

Array Programming with NumPy

Charles R. Harris¹, K. Jarrod Millman^{2,3,4,*}, Stéfan J. van der Walt^{5,2,4,*}, Ralf Gommers^{6,*}, Pauli Virtanen⁷, David Cournapeau⁸, Eric Wieser⁹, Julian Taylor¹⁰, Sebastian Berg⁴, Nathaniel J. Smith¹¹, Robert Kern¹², Matti Picus⁴, Stephan Hoyer¹³, Marten H. van Kerkwijk¹⁴, Matthew Brett^{2,15}, Allan Haldane¹⁶, Jaime Fernández del Río¹⁷, Mark Wiebe^{18,19}, Pearu Peterson^{6,20,21}, Pierre Gérard-Marchant^{22,23}, Kevin Sheppard²⁴, Tyler Reddy²⁵, Warren Weckesser⁴, Hameer Abbasi⁶, Christoph Gohlke²⁶, and Travis E. Oliphant⁶

¹Independent Researcher, Logan, Utah, USA

²Brain Imaging Center, University of California, Berkeley, Berkeley, CA, USA

³Division of Biostatistics, University of California, Berkeley, Berkeley, CA, USA

⁴Berkeley Institute for Data Science, University of California, Berkeley, Berkeley, CA, USA

⁵Applied Mathematics, Stellenbosch University, Stellenbosch, South Africa

⁶Quansight LLC, Austin, TX, USA

⁷Department of Physics and Nanoscience Center, University of Jyväskylä, Jyväskylä, Finland

⁸Mercari JP, Tokyo, Japan

⁹Department of Engineering, University of Cambridge, Cambridge, UK

¹⁰Independent Researcher, Karlsruhe, Germany

¹¹Independent Researcher, Berkeley, CA, USA

¹²Enthought, Inc., Austin, TX, USA

¹³Google Research, Mountain View, CA, USA

¹⁴Department of Astronomy & Astrophysics, University of Toronto, Toronto, ON, Canada

¹⁵School of Psychology, University of Birmingham, Edgbaston, Birmingham, UK

¹⁶Department of Physics, Temple University, Philadelphia, PA, USA

¹⁷Google, Zurich, Switzerland

¹⁸Department of Physics and Astronomy, The University of British Columbia, Vancouver, BC, Canada

¹⁹Amazon, Seattle, Washington, USA

²⁰Independent Researcher, Saue, Estonia

²¹Department of Mechanics and Applied Mathematics, Institute of Cybernetics at Tallinn Technical University, Tallinn, Estonia

²²Department of Biological and Agricultural Engineering, University of Georgia, Athens, GA

²³France-IX Services, Paris, France

²⁴Department of Economics, University of Oxford, Oxford, UK

²⁵CCS-7, Los Alamos National Laboratory, Los Alamos, NM, USA

²⁶Laboratory for Fluorescence Dynamics, Biomedical Engineering Department, University of California, Irvine, Irvine, CA, USA

*millman@berkeley.edu, stefanv@berkeley.edu, ralf.gommers@gmail.com

SUMMARY

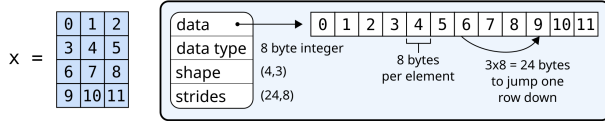
Array programming provides a powerful, compact, expressive syntax for accessing, manipulating, and operating on data in vectors, matrices, and higher-dimensional arrays. NumPy is the primary array programming library for the Python language. It plays an essential role in research analysis pipelines in fields as diverse as physics, chemistry, astronomy, geoscience, biology, psychology, material science, engineering, finance, and economics. For example, in astronomy, NumPy was an important part of the software stack used in the discovery of gravitational waves and the first imaging of a black hole. Here we review how a few fundamental array concepts lead to a simple and powerful programming paradigm for organizing, exploring, and analyzing scientific data. NumPy is the foundation upon which the entire scientific Python universe is constructed. It is so pervasive that several projects, targeting audiences with specialized needs, have developed their own NumPy-like interfaces and array objects. Because of its central position in the ecosystem, NumPy increasingly plays the role of an interoperability layer between these new array computation libraries. NumPy's API and its role as interoperability layer provides a flexible framework for the next decade of scientific and industrial analysis.

1 Two Python array packages existed before NumPy. The
2 Numeric package began in the mid-1990s and provided an
3 array object and array-aware functions in Python, written in
4 C, and linking to standard fast implementations of linear algebra^{1,2}. One of its earliest uses was to steer C++ applications
5 for inertial confinement fusion research at Lawrence Liver-
6 more National Laboratory³. To handle large astronomical

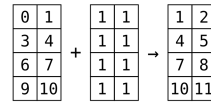
images coming from the Hubble Space Telescope, a reimple-
mentation of Numeric, called Numarray, added support for
structured arrays, flexible indexing, memory mapping, byte-
order variants, more efficient memory use, flexible IEEE error
handling capabilities, and better type casting rules⁴. While
Numarray was highly compatible with Numeric, the two pack-
ages had enough differences that it divided the community,

8
9
10
11
12
13
14

a Data Structure



d Vectorization



g Example

```
In [1]: import numpy as np
In [2]: x = np.arange(12)
In [3]: x = x.reshape(4, 3)

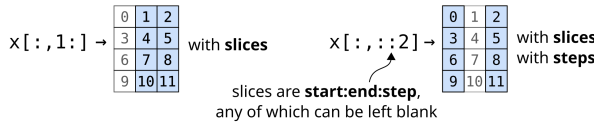
In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

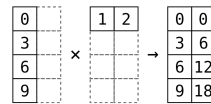
In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[ -4.5,  -4.5,  -4.5],
       [ -1.5,  -1.5,  -1.5],
       [  1.5,   1.5,   1.5],
       [  4.5,   4.5,   4.5]])
```

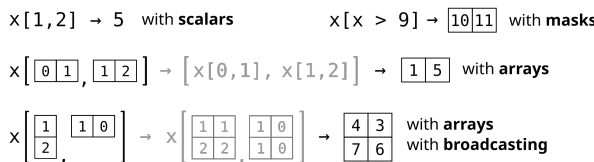
b Indexing (view)



e Broadcasting



c Indexing (copy)



f Reduction

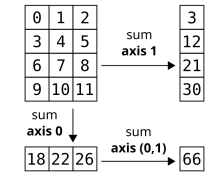


Figure 1. The NumPy array incorporates several fundamental array concepts. **a**, The NumPy array data structure and its associated metadata fields. **b**, Indexing an array with slices and steps. These operations return a *view* of the original data. **c**, Indexing an array with masks, scalar coordinates, or other arrays, so that it returns a copy of the original data. In the bottom example, an array is indexed with other arrays; this broadcasts the indexing arguments before performing the lookup. **d**, Vectorization efficiently applies operations to groups of elements. **e**, Broadcasting in the multiplication of two-dimensional arrays. **f**, Reduction operations act along one or more axes. In this example, an array is summed along select axes to produce a vector, or along two axes consecutively to produce a scalar. **g**, Example NumPy code, illustrating some of these concepts.

until 2005, when NumPy emerged as a “best of both worlds” unification⁵—combining Numarray’s features with Numeric’s performance on small arrays and its rich *C Application Programming Interface* (API).

