# Multtestlib: A Parallel Approach to Unit Testing in Python

1st Ricardo Ribeiro de Alvarenga
*Departamento de Ciencia da Computação*
*Instituto Tecnologico de Aeronautica (ITA)*
Sao Jose dos Campos, Brasil
alvarenga.r.ricardo@gmail.com

2nd Luiz Alberto Vieira Dias
*Departamento de Ciencia da Computação*
*Instituto Tecnologico de Aeronautica (ITA)*
Sao Jose dos Campos, Brasil
vdias@ita.br

3rd Adilson Marques da Cunha
*Departamento de Ciencia da Computação*
*Instituto Tecnologico de Aeronautica (ITA)*
Sao Jose dos Campos, Brasil
cunha@ita.br

4th Lineu Fernando Stege Mialaret
*Instituto Federal de Educacao, Ciencia e Tecnologia de Sao Paulo (IFSP)*
Campus Jacarei, Brasil
lmialaret@ifsp.edu.br

***Abstract*—In this article, we propose a solution to improve the speed of Unit Tests in Python Language by the creation of a package, also in Python, to run unit tests using parallel processing. It was necessary to side-step Python's GIL (Global Interpreter Lock) which prevents parallel processing. With the reuse of code from the multprocessing package, it was possible to execute code in parallel, using subprocesses instead of the threads used by the GIL. Performance tests were carried out, comparing the developed solution with the unittest, from the standard library, such tests demonstrated that it is possible to carry out unit tests, in Python, using multiprocessors to reduce running time. The authors believe that, based on the results presented, this is a promising technique and that the developed package may eventually be included in a Python library.**

***Index Terms*—High Performance Computing, Multiprocessing, Open Source, Parallel Computing, Python, Software Engineering, Software Quality, Software Testing, Test Driven Development, Unit Test, White-Box Testing.**

## I. Introduction

This article tackles the development of a package for software testing, in the form of proof of concept, written in the computer language Python, using parallel processing.

### A. *Software Testing*

Like any other product, computer programs are subject to defects. This happens for several reasons: Problems or insufficient communication, domain knowledge, lack of tools and processes, individual error, inadequate training, wrong or ambiguous methods, lack of understanding of the problem, failure to specify requirements, among others [1]. To mitigate these risks, it is necessary to apply software testing procedures [2].

Software testing is an activity within software quality control and seeks to ensure that the product meets the specified functional and non-functional requirements [3].

Software testing is a technical and complex task that aims to run a program to find possible defects. Such task consumes between 40% to 50% of software development resources [4] [5]. The specialized literature presents a wide range of approaches ranging from white and black boxes, unit tests, acceptance tests, equivalence tests, test coverage, TDD, among others.

The goal of a software development team is to deliver products and services with a high degree of quality [3], but according to the Consortium for Information & Software Quality (CISQ) software bugs in the United States, in 2020 , cost $1.56 trillion. The publication recommends that software errors be treated and corrected before deployment, which is up to 10 times cheaper than fixing software in operation [6].

Software failures can cause direct and indirect financial losses for companies, as a result of the negative image caused by its failure, in addition to its correction cost [3].

The literature presents us with several examples of software failures, the most famous - and discussed - being the Y2K (Y2K problem, Y2K scare, millennium bug, Y2K bug) [7], a floating point problem in INTEL microprocessors in 1994 [8], the explosion of the Rocket Ariane 5 [9] and the crash of the Boeing 737 Max 8 airplanes [10] [11].

Unit tests test the smallest unit, or module, of software capable of being tested, which can be a class, a method or a function [2] [3]. Regression tests are responsible for ensuring that corrections or changes made do not change the behavior of already tested software [3]. To do this, the unit tests are recorded and run again after the changes have been made [4].

In this type of situation, re-running several tests already applied, or even all of them, can require a lot of processing time, ranging from days to weeks, depending on the size and complexity of the system [12].

### B. *Parallel Processing*

Processes are running programs and, depending on their activity, can be classified in one of the following states:

- New;
- Running;
- On hold;
- Ready;
- Closed.

