Python vs Julia

This document is intended to compare the python and julia languages, in order to do that, we established different criteria based on our research and our own experience.

This comparison was carried out as part of Milo KOSON, Kévin SEN and Louison TENDRON's final year project at ESEO Angers, and is focused on the data field for AI and Machine Learning only.

Most of the documentation we used was provided by the Julia official documentation website : https://docs.julialang.org/en/v1/

We also developed 7 predictive models in Python and Julia, that allowed us to know how to code, use libraries and debug in Julia. The focus was given on the following popular time series and machine learning models :

- Arima
- Convolutional Neural Network (CNN)
- Keras Neural Network
- Keras Long Short-Term Memory (LSTM)
- Random Forest
- STL-Arima
- XGBoost

This coding phase helped us to develop the comparison criteria that we present below.

We decided to divide the comparison in 2 axis :

- First, we will show the global comparison between Python and Julia on common aspects that we felt were important to compare :

1. **Ease of use** : How easy is it to use the language ?
2. **Global Documentation** : How well is the language documented ?
3. **Package Management** : How does the language manage the packages ?
4. **Data processing** : How easy is it to process data in the language ?
5. **Data visualisation** : How easy is it to visualise data in the language ?
6. **Tests** : How easy is it to test the language ?
7. **Compilation Method** : How does the language compile the code
8. **Parallel Computing** : Ability to execute multiple calculations or processes simultaneously
9. **Multi-Threading** : Ability of an operating system program or process to be used by several users at the same time.

10. **Distributed Computing** : Ability to use multiple interconnected computers or nodes to work together as a single, unified computing resource.
11. **Interoperability** : ability for two or more languages to interact or communicate for effective data transmission in a system.
12. **Community Trend** : What is the trend in the use of the language over time ?
13. **Backward Compatibility** : Does the future language versions maintain backwards compatibility with previous versions ?
14. **Corporate Support** : What level of support is provided by companies/organizations for the language ?
15. **Support by AI Coding Assistant** : Do AI Coding Assistants (ChatGPT, Bard, . . .) are able to provide helpful advice during the coding phase ?
16. **Ecological impact** : How can the language be used to manage energy resources ?

We decided to describe each of these aspects in this document, but we also produced an Excel file in which we summarised each point and gave a score for each aspect in Julia and Python.

- In a second time, we defined criteria for every model we developed then we rated each one of them on a scale from 1 to 5 depending on how relevant it is :

1. **Documentation** : How well is the model documented ?
2. **Performance** : How fast is the model ?
3. **Packages** : Which packages were used for the model ?
4. **Code length** : How long is the code ?

This model comparison is also available in the Excel file along with the global comparison summary.

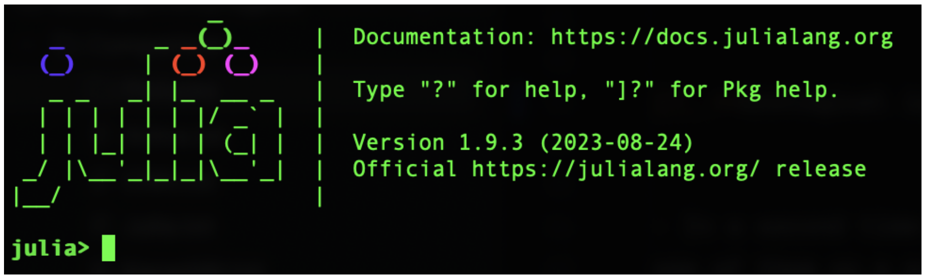
Global Comparison

In this section, we are going to analyse the criteria that we identified to compare the languages :

Ease of use

The thing that we noticed when we first used Julia was its high-level has a syntax similar to that of Python for declarations and the use of variables/functions, and is a little closer to the MatLab language when it comes to the use of mathematics. It makes the language easy to understand even if you have no experience with it.