Now, fifteen years later, NumPy underpins almost every Python library that does scientific or numerical computation^{6–9} including SciPy¹⁰, Matplotlib¹¹, pandas¹², scikit-learn¹³, and scikit-image¹⁴. It is a community-developed, open-source library, which provides a multidimensional Python array object along with array-aware functions that operate on it. Because of its inherent simplicity, the NumPy array is the *de facto* exchange format for array data in Python.

NumPy operates on in-memory arrays using the CPU. To utilize modern, specialized storage and hardware, there has been a recent proliferation of Python array packages. Unlike with the Numarray and Numeric divide, it is now much harder for these new libraries to fracture the user community—given how much work already builds on top of NumPy. However, to provide the ecosystem with access to new and exploratory technologies, NumPy is transitioning into a central coordinating mechanism that specifies a well-defined array programming API and dispatches it, as appropriate, to specialized array implementations.

NumPy arrays

The NumPy array is a data structure that efficiently stores and accesses multidimensional arrays¹⁵, also known as tensors, and enables a wide variety of scientific computation. It consists of a pointer to memory, along with metadata used to interpret the data stored there, notably *data type*, *shape*, and *strides* (Fig. 1a).

The *data type* describes the nature of elements stored in an array. An array has a single data type, and each array element occupies the same number of bytes in memory. Examples of data types include real and complex numbers (of lower and higher precision), strings, timestamps, and pointers to Python objects.

The *shape* of an array determines the number of elements along each axis, and the number of axes is the array’s dimensionality. For example, a vector of numbers can be stored as a one-dimensional array of shape N , while color videos are four-dimensional arrays of shape $(T, M, N, 3)$.

Strides are necessary to interpret computer memory, which stores elements linearly, as multidimensional arrays. It describes the number of bytes to move forward in memory to jump from row to row, column to column, and so forth. Consider, for example, a 2-D array of floating-point numbers with shape $(4, 3)$, where each element occupies 8 bytes in memory. To move between consecutive columns, we need to jump forward 8 bytes in memory, and to access the next row $3 \times 8 = 24$

bytes. The strides of that array are therefore (24, 8). NumPy can store arrays in either C or Fortran memory order, iterating first over either rows or columns. This allows external libraries written in those languages to access NumPy array data in memory directly.

Users interact with NumPy arrays using *indexing* (to access subarrays or individual elements), *operators* (e.g., +, −, × for vectorized operations and @ for matrix multiplication), as well as *array-aware functions*; together, these provide an easily readable, expressive, high-level API for array programming, while NumPy deals with the underlying mechanics of making operations fast.

Indexing an array returns single elements, subarrays, or elements that satisfy a specific condition (Fig. 1b). Arrays can even be indexed using other arrays (Fig. 1c). Wherever possible, indexing that retrieves a subarray returns a *view* on the original array, such that data is shared between the two arrays. This provides a powerful way to operate on subsets of array data while limiting memory usage.

To complement the array syntax, NumPy includes functions that perform *vectorized* calculations on arrays, including arithmetic, statistics, and trigonometry (Fig. 1d). Vectorization—operating on whole arrays rather than their individual elements—is essential to array programming. This means that operations that would take many tens of lines to express in languages such as C can often be implemented as a single, clear Python expression. This results in concise code and frees users to focus on the details of their analysis, while NumPy handles looping over array elements near-optimally, taking into consideration, for example, strides, to best utilize the computer’s fast cache memory.

When performing a vectorized operation (such as addition) on two arrays with the same shape, it is clear what should happen. Through *broadcasting*, NumPy allows the dimensions to differ, while still producing results that appeal to intuition. A trivial example is the addition of a scalar value to an array, but broadcasting also generalizes to more complex examples such as scaling each column of an array or generating a grid of coordinates. In broadcasting, one or both arrays are virtually duplicated (that is, without copying any data in memory), so that the shapes of the operands match (Fig. 1d). Broadcasting is also applied when an array is indexed using arrays of indices (Fig. 1c).

Other array-aware functions, such as `sum`, `mean`, and `maximum`, perform element-by-element *reductions*, aggregating results across one, multiple, or all axes of a single array. For example, summing an n -dimensional array over d axes results in a $(n - d)$ -dimensional array (Fig. 1f).

NumPy also includes array-aware functions for creating, reshaping, concatenating, and padding arrays; searching, sorting, and counting data; and reading and writing files. It provides extensive support for generating pseudorandom numbers, includes an assortment of probability distributions, and performs accelerated linear algebra, utilizing one of several backends such as OpenBLAS^{16,17} or Intel MKL optimized

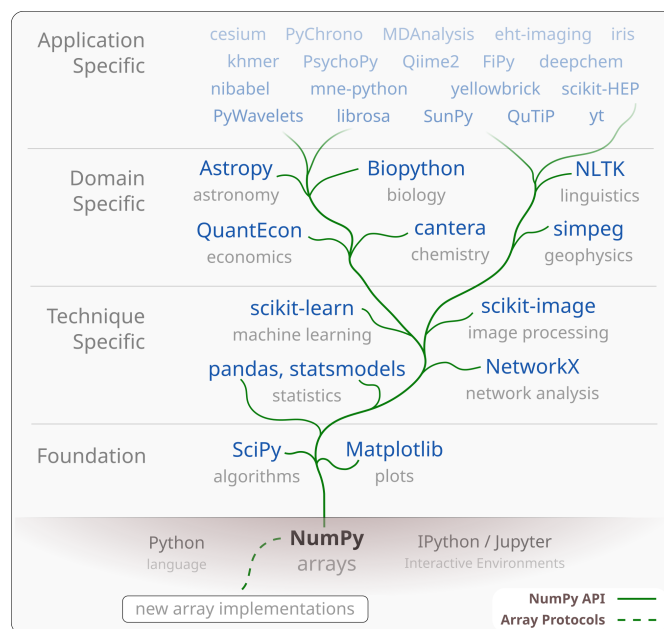


Figure 2. NumPy is the base of the scientific Python ecosystem. Essential libraries and projects that depend on NumPy’s API gain access to new array implementations that support NumPy’s array protocols (Fig. 3).

for the CPUs at hand (see Supplementary Methods for more details).

Altogether, the combination of a simple in-memory array representation, a syntax that closely mimics mathematics, and a variety of array-aware utility functions forms a productive and powerfully expressive array programming language.

Scientific Python ecosystem

Python is an open-source, general-purpose, interpreted programming language well-suited to standard programming tasks such as cleaning data, interacting with web resources, and parsing text. Adding fast array operations and linear algebra allows scientists to do all their work within a single language—and one that has the advantage of being famously easy to learn and teach, as witnessed by its adoption as a primary learning language in many universities.

Even though NumPy is not part of Python’s standard library, it benefits from a good relationship with the Python developers. Over the years, the Python language has added new features and special syntax so that NumPy would have a more succinct and easier to read array notation. Since it is not part of the standard library, NumPy is able to dictate its own release policies and development patterns.

SciPy and Matplotlib are tightly coupled with NumPy—in terms of history, development, and use. SciPy provides fundamental algorithms for scientific computing, including mathematical, scientific, and engineering routines. Matplotlib generates publication-ready figures and visualizations. The combination of NumPy, SciPy, and Matplotlib, together with an

advanced interactive environment like IPython¹⁸, or Jupyter¹⁹, provides a solid foundation for array programming in Python. The scientific Python ecosystem (Fig. 2) builds on top of this foundation to provide several, widely used *technique specific* libraries^{13,14,20}, that in turn underlay numerous *domain specific* projects^{21–26}. NumPy, at the base of the ecosystem of array-aware libraries, sets documentation standards, provides array testing infrastructure, and adds build support for Fortran and other compilers.

