The Python/Jupyter ecosystem: today's problem-solving environment for computational science

Lorena A. Barba[1]

[1]Affiliation not available

April 19, 2021

The March/April issue of CiSE's inaugural year (1999) carried an essay by eminent computer science professor John R. Rice (who at the time was area editor for Software, together with Matlab inventor Cleve Moler) titled A Perspective on Computational Science in the 21st Century (Rice, 1999). In it, he looked at the development directions for the future of computational science and engineering, and threaded across these was what he called "problem-solving environments." This routine-sounding term hides an ambitious vision, for the time. Rice imagined a software system for tackling problems within a science domain without all the agonizing toil of programming by hand every solution method. He and Ronald Boisvert had a previous article (1996) explaining the idea in more detail (Rice & Boisvert, 1996). A problem-solving environment would include a collection of mathematical and domain-specific software libraries, offer (semi-)automatic selection of solution method for a given problem, help check the problem formulation, display or assess the correctness of solutions, allow extensibility to add new methods, and even manage the overall computational process. They envisaged an environment that could be "all things to all people," meaning: it is effective when solving simple or complex problems, it supports rapid prototyping and detailed analysis, and it can be used both in introductory teaching and in productive research at the edges of knowledge. An ideal problem-solving environment would even make decisions for the user by means of an integrated knowledge base. What fabulous ambition!

Prof. Rice led a research group at Purdue University that worked to develop early problem-solving environments. The Ellpack system for solving elliptic boundary value problems, developed in the early 1980s, included dozens of software modules implementing solution methods and a descriptive language to formulate problems (today, we might call it a domain-specific language, DSL). For example, the line: `equation. uxx + uyy + 3 * ux - 4 * u = exp(x+y) * sin(pi * x)` would be used in an Ellpack program for defining the differential equation to be solved. Similarly expressive statements would define the boundary conditions, and the grid parameters to discretize the domain (a full example at https://www.cs.purdue.edu/ellpack/example.html). Later versions of the system offered parallel solvers and a graphical user interface (screenshots of the Ellpack system from Prof. John R. Rice's website at Purdue can be found in the Internet Archive https://web.archive.org/web/19990506040312/https://www.cs.purdue.edu/research/cse/index.html).

While Ellpack was licensed by Purdue University for a modest yearly fee, this project did not branch off commercially or otherwise. Perhaps a few hundred copies were distributed, mostly for use in university settings, and the project wound down by the early 2000s. By contrast, three commercial software packages for high-productivity scientific and engineering computation—Maple, Mathematica, and Matlab—had by then become very popular (Chonacky & Winch, 2005). These systems continue to be widely used in education, industry, and government settings. Their purchase price and proprietary implementations, however, led many champions of open-source software to conceive alternatives, oftentimes closely imitating their functionality.

In March/April 2011, twelve years after Prof. Rice's *Perspective* essay, CiSE ran a special issue on Python

for Scientific Computing, showcasing a maturing stack of tools and a highly productive environment for researchers. The issue included one of the most-widely cited articles in the history of the magazine, discussing the high-level multidimensional array structure at the core of NumPy (van der Walt et al., 2011). By this time, the scientific community had expanded Python for its purposes, and the four keystone libraries had been put in place in the first half of the decade:

1. **SciPy** was consolidated as a standard collection of modules for common mathematical and statistical functions.
2. The first version of **IPython**, an enhanced interactive shell for Python, was created by Fernando Pérez.
3. **Matplotlib**, the rich 2D visualization and now standard Python plotting library, was released by John Hunter.
4. Travis Oliphant created **NumPy** from a rewrite of the early Python array library Numeric, adding functionality from the competing array package called `numarray`.

CiSE had previously featured the developing Python support for scientific workflows in an issue organized by Paul F. Dubois, who was the project lead for Numeric from 1997 to 2002. Paul was an editor for Computer in Physics (which got merged into CiSE) since 1993, and joined CiSE with its founding. He wrote and edited for the Scientific Programming department until 2006, and continued with the column "Café Dubois" until 2008. The issue he led included the Hunter piece on Matplotlib (Hunter, 2007), the Pérez and Granger article about IPython (Perez & Granger, 2007), and Travis Oliphant's general overview of the Python language and its extensions with NumPy and SciPy (Oliphant, 2007). Other articles in the issue highlight applications in various science contexts: space observation, systems biology, robotics, nanophotonics, and more. An author team from the Simula Research Laboratory in Norway discussed new Python tooling for solving partial differential equations with finite element methods in what was the early development of the Fenics project (`http://www.fenics.org/`) (Mardal et al., 2007). This work heralded the compelling combination of symbolic mathematics and code generation, which Ellpack anticipated.