Just like python, Julia, in its "native form" is a built-in terminal that looks just like this :



```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.9.3 (2023-08-24)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

Figure 1: Julia built-in terminal

It allows you to run commands, functions or scripts, but is not very suitable for programming (from now on, we will refer to the built-in terminal by its real name : REPL (Read-Eval-Print Loop)).

That is why we chose to use an IDE, having a graphic interface is way more comfortable to code. But Julia does not have a dedicated IDE unlike Python that have many of them (Pycharm, Jupyter Notebooks, . . .) that is why we decided to use the Julia extension on VSCode. It works as every extension from the VSCode marketplace and is very easy to use, you just have to install the Julia software on your computer, the extension will detect it and set every parameter you need.

Getting packages and documentation is also easy, but we will come back to this point later during the analysis.

Global Documentation

Julia and Python both have their own official documentation website :

- Julia : https://docs.julialang.org/en/v1/
- Python : https://www.python.org/doc/

Both are very complete and can be used for different purposes like learning, getting help or exploring advanced language features. They are supposed to cover every aspect of each language and are regularly updated when a new version is released.

Package Management

In terms of packages, we found Julia more effective than Python for fetching and installing packages. In order to install a package in Julia, you just have to use the following command lines in the Julia REPL on your VSCode project:

```
using Pkg
Pkg.add("package_name")

// If you need a specific version of the package :
```

```
Pkg.add(name="package_name", version="0.1.0")
```

If the package exists, it will automatically install it and its dependencies, if you already possess the package, it will proceed to an update.

Furthermore, every package installed is listed in the **Project.toml** and the **Manifest.toml** files that are located at the root of our project.

So if you decide to use another computer that does not have any packages installed, and you don't want to install each package manually, you can just use the following command in the Julia REPL in your VSCode project to replicate your environment :