Many research groups have designed large, complex scientific libraries, which add *application specific* functionality to the ecosystem. For example, the `eht-imaging` library²⁷ developed by the Event Horizon Telescope collaboration for radio interferometry imaging, analysis, and simulation, relies on many lower-level components of the scientific Python ecosystem. NumPy arrays are used to store and manipulate numerical data at every step in the processing chain: from raw data through calibration and image reconstruction. SciPy supplies tools for general image processing tasks such as filtering and image alignment, while `scikit-image`, an image processing library that extends SciPy, provides higher-level functionality such as edge filters and Hough transforms. The `scipy.optimize` module performs mathematical optimization. `NetworkX`²⁰, a package for complex network analysis, is used to verify image comparison consistency. Astropy^{21,22} handles standard astronomical file formats and computes time/coordinate transformations. Matplotlib is used to visualize data and to generate the final image of the black hole.

The interactive environment created by the array programming foundation along with the surrounding ecosystem of tools—inside of IPython or Jupyter—is ideally suited to exploratory data analysis. Users fluidly inspect, manipulate, and visualize their data, and rapidly iterate to refine programming statements. These statements are then stitched together into imperative or functional programs, or notebooks containing both computation and narrative. Scientific computing beyond exploratory work is often done in a text editor or an integrated development environment (IDEs) such as Spyder. This rich and productive environment has made Python popular for scientific research.

To complement this facility for exploratory work and rapid prototyping, NumPy has developed a culture of employing time-tested software engineering practices to improve collaboration and reduce error²⁸. This culture is not only adopted by leaders in the project but also enthusiastically taught to newcomers. The NumPy team was early in adopting distributed revision control and code review to improve collaboration on code, and continuous testing that runs an extensive battery of automated tests for every proposed change to NumPy. The project also has comprehensive, high-quality documentation, integrated with the source code^{29–31}.

This culture of using best practices for producing reliable scientific software has been adopted by the ecosystem of libraries that build on NumPy. For example, in a recent award

given by the Royal Astronomical Society to Astropy, they state:

The Astropy Project has provided hundreds of junior scientists with experience in professional-standard software development practices including use of version control, unit testing, code review and issue tracking procedures. This is a vital skill set for modern researchers that is often missing from formal university education in physics or astronomy.

Community members explicitly work to address this lack of formal education through courses and workshops^{32–34}.

The recent rapid growth of data science, machine learning, and artificial intelligence has further and dramatically boosted the usage of scientific Python. Examples of its significant application, such as the `eht-imaging` library, now exist in almost every discipline in the natural and social sciences. These tools have become *the primary* software environment in many fields. NumPy and its ecosystem are commonly taught in university courses, boot camps, and summer schools, and are at the focus of community conferences and workshops worldwide.

NumPy and its API have become truly ubiquitous.

Array proliferation and interoperability

NumPy provides in-memory, multidimensional, homogeneously typed (i.e., single pointer and strided) arrays on CPUs. It runs on machines ranging from embedded devices to the world’s largest supercomputers, with performance approaching that of compiled languages. For most its existence, NumPy addressed the vast majority of array computation use cases.

However, scientific data sets now routinely exceed the memory capacity of a single machine and may be stored on multiple machines or in the cloud. In addition, the recent need to accelerate deep learning and artificial intelligence applications has led to the emergence of specialized accelerator hardware, including graphics processing units (GPUs), tensor processing units (TPUs), and field-programmable gate arrays (FPGAs). Due to its in-memory data model, NumPy is currently unable to utilize such storage and specialized hardware directly. However, both distributed data and the parallel execution of GPUs, TPUs, and FPGAs map well to the *paradigm* of array programming: a gap, therefore, existed between available modern hardware architectures and the tools necessary to leverage their computational power.

The community’s efforts to fill this gap led to a proliferation of new array implementations. For example, each deep learning framework created its own arrays; PyTorch³⁵, TensorFlow³⁶, Apache MXNet³⁷, and JAX arrays all have the capability to run on CPUs and GPUs, in a distributed fashion, utilizing lazy evaluation to allow for additional performance optimizations. SciPy and PyData/Sparse both provide sparse arrays—which typically contain few non-zero values and store only those in memory for efficiency. In addition, there are

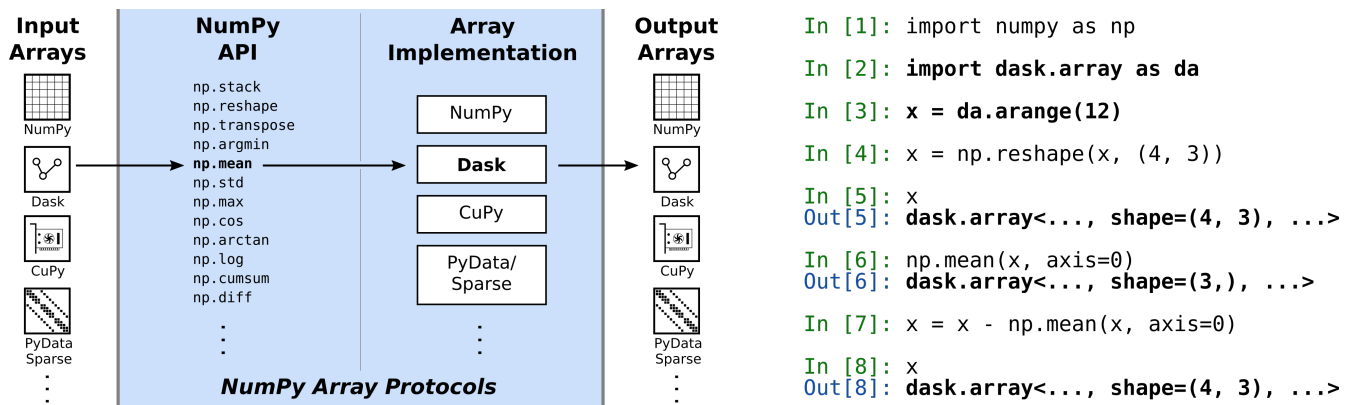


Figure 3. NumPy’s API and array protocols expose new arrays to the ecosystem. In this example, NumPy’s `mean` function is called on a Dask array. The call succeeds by dispatching to the appropriate library implementation (i.e., Dask in this case) and results in a new Dask array. Compare this code to the example code in Fig. 1g.

projects that build on top of NumPy arrays as a data container and *extend* its capabilities. Distributed arrays are made possible that way by Dask, and labeled arrays—referring to dimensions of an array by name rather than by index for clarity, compare `x[:, 1]` vs. `x.loc[:, 'time']`—by `xarray`³⁸.

Such libraries often mimic the NumPy API, because it lowers the barrier to entry for newcomers and provides the wider community with a stable array programming interface. This, in turn, prevents disruptive schisms like the divergence of Numeric and Numarray. But exploring new ways of working with arrays is experimental by nature and, in fact, several promising libraries—such as Theano and Caffe—have already ceased development. And each time that a user decides to try a new technology, they must change import statements and ensure that the new library implements all the parts of the NumPy API they currently use.

Ideally, operating on specialized arrays using NumPy functions or semantics would simply work, so that users could write code once, and would then benefit from switching between NumPy arrays, GPU arrays, distributed arrays, and so forth, as appropriate. To support array operations between external array objects, NumPy therefore added the capability to act as a central coordination mechanism with a well-specified API (Fig. 2).

