Computer Science and Information Systems 22(3):693–726 https://doi.org/10.2298/CSIS240228015J

Empirical Analysis of Python's Energy Impact: Evidence from Real Measurements

Elisa Jimenez, Alberto Gordillo, Coral Calero, Mª Ángeles Moraga, and Félix García

Instituto de Tecnologías y Sistemas de Información Camino de Moledores, s/n, 13005, Ciudad Real (Spain)
{elisa.jimenez,alberto.gordillo,coral.calero,mariaangeles.moraga,felix.garcia}@uclm.es

Abstract. Programming languages provide the notation for writing computer programs capable of granting our devices the desired functionalities. Even though they may seem intangible, the resulting programs involve an amount of energy consumption, which has an impact on the environment. Some studies on the consumption of programming languages indicate that while being one of the most widely used languages, Python is also one of the most demanding in terms of energy consumption. To provide developers using Python with a set of best practices on how to use it in the most energy-efficient way, this paper presents a study on whether the different ways of programming in Python have an impact on the energy consumption of the resulting programs. We have studied the relationship between Python's energy consumption and the fact that Python is a very versatile language that allows programs to be compiled and executed in many different ways. From the results obtained in our study, there seems to be a clear relationship between software energy consumption at runtime and: (1) the use of interpreted or compiled Python; (2) the use of dynamically or statically typed variables. Compiling Python code is a good option if it is done using the py_compile module. The use of interpreted code seems to decrease energy consumption over compiling using Nuitka. The use of dynamically typed variables seems to decrease considerably the graphics and processor energy consumption. In addition, we have observed that energy consumption is not always directly related to execution time. Sometimes, more power in less time increases consumption, due to the power required.

Keywords: python, efficiency, programming languages, green software

1. Introduction

In addition to serve as a means of communication and entertainment, technological devices have also become a great working tool, being almost impossible to imagine life without them. The functionalities of technological devices are enabled by software applications, which are often developed using multiple programming languages, although one language typically predominates.

In general terms, a programming language provides a structured way to express algorithms and instructions to create a software program capable of executing one or more functionalities. Obviously, it is possible to implement the same functionality using different programming languages.

In fact, the big amount of available programming languages means that developers should choose one of them to implement the software programs. As [46] points out, choosing the appropriate programming language can make or break a project.

There are several criteria that could be used to do this selection. For example [46] mentions programmer productivity, maintainability, efficiency, portability, tool support, and software and hardware interfaces as key factors, but also indicates that, depending on the type of code to develop, there is little room for choice. In [5] the authors propose a model to select the best programming language to be learned by novice programmers for Data Analytics Applications based on eight criteria: popularity, data analytics support, volume of data it can handle, speed of compiling, expressiveness, dreadfulness, programmers' recommendations and average reasonable financial cost. [32] present the results of a quantitative study about the language adoption process, identifying the availability of open-source libraries, existing code, and experience as the most influential factors. In contrast, intrinsic factors, such as a language's simplicity or safety, rank low.

These are just some examples but, as can be observed, there is no a standard set of aspects to be considered as important when choosing a programming language. Because, as remarked by [34], the nature of languages as a special software tool makes it difficult to find measures to draw objective conclusions about them.
Besides this lack of consensus on which are the best criteria to choose a programming language, it can be noted that energy efficiency does not appear as a key aspect to be considered.

However, nowadays, the energy consumption of IT (Information Technologies), including software, is becoming a concern. According to the BEREC report [12], some 2 to 4% of greenhouse gases currently come from the digital industry. Moreover, new programming trends, such as big data or Artificial Intelligence, could further increase these figures.

As remarked by [13] the existing studies that analyze the impact of choice of programming language suffer from several deficiencies with respect to methodology and the applications they consider. In [34] is indicated that most of the claims about programming languages are based on personal affinity, being the empirical comparison a good approach to provide objective information about languages. Looking into programming languages from an empirical perspective would provide supportive evidence and valuable conclusions about them.

The objective of this paper is to conduct an empirical study, which can be considered a benchmark study according to the Empirical Standard of ACM [42] (hereafter referred to as a study), to analyse the energy consumption of a programming language. In particular, the study focuses on Python's energy consumption. Python is a highly versatile programming language and is one of the most widely used languages in the current era. According to the PYPL (Popularity of Programming Language Index), [15], created by analysing how often language tutorials are searched on Google, the most popular programming language in 2023 compared to a year ago is Python, with a 27.27% increase, followed by Java, with a 16.35% increase (results of May 2023). Moreover, it is indicated that Python grew the most in the last 5 years (3.5%) [15]. Also the last updates of other rankings such as the IEEE Spectrum's Ranking of the Top Programming Languages 2022 [16] (a ranking based on nine metrics to know what languages the public is programming in) or the Tiobe Index [4] (which takes data from hundreds of different sources, compiles it, and dumps it into a list), rank Python at the top of the list.

However, according to [36] Python is one of the most energy-demanding programming languages. An analysis of the possible causes of the high consumption of Python

leads to the conclusion that it could be due to the fact that: (1) as indicated in [45], Python is a dynamically typed language or (2) the fact that Python code is typically executed without prior compilation. Therefore, we decided to study the impact of these two aspects on the energy consumption of Python to offer developers the most efficient way to use this language.

The remainder of this document is organised as follows: Section 2 presents the related work and the issues motivating this paper. Section 3 describes the background of this study, including some relevant aspects about the execution of programs written in Python language and the applied methodology to carry out the study. Section 4 contains the analysis and discussion of the results obtained, in which we analyse the difference in consumption between the different test cases to answer the research questions in Section 5. In Section 6, we present the threats to the validity of our study. Finally, Section 7 presents the conclusions of the study and gives some recommendations on the use of Python.

2. Related work

Programming languages have been studied and analysed from several points of view. For example, there are many studies that relate programming languages to the improvement of the developers' productivity from different points of view.

In [38] an experiment was conducted to compare programmer productivity and defect rates for Java and C++. They concluded that a typical C++ program had two to three times more bugs per line of code than a typical Java program. C++ also generated between 15 per cent and 50 per cent more defects per line, and perhaps took six times longer to debug. When comparing defects against development time, Java and C++ showed no difference, but C++ had two to three times more bugs per hour.

In [18] authors tried to test the Brooks assumption that annual lines-of-code programmer productivity is constant, independent of programming language used. They analysed 10 of the most popular programming languages in use in the open-source community, concluding that the programming language is a significant factor in determining the rate at which source code is written.

In an effort to study the effects of programming language fragmentation on productivity—and ultimately on a developer's problem-solving abilities—in [25] the authors presented a metric, namely language entropy, for characterising the distribution of a developer's programming efforts across multiple programming languages. They concluded that changes in language fragmentation affect a programmer working within a single paradigm less than a programmer working with multiple paradigms.

The objective of [26] is to identify how different programming languages may affect software development productivity. Each programming language has its own productivity level. The productivity of new development projects seems to be influenced by the programming language used, while the productivity of enhancement projects seems to be much less dependent on their specific programming language.

In [13] the authors propose a novel methodology which controls the development process and developer competence and quantifies how the choice of programming language impacts software quality and developer productivity. After conducting a study and statistical analysis on a set of long-running open-source projects written mainly in C and

C++ (Firefox, Blender, VLC, and MySQL), they found that the use of C++ instead of C improves software quality and reduces maintenance effort.

Finally, [34] presents a study to investigate the impact of high-level, general-purpose, programming languages on software development productivity and quality. The authors analyse 11 primary languages: JavaScript, Java, Python, Go, Objective-C, Swift, PHP, Ruby, C#, C++, and C. The conclusion of the study is that the choice of programming language can affect the development process.

Focusing on the energy consumption of programming languages, [6], studied quantitatively the impact of languages (C/C++/Java/Python), compiler optimization (GNU C/C++ compiler with O1, O2, and O3 flags) and implementation choices (e.g. using malloc instead of new to create dynamic arrays and using vector vs. array for Quicksort) on the energy-efficiency of three well-known programs: Fast Fourier Transform, Linked List Insertion/Deletion and Quicksort. Experiments showed that by carefully selecting an appropriate language, optimisation flag and data structure, a significant amount of energy can be conserved to solve the same problem with identical input size.

In [7] three metrics are proposed to categorize software implementation and optimization efficiency: Greenup, Powerup, and Speedup metrics (GPS-UP). GPS-UP metrics transform the performance, power, and energy of a program into a point on the GPS-UP software energy efficiency quadrant graph. In addition, eight categories of possible software optimisation scenarios (four energy-saving and four energy-wasting) are presented with examples on how to obtain them and the new metrics are compared with existing metrics such as the Energy Delay Product (EDP).

Connolly Bree and Ó Cinnéide [17] conducted an assessment on the impact of two popular design-level refactoring on energy consumption in the Java programming language. Specifically, they focused on the refactoring techniques of replacing Inheritance with Delegation and vice versa. The researchers assessed the energy consumption by running code snippets for both refactoring and measuring average power consumption and energy consumption. The study revealed that Inheritance proved to be more efficient than Delegation. It exhibited a 77% reduction in runtime and a 4% decrease in average power consumption when compared to Delegation. However, a significant limitation of the study was the experiments were conducted in an Interpreted mode, which does not accurately reflect real-life scenarios where Just-in-Time (JIT) enabled compilers are commonly utilized.

Pinto et al. [39] explore the energy efficiency of several Java Collection implementations, beyond their well-established characteristics in terms of performance, scalability, and thread-safety. The study involves 16 collection implementations (13 thread-safe, 3 non-thread-safe) categorized into lists, sets, and mappings. The research reveals that design decisions significantly influence energy consumption. Notably, adopting a newer hashtable version can result in a 2.19x energy savings in micro-benchmarks and up to 17% in real-world benchmarks compared to older associative implementations.

Also Pereira et al. [37] study the energy efficiency of Java Collections. They propose an approach to energy-aware development that combines application-independent energy profiling of Java Collections and static analysis to estimate the system's utilization of these collections and its intensity. The results indicate that some widely used collections, e.g. ArrayList, HashMap and Hashtable, are not energy efficient and should sometimes be avoided when energy consumption is a major concern.