By the time of the 2011 special issue, the scientific Python ecosystem had gained **SymPy**, the symbolic mathematics and computer algebra system, and **Cython** (Behnel et al., 2011), a solution to compile portions of a Python program to obtain faster execution. An important contribution for lowering the bar to adoption of scientific Python was the Enthought Python Distribution, which was free for academic users. It relieved users from the laborious installation of every library individually. The SciKits (`https://www.scipy.org/scikits.html`), or SciPy Toolkits, began appearing on the scene: `scikit-learn` (`https://scikit-learn.org/`), for predictive data analysis and machine learning (Pedregosa et al., 2011), and `scikit-image` (https://scikit-image.org), for image processing (van der Walt et al., 2014), have since become essential in many scientific contexts. And a new, powerful and flexible Python library for analysis and manipulation of data in the form of labeled tables and time series, `pandas`, sparked a wave of adoption for statistical modeling (McKinney, 2011). The state of the ecosystem at that time was expertly reviewed by Fernando Pérez, Brian Granger, and John Hunter (Perez et al., 2011). But the last ten years have seen an explosion of innovation, beyond the wildest dreams of these leaders.

In the March/April 2021 issue, CiSE is proud to feature several articles showcasing **Jupyter** in computational science. The showpiece is an invited article by Jupyter co-founders Brian Granger and Fernando Pérez (Granger & Perez, 2021). They shift the focus of our conversation about problem-solving environments to the human angle: the researcher interactively exploring a scientific question or analyzing data, and the community of people collaborating and advancing their field. Jupyter derives from the IPython Notebook, a browser-based application to compose documents that add computable content to all the other kinds of content that a browser can display: formatted text, images, video, equations, etc. It is a concept inspired by the Mathematica notebook interface, introduced in 1988, and translated to the Python world by the Sage Notebook, starting in 2006. The powerful idea of the Sage Notebook that lives on in Jupyter is connecting a web application serving as a graphical user interface—where the user's text inputs and the computational outputs are shown—to a back-end that runs the Python interpreter. While the server is running, the state

is available to continue interactively computing. On shutting it down, the notebook document can be saved thereby preserving the inputs and outputs, together with any text and multi-media content added by the user. Jupyter took these ideas into the modern web era by employing open formats, protocols and applications that work equally on a laptop and on remote cloud or HPC systems. The communication protocol between the web application and the back-end is language-agnostic, which quickly led to Notebook support for other languages, like Julia and R (and the new name Jupyter). Dozens of so-called Jupyter kernels now allow computing in many different languages (including interactive use of Fortran and C++), though Python continues to be the most popular by far. Serving Jupyter to multiple-users in a corporation or university became possible with **JupyterHub** (`https://jupyter.org/hub`), which removes the need for users to install any software on their local machine while delivering a uniform environment to them. This provides an ideal solution for academic settings, where large-scale computing and data science education initiatives have always been hampered by individual students' software and hardware needs. Jupyter was rapidly adopted by tech giants like Google, Amazon and Netflix; by financial behemoths like Bloomberg; by NASA and the LIGO collaboration, which released Jupyter notebooks with the analysis that proved the existence of gravitational waves (`https://www.gw-openscience.org/events/GW150914/`); and by computing and data science educators everywhere. On being awarded the ACM Software System Award in 2017, the citation reads that the tools of Project Jupyter "have become a de facto standard for data analysis in research, education, journalism, and industry" (`https://awards.acm.org/award_winners/perez_9039634`). Like Granger and Pérez note in the previous issue of CiSE, Jupyter now has many millions of users worldwide and many thousands of organizations use Jupyter in their day-to-day operations. They explain the project's success by virtue of being a tool of thought, a new medium for communication (via computational narratives), and a community of practice.

