tool which could extract occurrences of code inefficiencies discussed above and encourage users to adhere to better coding practices.

3 METHODOLOGY

This section elaborates on the identification of the four coding practices and the experiments performed to calculate their energy consumption. Furthermore, this section also introduces GreenCode, a Chrome extension to notify users about alternative coding practices when they use an energy-efficient paradigm in their code.

3.1 Scenario Comparisons

Through various blog posts and articles about green coding practices[9][10], we have identified four common code scenarios that arise during python development in which the energy consumption can be reduced by using alternative practices. These paradigms are also representative of scripts as they are used in many cases. For example, *for loops* are widely used in string and array manipulation and exponentiation operators are commonly used in mathematical scripts.

These scenarios and their alternative practices are as follows:

(1) **Using *for Loops* for List Operations** can consume higher energy than using list comprehensions. The latter offers a shorter syntax while performing sophisticated filtering, mapping, and conditional logic on existing lists
(2) **Comparing Different Data Types** can be useful for comparing different data types for various tasks such as sorting. However, it also consumes more energy.
(3) **Using the Exponentiation Operator (**)** can be necessary to work with negative and fractional exponents. But if a number simply needs to be raised to an integer power, a loop will consume much less energy.
(4) **Using addition operator to join strings** can consume significantly more energy. As an alternative, Python's inbuilt functions can be used in many instances instead of using manual operations, since they are tested and optimized for best performance. In this project, we have picked the case of joining strings together using the join() function. This function is more energy efficient, compared to using the addition operation (+) to join strings.

To substantiate these claims, we have created a notebook that uses codecarbon[11], a python library to measure the energy consumption of python code, to compare the energy consumption of the four scenarios with their alternatives. In this notebook, the original codes and their alternatives were each executed 90 million times to see the differences in their energy consumption. It was found that in all four cases, the alternative paradigms performed better. The findings are in table 1. These performance differences would make a bigger impact when employed in large-scale software projects.

3.2 GreenCode Chrome Extension

To notify users about their energy-inefficient coding practices, we have created an extension called GreenCode compatible with Jupyter Notebook running on Google Chrome. Jupyter Notebook was chosen as it is a preferred editor for data science and data analytics work in Python[12]. Google Chrome was also chosen due to its popularity. According to a 2024 survey, nearly two-thirds of the global population use Google Chrome to browse the internet[13]. GreenCode identifies parts of Python code in the Jupyter Notebook that may potentially be energy inefficient. It also informs the user about an alternative that may be used to make the code more energy-efficient. The source code can be found here: GreenCode github

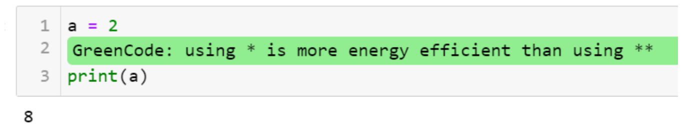
Coding Practice	**Energy Consumption (in Milliwatt-Hour)**	**Alternative Practice**	**Energy Consumption (in Milliwatt-Hour)**
Using *for loops* for lists	156	List Comprehension	87
Comparing different data types	271	Comparing same data types	59
Using exponentiation operator	378	Using multiplication operator	88
Using addition operator to join strings	327	Using inbuilt function to join strings	262

Table 1: Energy consumptions of four selected coding practices and their alternatives

GreenCode has been created using JavaScript. JavaScript was chosen as it is Google Chrome's primary scripting language. Since extensions are essentially web applications running within the browser, using JavaScript ensures compatibility with the browser's runtime environment. The JavaScript code parses through the Jupyter notebook's HTML code and identifies the four aforementioned use cases using regular expressions. Then, it replaces the code on the Jupyter Notebook with a suggestion. The user may choose to update the code according to this suggestion or ignore it altogether. In either case, the suggestion disappears if the user clicks on it. If the user wants to refer to the suggestion again, she can reload the webpage and the suggestion will be displayed again. For example, writing the following lines of code in a Jupyter notebook will prompt GreenCode to provide a suggestion:

```
a = 2
a = a**3
print(a)
```

A screenshot of this suggestion can be found in figure 1.



```
a = 2
GreenCode: using * is more energy efficient than using **
print(a)
```

```
8
```

Figure 1: GreenCode suggestion on Jupyter notebook

4 VALIDATION

In this section, we present the results of the validation of GreenCode which was performed on publicly available Python code.