Calero et al. [14] focus their efforts on investigating the suitability in the development of software applications in terms of energy consumption of Spring (a framework for the development of Java applications), and its conclusions point out that code developed using Spring require much more energy than those developed without Spring.

Finally, Lima et al. [27] investigated the energy behavior of programs written in Haskell. They conducted two in-depth and complementary studies to analyze the energy efficiency of programs from two different perspectives: strictness and concurrency. They found that making small changes can make a big difference. In one benchmark, under a specific configuration, choosing the use of the MVar (Mutable Variable) data sharing primitive instead of the TMVar (Transactional Mutable Variable), can result in up to 60% energy savings. In another benchmark, using TMVar instead of MVar can yield up to 30% energy savings.

To provide information about the differences in energy consumption of several programming languages Pereira et al. [36] conducted an investigation in which they analysed the energy behavior of twenty-seven programming languages, estimating and comparing the consumption required for the execution of ten different programs written in all of the selected programming languages. The 27 programming languages included compiled, interpreted, and virtual machine languages. As a result of the study, it was found that compiled languages tend to be, as expected, the fastest and most energy efficient. On average, virtual machine languages required almost five times more energy to execute the solutions than compiled languages. On the other hand, interpreted languages required almost 20 times more energy than compiled languages.

Some works specifically address the energy efficiency of the Python programming language. However, the existing literature mostly focuses on aspects such as performance, complexity, and optimisations, largely neglecting the crucial aspect of energy consumption. For example, in the work conducted by Redondo and Ortin [43] a meticulous evaluation of seven implementations of Python versions 2 and 3 is presented. Their aim is to assist in the selection of a suitable implementation by running 523 programs to each version. The evaluation encompasses runtime performance, memory consumption, and an exploration of significant qualitative characteristics inherent in each implementation. One of their main conclusions is that interpreter-based implementations (such as CPython) are the most energy-efficient, followed by statically compiled implementations of Python. In contrast, JIT-compiled approaches are found to be the least energy-efficient.

So far, work in the Python domain has focused predominantly on aspects such as performance, complexity, and optimisation, while energy efficiency has been neglected. The study developed by Reya [44] explores the energy efficiency of some coding patterns and techniques in Python, with the goal of guiding programmers towards more informed and energy-conscious coding practices. The research analyzes the energy consumption of a wide range of topics, such as data initialization, access patterns, structures, string formatting, sorting algorithms, dynamic programming, and performance comparisons between NumPy and Pandas, and personal computers versus cloud computing. The comparisons they present are very interesting and can offer programmers good practice in the use of Python.

Table 1 summarises the related work together with the method used to perform the energy consumption measurements.

Table 1. Summary of related works

Reference	Research goal and scope	Measurement method
[6]	Energy impact of the languages C, C++, Java, and Python based on the different implementations and compiler optimizations.	Software estimation. Intel Power Governor library (based on RAPL, Intel's Runtime Average Power Limit) estimates the energy consumption of CPU and DRAM power when implementing a few algorithms.
[14]	Execution time and energy consumption required by three applications, developed with and without Spring.	Hardware measurement. FEETINGS (Framework for Energy Efficiency Testing to Improvement eNviromental Goals of the Software), a specific framework for measuring software energy consumption, is used together with the EET (Energy Efficiency Tester) hardware measuring instrument.
[17]	How redundancy in an object-oriented design can contribute to unnecessary energy consumption and determine how software refactoring can eliminate this redundancy.	Hardware measurement. A Watts Up Pro power meter was used to record power consumption every second.
[27]	Energy behavior of programs written in Haskell. To do so, they make changes to benchmarks such as MVar and TMVar.	Software estimation. RAPL is used to collect processor energy information and extend two existing Haskell performance analysis tools (Criterion and GHC Profiler).
[36]	Power consumption of several programming languages of different types (interpreted, VM...) by using a set of benchmarks of different functionalities belonging to CBLG.	Software estimation. Intel's RAPL is used, which collects and analyzes and analyzes the resulting data on execution time execution time and memory usage.
[37]	Energy consumption of different Java Collection Framework (JFC) implementations. With the data obtained, they present an energy optimization approach for Java programs based on the calls to JFC methods in the source code of a program.	Software Estimation. To record CPU power consumption measurements, jRAPL

[39]	Energy efficiency of 16 implementations of Java collections grouped into 3 types (lists, sets, and allocations) to demonstrate that design decisions can greatly affect energy consumption.	Hardware measurement. The first type of architecture is measured using current meters across power supply lines to the CPU module. Software estimation. The second type of architecture energy values were estimated with jRAPL (framework for profiling Java programs using RAPL).
[43]	Runtime performance and memory consumption of seven language implementations of Python versions 2 and 3 to facilitate the selection of one of them.	Not specified. The methodology "Statistically Rigorous Java Performance Evaluation" is used to statistically analyze start-up and steady-state performance data. The work does not specify how the times are obtained.
[44]	Energy efficiency of various coding patterns and techniques in Python, with the objective of guiding programmers to a more informed and energy-conscious coding practices.	Software estimation. Intel's Power Gadget is used, which is a software-based tool for tracking power usage that is compatible with Intel Core i5 processors.

Two thirds of the studies use software estimation methods and therefore relatively few studies obtain more realistic measures of consumption using hardware devices. In this study we also try to contribute in this direction by providing real consumption measurements.

3. Background

This section provides an overview of the methods for executing Python programs and the methodology used in this study.

3.1. Python Execution Methods

Python generally uses dynamically typed variables, meaning that a variable can change its type during the lifetime of a program [33]. Previous studies have highlighted that this dynamic typing can impact performance, as the lack of compile-time type information reduces opportunities for compiler optimizations, and additional type checking at runtime can incur performance costs [43].

In addition to this inherent characteristic of Python, there are several methods for executing Python programs, each with its own characteristics and advantages in terms of performance and energy efficiency. These methods are described below.

3.1.1 Interpreted code Programs written in Python are generally interpreted by a specific implementation of the language, such as CPython, Jython, or IronPython [3]. In the case of CPython, the Python source code is first compiled to an intermediate format called

bytecode, which is closer to machine language but still independent of the processor architecture. This bytecode is then interpreted by the Python virtual machine.

The py_compile module [2] in CPython is used to precompile Python source files (.py) into bytecode files (.pyc). This precompilation process converts the source code into an intermediate bytecode form before execution, which helps reduce the time required to start the program. However, even though the bytecode is precompiled, its execution is still carried out by the CPython virtual machine, which interprets the bytecode at runtime. This distinction highlights that the precompilation occurs before the program's execution (at compile time), unlike Just-In-Time (JIT) compilation, which compiles code during program execution.

3.1.2 Just-In-Time (JIT) Compilation Just-In-Time (JIT) compilation [11] is a technique used to improve performance during the execution of interpreted programs. Instead of interpreting the bytecode every time it runs, JIT compiles parts of the bytecode into native machine code at runtime, which can result in faster execution.

GraalPy [9] is an implementation of Python on the GraalVM platform, which provides high-performance execution through JIT compilation. GraalPy utilizes the advanced JIT capabilities of GraalVM to dynamically compile Python code to native machine code during execution, resulting in significant performance improvements. GraalVM applies aggressive optimizations during JIT compilation, making GraalPy a powerful tool for executing Python code efficiently. Therefore, GraalPy combines features of an interpreter and a JIT compiler. It interprets and executes Python code in a manner similar to a traditional interpreter, but also performs JIT compilation to improve performance during program execution.

3.1.3 Ahead-Of-Time (AOT) Compilation Ahead-Of-Time (AOT) [10] compilation is a method where the source code is compiled directly into native machine code before execution. This process is completed during the build time, resulting in an executable that does not require further compilation or interpretation at runtime.

Nuitka [1] is a tool that compiles Python programs to C and then uses a C compiler to generate native machine code. This process is a form of Ahead-Of-Time (AOT) compilation, as it converts Python source code into an executable that can be run directly by the operating system without requiring Python to be installed. Nuitka eliminates the need to interpret bytecode at runtime and can offer significant advantages in terms of execution speed and energy efficiency. While the generated executables may still depend on certain Python libraries at runtime, Nuitka's approach aligns closely with the principles of AOT compilation, producing efficient and standalone executables.

GraalPy [9], is primarily considered a Just-In-Time (JIT) compiler but also supports AOT compilation. This feature allows GraalPy to compile Python scripts into native binaries, leveraging the performance optimizations provided by GraalVM. GraalPy's ability to perform AOT compilation can be utilized through the creation of native images [8], which are standalone executables generated by the native-image tool.

Fig. 1 also represents the cycle of a program written in Python, from source to machine code.

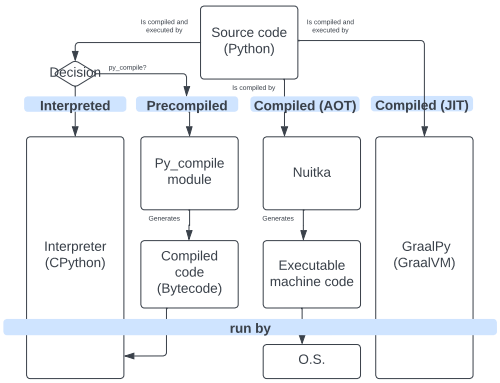

Fig. 1. Python code cycle

Although GraalPy presents potential benefits, during our study, we encountered multiple errors while executing several codes using the GraalPy compiler, likely due to the fact that GraalPy is still under development. As a result, it was not feasible to include it in our evaluation.

3.2. Green Software Measurement Process (GSMP)

As indicated by [50] Software Engineering requires a specific process for conducting experiments, as other sciences and engineering disciplines. For this reason, we have applied FEETINGS in this study, which has a methodological component, named GSMP (Green Software Measurement Process) [30] including all the activities and roles necessary to perform the measurement and analysis of the energy consumption of software, ensuring the reliability and consistency of the measurements. GSMP [29] is composed of seven phases (see Fig. 2). In a nutshell, the initial phase focuses primarily on the definition of the requirements and the software system to be evaluated. The next two phases focus on the configuration and preparation of the measurement environment. In phase four, energy consumption measurement activities are carried out. Finally, the last phases are the analysis and reporting of the data obtained. GSMP is intended to be performed in an iterative way, so the phases are interrelated to each other.