To facilitate this *interoperability*, NumPy provides “protocols” (or contracts of operation), that allow for specialized arrays to be passed to NumPy functions (Fig. 3). NumPy, in turn, dispatches operations to the originating library, as required. Over four hundred of the most popular NumPy functions are supported. The protocols are implemented by widely used libraries such as Dask, CuPy, `xarray`, and PyData/Sparse. Thanks to these developments, users can now, for example, scale their computation from a single machine to distributed systems using Dask. The protocols also *compose* well, allowing users to redeploy NumPy code at scale on distributed, multi-GPU systems via, for instance, CuPy arrays embedded

in Dask arrays. Using NumPy’s high-level API, users can leverage highly parallel code execution on multiple systems with millions of cores, all with minimal code changes³⁹.

These array protocols are now a key feature of NumPy, and are expected to only increase in importance. As with the rest of NumPy, we iteratively refine and add protocol designs to improve utility and simplify adoption.

Discussion

NumPy combines the expressive power of *array programming*, the performance of C, and the readability, usability, and versatility of Python in a mature, well-tested, well-documented, and community-developed library. Libraries in the scientific Python ecosystem provide fast implementations of most important algorithms. Where extreme optimization is warranted, compiled languages such as Cython⁴⁰, Numba⁴¹, and Pythran⁴², that extend Python and transparently accelerate bottlenecks, can be used. Because of NumPy’s simple memory model, it is easy to write low-level, hand-optimized code, usually in C or Fortran, to manipulate NumPy arrays and pass them back to Python. Furthermore, using array protocols, it is possible to utilize the full spectrum of specialized hardware acceleration with minimal changes to existing code.

NumPy was initially developed by students, faculty, and researchers to provide an advanced, open-source array programming library for Python, which was free to use and unencumbered by license servers, dongles, and the like. There was a sense of building something consequential together, for the benefit of many others. Participating in such an endeavor, within a welcoming community of like-minded individuals, held a powerful attraction for many early contributors.

These user-developers frequently had to write code from scratch to solve their own or their colleagues’ problems—often in low-level languages that precede Python, like Fortran⁴³ and C. To them, the advantages of an interactive, high-level array library were evident. The design of this new tool was informed by other powerful interactive programming lan-

guages for scientific computing such as Basis⁴⁴, Yorick⁴⁵, R⁴⁶, and APL⁴⁷, as well as commercial languages and environments like IDL and MATLAB.

What began as an attempt to add an array object to Python became the foundation of a vibrant ecosystem of tools. Now, a large amount of scientific work depends on NumPy being correct, fast, and stable. It is no longer a small community project, but is core scientific infrastructure.

The developer culture has matured: while initial development was highly informal, NumPy now has a roadmap and a process for proposing and discussing large changes. The project has formal governance structures and is fiscally sponsored by NumFOCUS, a nonprofit that promotes open practices in research, data, and scientific computing. Over the past few years, the project attracted its first funded development, sponsored by the Moore and Sloan Foundations, and received an award as part of the Chan Zuckerberg Initiative's Essentials of Open Source Software program. With this funding, the project was (and is) able to have sustained focus over multiple months to implement substantial new features and improvements. That said, it still depends heavily on contributions made by graduate students and researchers in their free time (see Supplementary Methods for more details).

NumPy is no longer *just* the foundational array library underlying the scientific Python ecosystem, but has also become the standard API for tensor computation and a central coordinating mechanism between array types and technologies in Python. Work continues to expand on and improve these interoperability features.

Over the next decade, we will face several challenges. New devices will be developed, and existing specialized hardware will evolve, to meet diminishing returns on Moore's law. There will be more, and a wider variety of, data science practitioners, a significant proportion of whom will be using NumPy. The scale of scientific data gathering will continue to expand, with the adoption of devices and instruments such as light sheet microscopes and the Large Synoptic Survey Telescope (LSST)⁴⁸. New generation languages, interpreters, and compilers, such as Rust⁴⁹, Julia⁵⁰, and LLVM⁵¹, will invent and determine the viability of new concepts and data structures.

Through various mechanisms described in this paper, NumPy is poised to embrace such a changing landscape, and to continue playing a leading role in interactive scientific computation. To do so will require sustained funding from government, academia, and industry. But, importantly, it will also need a new generation of graduate students and other developers to engage, to build a NumPy that meets the needs of the next decade of data science.

References

1. Dubois, P. F., Hinsén, K. & Hugunin, J. Numerical Python. *Comput. Phys.* **10**, 262–267 (1996).
2. Ascher, D., Dubois, P. F., Hinsén, K., Hugunin, J. & Oliphant, T. E. An open source project: Numerical Python (2001). DOI 10.5281/zenodo.3599567.
3. Yang, T.-Y., Furnish, G. & Dubois, P. F. Steering object-oriented scientific computations. In *Proceedings of TOOLS USA 97. International Conference on Technology of Object Oriented Systems and Languages*, 112–119 (IEEE, 1997).
4. Greenfield, P., Miller, J. T., Hsu, J. & White, R. L. numarray: A new scientific array package for Python. *PyCon DC* (2003).
5. Oliphant, T. E. *Guide to NumPy* (Trelgol Publishing USA, 2006), 1st edn.
6. Dubois, P. F. Python: Batteries included. *Comput. Sci. & Eng.* **9**, 7–9 (2007).
7. Oliphant, T. E. Python for scientific computing. *Comput. Sci. & Eng.* **9**, 10–20 (2007).
8. Millman, K. J. & Aivazis, M. Python for scientists and engineers. *Comput. Sci. & Eng.* **13**, 9–12 (2011).
9. Pérez, F., Granger, B. E. & Hunter, J. D. Python: an ecosystem for scientific computing. *Comput. Sci. & Eng.* **13**, 13–21 (2011).
10. Virtanen, P. *et al.* SciPy 1.0—fundamental algorithms for scientific computing in Python. *Nat. Methods* **17**, 261–272 (2020).
11. Hunter, J. D. Matplotlib: A 2D graphics environment. *Comput. Sci. & Eng.* **9**, 90–95 (2007).
12. McKinney, W. Data structures for statistical computing in Python. In van der Walt, S. & Millman, J. (eds.) *Proceedings of the 9th Python in Science Conference*, 51–56 (2010).
13. Pedregosa, F. *et al.* Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* **12**, 2825–2830 (2011).
14. van der Walt, S. *et al.* scikit-image: image processing in Python. *PeerJ* **2**, e453 (2014).
15. van der Walt, S., Colbert, S. C. & Varoquaux, G. The NumPy array: a structure for efficient numerical computation. *Comput. Sci. & Eng.* **13**, 22–30 (2011).
16. Wang, Q., Zhang, X., Zhang, Y. & Yi, Q. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 1–12 (IEEE, 2013).
17. Xianyi, Z., Qian, W. & Yunquan, Z. Model-driven level 3 blas performance optimization on loongson 3a processor.

In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, 684–691 (IEEE, 2012).