For each of the four coding practices, five notebooks were selected from The Algorithms Python Github repository to test GreenCode. The Algorithms Github account is an open-source resource for learning Data Structures and Algorithms and their implementations in many programming languages. Their Python repository has 177,000 stars and contains a wide variety of commonly used algorithms in Python. Thus, this repository was selected because of its comprehensiveness and popularity. The selection of the notebooks was contingent on the number of occurrences of the coding paradigm for which they were chosen, and also the context in which the paradigm occurred. The selection was done such that GreenCode can be tested on a higher number and a wider variety of occurrences of these paradigms.

The true positives and false positives for each practice mentioned in Section 3.1 have been calculated after a manual inspection of the code. If an occurrence of a coding practice can be replaced by its alternative, which is also mentioned in Section 3.1, without disrupting the logic of the code, it has been counted as a true positive. However, if the alternative practice is not relevant to an occurrence, it has been counted as a false positive. True negatives and false negatives are zero for each case since the tool does not identify any negative case. These values have been reported in a confusion matrix for each practice.

Finally, the precision for each practice has been reported in table 6. The precision metric was chosen over metrics like recall and F1 score because only true positive and false positive values are relevant in this experiment. Precision is calculated as follows:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

	Positive	Negative
Positive	TP (13)	FP (5)
Negative	FN (0)	TN (0)

Table 2: Confusion matrix: *for loop* detection

4.1 For Loops vs List Comprehension

The notebooks alternative_string_arrange.py, credit_card_validator.py, anagrams.py, check_anagrams.py, damerau_levenshtein_distance.py, and frequency_finder.py were selected from the folder *strings* in the repository. This folder was selected because it contains a high number of occurrences of *for loops* used for string manipulation. GreenCode identified 18 occurrences of *for loops*. 13 of these *for loops* were performing operations on lists which could have been replaced by list comprehension. The other 5 occurrences could not have been replaced with list comprehension. These results can be found in table 2.

4.2 Comparing Different Data Types

The notebooks histogram_stretch.py from the folder *histogram_equalization*, bisection_2.py from the folder *numerical_analysis*, inversions.py from the folder *divide_and_conquer*, and edit_distance.py and min_distance_up_bottom.py from the folder *dynamic_programming* were selected for this case. These notebooks were selected as they contain many occurrences of various comparison operators. GreenCode detected 20 occurrences of comparison operators. Only 3 of these were comparing variables of different types. These results can be found in table 3.

	Positive	Negative
Positive	TP (3)	FP (17)
Negative	FN (0)	TN (0)

Table 3: Confusion matrix: Comparison of different data types

	Positive	Negative
Positive	TP (20)	FP (1)
Negative	FN (0)	TN (0)

Table 4: Confusion matrix: Exponentiation operator

	Positive	Negative
Positive	TP (12)	FP (14)
Negative	FN (0)	TN (0)

Table 5: Confusion matrix: joining strings using addition operator

Coding Practice	**Precision**
Using *for loops* for lists	0.72
Comparing different data types	0.15
Using exponentiation operator	0.95
Using addition operator to join strings	0.60

Table 6: Precision for each coding practice

4.3 Using the Exponentiation Operator

The notebooks area.py, area_under_curve.py, basic_maths.py, binomial_distribution.py, and dodecahedron.py were selected from the folder *maths* in the repository as they contain many occurrences of the exponentiation operator. GreenCode detected 21 occurrences of this operator. All of these could have been implemented using a multiplication operator except for one case in which the exponent was a proper fraction. These results can be found in table 4.

4.4 Using Addition Operator to Join Strings

The notebooks camel_case_to_snake_case.py, detecting_english_programmatically.py, title.py, z_function.py, wave.py from the folder *strings* in the repository were selected for this case because they contain many occurrences of using the addition operator to join strings. GreenCode detected 26 such occurrences, 12 could have been replaced by Python's inbuilt join() function. 14 occurrences were using the addition operator for purposes other than string joining. These results can be found in the table 4.

5 DISCUSSION

For the first part of our study mentioned in Section 3.1, the energy consumption of certain Python codes was measured before and after applying energy-efficient practices. The findings are visualised in the graph shown in Figure 2. For most cases, except the string join method, it was found that the initial code used more than double the energy as compared to the suggested improvement. In the case of the exponentiation operator, the difference is most significant, where multiplication is over four times more efficient. This data indicates that it is possible to achieve certain functionalities by using better practices, without compromising on efficiency.

