# What Is Julia, and Why It Matters?

§



**Julia** is a reasonably new, open-source, high-level, dynamically-typed programming language. It's a multi-platform language supported on Linux, macOS, Windows and FreeBSD. It has been gaining attention recently because its package repository has been increasing with exciting tools, and the community has also been growing.

In this Blog Article, we'll discuss Julia's main features, why it's important, compare other popular Data Scientist's programming languages such as Python, R, and Spark, and conclude by providing additional resources for those interested in learning Julia.

We'll be using Julia scripts which can be found in the Blog Article Repo.

§

# Table of Contents



§

# Prelude

I remember some years ago, when Showtime's Billions was at its pinnacle, that Julia was featured in an episode in which Taylor Mason, the quantitative analyst, mentioned it as a critical tool used in the sophisticated quantitative strategies deployed at the fictional Taylor Mason Capital.

It seemed interesting, but I didn't investigate further since my main workhorse at the time was (*and still is)* Python (*with some specific tasks performed on R*). This combination covers all the data-processing and data-analysis-related functions I perform daily as a Data Scientist and content creator.

Some months ago, I came across an interesting TEDx MIT talk: "A programming language to heal the planet together: Julia". It immediately grabbed my attention and made me think: How comfortable have I gotten with Python? Maybe too comfortable? I mean, it's general-purpose, dynamically-typed, descriptive, easy to read and write, has tons of documentation available, a massive community, endless libraries to suit whichever need we may think of, and the list goes on and on. But let's face the elephant in the room: it can be s l o w, and the daily data volume generation will only increase.

Dr. Alan Edelman's conference appeared so convincing that I decided to adopt the language to see if it could substitute Python & R, at least to some degree.

For this to work, I first needed to do some preliminary checks:

- Is high-level and mostly dynamically-typed (*or at least has the option to do so*).
- Has a robust and up-to-date set of equivalent libraries.
- Has an active community (*or at least detailed and rigorous technical documentation*).
- Allows manipulating array-type and tabular-type objects as easily as with NumPy or Pandas.
- Has well-maintained support for at least a couple of IDE(s), preferably also Jupyter or an equivalent notebook-style environment.

I spent a couple of months testing this new programming language, and let me tell you, lads, it exceeded my expectations in every possible way.

§

# Julia, in a nutshell

The creators of Julia did a really good job at describing what this language would represent, its foundations, and what the main objectives were when creating it:

> We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, and as good at gluing programs together as the shell. Something that is dirt simple to learn yet keeps the most serious hackers happy. We want it interactive, and we want it compiled. (Did we mention it should be as fast as C?)

— *Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman*[1]

They basically intended to glue 6 of the most popular scientific languages to create one perfect language that will come here to stay, potentially replacing the most popular data science language right now: Python.

If we visit the official Julia website, we can see some of its key properties:

- Fast
- Dynamic
- Reproducible
- Composable
- General

- Open source

Some of the aspects above are natively covered in other languages such as Python; it's dynamic, can generate reproducible environments, is general-purpose, and is open-source.

Some of them, though, are not covered: Python is not a particularly fast language since it compiles into a format known as byte code. The source code compiled to byte code is then executed in Python's virtual machine line by line to carry out the operations. Internally, Python code is interpreted during run time rather than compiled into native code. Hence it's slower than some other compiled languages such as C, C++ and Rust.

Also, Python does not natively support multiple-dispatch (*we'll see what that is later on*). We have to install an external library to add this functionality.

§

# What makes Julia so special?

As stated in the TEDx MIT Talk we mentioned earlier, Julia is fast; it joined the Petaflop Club in 2017 with the `Celeste.jl` implementation. If we take a moment to realize that the other languages currently belonging to this club are C, C++ and Fortran, it really leaves something to think about in terms of where we're heading.

Apart from the fast-performing aspect, Julia is easy to write and read, and this is directly related to the point above; most low-level languages are hard to write simply because of their nature: they are faster-performing, but we also have to be more careful in how we design our programs, requiring a tremendous amount of expertise in computer science and algorithmic design.

As mentioned TEDx MIT Talk:

> The common wisdom for programming languages has always been that we could have an "either" or an "or"; either we can have a programming language that's easy to program, but we'll pay the price (*somehow the programs will execute much more slowly, and we will lose out on performance*). The other possibility, a much more complicated endeavor, involves much higher programming expertise, and only then can we get better performance. Julia showed that it wasn't one or the other but that we could have our cake and eat it too.

— *Dr. Alan Edelman*[2]

This quote is fantastic because it summarizes what Julia is all about in a single paragraph: it's fast-performing while at the same time being easy to write and read.