```
using Pkg
Pkg.instantiate()
```

All your packages will be installed on your new computer, we think it is a very clever way to handle packages.

With Python, we used PyCharm's "Python Packages" menu which is a user interface for Pip, the package manager of Python. It is very easy to use, you just have to search for the package you want to install, select it and click on the "Install" button. It will automatically install the package and its dependencies. It can also show you which version of python is required for the package to work, which is very useful.

Data Processing

Python shines in data science domains like data manipulation, scaling, normalization, encoding, time series preparation, and splitting datasets. Its extensive libraries such as Pandas and scikit-learn make data manipulation tasks like filtering, grouping, and joining seamless and efficient. For scaling and normalization, scikit-learn provides robust and easy-to-use tools, crucial for preprocessing data for machine learning models. Encoding categorical data is also straightforward with functions like LabelEncoder, OneHotEncoder, and get_dummies. When dealing with time series data, Pandas offers powerful features like shift() for lagging or leading data, essential in time series forecasting. Lastly, scikit-learn's train_test_split is a staple function for splitting datasets into training and test sets, a critical step in machine learning workflows. Python's well-documented libraries and large community support make it a top choice for these tasks.

Julia, though less mature in its ecosystem than Python, is rapidly gaining ground in data science domains due to its performance, especially with large datasets. In data manipulation, packages like DataFrames.jl offer Julia users good capabilities, although they come with a smaller community and fewer resources compared to Python. For scaling, normalization, and encoding tasks, Julia's packages like MLJ.jl and StatsBase.jl provide efficient tools, though they might not be as comprehensive as Python's offerings. Julia also caters to time series data manipulation with packages like TimeSeries.jl, offering functionalities akin to Python but with potentially better performance in handling larger datasets. Similarly, for splitting datasets, Julia uses packages like MLJ.jl, which, while efficient, may have fewer examples and a smaller user base. Julia's main advantage lies in its speed and efficiency, making it a compelling choice for data-intensive tasks.

Data Visualisation

Python shines in data visualization due to its vast array of well-established libraries like Matplotlib, Seaborn, and Bokeh. These tools are not only powerful but also versatile, catering to a wide range of visualization needs with ease and flexibility. The simplicity and readability of Python's syntax further enhance its appeal, especially for beginners. Community support is another major strength, offering abundant resources for learning and troubleshooting. However, Python's weaknesses lie in its performance with very large datasets or highly complex interactive visualizations, where it can be less efficient than some other languages

Julia, although newer in the field of data visualization, stands out for its high performance, particularly with large datasets and computationally intensive visualizations. Its syntax, while user-friendly for those with a background in scientific computing, may have a steeper learning curve for beginners. Julia's ecosystem of visualization libraries, such as Plots.jl, is growing but not as extensive or mature as Python's. The language's ability to interface with Python libraries mitigates this to some extent. The main drawback of Julia in data visualization is the relative scarcity of resources and community support compared to the well-established Python community

Tests

Python's testing frameworks are known for their diversity and maturity, with options like Pytest, Unittest, Doctest, and Robot Framework catering to various testing needs. Pytest is notable for its simplicity, while Unittest is conveniently included in the standard Python library. Selenium excels in web application testing, showcasing Python's adaptability. The community around Python provides extensive support and resources, making these frameworks highly accessible to both new and seasoned developers.

Julia integrates its testing frameworks, such as Test.jl and FactCheck.jl, closely with its language features. Specialized frameworks like Documenter.jl and Mocking.jl cater specifically to Julia's ecosystem, with Polymorphic.jl introducing a unique approach to property-based testing. This focus on integration and specificity in testing aligns well with Julia's overall efficiency in development.

Comparatively, Python's testing frameworks are often more user-friendly for beginners, backed by a wide-ranging community and abundant resources. Julia's frameworks, though newer, provide efficient integration with the language, making them advantageous for Julia-specific projects. The choice between the two depends on factors such as the project's requirements, familiarity with the language, and specific development needs. While Python offers accessibility for newcomers, Julia's frameworks are well-suited for leveraging the language's unique features.

Compilation Method

From what we have experienced and what we saw on the internet during ou researches is that Julia's compilation method was conceived to be more efficient than python's.

Indeed, Python uses an interpreted, dynamically-typed, and high-level programming language. It employs a two-step process : first, the source code is translated into bytecode by the Python interpreter, and then the bytecode is executed by the Python Virtual Machine (PVM). It is therefore by its interpreted nature that Python leads to a slower execution speeds compared to Julia and more generally to compiled languages such as C, Java, Rust, ...