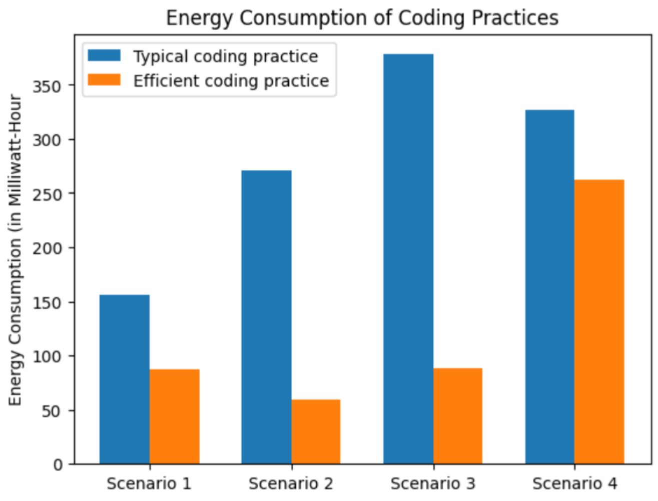


Figure 2: Comparison of Energy Consumption for Python Codes

To make developers aware of this information, the GreenCode plugin was developed for the Chrome web browser which parsed Jupyter code to identify the above scenarios and suggest improvements. The plugin ran successfully after it was added to Google Chrome. After testing it on existing Jupyter notebooks, suggestions showed up in the appropriate code blocks as designed.

Utilization of Regular Expressions (Regex) constituted a fundamental aspect of GreenCode's developmental framework, facilitating the identification of segments within the code potentially susceptible to energy inefficiencies. Emphasis was placed on the detection of all occurrences of the scenarios. This led to a trade-off with accuracy, by increasing the scope for detection of False Positives while minimising False Negatives. Consistent with the assumption, the results show that there were a considerable number of False Positives detected, but no False Negatives were detected. Hence, there were no cases of inefficient code that were not detected by GreenCode. Since the improvements are only suggested and disappear once the developer starts editing the code, there is no direct impact of choosing this approach.

The exponentiation operator scenario has the highest precision of 0.92 since Regex matching was the easiest for that case. There was only one false negative which occurred since the exponent was a proper fraction. In such cases, the multiplication operator cannot replace the exponentiation operator. On the other hand, the comparison of different data types had an extremely low precision of 0.15 due to a large number of False Positives. This is because it is not possible to find the data type of a variable without running the Python code. Hence, based on simple parsing of the code, it is not possible to identify the data type of a given variable. GreenCode was configured to show the prompt in case of all data comparison operations. The user can check to see if the data types are indeed inconsistent and may choose to ignore the prompt shown. Similarly, in the case of *for loops*, detection was done by finding all *for loops* in the code, leading to certain instances being detected where it was not possible to replace the code with list comprehension. The precision was 0.72 for this case. Similarly, for the string joins, GreenCode scanned for the + operator in the code, leading to a high number of False Positives while also detecting all the True Positives. The precision was 0.60 for this case.

Overall, GreenCode performed well with respect to the detection of energy-inefficient code within a Jupyter Notebook but was not able to distinguish the cases where code refactoring was not possible, hence giving recommendations even when they were not accurate. We discuss these limitations and the scope of improvement in the next section.

6 LIMITATIONS AND FUTURE WORK

The current version of GreenCode is only a proof of concept. There can be many improvements to the first version of this extension. The regular expressions used to identify the code on Jupyter Notebook are quite primitive.

The tool currently shows the user a suggestion for every *for loop* in the code, irrespective of whether the loop can logically be replaced by list comprehension or not. In future versions, the code can also be parsed to checklist operations occurring inside the *for loop*. Then, the tool will not unnecessarily provide suggestions to the user in cases where list comprehension is not a valid option to use.

Moreover, since it is impossible to check the data type of a variable in Python before runtime, GreenCode cannot detect whether the comparison is occurring between variables of the same or different types. Currently, it asks the user to check if a comparison operator is being used on variables of different data types. If so, it also advises the user to consider implementing her code differently.

Furthermore, GreenCode has only been implemented considering one example of Python's inbuilt functions, join(). Suggestions for using many other functions can also be integrated into the tool. For example, if a user is manually writing code which can be done more efficiently by functions like abs(), round(), len() etc., she can be advised to use the latter.

GreenCode can also be made more user-experience friendly. Currently, the suggestions disappear as soon as the user clicks on them. The user may want to refer to the suggestion again without refreshing the browser. In future versions, the suggestion can be given in a collapsible manner which would make it easier to access.

Currently, GreenCode is only limited to Jupyter notebooks running on Google Chrome. In future iterations, it can be extended to work for other browsers and development tools like IDEs etc.

Lastly, the tool is currently built for only four identified cases of energy-inefficient practices. More such cases can be explored and it can be extended to notify users about a wider variety of green coding practices.

