Julia vs. Python: Which is best for data science?

Python has turned into a data science and machine learning mainstay, while Julia was built from the ground up to do the job

By Serdar Yegulalp
Senior Writer, InfoWorld
MAY 27, 2020 3:00 AM PDT

Among the many use cases Python covers, data analytics has become perhaps the biggest and most significant. The Python ecosystem is loaded with libraries, tools, and applications that make the work of scientific computing and data analysis fast and convenient.

But for the developers behind the Julia language — aimed specifically at "scientific computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing"—Python isn't fast or convenient *enough*. Julia aims to give scientists and data analysts not only fast and convenient development, but also blazing execution speed.

What is the Julia language?

Created in 2009 by a four-person team and unveiled to the public in 2012, Julia is meant to address the shortcomings in Python and other languages and applications used for scientific computing and data processing. "We are greedy," they wrote. They wanted more:

> *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, 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?)*

Here are some of the ways Julia implements those aspirations:

- **Julia is compiled, not interpreted.** For faster runtime performance, Julia is just-in-time (JIT) compiled using the LLVM compiler framework. At its best, Julia can approach or match the speed of C.
- **Julia is interactive.** Julia includes a REPL (read-eval-print loop), or interactive command line, similar to what Python offers. Quick one-off scripts and commands can be punched right in.
- **Julia has a straightforward syntax.** Julia's syntax is similar to Python's—terse, but also expressive and powerful.
- **Julia combines the benefits of dynamic typing and static typing.** You can specify types for variables, like "unsigned 32-bit integer." But you can also create hierarchies of types to allow general cases for handling variables of specific types—for instance, to write a function that accepts integers without specifying the length or signing of the integer. You can even do without typing entirely if it isn't needed in a particular context.
- **Julia can call Python, C, and Fortran libraries.** Julia can interface directly with external libraries written in C and Fortran. It's also possible to interface with Python code by way of the PyCall library, and even share data between Python and Julia.
- **Julia supports metaprogramming.** Julia programs can generate other Julia programs, and even modify their own code, in a way that is reminiscent of languages like Lisp.
- **Julia has a full-featured debugger.** Julia 1.1 introduced a debugging suite, which executes code in a local REPL and allows you to step through the results, inspect

variables, and add breakpoints in code. You can even perform fine-grained tasks like stepping through a function generated by code.

Related video: How Python makes programming easier

Perfect for IT, Python simplifies many kinds of work, from system automation to working in cutting-edge fields like machine learning.