Regardless of the number of processors in a computer, only one process runs at a time, per processor. A basic unit of CPU (Central Processing Unit) utilization, known as a thread, indicates how processes within the CPU are managed. If the process is monothreaded we will have only one process running at a time, only one request being served at a time. On the other hand, multithreaded processes have the ability to manage tens or hundreds of requests simultaneously, reducing user waiting time, sharing resources and information more economically, when compared to monothread processes [13].

### C. Python

Python is an interpreted, multi-paradigm, object-oriented, interactive and functional programming language developed by Guido Van Rossum between 1989-91 with the aim of replacing the ABC language and capable of handling exceptions in the operating system *Amoeba* [14] [15].

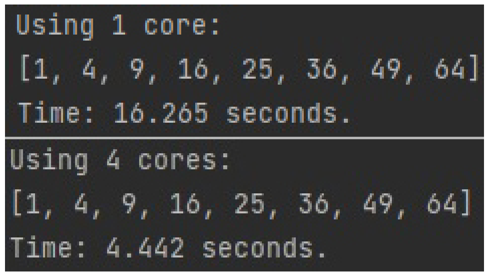
Because it is easy to learn and versatile, it has allowed developers around the world to use Python in a wide range of applications such as data science, machine learning, web development, application development, script automation, Database, Artificial Intelligence, among others. Such facts made the popularity of the language soar [16] [17] [18], until in 2022 Python became the most popular programming language, according to the Tiobe index [19].

Virtually all modern computers, regardless of their size, from handheld devices to large servers, have a multiprocessor architecture. This architectural design allows for the processing of tasks in parallel, providing greater throughput [13]. However, the Python language uses GIL (Global Interpreter Lock) which makes only one thread execute the bytecode at a time, this guarantees security against concurrent access at the cost of not allowing the language natively leverages the processing power of all cores provided by the application's host hardware [20].

One of the most important features of the Python language is the ease of code reuse, through its modularization. The modularization process starts when the programmer breaks large chunks of code into smaller pieces, which are then stored in *.py* files. These files are called modules, modules are grouped into packages and a set of packages forms a library [21] [22] [23].

The import command allows the incorporation of modules, or parts, into the program under development. PEP 8 (Python Enhancement Proposal 8), Style Guide for Python Code [24], directs that import commands be declared at the beginning of the file, in the following order:

- Standard library;
- Third party modules;
- Developer modules.



```
Using 1 core:
[1, 4, 9, 16, 25, 36, 49, 64]
Time: 16.265 seconds.
Using 4 cores:
[1, 4, 9, 16, 25, 36, 49, 64]
Time: 4.442 seconds.
```

Fig. 1. Results with 1 and 4 cores

## II. Development

### A. Specifications of the new package

The research and development of the multtestlib package used as benchmark the unittest library, Python's standard, which according to its documentation, is a test framework based on JUnit and capable of executing automated tests among other tasks [2] [25].

The tool to be developed should be faster than its benchmark while being easier to use and being able to document massive unit tests.

Based on these premises, seven basic characteristics were defined that the new solution should contain:

- Be able to perform unit testing in Python, using non-distributed parallel processing;
- Be faster than unittest;
- Easy syntax and application;
- Be flexible;
- Display the results on screen;
- Record results in files.
- Be easy to install.

### B. Multiprocessing

This stage of the project made use of the code reuse technique to be able to leverage existing Python packages. It was necessary to carry out a research work to understand the operation of parallel processing packages in Python and verify their degree of suitability for the project. The following packages were analyzed:

**Dispy:** Manages computational clusters capable of performing SMP (Symmetric Multi-Processing) calculations for clustered or cloud machines. Ideal for working with parallel data (SIMD) – when a Python function uses multiple Data Sets [26]. Due to its complex implementation (which would

```python
from multiprocessing import Process

def f(color):
    print(color, 'is a color.')

if __name__ == '__main__':
    p = Process(target=f, args=('Blue',))
    p.start()
    p.join()
```

Listing 1: Multiprocessing

not meet item 4 of the specifications), the library was not considered for the present research and development work.