7 CONCLUSION

In this paper, we have made two novel contributions. Firstly, we have introduced a Chrome plugin to identify energy-inefficient code and suggest actionable alternatives in Python code on Jupyter notebooks. Secondly, we have identified four common coding scenarios from online articles where energy consumption can be reduced and proposed alternative practices for each scenario. By conducting experiments and measuring energy consumption before and after applying these practices, we have demonstrated reductions in energy usage. We have also validated GreenCode's effectiveness through experiments on publicly available Python code. It showed promising results for two use cases. It was able to identify and suggest alternatives for energy-inefficient coding practices for using exponentiation operator and using *for loops* for lists with precisions of 0.95 and 0.72 respectively. The precision for the case of using the addition operator to join strings was 0.60, indicating the need for improvements. However, it did not show favourable results for the case of comparing different data types. The precision was only 0.15.

Despite some limitations, such as the tool's current focus on a limited set of coding scenarios and its compatibility with only Jupyter notebooks on Google Chrome, GreenCode presents a promising solution for promoting eco-friendly coding practices in Python development. Future work includes expanding the tool's capabilities, improving user experience, and extending its compatibility with other development environments and browsers. Through these efforts, GreenCode aims to contribute to the reduction of energy consumption in software development and promote sustainability in the computing industry.

REFERENCES

[1] Avita Katal, Susheela Dahiya, and Tanupriya Choudhury. "Energy efficiency in cloud computing data centers: a survey on software technologies". In: *Cluster Computing* 26.3 (Aug. 2022), pp. 1845–1875. ISSN: 1573-7543. DOI: 10.1007/s10586-022-03713-0. URL: http://dx.doi.org/10.1007/s10586-022-03713-0.

[2] Giuseppe Procaccianti, Héctor Fernández, and Patricia Lago. "Empirical evaluation of two best practices for energy-efficient software development". In: *Journal of Systems and Software* 117 (July 2016), pp. 185–198. ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.02.035. URL: http://dx.doi.org/10.1016/j.jss.2016.02.035.

[3] Gustavo Pinto, Francisco Soares-Neto, and Fernando Castor. "Refactoring for Energy Efficiency: A Reflection on the State of the Art". In: *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software.* IEEE, May 2015. DOI: 10.1109/greens.2015.12. URL: http://dx.doi.org/10.1109/GREENS.2015.12.

[4] R Manimegalai et al. "Energy Efficient Coding Practices for Sustainable Software Development". In: *Proceedings of the First International Conference on Science, Engineering and Technology Practices for Sustainable Development, ICSETPSD 2023, 17th-18th November 2023, Coimbatore, Tamilnadu, India.* ICSETPSD. EAI, 2024. DOI: 10.4108/eai.17-11-2023.2342635. URL: http://dx.doi.org/10.4108/eai.17-11-2023.2342635.

[5] Candy Pang et al. "What Do Programmers Know about Software Energy Consumption?" In: *IEEE Software* 33.3 (2016), pp. 83–89. DOI: 10.1109/MS.2015.83.

[6] K. R. Srinath. "Python–the fastest growing programming language". In: *International Research Journal of Engineering and Technology* 4.12 (Dec. 2017). Available at: https://www.irjet.net/archives/V4/i12/IRJET-V4I1266.pdf, pp. 354–357. ISSN: 2395-0072.

[7] Nurzihan Fatema Reya et al. "GreenPy: Evaluating Application-Level Energy Efficiency in Python for Green Computing". In: *Annals of Emerging Technologies in Computing* 7.3 (July 2023), pp. 92–110. ISSN: 2516-0281. DOI: 10.33166/aetic.2023.03.005. URL: http://dx.doi.org/10.33166/AETiC.2023.03.005.

[8] Cagri Sahin, Lori Pollock, and James Clause. "How do code refactorings affect energy usage?" In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* 2014, pp. 1–10.

[9] Pronod Bharatiya. *How to use python dict() in energy efficient way - some useful tips.* Jan. 2024. URL: https://www.linkedin.com/pulse/how-use-python-dict-energy-efficient-way-some-useful-bharatiya/.

[10] Metakratos Studio. *Writing eco-friendly code in Python: Cooling the planet.* May 2023. URL: https://medium.com/metakratos-studio/writing-eco-friendly-code-in-python-cooling-the-planet-3cb8f7a464e3.

[11] *codecarbon.* accessed on 14 March 2024. URL: https://codecarbon.io/.

[12] JetBrains. *The State of Developer Ecosystem 2022 - Data Science*. https://www.jetbrains.com/lp/devecosystem-2022/data-science/. Accessed: March 28, 2024. 2022.

[13] Oberlo. *Browser Market Share Statistics*. https://www.oberlo.com/statistics/browser-market-share/. Accessed: March 28, 2024. 2024.