On the other hand, Julia was designed for high-performance numerical and scientific computing. It uses Just-In-Time (JIT) compilation, which means that the code is translated into machine code just before execution. Julia's type system allows for multiple dispatch, enabling the JIT compiler to generate specialized machine code for different argument types, leading to efficient execution. This approach combines the flexibility of interpreted languages with the performance of compiled languages, making Julia suitable for numerical computing tasks.

Julia's Compilation method is therefore faster than Python's, but unless you have a highly-developed model, and as long as you don't get into real-time issues, we don't think the time saved during compilation will make much difference whether you choose Julia or Python.

Nb: Python is trying to evolve to a JIT compilation method with version 3.13, out in October 2024, so this analysis may evolve afterwards.

Parallel Computing

For the parallel computing part, we did not have the opportunity to test it, but we did some researches on the internet, just to give you some example of what can be done with Julia or Python.

Python has a module called "multiprocessing" that allows you to run multiple processes in parallel. It is very easy to use, you just have to import the module and use the "Pool" class to create a pool of workers that will execute your processes. You can also use the "Process" class to create a process that will execute your function. This will mainly work by creating multiple processes that will run on different cores of your CPU.

Concerning Julia, it seems to be more documented, with many possibility of parallel computing. You can choose between multi-threading, gpu computing, distributed computing or coroutines. Each one of these methods have it's own advantages and disadvantages. Coroutines are asynchronous tasks that we see in many programming languages, they are useful for I/O-bound operations (Open/Write files, HTTP requests, . . .). Gpu computing is provided via some packages that allow you to use your GPU to run your code, it is very useful for machine learning tasks. The other methods will be explained in the next sections.

If you need more information about Parallel Computing, here are the official documentation links :

- Python : https://docs.python.org/3/library/multiprocessing.html
- Julia : https://docs.julialang.org/en/v1/manual/parallel-computing/

Multi-Threading

Regarding multi-threading, each language has its own mechanism, we did not use this functionality because we did not need to, but it is always interesting to know how the language works and what are its limits.

Python has a something called a Global Interpreter Lock (GIL) which restricts the execution of multiple threads in the same process to one at a time. This can limit the effectiveness of multi-threading. However, Python's threading module can still be useful for I/O-bound operations (Open/Write files, HTTP requests, . . .) where threads spend time waiting for external resources.

On the other side, Julia was designed with a focus on parallel and concurrent computing, but it doesn't have a GIL mechanism. Its threading model allows concurrent execution of multiple tasks, making it a better support for multi-threading scenarios compared to Python. Julia's threading is light and enables efficient parallelism, making it adapted for tasks that can use concurrent execution.

If you need more information about Multi-Threading, here are the official documentation links :

- Python : https://docs.python.org/3/library/threading.html
- Julia : https://docs.julialang.org/en/v1/manual/multi-threading/

Nb: Python is trying to evolve and supress the GIL in the version 3.13, out in October 2024, so this analysis may change afterwards.

Distributed Computing

The main difference in terms of distributed computing is that Julia supports it natively via the "Distributed" library, whereas Python has to rely on external libraries and frameworks such as Dask, Ray or Dispy.

Indeed, distributed computing is a core part of the Julia language's design. It consists in launching parallel tasks and distribute computations seamlessly across multiple nodes. Julia's lightweight threading model and absence of a GIL allows the users to have a better scalability and performance.

Python gives access to the same functionality despite the fact that it requires access to external tools, which leaves less flexibility to the user and creates a certain 'technological dependency'. In addition, Python's Global Interpreter Lock (GIL) can limit the performance gains in certain cases.

If you are going to use distributed computing, using Julia would be a wiser choice in our opinion.

If you need more information about Distributed Computing, here are the official documentation links :

- Python : *(No official documentation because it is not a native feature)*
- Julia : https://docs.julialang.org/en/v1/manual/distributed-computing/

Interoperability

During our research, we found out that interoperability could be a very useful tool for programming, indeed, both Python and Julia are allowing interoperability through different libraries.

Concerning Julia, we found this article on the official Julia forum, it shows different examples of interoperability with Python, R and C++ languages : https://forem.julialang.org/ifihan/interoperability-in-julia-1m26.

For Python, we also found a very well documented article on Python's official documentation website that shows how to extend Python with C/C++ language : https://docs.python.org/3/extending/extending.html.