**Pandarallel:** Performs parallelization of operations, using Pandas for processing massive Data Sets. It is a tool aimed at using the Pandas [27] library, which made its use in the project unfeasible.

**IPython Parallel:** It is a package for using parallel processing in Jupyter Notebook environment via command line (CLI) [28]. Using the command line in a Jupyter Notebook environment did not allow its use in this project.

**Dask:** This is a package with similar characteristics to the Pandarallel and Ipyrallel packages, for local machines with multiple cores operating in cloud clusters [29]. Its architecture did not allow its use in this research.

**Multiprocessing:** This package allows the use of processes concurrently, bypassing the limitation of Python's GIL, which makes it possible to use all the multiprocessors of the machine responsible for executing the program. It uses the process pool paradigm generating a list of tasks to be submitted to the computer's processors, as soon as there is an available processor, it receives the next task in the queue executing it. Besides this package can be coupled with other Python packages, it can be used through simple and concise code [30]. Its unique these features enabled the multiprocessing package to be used in the project under development.

Both the official multiprocessing package documentation [30] and PEP 371 [31], which includes the multiprocessing package in the Python standard library, do not detail how the multiprocessing package side-steppes the GIL limitation. This documentation just states that the package uses subprocesses instead of threads.

The listing 1 demonstrates the simplest way to use the multiprocessing package. Basically, an object *(p)* is defined to trigger a function *(f)* that takes an argument. The process is started with the command *start()*. However, in a software testing environment, where the test list can contain dozens or even hundreds of tests to run, having a *start()* command for each unit test is not productive.

Another feature available by the multiprocessing package is the possibility of creating a processing pool. In the listing 2 we have an example of executing a function, *square(n)*, which uses a pool to receive its execution parameters. In this configuration 4 processors, *Pool(4)*, were allocated for the execution of the program. We can see that the function *square(n)* executes the command *time.sleep(2)*, included only for the purpose of this research and that pauses the execution of the processing for 2 seconds. In a sequential processing environment the code is executed in approximately sixteen seconds, while in a parallel processing environment with 4 processors the task is solved in approximately 4 seconds, the figure 1 illustrates the results obtained with the two configurations of processing.

```
(...)
def square(n):
    time.sleep(2)
    return n * n

def main():
(...)
with Pool(4) as pool:
    res = pool.map(square, (1, 2, 3, 4, 5, 6, 7, 8))
(...)
```

Listing 2: Pool with timer

The next development step consisted of tuning the *pool* class to run unit test commands. The test function *test_equal(a, b)* has been developed that takes two values and checks if they are identical. An example of using *pool* can be seen in the listing 3, where *multtestlib.Pool(cores)* indicates how many processors will be activated for the execution of the test queue *pool. starmap*, which is a list object. The created queue receives the parameters for the unit test execution. In the example, the values of *a*, *b* and *c* were verified using two computer cores.

The reuse of the multprocessing package in the project was done through the import command.

## C. Faster than Unittest

The second requirement for building the multtestlib package is to be faster than its benchmark, unittest. To verify this requirement, it was necessary to carry out performance tests, comparing the performance of the two packages running the same tests in a previously prepared environment.

For the execution of the tests, a Google Cloud instance of the e2-standard-8 type was configured, with 8 processors, 32 GB of memory, x86/64 architecture running the Debian 11 operating system.

Once the virtual machine that would process the performance tests was configured, twenty test files were created. Half of the files ran unittest tests and the other half ran multtestlib tests. The tests consisted of blocks of 50 to 500 tests, with an increment of 50 tests from one file to another, where *assertEqual* commands were executed in a function that simulated processing from 1/1,000s to 10/1,000s in duration, according to the parameter informed to the tested function. Each package was subjected to the same number of test lines, with the same sequence of parameters.