18. Pérez, F. & Granger, B. E. IPython: a system for interactive scientific computing. *Comput. Sci. & Eng.* **9**, 21–29 (2007).
19. Kluyver, T. *et al.* Jupyter Notebooks—a publishing format for reproducible computational workflows. In Loizides, F. & Schmidt, B. (eds.) *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 87–90 (IOS Press, 2016).
20. Hagberg, A. A., Schult, D. A. & Swart, P. J. Exploring network structure, dynamics, and function using NetworkX. In Varoquaux, G., Vaught, T. & Millman, K. J. (eds.) *Proceedings of the 7th Python in Science Conference*, 11–15 (Pasadena, CA USA, 2008).
21. Astropy Collaboration *et al.* Astropy: A community Python package for astronomy. *Astron. & Astrophys.* **558**, A33 (2013).
22. Price-Whelan, A. M. *et al.* The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package. *The Astron. J.* **156**, 123 (2018).
23. Cock, P. J. *et al.* Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinforma.* **25**, 1422–1423 (2009).
24. Millman, K. J. & Brett, M. Analysis of functional Magnetic Resonance Imaging in Python. *Comput. Sci. & Eng.* **9**, 52–55 (2007).
25. SunPy Community, T. *et al.* SunPy—Python for solar physics. *Comput. Sci. Discov.* **8**, 014009 (2015).
26. Hamman, J., Rocklin, M. & Abernathy, R. Pangeo: A Big-data Ecosystem for Scalable Earth System Science. In *EGU General Assembly Conference Abstracts*, EGU General Assembly Conference Abstracts, 12146 (2018).
27. Chael, A. A. *et al.* ehtim: Imaging, analysis, and simulation software for radio interferometry. *Astrophys. Source Code Libr.* (2019).
28. Millman, K. J. & Pérez, F. Developing open-source scientific practice. *Implement. Reproducible Res. CRC Press. Boca Raton, FL* 149–183 (2014).
Describes the software engineering practices embraced by the NumPy and SciPy communities with a focus on how these practices improve research.
29. van der Walt, S. The SciPy documentation project (technical overview). In Varoquaux, G., Vaught, T. & Millman, K. J. (eds.) *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, 27–28 (2008).
30. Harrington, J. The SciPy documentation project. In Varoquaux, G., Vaught, T. & Millman, K. J. (eds.) *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, 33–35 (2008).

31. Harrington, J. & Goldsmith, D. Progress report: NumPy and SciPy documentation in 2009. In Varoquaux, G., van der Walt, S. & Millman, K. J. (eds.) *Proceedings of the 8th Python in Science Conference (SciPy 2009)*, 84–87 (2009).
32. Wilson, G. Software carpentry: Getting scientists to write better code by making them more productive. *Comput. Sci. & Eng.* (2006).
33. Hannay, J. E. *et al.* How do scientists develop and use scientific software? In *Proc. 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering* (2009).
34. Millman, K. J., Brett, M., Barnowski, R. & Poline, J.-B. Teaching computational reproducibility for neuroimaging. *Front. Neurosci.* **12**, 727 (2018).
35. Paszke, A. *et al.* Pytorch: An imperative style, high-performance deep learning library. In Wallach, H. *et al.* (eds.) *Advances in Neural Information Processing Systems* 32, 8024–8035 (Curran Associates, Inc., 2019).
36. Abadi, M. *et al.* Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).
37. Chen, T. *et al.* Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).
38. Hoyer, S. & Hamman, J. xarray: N-D labeled arrays and datasets in Python. *J. Open Res. Softw.* **5** (2017).
39. Entschew, P. Distributed multi-GPU computing with Dask, CuPy and RAPIDS. *EuroPython 2019* (2019).
40. Behnel, S. *et al.* Cython: The best of both worlds. *Comput. Sci. & Eng.* **13**, 31–39 (2011).
41. Lam, S. K., Pitrou, A. & Seibert, S. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM '15*, 7:1–7:6 (ACM, New York, NY, USA, 2015).
42. Guelton, S. *et al.* Pythran: Enabling static optimization of scientific python programs. *Comput. Sci. & Discov.* **8**, 014001 (2015).
43. Dongarra, J., Golub, G. H., Grosse, E., Moler, C. & Moore, K. Netlib and na-net: Building a scientific computing community. *IEEE Annals Hist. Comput.* **30**, 30–41 (2008).
44. Dubois, P. F. The basis system. Tech. Rep., Lawrence Livermore National Laboratory, CA (USA) (1989). UCRL-MA-118543, Parts I-VI.
45. Munro, D. H. & Dubois, P. F. Using the yorick interpreted language. *Comput. Phys.* **9**, 609–615 (1995).
46. Ihaka, R. & Gentleman, R. R: a language for data analysis and graphics. *J. Comput. Graph. Stat.* **5**, 299–314 (1996).

- 529 **47.** Iverson, K. E. A programming language. In *Proceedings*
530 *of the May 1-3, 1962, Spring Joint Computer Conference*,
531 345–351 (1962).
- 532 **48.** Jenness, T. *et al.* Lsst data management software devel-
533 opment practices and tools. In *Software and Cyberinfras-*
534 *tructure for Astronomy V*, vol. 10707, 1070709 (Interna-
535 tional Society for Optics and Photonics, 2018).
- 536 **49.** Matsakis, N. D. & Klock, F. S. The rust language. *Ada*
537 *Lett.* **34**, 103–104 (2014).
- 538 **50.** Bezanson, J., Edelman, A., Karpinski, S. & Shah, V. B.
539 Julia: A fresh approach to numerical computing. *SIAM*
540 *Rev.* **59**, 65–98 (2017).
- 541 **51.** Lattner, C. & Adve, V. LLVM: A compilation framework
542 for lifelong program analysis and transformation. 75–88
543 (San Jose, CA, USA, 2004).
- 544 **52.** Peterson, P. F2PY: a tool for connecting Fortran and
545 Python programs. *Int. J. Comput. Sci. Eng.* **4**, 296–305
546 (2009).
- 547 **53.** Anderson, E. *et al.* *LAPACK Users' Guide* (Society for
548 Industrial and Applied Mathematics, Philadelphia, PA,
549 1999), 3rd edn.
- 550 **54.** Dongarra, J. & Hammarling, S. Evolution of numeri-
551 cal software for dense linear algebra. In Cox, M. G. &
552 Hammarling, S. (eds.) *Reliable Numerical Computation*,
553 297–327 (Clarendon Press, 1990).
- 554 **55.** O'Neill, M. E. Pcg: A family of simple fast space-
555 efficient statistically good algorithms for random number
556 generation. Tech. Rep. HMC-CS-2014-0905, Harvey
557 Mudd College, Claremont, CA (2014).
- 558 **56.** Salmon, J. K., Moraes, M. A., Dror, R. O. & Shaw, D. E.
559 Parallel random numbers: As easy as 1, 2, 3. In *Pro-*
560 *ceedings of 2011 International Conference for High Per-*
561 *formance Computing, Networking, Storage and Analysis*,
562 SC '11, 16:1–16:12 (ACM, New York, NY, USA, 2011).
- 563 **57.** Doty-Humphrey, C. Praxtrand, version 0.94. [Online;
564 accessed 1-June-2020].
- 565 **58.** Matsumoto, M. & Nishimura, T. Mersenne Twister:
566 A 623-dimensionally equidistributed uniform pseudo-
567 random number generator. *ACM Transactions on Model.*
568 *Comput. Simul.* **8**, 3–30 (1998).
- 569 **59.** Sheppard, K., Duvenhage, B., de Buyl, P. & Ham,
570 D. A. bashtage/randomgen: Release 1.16.2 (2019). DOI
571 10.5281/zenodo.2621273.
- 572 **60.** Lemire, D. Testing non-cryptographic random number
573 generators: my results (2017). [Online; accessed 1-June-
574 2020].
- 575 **61.** Marsaglia, G. & Tsang, W. W. The ziggurat method for
576 generating random variables. *J. Stat. Software, Articles* **5**,
577 1–7 (2000).
- 62.** Lemire, D. Fast random integer generation in an interval. 578
ACM Transactions on Model. Comput. Simul. **29**, 1–12 579
(2019). 580

Supplementary Methods

We use Git for version control and GitHub as the public hosting service for our official *upstream* repository (<https://github.com/numpy/numpy>). We each work in our own copy (or fork) of the project and use the upstream repository as our integration point. To get new code into the upstream repository, we use GitHub's pull request (PR) mechanism. This allows us to review code before integrating it as well as to run a large number of tests on the modified code to ensure that the changes do not break expected behavior.