In addition to the methodological component, FEETINGS [30] has two other components: a conceptual component (GSMO-Green Software Measurement Ontology, with the terminology related to the measurement of software energy consumption) and a technological component composed by EET (Energy Efficiency Tester) [28] and ELLIOT [19]. EET is a hardware device built to capture the energy consumption of the hard disk, graphic card, and processor, as well as the overall energy consumption of the computer (namely DUT-Device Under Test) when running software. The data captured by EET are analysed by the ELLIOT tool.

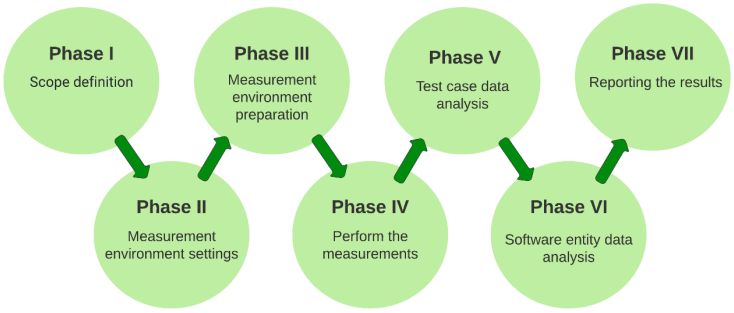

Fig. 2. GSMP phases for evaluating the energy efficiency of a software

4. Python energy consumption study

With the above considerations in mind, our research aims to quantify the energy savings achieved by statically declaring variable types. In addition, we extend our research to different Python execution methods to study the impact on consumption as well. With the results obtained, we intend to offer a set of recommendations to Python programmers on how to make better use of this programming language from an energy efficiency point of view.
Table 2 shows the research questions together with their motivation.

Table 2. Research Questions and Their Motivation

Research question	**Motivation**
RQ1. Is there any relationship between the energy consumption (by a software application at runtime) and the way it is executed?	With this research question, we want to check the difference in energy consumption between the different ways of executing programs written in Python (interpreted or compiled), in order to offer programmers some recommendations on the most efficient way to use this language.
RQ2. Is there any relationship between the energy consumption and time required (by a software application at runtime) and the way the variables are typed in the language used?	The most common use of the Python language involves the use of dynamic variables. Therefore, with this question, we want to determine whether statically declaring variables could improve the energy consumption of the language.

4.1. Application of GSMP to this study

Table 3 summarises the application of the GSMP phases and activities to our study.

Table 3. GSMP phases summary

Phase	**Application**
I. Scope Definition	– **Software Entity Class:** Python programming language – **Software Entities (SE):** Interpreted & Dinamically Typed Variables (DTV), Interpreted & Statically Typed Variables (STV), Py_compile & DTV, Py_compile & STV, Nuitka & DTV and Nuitka & STV. – **Test cases:** Ten algorithms of the Computer Language Benchmarks Game (CLBG): Binary trees, Fannkuch-redux, Fasta, Mandelbrot, K-nucleotide, N-body, Pi-digits, Reverse-complement, Regex-redux and Spectral-norm. – **Run test cases:** Each algorithm in the different ways of executing Python language.
II. Measurement Environment Setting	– **Hardware measuring instrument:** EET (Energy Efficiency Tester) – **Device Under Test (DUT):** Monitor: Philips 170s6fs LCD Motherboard: ASUS Prime B460-Plus Processor: Intel i7 10700 2900MHz RAM: 2 modules of 16GB Kingston Hiperx Fury DDR4 Graphics card: Sapphire ATI Radeon X1950 GT, 256mb RAM DDR3 Hard disk: Western Digital Blue 500GB SSD Power supply: 360 PS5805 – 580W O.S.: Gnu/Linux Ubuntu 20.4 LTS. – **Measures:** Execution time. DUT Energy Consumption. Processor Energy Consumption. Graphics Card Energy Consumption.
III. Measurement Environment Preparation	– **Before starting the measurements:** Install the Nuitka compiler for Python 3.11. – **For each Python execution way under study:** Clean the DUT, Check that there is not any software running in the background.

IV. Performing the measurement	– **For Python interpreted:** Execute algorithms using CPython interpreter. – **For py_compile:** Compile with py_compile. Execute the compiled algorithms using CPython interpreter. Delete _pycache_ folder between measurements. – **For Nuitka:** Compile with Nuitka and execute the algorithms.
Phases III and IV are repeated for each algorithm.	
V. Test Case Data Analysis	Analyse the energy consumption data for each test case. Check that the measurements are correct (outliers, wrong executions and so on) and eliminate the wrong measures if it is necessary.
VI. Software Entity Data Analysis	– **For each SE:** Calculate the mean of the energy consumption for each algorithm and for each component (DUT, processor, and graphic card). Calculate the mean of the energy consumption for each SE considering the mean of energy consumption of all the algorithms. State conclusions (see “Results” section).
VII. Reporting the result	This paper.

In the first phase of GSMP the scope of the study must be defined (see Fig. 2). As we mentioned previously, the purpose of this study was to examine programs written in Python to determine whether different execution forms and programming approaches (dynamically typed variables (DTV) or statically typed variables (STV)) have an impact on the energy consumption required to run the resulting application.

As modes of execution, we have selected CPython, the py_compile module and the Nuitka compiler for several reasons that contribute to the richness and representativeness of our study, which are described below:

- CPython, the reference implementation of Python [23], was chosen because of its wide adoption and widespread use in the Python development community. As the standard implementation, CPython provides a representative perspective of how Python programs run in most production environments.
- The py_compile module is also an essential tool in the Python development environment, as it is used to speed up the execution of programs. The inclusion of py_compile

in our measurement allows us to explore the energy impact associated with the compilation phase.
- The inclusion of the Nuitka compiler adds an interesting dimension to our research as it directly converts Python code to machine code through Ahead-Of-Time (AOT) compilation, thus avoiding the use of CPython for execution. This enables us to evaluate the energy efficiency and performance benefits of generating native executables.

As mentioned in the previous section, it was impossible to include the GraalPy compiler in our study. However, measurements have been performed with GraalPy and the results are available in the study repository at [22].

By comparing the energy consumption between CPython, py_compile and Nuitka, we can assess how choices in execution tools and technologies impact power consumption, thus providing valuable information for developers looking to optimise their processes. The aspects under study according to the research questions, result on eight different combinations (SEs), shown in Table 4

Table 4. Combinations to be studied and compared respect to their energy consumption

	DTV	**STV**
Interpreted (CPython)	SE1	SE4
Pre-compiled (Py-compile)	SE2	SE5
Compiled (Nuitka)	SE3	SE6

SE1 is the 'usual' way to use Python. The CPython interpreter compiles Python code to bytecode before executing it, but it does not use Just-In-Time (JIT) compilation. CPython follows a traditional rendering approach, so it did not require any additional action beyond executing the code. Similarly, SE5 did not require any additional action to execute, but it was necessary to adapt the algorithms to declare variables statically.

Similarly, SE4 did not require any additional action for its execution, but it was necessary to adapt the algorithms to declare the variables statically. For SE2 and SE5, each algorithm was pre-compiled using the py_compile command and then the resulting file was executed using the CPython interpreter. The bytecode code is stored in a folder called 'pycache', which was deleted between measurements so as not to affect the results of the experiment.

For SE3 and SE6, each of the algorithms was compiled using Nuitka [1] and then run without using CPython.

Ten algorithms written in Python (Binary-trees, Fannkuch, Fasta, Mandelbrot, K-nucleotide, N-body, Pidigits, Reverse-complement, Regex-redux and Spectral-norm) selected from "The Computer Language Benchmarks Game" [20] were used to measure energy consumption. The CLBG Initiative has developed a framework for testing and comparison of multiple programming languages using a collection of general programming problems. Although there is a specific tool for performing experiments in Python called "The Python Performance Benchmark Suite" [47], we have considered performing our experiments using CLBG because it was the one used by [36] where Python resulted

as the most energy consumer.

Table 5 shows the selected algorithms together with their description and the size of the input used for their execution.

Table 5. Algorithm for test cases

Algorithm	Description	Data size
Binary-trees	Allocate and deallocate many binary trees.	21
Fannkuch-redux	Indexed-access to tiny integer-sequence.	12
Fasta	Generate and write random DNA sequences.	25000000
K-nucleotide	Hashtable update and k-nucleotide strings.	25000000
Mandelbrot	Generates a Mandelbrot set	16000
N-body	Double-precision N-body simulation.	50000000
Pi-digits	Calculates all digits in pi till the nth position.	10000
Regex-redux	Match DNA 8-mers and substitute magic patterns.	5000000
Reverse-complement	Converts a DNA sequence into its reverse-complement.	25000000
Spectral norm	Eigenvalue using the power method.	5500

The 10 algorithms chosen will be implemented with dynamically (as usual in Python) or statically typed variables. Moreover, these implementations will be executed in an interpreted (CPython), precompiled (Py_compile) and compile way (Nuitka). The experiment will be performed on the most updated CPython version to date, 3.11 and the compilation with Nuitka will be done using the –standalone option [40]

The software entity class was defined as the Python programming language, the software entities are the already described forms of execution, the test cases were the ten algorithms selected and the run test cases is defined as the combination of each way of executing and programming Python for each algorithm.

To answer the research questions, we will measure the energy consumption for each one of the combinations (see Table 4), what means a total of 80 data sets of energy consumption (10 algorithms vs. 8 SEs).

As a result of the second phase of the GSMP process, we selected EET [28] as the measuring instrument (the technological component of FEETINGS) and defined the specification of the Device Under Test (DUT) where the test cases were executed.

From the measurements provided by EET, we would analyse the ones of the DUT (i.e. the energy consumption of the entire PC), the graphics card and the processor. The data samples obtained by EET will be analysed with ELLIOT.

In the third phase of the process, the measurement environment was prepared. The versions of Python, and Nuitka needed to carry out our study were installed, and between measurements, other specific actions were applied.

In the fourth phase, measurements are carried out in EET. Each run test case corresponds to the execution of an algorithm on a SE given. And each test case run was repeated 30 times, our decision regarding the number of measurements is based on the recommendation of authors such as [24] for evaluations of software power consumption in a controlled environment. Generally, a sample of 30 measurements is sufficient for the

analysis of each intended test case, as it tends to produce a near-normal sampling distribution.