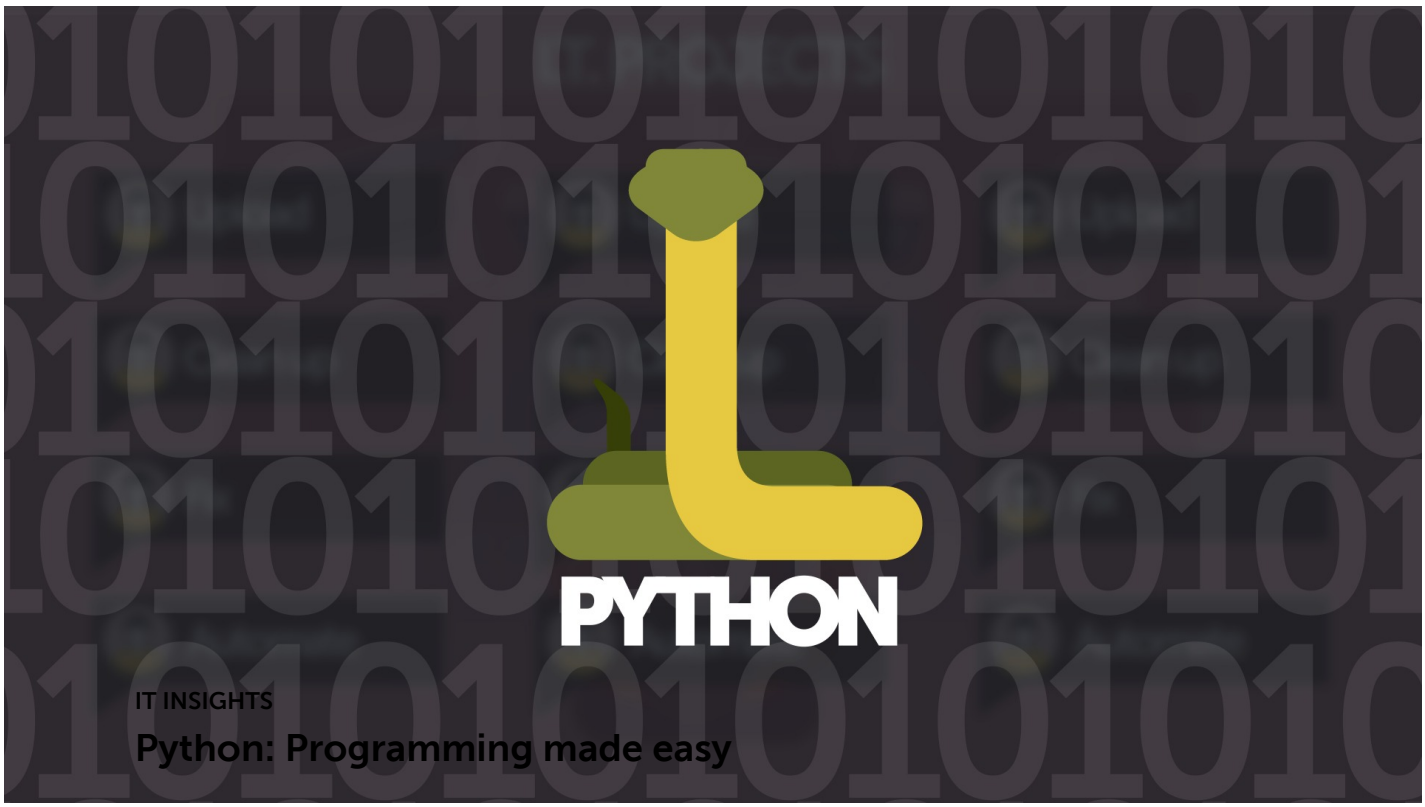

Table of Contents

Julia vs. Python: Julia language advantages

Julia was designed from the start for scientific and numerical computation. Thus it's no surprise that Julia has many features advantageous for such use cases:

- **Julia is fast.** Julia's JIT compilation and type declarations mean it can routinely beat "pure," unoptimized Python by orders of magnitude. Python can be *made* faster by way of external libraries, third-party JIT compilers (PyPy), and optimizations with tools like Cython, but Julia is designed to be faster right out of the gate.

- **Julia has a math-friendly syntax.** A major target audience for Julia is users of scientific computing languages and environments like Matlab, R, Mathematica, and Octave. Julia's syntax for math operations looks more like the way math formulas are written outside of the computing world, making it easier for non-programmers to pick up on.
- **Julia has automatic memory management.** Like Python, Julia doesn't burden the user with the details of allocating and freeing memory, and it provides some measure of manual control over garbage collection. The idea is that if you switch to Julia, you don't lose one of Python's common conveniences.
- **Julia offers superior parallelism.** Math and scientific computing thrive when you can make use of the full resources available on a given machine, especially multiple cores. Both Python and Julia can run operations in parallel. However, Python's methods for parallelizing operations often require data to be serialized and deserialized between threads or nodes, while Julia's parallelization is more refined. Further, Julia's parallelization syntax is less top-heavy than Python's, lowering the threshold to its use.
- **Julia is developing its own native machine learning libraries.** Flux is a machine learning library for Julia that has many existing model patterns for common use cases. Since it's written entirely in Julia, it can be modified as needed by the user, and it uses Julia's native just-in-time compilation to optimize projects from inside out.

Julia vs. Python: Python advantages

Although Julia is purpose-built for data science, whereas Python has more or less evolved into the role, Python offers some compelling advantages to the data scientist. Some of the reasons "general purpose" Python may be the better choice for data science work:

- **Python uses zero-based array indexing.** In most languages, Python and C included, the first element of an array is accessed with a zero—e.g., `string[0]` in Python for the first character in a string. Julia uses 1 for the first element in an array. This isn't an arbitrary decision; many other math and science applications, like Mathematica,

use 1-indexing, and Julia is intended to appeal to that audience. It's possible to support zero-indexing in Julia with an experimental feature, but 1-indexing by default may stand in the way of adoption by a more general-use audience with ingrained programming habits.

- **Python has less startup overhead.** Python programs may be slower than Julia programs, but the Python runtime itself is more lightweight, and it generally takes less time for Python programs to start and deliver first results. Also, while JIT compilation speeds up execution time for Julia programs, it comes at the cost of slower startup. Much work has been done to make Julia start faster, but Python still has the edge here.
- **Python is mature.** The Julia language is young. Julia has been under development only since 2009, and has undergone a fair amount of feature churn along the way. By contrast, Python has been around for almost 30 years.
- **Python has far more third-party packages.** The breadth and usefulness of Python's culture of third-party packages remains one of the language's biggest attractions. Again, Julia's relative newness means the culture of software around it is still small. Some of that is offset by the ability to use existing C and Python libraries, but Julia needs libraries of its own to thrive. Libraries like Flux and Knet make Julia useful for machine learning and deep learning, but the vast majority of that work is still done with TensorFlow or PyTorch.
- **Python has millions of users.** A language is nothing without a large, devoted, and active community around it. The community around Julia is enthusiastic and growing, but it is still only a fraction of the size of the Python community. Python's huge community is a huge advantage.
- **Python is getting faster.** Aside from gaining improvements to the Python interpreter (including improvements to multi-core and parallel processing), Python has become easier to speed up. The mypyc project translates type-annotated Python into native C extensions. Python 3.11 typically runs four-fold performance improvements, and often much more for pure mathematical operations.

Serdar Yegulalp is a senior writer at InfoWorld, focused on machine learning, containerization, devops, the Python ecosystem, and periodic reviews.