On our side, we did not experiment interoperability because we were asked to explore Julia's language using only native features, every library that we used during our coding sessions are 100% coded in Julia, allowing better performances and a better reliability.

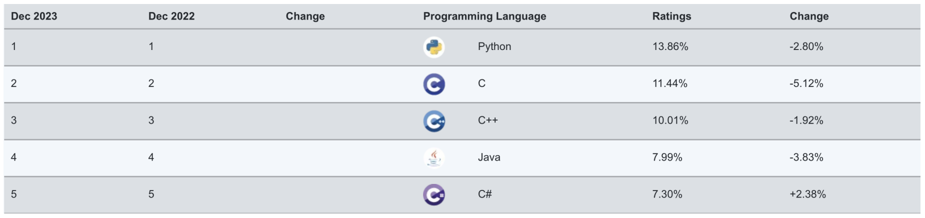
Community Trend

For this part, we decided to rely on the TIOBE Index from december 2023 : https://www.tiobe.com/tiobe-index/.
Here is the purpose on the index, explained on its website :

"The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the best programming language or the language in which most lines of code have been written."

And if you are interested in how they manage to rate languages, here is the link they also provide : https://www.tiobe.com/tiobe-index/programminglanguages_definition/.

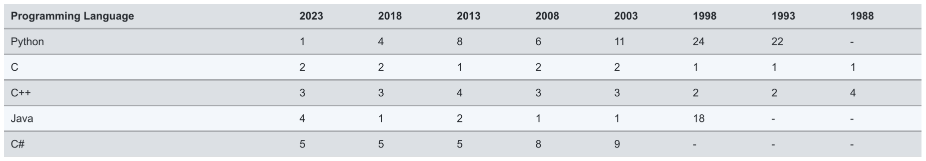
In the picture below, we can see Python's "popularity" between 2022 and 2023 :



Dec 2023	Dec 2022	Change	Programming Language	Ratings	Change
1	1		Python	13.86%	-2.80%
2	2		C	11.44%	-5.12%
3	3		C++	10.01%	-1.92%
4	4		Java	7.99%	-3.83%
5	5		C#	7.30%	+2.38%

Figure 2: Pyhton's rating by TIOBE

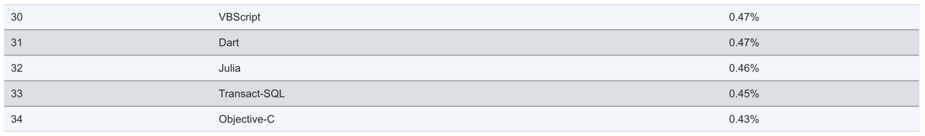
It occupies the first place in 2022 and 2023 with a rating of 13.86%, making it the "trendiest" programming language. And if we look a little further into the past, we can see that overall, Python remains in top 5 position if we observe it's rank since 1993 (2 years after its creation) :



Programming Language	2023	2018	2013	2008	2003	1998	1993	1988
Python	1	4	8	6	11	24	22	-
C	2	2	1	2	2	1	1	1
C++	3	3	4	3	3	2	2	4
Java	4	1	2	1	1	18	-	-
C#	5	5	5	8	9	-	-	-

Figure 3: Python's rating over the years

Whereas Julia (created in 2012) occupies a not so bad 32nd place with a rating of 0.46% :



30	VBScript	0.47%
31	Dart	0.47%
32	Julia	0.46%
33	Transact-SQL	0.45%
34	Objective-C	0.43%

Figure 4: Julia's rating by TIOBE

This shows that despite its efficiency and its reliability, Julia remains a very specific language used only by a minority of users that know how it works and that understood its potential over other languages such as Python.

The Community Trend remains a good indicator to observe a language's influence, but it is important to note that it does not take into consideration the efficiency of the language.

Backward Compatibility

Regarding the backward compatibility, we did not find much information, so we based our analysis on the obsevation that we made during our coding sessions.

Python seems to have an overall good backward compatibility, we did not encounter any issues when we tried to run our code on a previous version of Python. We also found this article on the official Python documentation website that explains how to handle backward compatibility : https://docs.python.org/3/howto/pyporting.html.

On the other hand, Julia seems to have a more complicated backward compatibility, even though it is possible to run previous versions of Julia, it is not recommended to do so, because it can lead to some issues with some libraries that became stable only recently. But there is no breaking changes since Julia 1.0, so if you want to use Julia, we would recommend you to use the latest version.

Corporate Support

For this part, we have clear differences between Python and Julia, thus can influence the choice of a language for a company.

For Python, the language is constantly developed by the Python Software Foundation (PSF), a non-profit organization that is dedicated to making this language accessible to everyone. It is also supported by many companies such as Google, Microsoft, Facebook, Amazon… And with such a huge community of developers, library creators and users, Python is constantly evolving and improving.

On the other hand, Julia is only improving via the contribution of open-source developers, and is not supported by any company. Even though it is a very efficient language, it is not as popular as Python, and it is not evolving as fast as Python. But that means that investing in Julia could be a good idea for a company, because the first company that will support Julia will have a huge advantage over its competitors, by developing the language and its libraries, and by creating a community around it.

Support by AI Coding Assistant

During our coding sessions, we wanted to see if the coding assistants (ChatGPT 3.5 from OpenAI and Bard from Google) could help us to code in Julia.

(We are specifying that we used the ChatGPT 3.5 version because we wanted to use the new GPT-4 version but OpenAI opened a waiting list, but we only received the access when we finished the coding phase, so we could not try it.)

- **ChatGPT 3.5** : First of all, the main problem with GPT 3.5 is that its “knowledge” does not exceed January 2022, meaning that any change

that occurred between January 2022 and today will not be taken into consideration.
For Python, it was pretty easy to obtain a good code that could be run without problem, the language is well known and widely used, of course, the libraries are still evolving, but they remain quite similar over time, so it is not a problem.
On the other hand for Julia it was more complicated, the libraries are evolving quite fast and are drastically changing their operating mechanisms, GPT 3.5 was therefore not able to deliver a good code for most of the time, specially for machine learning algorithms and neural networks that are working with the "Flux" library.
GPT 3.5 can be a good tool if you want to code a quick program in Python, but it will not be very suitable for a specific Julia code in AI/Machine Learning like we had to do.