Also, Julia supports a fascinating concept known as multiple dispatch. This feature refers to the fact that a function or method can be dynamically dispatched based on the runtime (*dynamic*) type or, in the more general case, some other attribute of more than one of its arguments. This is extremely useful and adds flexibility to our programs.

We also mentioned that Julia is **dynamically-typed**. This means most of its type checking is performed at runtime instead of compile-time. The nice thing about Julia is that this feature is optional since it can statically type the data types, meaning more robustness, runtime speed, and safer data handling.

Having enumerated all these fabulous characteristics, why did Dr. Edelman refer to Julia as a "*Language to heal the planet*"? Well, let's unravel this step-by-step:

## 1. Efficient and easy-to-read syntax

Julia features math-friendly syntax as it was designed with the scientific community in mind using R, Matlab, Octave, and so on. Its syntax is very much like Python's and R's in that it's effortless to read and presents a pseudocode-like writing structure. As we have mentioned, it even supports Unicode characters so that a mathematical expression can be written using actual symbols.

This makes Julia a language whose programs can be understood by various academics and professionals in the field of scientific experimenting and mathematical modelling while keeping the characteristic low-level language performance.

Apart from the readability advantages, Julia's language semantics allow a well-written Julia program to give more opportunities to the compiler to generate efficient code and memory layouts.

# 2. Fast-performing

Sometimes we think of energy resources as given. Each morning we're woken up by a phone powered by electricity. We have breakfast and use all kinds of kitchen appliances powered mainly by electricity. We then take a shower whose water flow is powered by a hydraulic system, which in turn is powered by electricity. And then, we sit at our desks, turn on our computers powered by electricity, and start programming. Each program execution requires resources; the CPU processes billions of operations per second, and the computer fans spin to dissipate heat.

We might think that our energy consumption is limited since we're running programs on a personal computer, but what if we scale that to a production environment where racks of servers are live 24/7, processing huge amounts of data each second? The largest data centres require more than 100 megawatts of power capacity, which is enough to power roughly 80,000 U.S. households. [3]

Now, let us think of Machine Learning, a brilliant but computationally-expensive invention. Some algorithms' actual training processes often result in high wattage, primarily if we use Deep Learning models on GPU. Sometimes, training a large model could involve Terabytes or even Petabytes of information in the form of inputs.

For example, OpenAI trained its GPT-3 model on 45 terabytes of data. To train the final version of MegatronLM, a language model similar to but smaller than GPT-3, Nvidia ran 512 V100 GPUs over nine days. A single V100 GPU can consume between 250 and 300 watts. If we assume 250 watts, then 512 V100 GPUS consumes 128,000 watts, or 128 kilowatts (kW). Running for nine days means the MegatronLM's training costs 27,648-kilowatt hours (kWh). According to the U.S. Energy Information Administration, the average household uses 10,649 kWh annually. Therefore, training the final version of MegatronLM used almost the amount of energy three homes use in a year.[4]

The concept of Environmentally-Responsible programming or Green Coding has recently gained attention because of the exponential growth in data generation and the evident increase in global temperature due to global warming. Particularly, there's one exciting branch called Sustainable Software Design, which aims to combat climate change by designing better algorithms in higher-performing and less computationally expensive languages using greener routines.

Parallel Green Computing computing is one of the techniques that can reduce energy consumption vs single-thread computing. Another parameter affecting energy consumption is how the code is compiled and how much memory is required.

There's a very interesting GitHub repo, greensoftwarelab / Energy-Languages which includes 31 languages and ten routines for each tested language, along with some metrics:

- CPU (*J*)
- GPU (*J*)
- DRAM (*J*)

- Execution Time (*ms*)

Finally, Julia's advantage over other languages is that good performance is not limited to a small subset of "*built-in*" types and operations, and we can write high-level type-generic code that works on arbitrary user-defined types while remaining fast and memory-efficient. Types in languages like Python simply don't provide enough information to the compiler for similar capabilities.

## 3. Parallel computing support

We have already mentioned that Julia includes support for Spark computation. Still, it also natively supports multi-threading on CPU and distributed computing using the `Distributed` standard library.

Julia also supports native GPU computing. There is a rich ecosystem of Julia packages that target GPUs. The JuliaGPU.org website provides a list of capabilities, supported GPUs, related packages and documentation.

Making correct use of parallel computing can increase energy and runtime efficiency.

§

# To whom is Julia targeted?

Although Julia is defined as a general-purpose programming language, it shines on **scientific computing tasks**; it has a wide variety of scientific libraries covering linear algebra, differential and integral calculus, advanced probabilistic and statistical modelling, discrete and continuous mathematics, analytical geometry and much more. There's even a specific monospaced font called JuliaMono, tailored explicitly for scientific computing; we can use font ligatures to produce beautifully written mathematical expressions.