It is worth emphasizing that EET obtains 100 samples (instantaneous power values) per second, resulting in many values per test case that must be managed and analysed by ELLIOT. As an example, and to facilitate the reader's understanding, for this study, between 6800 and 1,435,535 samples (depending on the runtime of each algorithm) of instantaneous processor power have been obtained for the run test cases. Therefore, for each measurement, the average value of the 30 runs of the algorithm is obtained and the resulting data sets are analysed using ELLIOT being then possible to interpret them according to the test cases defined, to, at the end, answer the research questions (Phases V and VI). As a final remark, the consumption data analysed are the ones obtained after subtracting the baseline, i.e., the consumption of the operating system and the hardware devices in the background. All the information and data about the study can be found at [22].

5. Results and Discussion

Table 6 shows the time (s) and energy consumption (J) results for each one of the ten algorithms for Software Entities SE1, SE2 and SE3. Table 7 shows the corresponding to SE4, SE5 and SE6 (see Table 4 for details of the SEs).

Table 6. SE1-3 time and consumption results

	SE 1		SE 2		SE 3	
Algorithm	Time (s)	DUT (J)	Time (s)	DUT (J)	Time (s)	DUT (J)
Binary-trees	13.3242	2985.7902	12.9068	2860.2744	15.8172	3357.8365
Fannkuch-redux	99.5651	19140,0541	99.1313	19806.2811	107.9207	21544.9500
Fasta	29.1941	2480.1973	29.2924	2456.5435	35.9573	3312.2393
K-nucleotide	14.0083	3230.7826	13.7028	3370.1803	17.5856	3973.8959
Mandelbrot	57.0628	11576.6236	57.0012	11201.6928	53.0170	10462.8838
N-body	182.1192	17957.4216	180.9061	17493.4335	278.6027	29050.4140
Pi-digits	310.0411	30235.2839	312.7975	27114.8860	312.6554	28845.3751
Regex-redux	4.7917	549.2286	4.7883	554.7591	4.7883	563.2351
Reverse-complement	2.0077	101.4849	2.1446	90.8980	2.1456	107.6565
Spectral-norm	48.2717	9627.9359	47.9484	9084.3719	61.8417	11750.8619

To answer the research questions proposed, the results of the possible combinations of the SE will be compared. The idea is to analyse the influence of each factor separately. Therefore, nine different comparisons will be presented, as shown in Fig. 3

To help to better interpret the comparisons, for each case, we will use the SEn with the best average results in terms of energy consumption for most of the 10 algorithms as basis and then we will calculate the percentage of increase on the energy consumption required by the other SEn of the comparison.

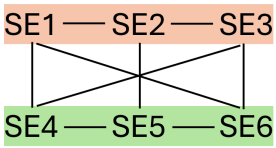

Fig. 3. Comparisons analysed

Table 7. SE4-6 time and consumption results

	SE 4		SE 5		SE 6	
Algorithm	Time (s)	DUT (J)	Time (s)	DUT (J)	Time (s)	DUT (J)
Binary-trees	13.3137	28549624	13.1434	2854.9970	15.6363	3293,1061
Fannkuch-redux	99.5693	1979.4,1939	99.7754	20306.4685	108.3222	21417,3473
Fasta	28.7633	2858.1650	29.3463	2522.0431	36.5822	3202,7499
K-nucleotide	13.8507	3309.3222	13.7778	3283.5624	17.5838	4049,7263
Mandelbrot	56.9158	11374.2202	56.9270	11493.3948	51.1548	10419,3156
N-body	182.1336	16834.5039	180.7359	15494.6057	283.2040	30017,9264
Pi-digts	310.1995	28704.6007	311.6585	27360.2596	312.8101	28983,7958
Regex-redux	4.7935	535.7790	4.7930	536.2314	4.7899	569,0573
Reverse-complement	2.1445	108.2303	2.1445	89.5619	2.1428	94,4505
Spectral-norm	50.8669	9316.0233	51.0056	9977.2440	58.9530	11701,2531

5.1. SE1, SE2 and SE3: Python Interpreted, compiled (py_compile) and Compiled (Nuitka) with DTV

This comparison aims to study the influence on consumption of the way in which the code to be executed is obtained when the variables are dynamically typed. In relation to the power consumption of the DUT, the best SE is SE2 (compiled (py_compile) and DTV), therefore, Table 8 shows the relative percentage increase in power consumption of SE1 (interpreted and DTV) and SE3 (compiled (Nuitka) and DTV) with respect to SE2.

Table 8. Time and energy consumption comparison: interpreted/compiled (Nuitka) vs. compiled (py_compile); with DTV

	SE1 vs. SE2		SE3 vs. SE2	
Algorithms	Time (%)	Energy consumption of DUT (%)	Time (%)	Energy consumption of DUT (%)
Binary-trees	3.2338	4.3882	22.5497	17.3956
Fannkuch-redux	0.4376	-3.3637	8.8665	8.7784
Fasta	-0.3357	0.9629	22.7529	34.8333
K-nucleotide	2.2295	-4.1362	28.3359	17.9135
Mandelbrot	0.1082	3.3471	-6.9897	-6.5955
N-body	0.6705	2.6524	54.0040	66.0647
Pi-digits	-0.8812	11.5081	-0.0454	6.3821
Regex-redux	0.0714	-0.9969	0.0281	1.5279
Reverse-complement	-6.3830	11.6470	0.0461	18.4365
Spectral-norm	0.6742	5.9835	28.9755	29.3525

As observed in Table 8 (left part), interpreting code with dynamically typed variables generally increases both time and energy consumption compared to compiling with the py_compile module.

From the above results, we can deduce that compiling Python using the py_compile module, with dynamically typed variables, seem to lead to a noteworthy improvement, pri-

marily in terms of energy consumption. However, compiling the code with Nuitka entails a significant increase in both runtime and energy consumption, making it an unfavourable choice.

Fig. 4 shows the average consumption of the graphics card and the processor for SE1, SE2 and SE3.

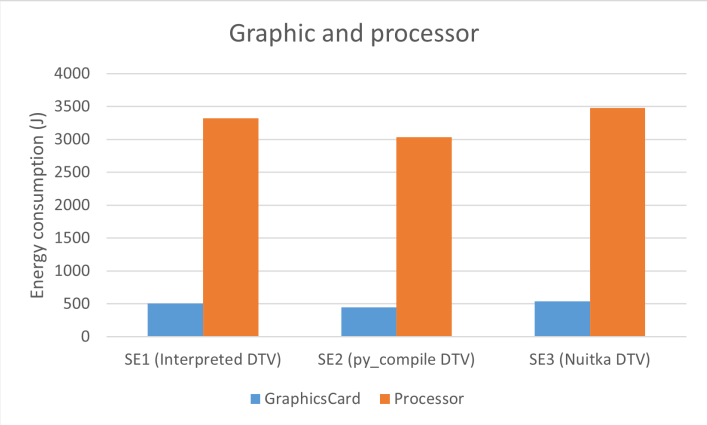

Fig. 4. Graphics and Processor analysis in DTV SEs

The consumption of the graphics card is minimal as it is hardly used in the algorithms analysed. However, both the graphics card and the processor show a slight decrease in consumption in SE2. The data for the graphics card and the processor can be found in the repository. Based on the obtained results, we can conclude that:

> When employing dynamically typed variables (DTV), the optimal choice is to compile the code using the pycompile module.

5.2. SE4, SE5 and SE6: Python Interpreted, compiled (py_compile) and Compiled (Nuitka) with STV

As in the previous section, we are going to compare again the consumption of the three possible options to obtain running software (interpreted, compiled (py_compile) and compiled (Nuitka)) but this time when variables are statically typed.

The best SE for the DUT consumption is SE5 (compiled, py_compile). Therefore, Table 9 shows the percentage of relative increase in consumption of SE4 (interpreted) and SE6 (complied, Nuitka) against SE5 consumption. In the comparison of SE4 (interpreted) and SE5 (compiled, py_compile) it can be observed that half of the algorithms obtain better results in the former and the other half in the latter.

So, it seems than in four of the algorithms, the use of compiling the code with py_compile is much more energy efficient, whereas in three of them there is not a big difference between interpreting or using py_compile. The other three algorithms have better energy

Table 9. Time and energy consumption comparison: interpreted and compiled (Nuitka) vs. compiled (py_compile); with STV

	SE4 vs. SE5		SE6 vs. SE5	
Algorithms	Time (%)	Energy consumption (%)	Time (%)	Energy consumption (%)
Binary-trees	1.2953	-0.0012	18.9672	15.3453
Fannkuch-redux	-0.2066	-2.5227	8.5661	5.4706
Fasta	-1.9867	13.3274	24.6566	26.9903
K-nucleotide	0.5296	0.7845	27.6245	23.3333
Mandelbrot	-0.0196	-1.0369	-10.1396	-9.3452
N-body	0.7733	8.6475	56.6949	93.7315
Pi-digits	-0.4681	4.9135	0.3695	5.9339
Regex-redux	0.0097	-0.0844	-0.0643	6.1216
Reverse-complement	0.0001	20.8442	-0.0786	5.4584
Spectral-norm	-0.2719	-6.6273	15.5815	17.2794

behaviour when are interpreted, but with scarce difference respect to compiling with py_compile.

In this case, there seem that there is no relationship between the energy consumption and the runtime.

Focusing now on the comparison between SE6 (compiled, nuitka) respect to SE5 (compiled, py_compile) the results show that the use of Nuitka is less energy efficient than the use of py_compile, arriving up to the 90% of increment in the N-body algorithm.

Fig. 5 shows the average consumption of the graphics card and the processor for SE4, SE5 and SE6.