- **Bard** : Compared to GPT 3.5, Bard has the advantage of being connected to the internet.
  Once again, for Python, there was no problem in order to have a not so complex piece of code that could be run right away, but we identified a bigger problem when we tried to discuss with it.
  While with GPT 3.5 we could try to specify things or modify the code many times, with Bard, we had a first code that did not work well in Julia, so we explained the error and where it came from, Bard analysed the problem and gave us another code to try. When we tried that second code, there was another error, so we did the same thing and gave it instructions on what we wanted to have. Bard now entered a phase where it only offered us 2 possibilities, run the old piece of code it gave us or run the new one. We could not have any other option, and it occurred multiple times, making Bard very inefficient very quick.

We could also talk about GitHub Copilot that can be really helpful in Python but as it is based on OpenAI learning models, it encounters the same problems as ChatGPT 3.5. The main problem is that most of the coding assistants are pulling their code from platforms like GitHub and Julia being a very specific language, there less information about it. On the following link, you can observe its representation compared to other programming languages : https://madnight.github.io/githut/#/pull_requests/2023/3.
Julia has a percentage of pulling request of 0.087% while Python has 17.715%. This also explains why these coding assistants are not very suitable with Julia for the moment. To counterbalance these problems, some assistants like Phind (https://www.phind.com/) could be useful as it also pulls information from the internet, but we have not tested it during our coding phase.

To conclude this part, if you want to use an AI coding assistant to improve your productivity with Python and Julia, it is possible (even if it is way easier with Python) we would recommend you to use ChatGPT 3.5 and try GPT-4 or Phind if you can.

Ecological Impact

This part is very important for us, because we think that it is important to use a language that is not too energy consuming, especially in the context of climate change.

Python by its nature is not very ecological, indeed, it is an interpreted language, meaning that it is not directly translated into machine code, but into bytecode that will be interpreted by the Python Virtual Machine (PVM). This process is very energy consuming, and it is the main reason why Python is not very ecological. In fact, running a Python program for 1 hour is equivalent to running a C program for 1 minute. And all this time is more work for your CPU, which means more energy consumption.

On the other hand, Julia is a far better programming language in terms of ecological impact, indeed, it is a mix between interpreted and compiled languages, it looks like an interpreted language, but it is compiled just before execution, and it is compiled in a very efficient way, making it very ecological. This technique allow Julia to be really efficient, ecologically and in terms of performance.
You can find more information on Julia and its ecological impact on this article : https://juliazoid.com/julia-the-little-known-programming-language-that-might-be-the-key-to-understanding-climate-change-ff91be18973a

Conclusion

We have concluded that each language has its strengths and weaknesses, but depending on the model and the situation, it may be more interesting to use one more than the other.

In terms of performance, Julia is way ahead of Python. This is to be expected, as Julia has been designed and optimised for AI and machine learning. The choice of language will therefore be hugely influenced by the context of the project.

We have identified 2 parameters to take into account :

- **The size of the dataset** : the greater the amount of data to be analysed, the greater the speed of the language to be taken into account.
- **Real time** : if you plan to work with real time constraints, the choice of language will also have to be based on the language's ability to handle this.

If we base ourselves solely on these two criteria and the performance we know about each language, we can have a table modelled as follows :

	Real Time	No Real Time
Small Dataset	Julia	Python
Medium Dataset	Julia	Python
Large Dataset	Julia	Julia

However, we also need to be cautious, throughout the project, we encountered a large number of obstacles (which can surely be overcome with a more experienced team) concerning the Julia language, and we believe that the language is still in a “maturing” phase in certain respects. What’s more, if the company wants to use Julia from now on, it will have to make a long-term investment.

In fact, in order to be able to use Julia under optimum conditions, we advise the company to dedicate a team to setting up and maintaining the packages that will be used. These teams will also need to be trained in the language in order to understand its specific operating mechanisms. This will require significant financial and time resources, as existing models will also have to be adapted, while ensuring their reliability.

It should also be borne in mind that Python, for its part, is working on the implementation of a JIT compilation method and also plans to remove the GIL (Global Interpreter Lock) very soon, which limits the language’s multi-threading and parallel computing functionalities. This could considerably improve Python’s performance in the coming months/years.

That’s why, if after that a company still wants to use Julia, we advise it to begin a transition phase over several years so that it doesn’t change completely too quickly and its teams have time to adapt to this new technology.