We can also use actual Unicode characters to assign variables:

### Code

```
α, β = [1, 2]

println(α)
println(β)
```

### Output

```
1
2
```

### Code

```
if α ≥ β
    println("α is greated than β")
else
    println("β is greated than α")
end
```

### Output

```
β is greated than α
```

If we would like to spice things up a little bit, we could use emojis to assign variables:

### Code

```
👴 = "We "
❤ = "love "
⭕ = "Julia"

println(👴, ❤, ⭕)
```

### Output

```
We love Julia
```

Or the other way around:

### Code

```
a = "🚀"
b = "👩‍🚀"
c = "🟡"

println(a, b, c)
```

### Output

```
🚀👩‍🚀🟡
```

Fun, right? Well, we're just getting started. This is a sample of what can be done with Unicode characters inside Julia.

Julia also excels at Data Science tasks; it has multiple libraries covering advanced and dynamic visualization, data querying, data processing and manipulation, problem optimization, machine learning, and more. It's supported by JupyterLab, and even has its own notebook environment available as a package: `Pluto.jl`.

But that's not all because it's also extremely high performing and capable of parallel computing; it can generate native code for GPUs and directly integrates with the Spark ecosystem.

All of these aspects make Julia extremely versatile and fun to work with.

§

# Side-by-side comparison between similar languages

| Feature | Julia | Python | R |
|---|---|---|---|
| Type | Just-in-time (JIT) compiled | Interpreted | Functional and interpreted |
| Runtime Speed (*for each search test*) | 0.32 s | 40.75 s (with built-in lists) | 21.94 s |
| Community | Smaller community than R. | Vast community with 15.7 million+ developers as of 2022 | Smaller community than Python's with 2 million+ users as of 2023. |
| Array Indexing | 1-indexed array start | 0-indexed array start | 1-indexed array start |
| Libraries | Over 7,400 libraries as of 2022. | Over 137,000 libraries as of 2022. | Over 11,794 libraries as of 2022. |
| Typing | High-performance dynamically typed, with option to statically type. | Dynamically-typed | Strongly but dynamically typed |
| Multiple-dispatch support | Native | With packages | Only by using the S4 system |
| Downloaded | 35 million times as of Jan 1, 2022 | NA | NA |

*Table 1. Comparison Table Between Julia, Python & R*

The main drawbacks with Julia right now are Python's main strengths; its massive adoption rate, a huge active community, and the vast number of libraries currently available. The leading cause is that Julia, released in 2012, is still a new language while Python, released in 1991, has been around for roughly triple the time.

The adoption rate could increase in the future since Julia is gradually receiving more attention; the more people using the language, the higher the adoption rate and, consequently, the better the community support and package coverage.

For a more detailed performance analysis including the C language as a benchmark, we can refer to this informative set of tests run by Daniel Moura published on Data Science Central:

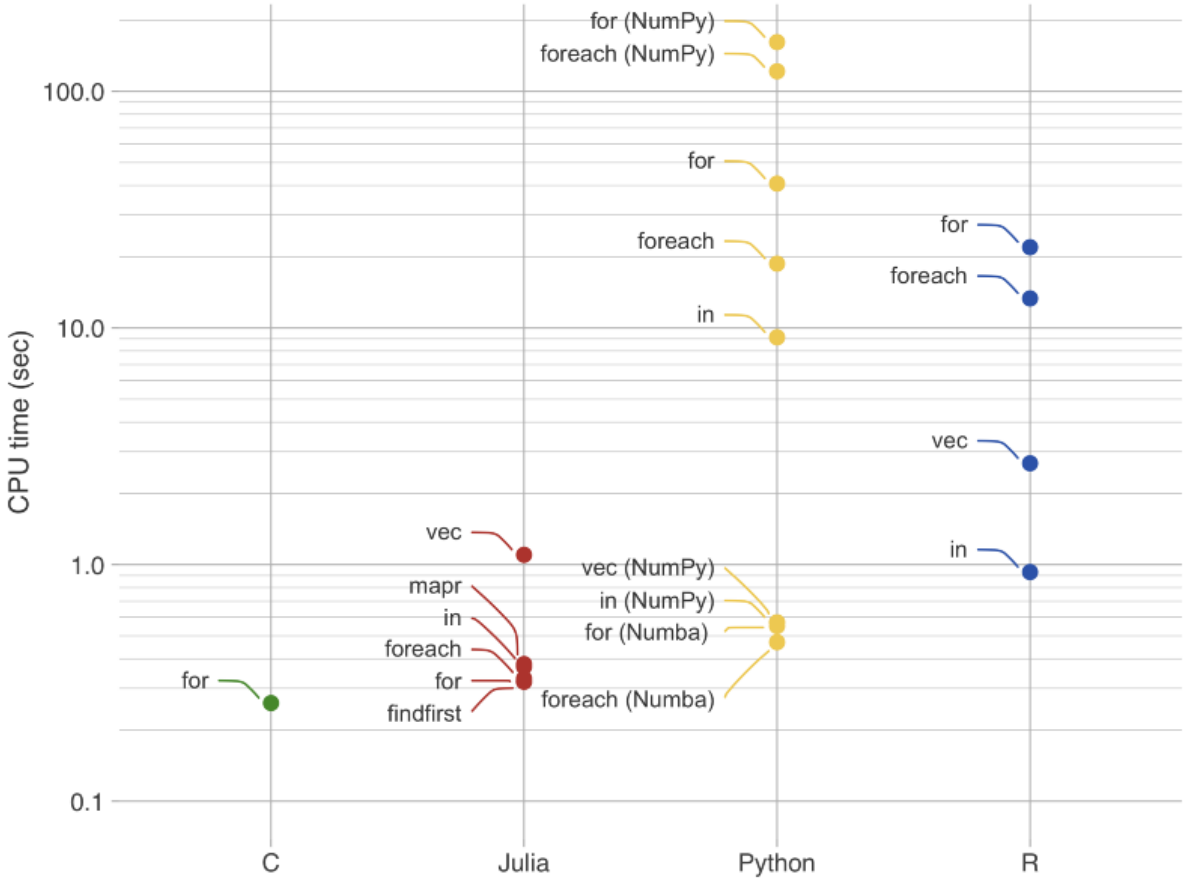


Figure 01: CPU Time Comparison Of Common Routines Between C, Julia, Python And R

The following routines were tested:

- Built-in functions/operators (*in, findfirst*).
- Vectorized (*vec*).
- Map-reduce (*mapr*).
- Loops (*for, foreach*).

We can see that Julia is close to C independently on the implementation. The only routines lagging are vectorized operations, with Python presenting faster execution times when using NumPy.

§

# Time to learn Julia

Now that we have a general understanding of what Julia is, there are multiple free online resources to learn it and become proficient:

- **Julia From Scratch**: A hands-on, step-by-step guide to getting started with Julia, including the Julia Language installation, installation and usage of a helpful VS Code Extension, installation of JuliaMono type font, using the Julia REPL, creating an environment, installing packages, basic data structures and data types, and basic operations.

- **Julia Community**: A fantastic index of resources ranging from YouTube Channels to Slack chats & Discord Servers to Julia's Forem.
- **Julia's Forem**: The best place to write, share, and discuss Julia content.
- **Julia Academy, Julia Programming for Nervous Beginners**: A great resource to get familiarized with Julia from scratch, with no previous programming experience required..
- **Julia Academy, Introduction to Julia (for programmers)**: A 10-part course to get started with Julia from scratch, provided we already know some other language like Python or R beforehand.
- **MIT Open Courseware, 18.S190**: A Julia-flavored iteration of MIT's 6.00.1x for a more complete and demanding programming experience.
- **The Julia Programming Language YouTube Channel**: Contains a wide variety of videos ranging from beginner-friendly to more advanced topics.

§

# Conclusions

In this segment, we performed a general overview of the Julia programming language. We also mentioned some of its advantages over similar languages, such as Python and R, and introduced a small set of features that Julia natively supports. We also discussed some downsides when working with Julia and mentioned how these could change when it's more widely adopted. Finally, we provided some helpful next steps for those interested in learning Julia programming.

Today, Julia is seen as a young and more bleeding-edge, experimental language that, even though it has enjoyed much attention from academics and professionals in recent years, is still in an early stage and requires a broader adoption for it to become supported as an industry standard, just as Python is right now.

Hopefully, more people will get to know this fresh new approach to reimagining what a programming language can be.

§

# References


- Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs
- Energy Innovation, How Much Energy Do Data Centers Really Use?
- Tech Target, Energy consumption of AI poses environmental problems
- CodeAcademy, 6 Ways To Be A More Environmentally Friendly Programmer
- CodeAcademy, How Sustainable Software Design Combats Climate Change — & How To Get Involved
- Julia, Parallel Computing
- Julia FAQ
- Data Science Central, R vs. Python vs. Julia: How easy it is to write efficient code?
- Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman, Why We Created Julia


§

# Copyright

§

1. Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman, Why We Created Julia↩
2. A programming language to heal the planet together: Julia↩
3. Energy Innovation, How Much Energy Do Data Centers Really Use?↩
4. Tech Target, Energy consumption of AI poses environmental problems↩