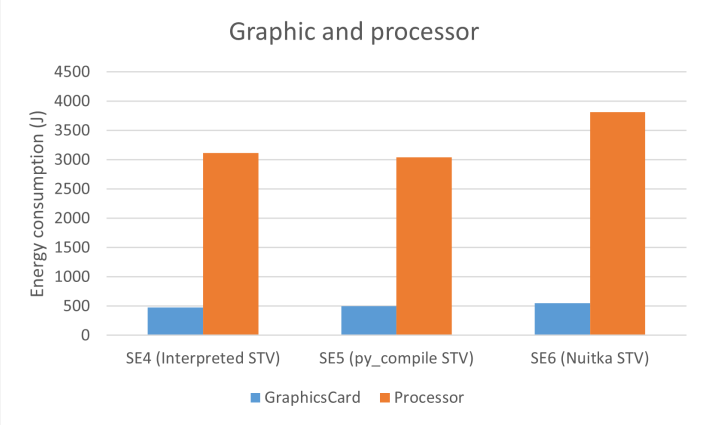

Fig. 5. Graphics and Processor analysis in STV SEs

The consumption of the graphics card is minimal, as it is hardly used in the algorithms analysed, and has no influence on the execution and development methods analysed in these three SEs. The consumption of the graphics card shows hardly any differences be-

tween the SEs. However, there is a slight decrease in processor consumption in SE5, with SE6 consuming the most. The data for the graphics card and the processor can be found in the repository.

Based on the obtained results, we observe that it is better to compile the code using the py_compile module instead of Nuitka. Although it is more difficult to draw a conclusion about the election between interpret the code or using py_compile, taking into consideration the differences in both cases and the results of the hardware components, SE5 also seems to be better option. So, we can conclude that:

> When employing statically typed variables (STV), the optimal choice is to compile the code using the py_compile module.

5.3. DTV vs. STV analysis

One of the reasons why Python might be so inefficient in terms of energy consumption is its great dynamism in typing variables. In this section we present the comparison that aims to check if there is any difference in energy consumption and runtime when using dynamically or statically typed variables, using an interpreted, compiled (py_compile) and compiled (Nuitka) versions of Python.

5.3.1 SE1 and SE4: Python Interpreted DTV and Python Interpreted STV About DUT consumption, the SE that gave us the best results in the interpreted version of Python is SE4, considering the averages of the algorithms. Therefore, Table 10 shows the percentage of relative increase in runtime and consumption of SE1 with respect to SE4.

Table 10. Time and energy consumption comparison: interpreted DTV and interpreted STV

	SE1 vs. SE4	
Algorithms	Time (%)	Energy consumption (%)
Binary-trees	0.0791	4.5825
Fannkuch-redux	-0.0042	-3.3047
Fasta	1.4977	-13.2241
K-nucleotide	1.1377	-2.3733
Mandelbrot	0.2583	1.7795
N-body	-0.0079	6.6703
Pi-digits	-0.0511	5.3325
Regex-redux	-0.0372	2.5103
Reverse-complement	-6.3787	-6.2324
Spectral norm	-5.1020	3.3481

Regarding energy consumption, most algorithms exhibit an increment in SE1 compared to SE4. However, in some algorithms (Fannkuch-redux, Fasta, K-nucleotide, and Reverse-complement), energy consumption improves in their STV version.

Looking at the runtime, there does not seem to be a relationship between run time and energy consumption.

So, we can conclude that:

> When using interpreted Python, the optimal choice is to use statically declared variables.

5.3.2 SE2 and SE5: py_compile DTV vs. py_compile STV When running the compiled code using the py_compile module, we obtained that half of the algorithms had better energy consumption for one of the SE and the other half for the other. So, to provide a simple interpretation of the results, Table 11 shows the percentage increase of the worst average (SE2) over the best average (SE5).

Table 11. Time and energy consumption comparison: compiled (py_compile) DTV and compiled (py_compile) STV

	SE2 vs. SE5	
Algorithms	Time (%)	Energy consumption (%)
Binary-trees	-1.8001	0.1848
Fannkuch-redux	-0.6456	-2.4632
Fasta	-0.1837	-2.5971
K-nucleotide	-0.5440	2.6379
Mandelbrot	0.1304	-2.5380
N-body	0.0942	12.9002
Pi-digits	0.3655	-0.8968
Regex-redux	-0.0989	3.4552
Reverse-complement	0.0047	1.4919
Spectral norm	-5.9938	-8.9491

As can be observed, there is a notable difference of energy consumption in two of the algorithms (N-body and Spectral-norm), each showing better performance in different scenarios. For the rest of the algorithms, there are no significant differences in energy consumption.

In terms of time, there does not seem to be a clear relationship between execution time and energy consumption.

So, we can conclude that:

> When using compiled code using the py_compile module, it does not seem to matter the choice between statically or dynamically declared variables.

5.3.3 SE3 and SE6: Python Compiled (Nuitka) DTV and Python Compiled (Nuitka) STV Finally, when we compile the code using Nuitka, SE3, which is the version with DTV, has yielded the best results, in contrast to the previous cases. Therefore, Table

12 presents the percentage increases of SE6 with respect to SE3.

Table 12. Time and energy consumption comparison: compiled (Nuitka) DTV and compiled (Nuitka) STV

	SE6 vs. SE3	
Algorithms	Time (%)	Energy consumption (%)
Binary-trees	-1.1437	-1.9656
Fannkuch-redux	0.3721	-0.5958
Fasta	1.7378	-3.4186
K-nucleotide	-0.0103	1.8725
Mandelbrot	-3.5124	-0.4181
N-body	1.6516	3.2231
Pi-digits	0.0495	0.4776
Regex-redux	0.0065	1.0231
Reverse-complement	-0.1293	-13.9819
Spectral-norm	-4.6711	-0.4240

In terms of algorithm runtime comparison, we find that only half of the algorithms (Fankuch-redux, Fasta, N-body, Pi-digits and Regex-redux) experience an increase. The rest of the algorithms decrease their execution time when running in SE3.

On the other hand, in terms of energy consumption, only four algorithms show an increase compared to SE3. If we make the opposite comparison (SE3 vs. SE6), most algorithms consume less in SE6.

In this case, again there does not seem to be a relationship between execution time and energy consumption.

So, we can conclude that:

> When using compiled code using Nuitka, the optimal choice is to use dinamically declared variables.

5.4. Comparison of opposites SEs

In this section we present the comparison of the most opposite versions of those included in the study. Concretely we have selected interpreted SEs and compiled by Nuitka SEs, because to run interpreted code is the opposite to run it as machine code.

5.4.1 SE1 and SE6: Python Interpreted with DTV and Python Compiled (Nuitka) with STV First, we are going to compare the most usual way of using the Python language (SE1-Interpreted and DTV) with the version compiled with Nuitka and STV (SE6). The version that has given us the best consumption results is Interpreted DTV (SE1), so Table 13 shows the percentage increase of SE6 with respect to SE1.

Looking at the consumption, 7 out of 10 algorithms increase their consumption when compiled with Nuitka and using statically declared variables. In fact, four of them (Fasta,

Table 13. Comparison of opposing SEs: Interpreted DTV and Compiled (Nuitka) STV

	SE6 vs. SE1	
Algorithms	Time (%)	Energy consumption (%)
Binary-trees	17.3530	10.2926
Fannkuch-redux	8.7954	11.8980
Fasta	25.3067	29.1329
K-nucleotide	25.5240	25.3482
Mandelbrot	-10.3536	-9.9969
N-body	55.5048	67.1617
Pi-digits	0.8931	-4.1392
Regex-redux	-0.0368	3.6103
Reverse-complement	6.7293	-6.9315
Spectral-norm	22.1275	21.5344

K-nucleotide, N-body and Sprectral.norm) increase more than a 20% the consumption. In the counterpart, the three algorithms that decrease their consumption on its version compiled with Nuitka and using statically declared variables (Mandelbrot, Pi-digits, and Reverse-complement) show a decrement less than a 10%.

Also, the execution time increases for most of the algorithms, being over the 55% for one algorithm and over the 25% in other two. In general, there seem to be a relationship between the energy consumption and the execution time.

So, we can conclude that:

> The use of interpreted code with DTV seems to decrease considerably the energy consumption and the execution time than compiled with STV code using Nuitka.

5.4.2 SE3 and SE4: Python Compiled (Nuitka) with DTV and Python Interpreted with STV In this case, we undertake a comparative analysis of two divergent configurations, with the aim of deriving further insights into the impact of using Python and the variable typing approach. Specifically, we contrast the effects of employing compiled Python (Nuitka) with DTV against interpreted Python with STV. Table 14 shows the percentage increase of SE3 with respect to SE4.

In terms of energy consumption, most of the algorithms (8 out of 10) consume more in SE3, being N-body the algorithm with the biggest increase (72%). Exceptions are Mandelbrot (which decreases an 8% the consumption in SE3) and Reverse-complement (but although being less consumer in SE3, the percentage of savings is minimal, around half point).

The execution time is, in general, proportionally related to the energy consumption, there seem to be a relationship between the energy consumption and the execution time.

Therefore, we can affirm that working with a compiled version with Nuitka that also has statically typed variables means a considerable increase in energy consumption and a longer execution time than working with its interpreted version and with dynamically typed variables.

So, we can conclude that:

Table 14. Time and energy consumption comparison: compiled (Nuitka) DTV and interpreted STV.

	SE3 vs. SE4	
Algorithms	Time (%)	Energy consumption (%)
Binary-trees	18.8047	17.6140
Fannkuch-redux	8.3875	8.8448
Fasta	25.0110	15.8869
K-nucleotide	26.9653	20.0819
Mandelbrot	-6.8502	-8.0123
N-body	52.9661	72.5647
Pi-digits	0.7917	0.4904
Regex-redux	-0.0805	5.1245
Reverse-complement	0.0507	-0.5302
Spectral-norm	21.5755	26.1360

The use of interpreted code with STV seems to decrease considerably the energy consumption and the execution time than compiled with DTV code using Nuitka.

And from analysis of sections 5.4.1 and 5.4.2. we can conclude that:

The use of interpreted code seems to decrease considerably the energy consumption and the execution time than compiled code using Nuitka.

5.5. Comparing the algorithms

Having presented the comparisons of the results of the SEs, in this section we show the energy consumption results grouped by algorithm. Fig. 6 shows an overall comparison of the DUT consumption of each algorithm in the different SEs.

As discussed in the previous sections, most algorithms show higher consumption in SE3 and SE6 (Nuitka). However, the Pi-digits algorithm shows an increase in consumption in SE1.