We also use GitHub's issue tracking system to collect and triage problems and proposed improvements.

Library organization

Broadly, the NumPy library consists of the following parts: the NumPy array data structure `ndarray`; the so-called *universal functions*; a set of library functions for manipulating arrays and doing scientific computation; infrastructure libraries for unit tests and Python package building; and the program `f2py` for wrapping Fortran code in Python⁵². The `ndarray` and the universal functions are generally considered the core of the library. In the following, we give a brief summary of these components of the library.

Core

The `ndarray` data structure and the universal functions make up the core of NumPy.

The `ndarray` is the data structure at the heart of NumPy. The data structure stores regularly strided homogeneous data types inside a contiguous block memory, allowing for the efficient representation of n -dimensional data. More details about the data structure are given in “The NumPy array: a structure for efficient numerical computation”¹⁵.

The *universal functions*, or more concisely, *ufuncs*, are functions written in C that implement efficient looping over NumPy arrays. An important feature of *ufuncs* is the built-in implementation of *broadcasting*. For example, the function `arctan2(x, y)` is a *ufunc* that accepts two values and computes $\tan^{-1}(y/x)$. When arrays are passed in as the arguments, the *ufunc* will take care of looping over the dimensions of the inputs in such a way that if, say, `x` is a 1-D array with length 3, and `y` is a 2-D array with shape 2×1 , the output will be an array with shape 2×3 (Fig. 1c). The *ufunc* machinery takes care of calling the function with all the appropriate combinations of input array elements to complete the output array. The elementary arithmetic operations of addition, multiplication, etc., are implemented as *ufuncs*, so that broadcasting also applies to expressions such as `x + y * z`.

Computing libraries.

NumPy provides a large library of functions for array manipulation and scientific computing, including functions for: creating, reshaping, concatenating, and padding arrays; searching, sorting and counting data in arrays; computing elementary

statistics, such as the mean, median, variance, and standard deviation; file I/O; and more.

NumPy's linear algebra library includes functionality for: solving linear systems of equations; calculating various matrix properties such as the determinant, the norm, the inverse, the pseudo-inverse; and computing the Cholesky, eigenvalue, and singular value decompositions of a matrix.

The NumPy module mainly provides interfaces to established implementations of the LAPACK (Linear Algebra PACKage)⁵³ interface. LAPACK itself, which was first released in 1992, is, in turn, based on the EISPACK and LINPACK open-source libraries. It is part of Netlib, a repository of mathematical software, papers, and databases, that has a long history of open, community-wide development^{43,54}. For improved performance, NumPy links to an accelerated BLAS (Basic Linear Algebra Subprograms) implementation, most commonly OpenBLAS (<http://www.openblas.net/>).

The random number generator library in NumPy provides alternative *bit stream generators* that provide the core function of generating random integers. A higher-level generator class that implements an assortment of probability distributions is provided. It includes the beta, gamma and Weibull distributions, the univariate and multivariate normal distributions, and more.

A suite of functions for computing the *fast Fourier transform (FFT)* and its inverse is provided.

Infrastructure libraries

NumPy provides utilities for writing tests and for building Python packages.

The testing subpackage provides functions such as `assert_allclose(actual, desired)` that may be used in test suites for code that uses NumPy arrays.

NumPy provides the subpackage `distutils` which includes functions and classes to facilitate configuration, installation, and packaging of libraries depending on NumPy. These can be used, for example, when publishing to the PyPI website.

F2PY

The program `f2py` is a tool for building NumPy-aware Python wrappers of Fortran functions. NumPy itself does not use any Fortran code; F2PY is part of NumPy for historical reasons.

Governance

NumPy adopted an official Governance Document on October 5, 2015 (<https://numpy.org/devdocs/dev/governance/governance.html>). Project decisions are usually made by consensus of interested contributors. This means that, for most decisions, everyone is entrusted with veto power. A Steering Council, currently composed of 12 members, facilitates this process and oversees daily development of the project by contributing code and reviewing contributions from the community.

NumPy’s official Code of Conduct was approved on September 1, 2018 (https://numpy.org/devdocs/dev/conduct/code_of_conduct.html). In brief, we strive to: *be open; be empathetic, welcoming, friendly, and patient; be collaborative; be inquisitive; and be careful in the words that we choose*. The Code of Conduct also specifies how breaches can be reported and outlines the process for responding to such reports.

Funding

In 2017, NumPy received its first large grants totaling 1.3M USD from the Gordon & Betty Moore and the Alfred P. Sloan foundations. Stéfan van der Walt is the PI and manages four programmers working on the project. These two grants focus on addressing the technical debt accrued over the years and on setting in place standards and architecture to encourage more sustainable development.

NumPy received a third grant for 195K USD from the Chan Zuckerberg Initiative at the end of 2019 with Ralf Gommers as the PI. This grant focuses on better serving NumPy’s large number of beginning to intermediate level users and on growing the community of NumPy contributors. It will also provide support to OpenBLAS, on which NumPy depends for accelerated linear algebra.

Finally, since May 2019 the project receives a small amount annually from Tidelift, which is used to fund things like documentation and website improvements.

Developers

NumPy is currently maintained by a group of 23 contributors with commit rights to the NumPy code base. Out of these, 17 maintainers were active in 2019, 4 of whom were paid to work on the project full-time. Additionally, there are a few long term developers who contributed and maintain specific parts of NumPy, but are not officially maintainers.

Over the course of its history, NumPy has attracted PRs by 823 contributors. However, its development relies heavily on a small number of active maintainers, who share more than half of the contributions among themselves.

At a release cycle of about every half year, the five recent releases in the years 2018 and 2019 have averaged about 450 PRs each,¹ with each release attracting more than a hundred new contributors. Figure 4 shows the number of PRs merged into the NumPy master branch. Although the number of PRs being merged fluctuates, the plot indicates an increased number of contributions over the past years.

Community calls

The massive number of scientific Python packages that built on NumPy meant that it had an unusually high need for stabil-

¹ Note that before mid 2011, NumPy development did not happen on github.com. All data provided here is based on the development which happened through GitHub PRs. In some cases contributions by maintainers may not be categorized as such.

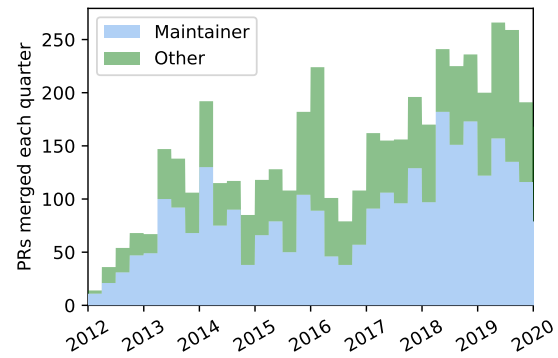


Figure 4. Number of pull requests merged into the NumPy master branch for each quarter since 2012. The total number of PRs is indicated with the lower blue area showing the portion contributed by current or previous maintainers.

ity. So to guide our development we formalized the feature proposal process, and constructed a development roadmap with extensive input and feedback from the community.