Each test file was executed five times, with its processing time in seconds recorded. The tests performed with the multtestlib package were performed using 1, 2, 4 and 8 processors. After the execution of all the tests, the averages of the execution time of each test block (from 50 to 500 tests) were calculated and grouped by the number of processors used.

After obtaining the average of the test execution times, the times spent with unittest and multtestlib were compared. For performance comparison purposes, the unittest times became 100% and the multtestlib values were adjusted as a percentage of the benchmark time, as shown in the figure 2.

It can be seen that the processing time of multtestlib with 1 core is longer than the time recorded by the benchmark. This is because task pool management, screen output procedures and multtestlib output files consume processor resources. When tasks are performed in parallel, the effort performed by each

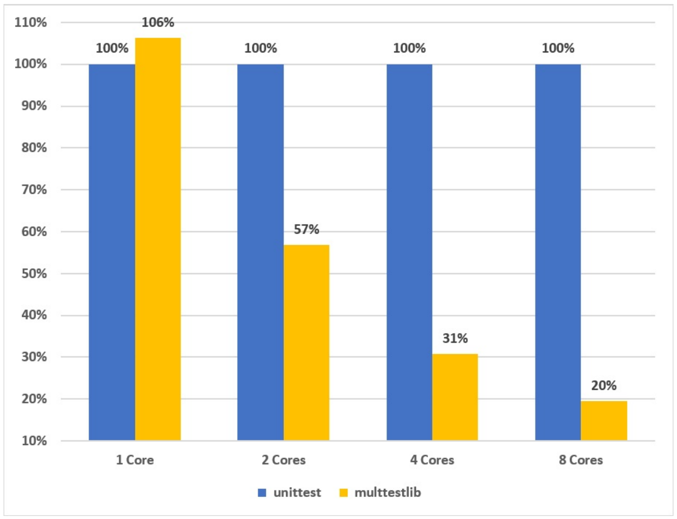


Fig. 2. Performance unittest x multtestlib

processor dilutes the time for managing the pool and the output tasks.

As the objective of the project is to build a tool for parallel processing, the performance with only one processor core is not a concern.

## D. Syntax and Application

The multtestlib package was developed as a proposal for software testing that would allow the developer to focus their effort on creating unit tests quickly and clearly, through a lean syntax. The listing 4 generically demonstrates the basic structure of a test file.

The test file, of type *.py*, is started by importing the multtestlib package and other files that contain functions or object classes to be tested. Next, the *main()* function is defined, which contains the tests that will be executed using the processing pool. This pool, which is a list object, is created by the commands *multtestlib.Pool(cores) as pool* and the list is fed by the *.starmap* method. To this list are added the parameters *subject_to_test, given_1, given_2, expected_1*, and so on, which will be supplied to the test command indicated by *multtestlib.command*. It is possible to use more than one pool in the same test file.

```python
def main():
    a = 90
    b = 35
    c = 12
    cores = 2

    with multtestlib.Pool(cores) as pool:
        pool.starmap(multtestlib.test_equal,
            [(a, 90),
             (b, 35),
             (c, 10),
            ])
```

Listing 3: Function test_equal example

| Multtestlib Command | Test |
|---|---|
| test_equal | a == b |
| test_not_equal | a != b |
| test_func | function(a, b) == c |
| test_not_func | function(a, b) != c |
| test_is | is a b |
| test_is_not | is a not b |
| test_in | is a in b |
| test_not_in | is a not in b |
| test_instance | is instance(a, b) |
| test_not_instance | is not instance(a, b) |
| test_almost_equal | round(a - b, 7) == 0 |
| test_not_almost_equal | round(a - b, 7) != 0 |
| test_greater | a > b |
| test_greater_equal | a >= b |
| test_less | a < b |
| test_less_equal | a <= b |

TABLE I
MULTTESTLIB COMMANDS

The table I displays the test commands of the multtestlib package.