The traction of Python in the context of scientific computing used to be ascribed to its effectiveness as a "glue language" (easy interoperability with other languages like C/C++ and Fortran), its full set of scientific libraries (NumPy, SciPy, SymPy, Matplotlib), and its notoriously shallow learning curve. But its downside was proclaimed to be performance: being an interpreted, dynamically typed language meant execution would be much slower than low-level implementations in compiled languages. In the past decade, solutions to this performance penalty have been multiplying. At first, researchers would identify performance bottlenecks in their Python applications, re-write the relevant portions of code in C/C++, and use `swig` to wrap this code and interface it with the main program in Python (Beazley, 2003). Later, they gained Cython for numerical loops that cannot be expressed in NumPy operations. Cython compiles Python code extended with type declarations, generating code that can take advantage of the optimizations provided by the C compiler and achieve high-performance on those hotspots that dominate runtime. Parallel distributed computing with Python programs using message passing became available with the `mpi4py` package, which supports communication of Python data types and definition of communicator objects according to the MPI specification. A related package, `petsc4py`, provides access to the algorithms and data structures of the PETSc library (`https://www.mcs.anl.gov/petsc/`). It allows assembling distributed vectors and sparse matrices, solving linear systems of equations with Krylov iterative methods, and solving nonlinear equations with Newton methods—all of which are core needs in many scientific applications such as those using finite element methods (Dalcin et al., 2011). Access to many-core hardware from Python programs was made possible with run-time code generation via **PyCUDA** and **PyOpenCL** (Klöckner et al., 2012). These tools pioneered the pursuit of high-performance computing with Python, at the cost of increasingly specialized programmer effort. In the last few years, however, a new wave of innovation in scientific Python has sprung from the widespread use of Python in industry settings. Fortunately, much of this innovation has occurred under the open-source model of development and licensing pervading the Python ecosystem.

New tools to defeat the performance penalty include **Numba** (https://numba.pydata.org), which accelerates Python code by just-in-time compilation of functions to optimized machine code. The programmer only needs to add decorators, e.g., `@numba.jit(nopython=True)`, ahead of the function definition and Numba will compile the function at runtime, and it will subsequently run without involving the Python interpreter. Numba can also compile a subset of Python code to CUDA kernels for execution on Nvidia GPUs (this

could possibly be the easiest way to exploit GPUs for high-throughput computations). And the new parallel computing library for Python, **Dask** (https://dask.org), offers distributed data structures that stand in for NumPy arrays and `pandas` dataframes, scalable machine learning integrating with `scikit-learn`, and high-level tools for scheduling and distributing tasks in a cluster. This allows transitioning to parallel and distributed clusters with very little code rewrite. Numba and Dask—like NumPy, Matplotlib, SciPy, SymPy, pandas and Jupyter—are fiscally sponsored projects of **NumFOCUS** (https://numfocus.org), a 501(c)(3) public charity in the United States (I served on its Board of Directors from 2014 to 2020). This means that they are community governed projects developing software under a standard public license, and they both raise funds for their development and receive services (e.g., financial administration, legal support) via a non-profit. The core developers, maintainers, and users of these projects come together at conferences where they give technical talks and offer tutorials, participate in online conversations using discussion boards and code-repository issue trackers, and build value together tenaciously.

Technology companies often participate actively and benevolently in this activity. NumFOCUS receives corporate sponsorships that benefit the projects, certainly, but another impactful way companies contribute is by allowing or assigning their paid employees to work on the open-source projects of this large ecosystem. Numba and Dask were started at Anaconda, Inc., an Austin-based software and consulting company that also created the hugely popular Anaconda Python distribution. The explosion of machine learning and AI saw the tech giants developing and releasing open-source Python libraries for these applications: Google's **TensorFlow** (https://www.tensorflow.org) and Facebook's **PyTorch** (https://pytorch.org) being the most notable. Both libraries provide a Python interface while executing core operations in compiled languages and also CUDA for access to Nvidia GPUs. Google also developed a Jupyter-based cloud notebook, **Colab** (`http://colab.research.google.com/`), providing a hosted solution to run TensorFlow with access to GPUs and Google's own TPUs (Tensor Processing Units). The Japanese company Preferred Networks led the development of **CuPy**, a NumPy-like open-source library of matrix functions for Nvidia GPUs. And Nvidia embraced the PyData ecosystem creating its RAPIDS AI team (https://rapids.ai) to develop open-source libraries like **cuDF**, with a pandas-like API for manipulating dataframes on GPUs, and the machine learning library **cuML**, with a growing set of algorithms from `scikit-learn`. The data science community can now build high-performance workflows in Python and Jupyter, taking advantage of the latest hardware on cloud resources.