Weekly community calls alternate between triage and higher level discussion. The calls not only involve developers from the community, but provide a venue for vendors and other external groups to provide input. For example, after Intel produced a forked version of NumPy, one of their developers joined a call to discuss community concerns.

NumPy enhancement proposals

Given the complexity of the codebase and the massive number of projects depending on it, large changes require careful planning and substantial work. NumPy Enhancement Proposals (NEPs) are modeled after Python Enhancement Proposals (PEPs) for “proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python”². Since then there have been 19 proposed NEPS—6 have been implemented, 4 have been accepted and are being implemented, 4 are under consideration, 3 have been deferred or superseded, and 2 have been rejected or withdrawn.

Central role

NumPy plays a central role in building and standardizing much of the scientific Python community infrastructure. NumPy’s docstring standard is now widely adopted. We are also now using the NEP system as a way to help coordinate the larger scientific Python community. For example, in NEP 29, we recommend, along with leaders from various other projects, that all projects across the Scientific Python ecosystem adopt a common “time window-based” policy for support

²<https://numpy.org/neps/nep-0000.html>

of Python and NumPy versions. This standard will simplify downstream project and release planning.

Wheels build system

A Python *wheel* (<https://www.python.org/dev/peps/pep-0427/>) is a standard file format for distributing Python libraries. In addition to Python code, a wheel may include compiled C extensions and other binary data. This is important, because many libraries, including NumPy, require a C compiler and other build tools to build the software from the source code, making it difficult for many users to install the software on their own. The introduction of wheels to the Python packaging system has made it much easier for users to install precompiled libraries.

A GitHub repository containing scripts to build NumPy wheels has been configured so that a simple commit to the repository triggers an automated build system that creates NumPy wheels for several computer platforms, including Windows, Mac OSX and Linux. The wheels are uploaded to a public server and made available for anyone to use. This system makes it easy for users to install precompiled versions of NumPy on these platforms.

The technology that is used to build the wheels evolves continually. At the time this paper is being written, a key component is the `multibuild` suite of tools developed by Matthew Brett and other developers (<https://github.com/matthew-brett/multibuild>). Currently, scripts using `multibuild` are written for the continuous integration platforms Travis-CI (for Linux and Mac OSX) and Appveyor (for Windows).

Recent technical improvements

With the recent infusion of funding and a clear process for coordinating with the developer community, we have been able to tackle a number of important large scale changes. We highlight two of those below, as well as changes made to our testing infrastructure to support hardware platforms used in large scale computing.

Array function protocol

A vast number of projects are built on NumPy; these projects are consumers of the NumPy API. Over the last several years, a growing number of projects are providers of a *NumPy-like API* and array objects targeting audiences with specialized needs beyond NumPy's capabilities. For example, the NumPy API is implemented by several popular tensor computation libraries including CuPy³, JAX⁴, and Apache MXNet⁵. Py-

Torch⁶ and Tensorflow⁷ provide tensor APIs with NumPy-inspired semantics. It is also implemented in packages that support sparse arrays such as `scipy.sparse` and `PyData/Sparse`. Another notable example is Dask, a library for parallel computing in Python. Dask adopts the NumPy API and therefore presents a familiar interface to existing NumPy users, while adding powerful abilities to parallelize and distribute tasks.

The multitude of specialized projects creates the difficulty that consumers of these NumPy-like APIs write code specific to a single project and do not support all of the above array providers. This is a burden for users relying on the specialized array-like, since a tool they need may not work for them. It also creates challenges for end-users who need to transition from NumPy to a more specialized array. The growing multitude of specialized projects with NumPy-like APIs threatened to again fracture the scientific Python community.

To address these issues NumPy has the goal of providing the fundamental API for *interoperability* between the various NumPy-like APIs. An earlier step in this direction was the implementation of the `__array_ufunc__` protocol in NumPy 1.13, which enabled interoperability for most mathematical functions (<https://numpy.org/neps/nep-0013-ufunc-overrides.html>). In 2019 this was expanded more generally with the inclusion of the `__array_function__` protocol into NumPy 1.17. These two protocols allow providers of array objects to be interoperable with the NumPy API: their arrays work correctly with almost all NumPy functions (<https://numpy.org/neps/nep-0018-array-function-protocol.html>). For the users relying on specialized array projects it means that even though much code is written specifically for NumPy arrays and uses the NumPy API as `import numpy as np`, it can nevertheless work for them. For example, here is how a CuPy GPU array can be passed through NumPy for processing, with all operations being dispatched back to CuPy:

```
import numpy as np
import cupy as cp

x_gpu = cp.array([1, 2, 3])
y = np.sum(x_gpu)  # Returns a GPU array
```

Similarly, user defined functions composed using NumPy can now be applied to, e.g., multi-node distributed Dask arrays:

```
import numpy as np
import dask.array as da

def f(x):
    """Function using NumPy API calls"""
    y = np.tensordot(x, x.T)
```

³<https://cupy.chainer.org/>

⁴<https://jax.readthedocs.io/en/latest/jax.numpy.html>

⁵<https://numpy.mxnet.io/>

⁶https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html

⁷<https://www.tensorflow.org/tutorials/customization/basics>

```

858         return np.mean(np.log(y + 1))
859
860
861     x_local = np.random.random([10000, 10000]) #
862         random local array
863     x_distr = da.random.random([10000, 10000]) #
864         random distributed array
865
866     f(x_local) # returns a NumPy array
867     f(x_distr) # works, returns a Dask array

```

868 Random number generation

869 The NumPy random module provides pseudorandom num-
870 bers from a wide range of distributions. In legacy ver-
871 sions of NumPy, simulated random values are produced by
872 a RandomState object that: handles seeding and state ini-
873 tialization; wraps the core pseudorandom number generator
874 based on a Mersenne Twister implementation⁸; interfaces with
875 the underlying code that transforms random bits into variates
876 from other distributions; and supplies a singleton instance
877 exposed in the root of the random module.

878 The RandomState object makes a compatibility guaran-
879 tee so that a fixed seed and sequence of function calls produce
880 the same set of values. This guarantee has slowed progress
881 since improving the underlying code requires extending the
882 API with additional keyword arguments. This guarantee con-
883 tinues to apply to RandomState.

884 NumPy 1.17 introduced a new API for generating random
885 numbers that use a more flexible structure that can be extended
886 by libraries or end-users. The new API is built using com-
887 ponents that separate the steps required to generate random
888 variates. Pseudorandom bits are generated by a bit generator.
889 These bits are then transformed into variates from complex
890 distributions by a generator. Finally, seeding is handled by an
891 object that produces sequences of high-quality initial values.