On the other hand, the Mandelbrot and Reverse-complement algorithms have a higher consumption in SE1 and SE4 respectively. It is also remarkable the behaviour of the Fasta algorithm, which shows a similar trend in SE1-SE5, however, in SE6 its consumption increases a lot.

Obviously, performance and resource consumption can vary depending on the specific algorithm, code characteristics, compiler optimisations and other factors. Without knowing specific details of each algorithm in question, some general reasons why algorithms perform better on some SEs than on others are as follows:

- Interpreter Optimisations: CPython, is the reference interpreter for Python, so it is highly optimised and can perform certain run-time optimisations. However, not in all algorithms, these optimisations result in lower power consumption.

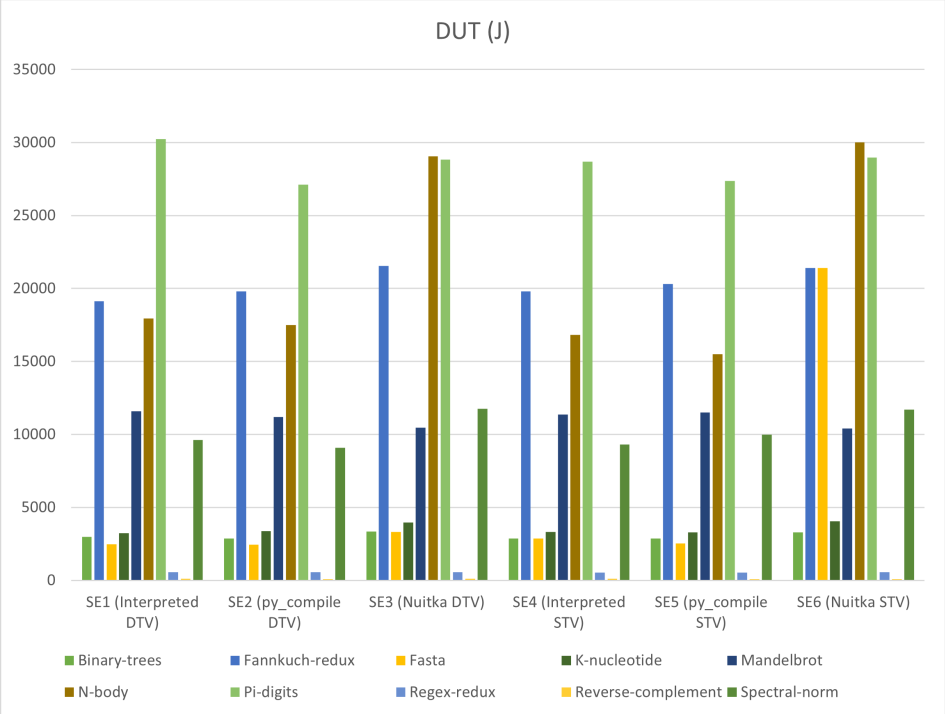

Fig. 6. Comparison of DUT consumption in the algorithms

- Code Characteristics: Code efficiency can vary between different algorithms. Some algorithms may benefit more from optimisations made by the CPython interpreter, while others may show improved performance when compiled.

Fig. 7 and Fig. 8 show the results of processor and graphics card consumption for the algorithms. Although the processor and the graphics card are two independent components, in some algorithms it can be seen how both components have a similar consumption tendency, as is the case of Fasta, N-body and Spectral-norm.

The energy consumption behavior of the algorithms is mainly due to the nature of each algorithm. However, the fact that the fasta, n-body and spectral-norm algorithms have similar resource consumption on both the processor (CPU) and graphics card (GC) may be due to several reasons related to the way the data is processed on both the CPU and the GPU of the graphics card, according to some sources as [21] and [31]:

- Nature of the Algorithm: They may have features that do not benefit significantly from the massive parallelization that a GPU could offer. Some algorithms, especially those with data dependencies or complex control flow structures, may not be as efficient on a GPU.
- Data Transfer Overhead: If algorithms involve large amounts of data transfer between the CPU and GPU, or if the data sets are small, the benefit of using a GPU may be offset by the data transfer overhead.

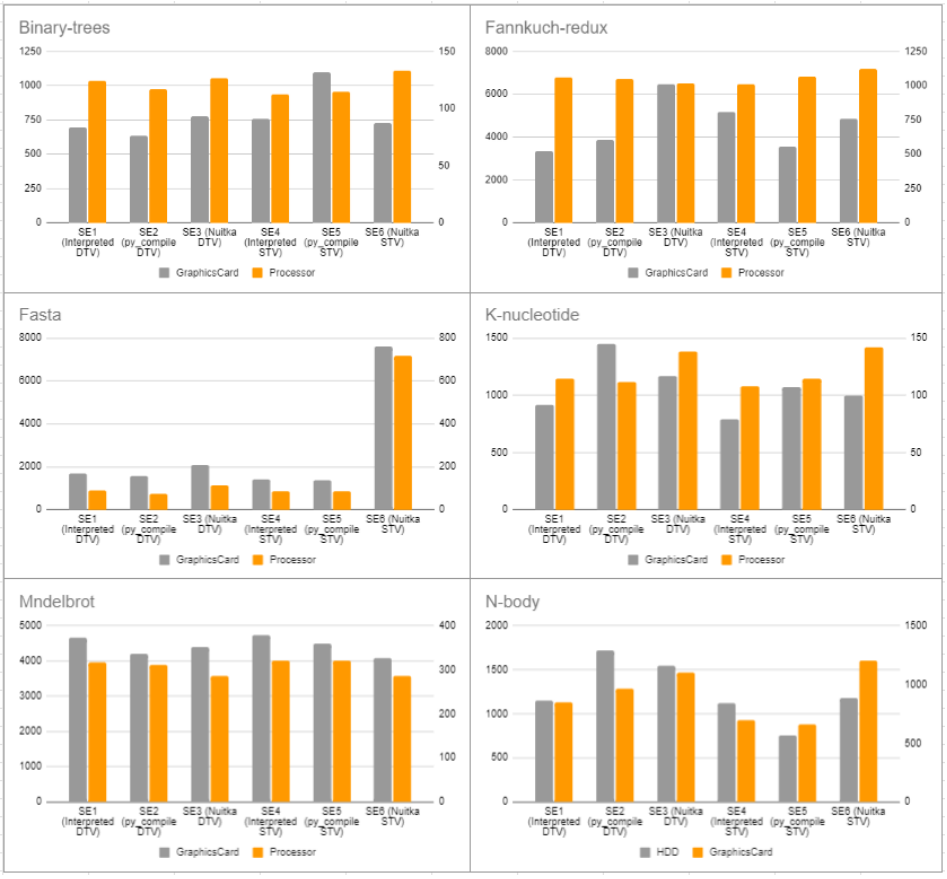

Fig. 7. Graphics and processor energy consumption results for Algorithms 1-6

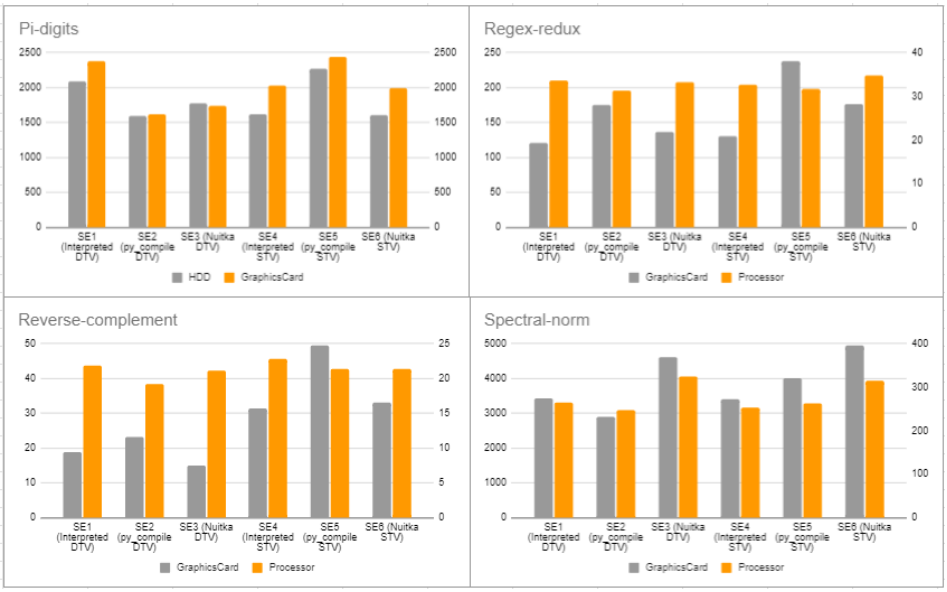

Fig. 8. Graphics and processor energy consumption results for Algorithms 7-10

- Implementation and Optimizations: GPU efficiency can depend heavily on how algorithms are implemented, and the specific optimizations made to take advantage of the GPU architecture. If the implementation has not been optimized to take advantage of GPU-specific features, performance may not differ significantly from CPU performance.
- GPU characteristics: Not all tasks are suitable for GPU execution. Some tasks, especially those involving intensive matrix operations or massively parallel computations, are more suitable for GPU execution.

On the other hand, it is noteworthy that the consumption of the graphics card and the processor in 8 of the 10 algorithms is higher in the SEn with statically typed variables. However, the algorithms are not the same in both cases. In the Pi-digits and Fasta algorithms, graphics consumption is higher with STV, but processor consumption is higher with DTV. In the Fannkuch and K-nucleotide algorithms the opposite is the case.

Regarding the way Python programs are executed, most algorithms (except Binary-trees and K-nucleotide) have a lower processor consumption in compiled SEs, either with the py_compile module or with Nuitka. Similarly, most algorithms (except for Regex-redux and Mandelbrot) also have lower graphic card consumption in compiled versions.

So, we can conclude that:

> The use of statically typed variables seems to considerably increase the power consumption of the graphics card and the processor.

And on how to run the Python code:

The use of compiled code, either with the Python module or with Nuitka, seems to decrease graphics card and processor energy consumption.

6. Answering the research questions and recommendations

Once the results obtained from the measurement have been analysed according to different comparison, we can answer the stated research questions, as follows:

RQ1. Is there any relationship between the energy consumption and time required (by a software application at runtime) and the use of Cpython, py_compile module or Nuitka?
After analysing the results obtained, we can affirm that yes, there is a significant relationship between energy consumption during software runtime and the use of interpreted or compiled Python.