The final part of the test file contains the commands of automatic preparation for the execution of the test file (*multtestlib.init()*), the command to protect against the freezing of execution by the Windows Operating System (*multtestlib. freeze_support()*) (function not necessary when using Linux), the call to the *main()* function and the test environment auto-close command (*multtestlib.end()*).

The *.init()* method of the multtestlib package configures, using methods from the *os* package, the files that will receive the results of the tests to be executed. At the end of the tests execution, the *end()* method completes the tests and closes the output files.

## E. Flexibility

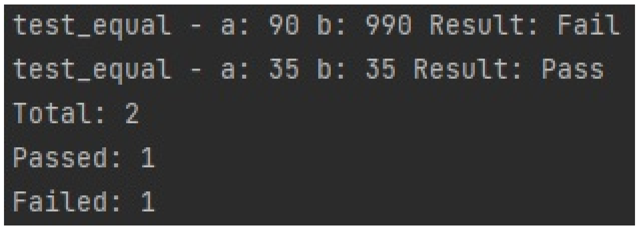
An agile development and test environment must have flexible and adaptable test people and tools [32]. To meet this need, the multtestlib package was developed for flexibility in use.

```python
import multtestlib
<other imports>

def main():
    with multtestlib.Pool(cores) as pool:
        pool.starmap(multtestlib.command,
            [(subject_to_test, given_1, given_2,
            ↪  expected_1),
             <...>
             (subject_to_test, given_n-1, given_n,
             ↪  expected_n),
            ])

if __name__ == "__main__":
    multtestlib.init()
    multtestlib.freeze_support()
    main()
    multtestlib.end()
```

Listing 4: Anatomy of a test file



```
test_equal - a: 90 b: 990 Result: Fail
test_equal - a: 35 b: 35 Result: Pass
Total: 2
Passed: 1
Failed: 1
```

Fig. 3. Screen output

The tool allows more than one pool queue to be created per test program. Each queue can contain tests of more than one object or method, according to the needs and characteristics of each project to be tested, the listing 5 exemplifies this type of situation.

In the case of regression tests, it is possible to use the files from the previous tests, managing their execution by creating a master program, which imports the unit test scripts and which will be configured and executed according to the needs of the regression test to be performed.

### *F. Results on Screen*

During the execution of the unit tests, the multtestlib package automatically generates a standard screen output. This output allows the tester to track the evolution of the running test script. This resource, figure 3, informs which test is running, expected value, received value and the result (Pass or Fail). At the end of the execution, the program also displays a total of the tests.

### *G. Log Files*

In order to allow the tester to verify the results of the tests carried out, with the reuse of code from the OS package, file and folder management methods were implemented for the storage of three *.txt* files containing the logs of the tests performed.

The generated files are:

- **filetot.txt**: Stores data from all tests performed;
- **filepass.txt**: Stores data referring to positive tests, those that worked;
- **filefail.txt**: Logs data from failed tests.

The information sent to the files are exactly the same as those sent to the screen during the execution of the tests, and all the generated files present in their last line information about the total number of tests performed, how many passed and how many failed, according to the function of each file.

### *H. Easy to Install*

The Python language allows the easy installation of any package available in PyPI (The Python Package Index), just by using the command *pip install* that can be executed in any Windows or Linux terminal [33].

The first version of the multtestlib package is available for PyPI and can be installed via the command *pip install multtestlib* [34].

```python
with multtestlib.Pool() as pool:
    pool.starmap(multtestlib.test_equal,
                 [(a, 90),
                  (b, 35),
                  ])

    pool.starmap(multtestlib.test_instance,
                 [(a, int),
                  (c1, Cube),
                  (c2, int),
                  ])

    pool.starmap(multtestlib.test_equal,
                 [(c1.volume, 1),
                  (c2.volume, 8),
                  ])
```