892 Bit generators are simple classes that manage the state
893 of an underlying pseudorandom number generator. NumPy
894 ships with four bit generators. The default bit generator is
895 a 64-bit implementation of the Permuted Congruential Gen-
896 erator⁵⁵ (PCG64). The three other bit generators are a 64-
897 bit version of the Philox generator⁵⁶ (Philox), Chris Doty-
898 Humphrey’s Small Fast Chaotic generator⁵⁷ (SFC64), and
899 the 32-bit Mersenne Twister⁵⁸ (MT19937) which has been
900 used in older versions of NumPy.⁹ Bit generators provide
901 functions, exposed both in Python and C, for generating ran-
902 dom integer and floating point numbers. The three new bit
903 generators were chosen for their combination of statistical
904 soundness and performance. The PCG family of generators
905 have been widely tested using state-of-the-art statistical tests
906 and found to perform well⁶⁰. All three of the new generators
907 were tested using the PractRand test suite⁵⁷ with four TB
908 of random values. Each was tested using a single generator

and using a composite generator assembled from multiple
copies of the same bit generator, each seeded from a shared
SeedSequence.

The Generator consumes one of the bit generators and
produces variates from complicated distributions. Many im-
proved methods for generating random variates from com-
mon distributions were implemented, including the Ziggur-
rat method for normal, exponential and gamma variates⁶¹,
and Lemire’s method for bounded random integer gener-
ation⁶². The Generator is more similar to the legacy
RandomState, and its API is substantially the same. The
key differences all relate to state management, which has been
delegated to the bit generator. The Generator does not
make the same stream guarantee as the RandomState ob-
ject, and so variates may differ across versions as improved
generation algorithms are introduced.¹⁰

Finally, a SeedSequence is used to initialize a bit gen-
erator. The seed sequence can be initialized with no argu-
ments, in which case it reads entropy from a system-dependent
provider, or with a user-provided seed. The seed sequence
then transforms the initial set of entropy into a sequence of
high-quality pseudorandom integers, which can be used to
initialize multiple bit generators deterministically. The key
feature of a seed sequence is that it can be used to spawn
child SeedSequences to initialize multiple distinct bit gen-
erators. This capability allows a seed sequence to facilitate
large distributed applications where the number of workers re-
quired is not known. The sequences generated from the same
initial entropy and spawns are fully deterministic to ensure
reproducibility.

The three components are combined to construct a complete
random number generator.

```

from numpy.random import (
    Generator,
    PCG64,
    SeedSequence,
)

seq = SeedSequence(1030424547444117993331016959)
pcg = PCG64(seq)
gen = Generator(pcg)

```

This approach retains access to the seed sequence which
can then be used to spawn additional generators.

```

children = seq.spawn(2)
gen_0 = Generator(PCG64(children[0]))
gen_1 = Generator(PCG64(children[1]))

```

While this approach retains complete flexibility, the method
np.random.default_rng can be used to instantiate a
Generator when reproducibility is not needed.

The final goal of the new API is to improve extensibility.
RandomState is a monolithic object that obscures all of the
underlying state and functions. The component architecture

⁸to be precise, the standard 32-bit version of MT19937

⁹The [randomgen project](#) supplies a wide range of alternative bit generators
such as a cryptographic counter-based generators (AESCTR) and generators
that expose hardware random number generators (RDRAND)⁵⁹.

¹⁰Despite the removal of the compatibility guarantee, simple reproducibil-
ity across versions is encouraged, and minor changes that do not pro-
duce meaningful performance gains or fix underlying bug are not generally
adopted.

is one part of the extensibility improvements. The underlying functions (written in C) which transform the output of a bit generator to other distributions are available for use in CFFI. This allows the same code to be run in both NumPy and dependent that can consume CFFI, e.g., Numba. Both the bit generators and the low-level functions can also be used in C or Cython code.¹¹

Testing on multiple architectures

At the time of writing the two fastest supercomputers in the world, Summit and Sierra, both have IBM POWER9 architectures (<https://www.top500.org/lists/2019/11/>). In late 2018, Astra, the first ARM-based supercomputer to enter the TOP500 list, went into production (<https://en.wikichip.org/wiki/supercomputers/astra>). Furthermore, over 100 billion ARM processors have been produced as of 2017 (https://en.wikipedia.org/wiki/ARM_architecture), making it the most widely used instruction set architecture in the world.

Clearly there are motivations for a large scientific computing software library to support POWER and ARM architectures. We've extended our continuous integration (CI) testing to include ppc64le (POWER8 on Travis CI) and ARMv8 (on Shippable service). We also test with the s390x architecture (IBM Z CPUs on Travis CI) so that we can probe the behavior of our library on a big-endian machine. This satisfies one of the major components of improved CI testing laid out in a version of our roadmap—specifically, “CI for more exotic platforms.”

PEP 599 (<https://www.python.org/dev/peps/pep-0599/>) lays out a plan for new Python binary wheel distribution support, manylinux2014, that adds support for a number of architectures supported by the CentOS Alternative Architecture Special Interest Group, including ARMv8, ppc64le, as well as s390x. We are thus well-positioned for a future where provision of binaries on these architectures will be expected for a library at the base of the ecosystem.

Acknowledgments

We thank Ross Barnowski, Paul Dubois, Michael Eickenberg, and Perry Greenfield, who suggested text and provided helpful feedback on the manuscript.

We also thank the many members of the community who provided feedback, submitted bug reports, made improvements to the documentation, code, or website, promoted NumPy's use in their scientific fields, and built the vast ecosystem of tools and libraries around NumPy. We also gratefully acknowledge the Numeric and Numarray developers on whose work we built.

Jim Hugunin wrote Numeric in 1995, while a graduate student at MIT. Hugunin based his package on previous work

by Jim Fulton, then working at the US Geological Survey, with input from many others. After he graduated, Paul Dubois at the Lawrence Livermore National Laboratory became the maintainer. Many people contributed to the project including T.E.O. (a co-author of this paper), David Ascher, Tim Peters, and Konrad Hinsen.

In 1998 the Space Telescope Science Institute started using Python and in 2000 began developing a new array package called Numarray, written almost entirely by Jay Todd Miller, starting from a prototype developed by Perry Greenfield. Other contributors included Richard L. White, J. C. Hsu, Jochen Krupper, and Phil Hodge. The Numeric/Numarray split divided the community, yet ultimately pushed progress much further and faster than would otherwise have been possible.

Shortly after Numarray development started, T.E.O. took over maintenance of Numeric. In 2005, he led the effort and did most of the work to unify Numeric and Numarray, and produce the first version of NumPy.

Eric Jones co-founded (along with T.E.O. and P.P.) the SciPy community, gave early feedback on array implementations, and provided funding and travel support to several community members. Numerous people contributed to the creation and growth of the larger SciPy ecosystem, which gives NumPy much of its value. Others injected new energy and ideas by creating experimental array packages.

K.J.M. and S.J.v.d.W. were funded in part by the Gordon and Betty Moore Foundation through Grant GBMF3834 and by the Alfred P. Sloan Foundation through Grant 2013-10-27 to the University of California, Berkeley. S.J.v.d.W., S.B., M.P., and W.W. were funded in part by the Gordon and Betty Moore Foundation through Grant GBMF5447 and by the Alfred P. Sloan Foundation through Grant G-2017-9960 to the University of California, Berkeley.

Author Contributions Statement

K.J.M. and S.J.v.d.W. composed the manuscript with input from others. S.B., R.G., K.S., W.W., M.B., and T.J.R. contributed text. All authors have contributed significant code, documentation, and/or expertise to the NumPy project. All authors reviewed the manuscript.

Competing Interests

The authors declare no competing interests.

¹¹As of 1.18.0, this scenario requires access to the NumPy source. Alternative approaches that avoid this extra step are being explored.