The most efficient option in terms of energy consumption is to use Python compiled with the py_compile module, while the least efficient option is Python compiled with Nuitka, which shows an overwhelming increase in energy consumption. Throughout section 4.1, the increase in time and consumption can be clearly seen when comparing the use or not of py_compile, with an increase in consumption of up to 66%, while in time there are also differences, albeit smaller. Similarly, in section 4.2, very significant differences are found in all the algorithms, with the N-body algorithm standing out, whose difference when compiled with Nuitka represents a 93% increase in energy consumption and a 56% difference in execution time.

In addition, the use of interpreted code seems to significantly decrease energy consumption compared to code compiled with Nuitka. As for the way variables are declared, it does not seem to affect the power consumption of the software when compiling code using the py_compile module. However, when using code compiled with Nuitka, the best option in terms of variable declaration is to use dynamically typed variables.

RQ2. Is there any relationship between the energy consumption required (by a software application at runtime) and the use of variables dynamically or statically typed (during the development)?
After analysing the results obtained, we can conclude that it depends on the type of execution:

- When using interpreted Python, the optimal choice is to use statically declared variables.
- When using compiled code using the py_compile module, the choice between statically or dynamically declared variables does not seem to matter.
- Using code interpreted with DTV seems to considerably decrease energy consumption than compiling with STV code using Nuitka.
- Using code interpreted with STV seems to considerably decrease energy consumption over code compiled with DTV using Nuitka.

Finally, Fig. 9 ranks the software entities studied in the presented study according to the energy consumption measurements obtained in our study. Python compiled with py_compile module and declaring variables occupies the top of the list whilst the bottom is for Python compiled using Nuitka and declaring variables.

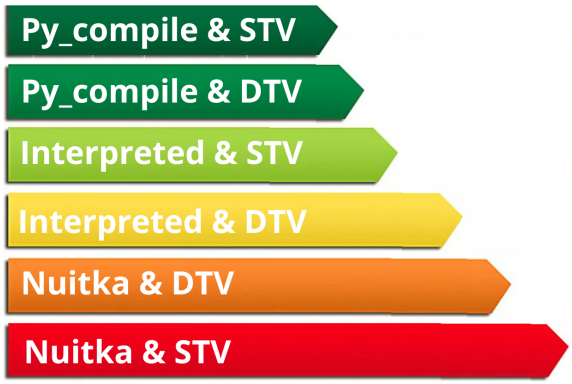

Fig. 9. Python energy-consumption classification

7. Threats to validity

This section tackles the threats to the validity of the study by following the recommendations in [49] and how we have tried to minimize their effects:

Construct validity. The first point is about the reliability of the measurements. We have used EET to measure consumption, which enables exact measurements of the energy consumed by the different hardware components in a very small interval of time (approximately 100 samples per second). Obviously, the measurements obtained are specific to EET and may differ if we use other mechanisms as an estimate, or if we employ other components (where they exist). However, EET has been already validated proving its reliability [34] and has been used previously in other measurements of this type.

Internal validity. With regards to those uncontrolled factors that may affect the results of the experiment, the most remarkable ones are related in the conditions in which the measurements were performed.

Firstly, the algorithms executed were the same in the six SEs (adapting the specific aspects on the way of using Python). Moreover, several executions were performed to mitigate the possible atypical values related with consumption. We used the same DUT to perform the executions and capture the energy consumption and measures have been taken to ensure that the DUT was always in the same conditions for the running of each different execution. To avoid the possible execution of background processes, before starting each measurement, all programs that could cause interference were closed, and the base consumption of the DUT was subtracted from the measurements. Regarding the number of measurements performed, there is no ideal number of measurements. Our decision is based on the recommendation of authors such as [24] for evaluations of software power consumption in a controlled environment. Generally, a sample of 30 measurements is sufficient for the analysis of each intended test case, as it tends to produce a near-normal sampling distribution. However, such as reported in the experimental package [22] statistical significance analysis did not obtain significant difference between most of the compared scenarios with resulting small effect sizes. New empirical studies will be conducted with more complex scenarios to confirm if the obtained scenarios differences in this study can be significant from statistical point of view in more complex settings.

External validity. Finally, related to the power of generalising the results obtained in this experiment. The results are based on a specific combination of algorithms and configuration of Python. Our experiment was performed on CPython interpreter version 3.11 and compilation with Nuitka was performed using the –standalone option.

Therefore, different interpreter versions and other compilation options could differ in the results. So, the study could be repeated using other interpreter versions, as well as extending it by considering other compilation options, other algorithms, the use of libraries, etc. A more exhaustive analysis could also be performed to report the parts of the code that involve higher consumption in the different components. Nevertheless, the current study is a good starting point on how to improve the large amount of power required by Python.

Throughout the document we have mentioned the existence of a new compiler, GraalPy, which promises to be very efficient. We have made some energy consumption measurements with the aim of including it in our study. However, its current state of development has prevented us from performing exhaustive measurements of all our algorithms due to its limitations.

Likewise, we are also aware of the existence of another compiler called Numba [35] which, due to its limitations with some libraries [41], we have not been able to include in our study either.

However, for both cases we have made some measurements. In the case of Numba, we performed the measurements on another version of the Mandelbrot algorithm compatible with this compiler. The same algorithm has also been measured on Interpreted and on Nuitka (DTV and STV) and the results obtained have been compared. In the case of GraalPy, we have measured all the algorithms. The results of the compatible algorithms together with the errors of the remaining algorithms can be found in the experimental package in our repository [22].

8. Conclusions

Sustainability has emerged as a paramount concern within contemporary society, prompting an increasing number of companies to integrate it into their product development practices. However, within the realm of software development companies, the consideration of sustainability remains an area with significant room for improvement. Despite the growing body of research addressing the sustainability of software development, substantial gaps persist. Part of the software sustainability is concerned by the obtaining of green software and the consumption required by software.

Numerous studies have contributed valuable insights into the green software development in general and in programming languages (focus of this paper) in particular. For instance, the work of [36] delves into the energy efficiency of various programming languages. Notably, the study's findings highlight Python, a language widely acclaimed by developers, as one of the most energy-intensive languages. Despite ongoing efforts to optimize Python, these endeavours have yet to yield comprehensive insights into the optimal methods for its development and execution.

Against this backdrop, the present research endeavours to gauge the real energy consumption associated with three distinct methods of executing Python programs, namely CPython, the Py_compile module, and Nuitka. Additionally, two different approaches to

Python development, involving dynamically typed variables and statically typed variables, are considered in this study.

Using the FEETINGS framework [30] and the EET [28], we measured the energy consumption of 10 algorithms written in Python. Each of these algorithms was adapted in two versions (Dynamically typed variables and statically typed variables) and executed 30 times in three different ways (Interpreted, compiled with the py_compile module and compiled with Nuitka). With the resulting consumption data, different comparisons were made to answer our research questions.

Based on the findings, a set of recommendations have been developed to make it easier for Python programmers to apply it in real life. Each of these recommendations is detailed below:

- R1. If you want to speed up loading and execution, it is better to compile using the py_compile module rather than Nuitka.
- R2. If the code is compiled using py_compile, it is better to declare the variables (STV).
- R3. When using interpreted Python is required, the best option is to use statically typed variables (STV).
- R4. When using dynamically typed variables (DTV) is required, the best option is to use compiled Python with py_compile module.
- R5. It is preferable to run the interpreted code with DTV, rather than to compile it with Nuitka.
- R6. A compiled version of Python with dynamically typed variables is better than an interpreted version with statically typed variables.
- R7. The use of statically typed variables does not always save energy, it depends on the algorithm executed.
- R8. It is better to use Python on its classical way (interpreted with DTV) than using STV and compile using Nuitka.
- R9. To reduce GC and processor consumption, it is better to use dynamically typed variables.
- R10. It is better to compile the code, either with the Python module or with Nuitka, to reduce the consumption of the graphics card and the processor.

Considering that Python is a very versatile language, several lines of future work are open. Python offers a multitude of libraries [48] that eliminate the need to write code from scratch, providing the programmer with various functionalities such as processing large amounts of data or image processing. It would therefore be interesting to check the influence of these libraries on energy consumption.

We are also aware of the existence of other types of benchmarks, for example ”The Python Performance Benchmark Suite” [47], specifically for measuring the performance of programs written in this language. It would therefore be interesting to replicate our study using this benchmark to compare the results obtained and offer more specific recommendations.

Other compilers (Numba and GraalPy) were considered to be included in our study. Due to their limitations, mainly because they are yet under construction, they were not included, but they will be considered in future works. Work is also underway to carry out the study presented in this paper in other programming languages, with the aim of providing relevant information on the influence that the use of different compilers has

on the energy consumption of the software. Finally, we also consider of main interest to study the energy efficiency of compiler optimisations, being another line of future work.

Acknowledgments. This work has been supported by the following projects: OASSIS (PID2021-122554OB-C31/ AEI/10.13039/ 501100011033/FEDER, UE); EMMA (Project SBPLY/21/180501/000115, funded by CECD (JCCM) and FEDER funds); SEEAT (PDC2022-133249-C31 funded by MCIN/AEI/ 10.13039/501100011033 and European Union NextGenerationEU/PRTR); PLAGEMIS (TED2021-129245B-C22 funded by MCIN/AEI/ 10.13039/501100011033 and European Union NextGenerationEU/PRTR). Financial support for the execution of applied research projects, within the framework of the UCLM Own Research Plan, co-financed at 85% by the European Regional Development Fund (FEDER) UNION (2022-GRIN-34110).