Python and Jupyter are also playing an increasingly important role in high-performance computing. Rollin Thomas and Shreyas Cholia, in the previous issue of CiSE, explained how the National Energy Research Scientific Computing Center (NERSC) began their voyage to Jupyter five years ago. They tell us that today about 25% of user interactions with the Cori supercomputer are via JupyterHub, and several scientific workflows have been made more user-friendly while at the same time enjoying parallel speed-ups with Dask. And they conclude: "Jupyter is quickly becoming the entry point to HPC for a growing class of users." (Thomas et al., 2021). In the next issue, CiSE will feature several scientific applications that embody the powerful idea of combining high researcher productivity via Python and high performance through code generation, just-in-time compilation, or exploiting advanced Python libraries. John Rice's perspective of a scientific problem-solving environment for the 21st century may finally be realized, as long as we continue to engage and support the thriving communities of practice of the Python/Jupyter ecosystem. *Don't miss our next issue!*

References

A perspective on computational science in the 21st Century. (1999). *Computing in Science & Engineering*, *1*(2), 14–16. https://doi.org/10.1109/5992.753042

From scientific software libraries to problem-solving environments. (1996). *IEEE Computational Science and Engineering*, *3*(3), 44–53. https://doi.org/10.1109/99.537091

Maple Mathematica, and Matlab: the 3M's without the tape. (2005). *Computing in Science & Engineering*, *7*(1), 8–16. https://doi.org/10.1109/mcse.2005.18

The NumPy Array: A Structure for Efficient Numerical Computation. (2011). *Computing in Science & Engineering*, *13*(2), 22–30. https://doi.org/10.1109/mcse.2011.37

Matplotlib: A 2D Graphics Environment. (2007). *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/mcse.2007.55

IPython: A System for Interactive Scientific Computing. (2007). *Computing in Science & Engineering*, *9*(3), 21–29. https://doi.org/10.1109/mcse.2007.53

Python for Scientific Computing. (2007). *Computing in Science & Engineering*, *9*(3), 10–20. https://doi.org/10.1109/mcse.2007.58

Using Python to Solve Partial Differential Equations. (2007). *Computing in Science & Engineering*, *9*(3), 48–51. https://doi.org/10.1109/mcse.2007.64

Cython: The Best of Both Worlds. (2011). *Computing in Science & Engineering*, *13*(2), 31–39. https://doi.org/10.1109/mcse.2010.118

Scikit-learn: Machine Learning in Python. (2011). *The Journal of Machine Learning Research*, *12*, 2825–2830. https://www.jmlr.org/papers/v12/pedregosa11a.html

scikit-image: image processing in Python. (2014). *PeerJ*, *2*, e453. https://doi.org/10.7717/peerj.453

pandas: a foundational Python library for data analysis and statistics. (2011). In *Proc. Workshop Python High Perform. Sci. Comput. (PyHPC)*. https://www.dlr.de/sc/en/Portaldata/15/Resources/dokumente/pyhpc2011/submissions/pyhpc2011_submission_9.pdf

Python: An Ecosystem for Scientific Computing. (2011). *Computing in Science & Engineering*, *13*(2), 13–21. https://doi.org/10.1109/mcse.2010.119

Jupyter: Thinking and Storytelling With Code and Data. (2021). *Computing in Science & Engineering*, *23*(2), 7–14. https://doi.org/10.1109/mcse.2021.3059263

Automated scientific software scripting with SWIG. (2003). *Future Generation Computer Systems*, *19*(5), 599–609. https://doi.org/10.1016/s0167-739x(02)00171-1

Parallel distributed computing using Python. (2011). *Advances in Water Resources*, *34*(9), 1124–1139. https://doi.org/10.1016/j.advwatres.2011.04.013

PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. (2012). *Parallel Computing*, *38*(3), 157–174. https://doi.org/10.1016/j.parco.2011.09.001

Interactive Supercomputing With Jupyter. (2021). *Computing in Science & Engineering*, *23*(2), 93–98. https://doi.org/10.1109/mcse.2021.3059037