Listing 5: Multiples pools

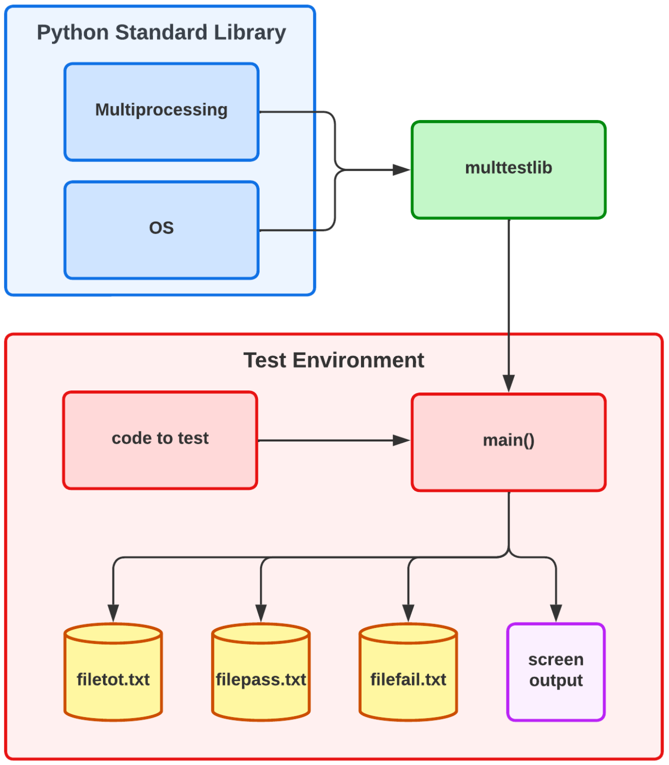


Fig. 4. Multtestlib architecture

## III. Architecture

The development of this proof of concept resulted in a Python package that reuses code from the Multiprocessing and OS packages. While Multiprocessing manages a pool of tasks to be submitted to execution by processors working in parallel, the OS package allows the management of the three log files generated during the execution of the tests. Figure 4 illustrates the typical architecture of a test that uses the multtestlib package.

## IV. Conclusion

The proof of concept was developed entirely in the Python language, proving its reputation as a powerful, concise and flexible language. It was possible to change the normal behavior of the language, which uses the GIL to guarantee sequential processing, and to perform parallel processing to perform tasks faster, optimizing computational resources and, consequently, saving hours of work for the professionals responsible for carrying out software testing.

The authors believe it is possible to expand parallel processing use in Python language for other computational problems in different software engineering areas.

## V. Future Works

For future works, it is suggested the integration of the multitestlib package with integration and continuous delivery tools such as jenkins.

Another suggestion is the evolution of the package to a framework, which contains features such as a graphical interface, test schedule, more detailed and dynamic reports, monitoring of test execution by web browsers and analysis of code coverage.

And finally, the authors hopes to be able to contribute to the Python community by promoting the multtestlib package, showing its features and advantages so that it can be widely used and improved.

## VI. Acknowledgments

The authors would like to thank the support granted by the following institutions: Banco do Brasil (BB), The Brazilian Ministry of Education (MEC) and the Aeronautics Institute of Thecnology (ITA), throught the Casimiro Montenegro Filho Foundation (FCMF) for allowing the completion of this work. This paper is part of MEC-ITA project.

## References

[1] F. Huang, B. Liu, and B. Huang, "A taxonomy system to identify human error causes for software defects," pp. 44–49, 2012.

[2] L. Copeland, *A practitioner's guide to software test design*. Artech House, 2004.

[3] G. O'Regan, *Concise Guide to Software Testing*, 1st ed., ser. Undergraduate Topics in Computer Science. Springer International Publishing, 2019.

[4] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. John Wiley & Sons, 2011.

[5] I. Sommerville, *Software Engineering, 10th Edition*. Pearson, 2016.

[6] H. Krasner, *The Cost of Poor Software Quality in the US: A 2020 Report*. Consortium for Information & Software Quality - CISQ, 2021.

[7] R. F. Bennett, "The y2k problem," pp. 438–439, 1999.

[8] H. Sharangpani and M. Barton, "Statistical analysis of floating point flaw in the pentiumtm processor (1994)," *Intel Corporation*, vol. 30, 1994.