References

1. Nuitka (2023), https://nuitka.net/index.html
2. Py_compile (2023), https://docs.python.org/es/3/library/py_compile.html
3. Python implementations (2023), https://wiki.python.org/moin/PythonImplementations#Other_Implementations
4. TIOBE Index (Feb 2024), https://www.tiobe.com/tiobe-index/
5. Abdelnabi, A.A.B.: An analytical hierarchical process model to select programming language for novice programmers for data analytics applications. pp. 128–132. IEEE (2019)
6. Abdulsalam, S., Lakomski, D., Gu, Q., Jin, T., Zong, Z.: Program energy efficiency: The impact of language, compiler and implementation choices. pp. 1–6. IEEE (2014)
7. Abdulsalam, S., Zong, Z., Gu, Q., Qiu, M.: Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency. In: 2015 Sixth International Green and Sustainable Computing Conference (IGSC). pp. 1–8 (2015)
8. and/or its affiliates, O.: GraalPy native-image, https://www.graalvm.org/latest/reference-manual/python/native-applications/
9. and/or its affiliates, O.: GraalPy (2024), https://www.graalvm.org/latest/reference-manual/python/
10. Anaconda, I.a.o.: AOT compilation (2020), https://numba.pydata.org/numba-doc/dev/user/pycc.html
11. Aycock, J.: A brief history of just-in-time. ACM Comput. Surv. 35(2), 97–113 (Jun 2003), https://doi.org/10.1145/857076.857077, place: New York, NY, USA Publisher: Association for Computing Machinery
12. BEREC: BEREC Report on Sustainability: Assessing BEREC's contribution to limiting the impact of the digital sector on the environment. Tech. rep. (Jun 2022), https://www.berec.europa.eu/en/document-categories/berec/reports/berec-report-on-sustainability-assessing-berecs-contribution-to_-limiting-the-impact-of-the-digital-sector-on-the-environment
13. Bhattacharya, P., Neamtiu, I.: Assessing programming language impact on development and maintenance: A study on C and C++. pp. 171–180 (2011)
14. Calero, C., Polo, M., Moraga, M.Ã.: Investigating the impact on execution time and energy consumption of developing with Spring. Sustainable Computing: Informatics and Systems 32, 100603 (2021), publisher: Elsevier
15. Carbonnelle, P.: PYPL (Oct 2023), https://pypl.github.io/PYPL.html
16. Cass, S.: Top Programming Languages 2022 (Aug 2022), https://spectrum.ieee.org/top-programming-languages-2022

17. Connolly Bree, D., Ó Cinnéide, M.: Inheritance versus Delegation: which is more energy efficient? pp. 323–329 (Jun 2020)
18. Delorey, D.P., Knutson, C.D., Chun, S.: Do programming languages affect productivity? a case study using data from open source projects. pp. 8–8. IEEE (2007)
19. Gordillo, A., Mancebo, J.: ELLIOT: GESTIÓN Y ANÁLISIS DE DATOS DE CONSUMO DE SOFTWARE. In: Calidad y sostenibilidad de sistemas de información en la práctica. 1 edn. (Jan 2022)
20. Gouy, I.: The Computer Language Benchmarks Game (2008), https://benchmarksgame-team.pages.debian.net/benchmarksgame/sometimes-people-just-make-up-stuff.html
21. Grama, A.: Introduction to parallel computing. Pearson Education (2003)
22. Jimenez, E., Gordillo, A., Calero, C., Moraga, M.Ā., Garcia, F.: Repository of energy consumption of Python results, https://github.com/GrupoAlarcos/PythonEnergyConsumptionStudy
23. Kaushik, S.: Best Python Interpreters: Choose the Best in 2023 (Nov 2022), https://hackr.io/blog/python-interpreters
24. Kern, E., Hilty, L.M., Guldner, A., Maksimov, Y.V., Filler, A., Gröger, J., Naumann, S.: Sustainable software products—Towards assessment criteria for resource and energy efficiency. Future Generation Computer Systems 86, 199–210 (2018), publisher: Elsevier
25. Krein, J.L., MacLean, A.C., Knutson, C.D., Delorey, D.P., Eggett, D.L.: Impact of programming language fragmentation on developer productivity: a sourceforge empirical study. International Journal of Open Source Software and Processes (IJOSSP) 2(2), 41–61 (2010), publisher: IGI Global
26. Lavazza, L., Morasca, S., Tosi, D.: An empirical study on the effect of programming languages on productivity. pp. 1434–1439 (2016)
27. Lima, L.G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., Fernandes, J.P.: On Haskell and energy efficiency. Journal of Systems and Software 149, 554–580 (2019), https://www.sciencedirect.com/science/article/pii/S0164121218302747
28. Mancebo, J., Arriaga, H.O., García, F., Moraga, M.Ā., García-Rodríguez de Guzmán, I., Calero, C.: EET: A Device to Support the Measurement of Software Consumption. In: 2018 IEEE/ACM 6th International Workshop on Green And Sustainable Software (GREENS). pp. 16–22 (2018)
29. Mancebo, J., Calero, C., Garcia, F.: GSMP: Green Software Measurement Process. pp. 43–67 (Oct 2021)
30. Mancebo, J., Calero, C., García, F., Moraga, M.Ā., de Guzmán, I.G.R.: FEETINGS: Framework for energy efficiency testing to improve environmental goal of the software. Sustainable Computing: Informatics and Systems 30, 100558 (2021), publisher: Elsevier
31. Merrit, R.: What Is Accelerated Computing? (Sep 2021), https://blogs.nvidia.com/blog/what-is-accelerated-computing/
32. Meyerovich, L.A., Rabkin, A.S.: Empirical analysis of programming language adoption. pp. 1–18 (2013)
33. Montanaro, S.: Python dynamic language (Feb 2012), https://n9.cl/pyaxq8
34. Muna, A.: Assessing programming language impact on software development productivity based on mining oss repositories. ACM SIGSOFT Software Engineering Notes 44(1), 36–38 (2022), publisher: ACM New York, NY, USA
35. Oliphant, T.: Numba (2012), https://numba.pydata.org/
36. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: 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. pp. 256–267. SLE 2017, Association for Computing Machinery, New York, NY, USA (2017), https://doi.org/10.1145/3136014.3136031, event-place: Vancouver, BC, Canada

37. Pereira, R., Couto, M., Saraiva, J., Cunha, J., Fernandes, J.P.: The Influence of the Java Collection Framework on Overall Energy Consumption. In: Proceedings of the 5th International Workshop on Green and Sustainable Software. pp. 15–21. GREENS '16, Association for Computing Machinery, New York, NY, USA (2016), https://doi.org/10.1145/2896967.2896968, event-place: Austin, Texas
38. Phipps, G.: Comparing observed bug and productivity rates for Java and C++. Software: Practice and Experience 29(4), 345–358 (1999), publisher: Wiley Online Library
39. Pinto, G., Liu, K., Castor, F., Liu, Y.D.: A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 20–31 (2016)
40. public: Nuitka options (2023), https://github.com/Nuitka/Nuitka
41. python: Numba limitations (2020), https://numba.readthedocs.io/en/stable/reference/pysupported.html
42. Ralph, P., Nauman, A.: Empirical Standards ACM SIGSOFT (Mar 2021), https://github.com/acmsigsoft/EmpiricalStandards
43. Redondo, J.M., Ortin, F.: A Comprehensive Evaluation of Common Python Implementations. IEEE Softw. 32(4), 76–84 (Jul 2015), https://doi.org/10.1109/MS.2014.104, place: Washington, DC, USA Publisher: IEEE Computer Society Press
44. Reya, N.F., Ahmed, A., Islam, T.Z.M.M.: GreenPy: Evaluating Application-Level Energy Efficiency in Python for Green Computing. Annals of Emerging Technologies in Computing (AETiC) 7(3) (2023)
45. Shaw, A.: Why is Python so slow? (Jul 2018)
46. Spinellis, D.: Choosing a programming language. IEEE software 23(4), 62–63 (2006), publisher: IEEE
47. Stinner, V.: The Python Performance Benchmark Suite (2017), https://pyperformance.readthedocs.io/
48. Team, G.L.: Top 30 Python Libraries To Know in 2024 (Nov 2023), https://www.mygreatlearning.com/blog/open-source-python-libraries/
49. Wohlin, C., Runeson, P., Hst, M., Ohlsson, C., Regnel, Wessln, A.: Experimentation in Software Engineering. Springer (2012)
50. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)

Elisa Jimenez is currently a PhD student in Advanced Information Technologies. She graduated in Computer Engineering in 2021 and obtained a Master's degree also in Computer Engineering in 2022. She is a member of the Alarcos Research Group, University of Castilla-La Mancha (UCLM), Ciudad Real, Spain. She works on software sustainability by performing energy consumption measurements among other research. Contact her at elisa.jimenez@uclm.es

Alberto Gordillo is currently a PhD student in Advanced Information Technologies. He graduated in Computer Engineering in 2020 and obtained a master's degree also in Computer Engineering in 2022. He is a member of the Alarcos Research Group, University of Castilla-La Mancha (UCLM), Ciudad Real, Spain. He works on process automation and software sustainability by performing energy consumption measurements among other research. He holds the ITIL4 professional certification. (Information Technology Infrastructure Library). Contact him at alberto.gordillo@uclm.es

Coral Calero is currently a Full Professor with the University of Castilla-La Mancha (UCLM), Ciudad Real, Spain. She is a member of the Alarcos Research Group, being responsible of the "Green and Sustainable software" line research. She received the M.Sc. degree in 1996 from the University of Seville and the Ph.D. degree in Computer science in 2001 from the UCLM. She holds the professional certifications PMP (Project Management Professional), Scrum Master, and Scrum Manager. She is member of the Spanish Committee on Research Ethics. Contact her at Coral.Calero@uclm.es

Mª Ángeles Moraga is currently an Associate Professor and a Member of the Alarcos Research Group, University of Castilla-La Mancha (UCLM), Ciudad Real, Spain. Mª Angeles Moraga received the M.Sc. degree in 2003, and the Ph.D. degrees in computer science in 2006, from the UCLM. She works on software quality and measures, and software sustainability. She holds the following professional certifications: PMP (Project Management Professional), Scrum Master I - PSM I, and Scrum Manager. Contact her at MariaAngeles.Moraga@uclm.es

Félix García is Full Professor at the University of Castilla-La Mancha (UCLM) and a member of the Alarcos Research Group, Ciudad Real, Spain. His research interests include software sustainability, business process management, software processes, software measurement, and agile methods. He holds the following professional certifications: PMP (Project Management Professional), CISA (Certified Information Systems Auditor), Scrum Master I - PSM I, and Scrum Manager. Contact him at Felix.Garcia@uclm.es.

Received: February 28, 2024; Accepted: September 25, 2024.