[9] B. Nuseibeh, "Ariane 5: who dunnit?" *IEEE Software*, vol. 14, no. 3, p. 15, 1997.

[10] P. Johnston and R. Harris, "The boeing 737 max saga: lessons for software organizations," *Software Quality Professional*, vol. 21, no. 3, pp. 4–12, 2019.

[11] G. Travis, "How the boeing 737 max disaster looks to a software developer," *IEEE Spectrum*, vol. 18, 2019.

[12] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[13] P. B. G. Abraham Silberschatz, Greg Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018.

[14] "Beginnersguide/overview - python wiki," https://wiki.python.org/moin/BeginnersGuide/Overview, (Accessed on 09/30/2022).

[15] "General python faq — python 3.10.7 documentation," https://docs.python.org/3/faq/general.html, (Accessed on 10/02/2022).

[16] P. Carbonnelle, "Pypl popularity of programming language," Jul 2021. [Online]. Available: https://pypl.github.io/PYPL.html

[17] "Programming languages: Python's growth is "absolutely explosive," says anaconda ceo, and not slowing down — techrepublic," https://www.techrepublic.com/article/programming-languages-pythons-growth-is-absolutely-explosive-says-anaconda-ceo-and-not-slowing-down/, (Accessed on 09/20/2022).

[18] S. Veeraraghavan, "Best programming languages to learn in 2021," Jun 2021. [Online]. Available: https://www.simplilearn.com/best-programming-languages-start-learning-today-article

[19] "Tiobe index - tiobe," https://www.tiobe.com/tiobe-index/, (Accessed on 10/02/2022).

[20] "Glossary — python 3.10.7 documentation," https://docs.python.org/3/glossary.htmlterm-global-interpreter-lock, (Accessed on 10/02/2022).

[21] M. Summerfield, *Programação em Python 3: Uma Introdução Completa à Linguagem Python*. Alta Books, 2012.

[22] P. Barry, *Head First Python A Brain-Friendly Guide*, 2nd ed. O'Reilly Media, 2016. [Online]. Available: libgen.li/file.php?md5=fd6123431b4cc1c87c9024825728b014

[23] "Difference between python modules, packages, libraries, and frameworks — learnpython.com," https://learnpython.com/blog/python-modules-packages-libraries-frameworks/, (Accessed on 10/02/2022).

[24] "Pep 8 – style guide for python code — peps.python.org," https://peps.python.org/pep-0008/, (Accessed on 10/02/2022).

[25] "unittest — unit testing framework — python 3.10.7 documentation," https://docs.python.org/3/library/unittest.html, (Accessed on 10/02/2022).

[26] "dispy: Distributed and parallel computing with/for python — dispy 4.15.0 documentation," https://dispy.org/, (Accessed on 09/18/2022).

[27] "Pandaral·lel documentation," https://nalepae.github.io/pandarallel/, (Accessed on 09/18/2022).

[28] "Using ipython for parallel computing — ipyparallel 8.4.1 documentation," https://ipyparallel.readthedocs.io/en/latest/, (Accessed on 09/18/2022).

[29] "Dask — scale the python tools you love," https://www.dask.org/, (Accessed on 09/18/2022).

[30] "multiprocessing — process-based parallelism — python 3.10.7 documentation," https://docs.python.org/3/library/multiprocessing.html, (Accessed on 09/18/2022).

[31] "Pep 371 – addition of the multiprocessing package to the standard library — peps.python.org," https://peps.python.org/pep-0371/, (Accessed on 10/12/2022).

[32] J. Crispin, Lisa;Gregory, *Agile testing a practical guide for testers and agile teams*, 1st ed., ser. The Addison-Wesley signature series; A Mike Cohn signature book. Addison-Wesley, 2014.

[33] "Pypi · the python package index," https://pypi.org/, (Accessed on 09/28/2022).

[34] "multtestlib · pypi," https://pypi.org/project/multtestlib/, (Accessed on 09/20/2022).