Python: An Ecosystem for Scientific Computing

As the relationship between research and computing evolves, new tools are required to not only treat numerical problems, but also to solve various problems that involve large datasets in different formats, new algorithms, and computational systems such as databases and Internet servers. Python can help develop these computational research tools by providing a balance of clarity and flexibility without sacrificing performance.

Scientific computing, a discipline at the intersection of scientific research, engineering, and computing, has traditionally focused either on raw performance (in languages such as Fortran and C/C++) or generality and ease of use (in systems such as Matlab or Mathematica), and mainly for numerical problems. Today, scientific researchers use computers for problems that extend far beyond pure numerics, and we need tools flexible enough to address issues beyond performance and usability.

As we describe here, the Python programming language, augmented with a stack of open source tools developed over the past decade by a diverse group of scientists and engineers, provides a *computational ecosystem* that is quite capable of tackling these new challenges.

Scientific Computing's Changing Landscape

For a long time, scientific computing was focused almost exclusively on raw performance for floating-point numerical tasks. It's no accident that Fortran is an abbreviation of *for*mula *tran*slator: for a long time, computing numerical formulas was a computer's main purpose. Today, however, scientific computing's algorithmic needs go far beyond floating-point numerics. Despite the lasting importance and usefulness of array-oriented libraries such as Lapack (and its many descendants), modern scientific codes routinely tackle problems that go beyond number crunching with arrays.

Problems such as graph traversal, finite state machines, or branch and bound algorithms are now a staple of many scientific codes.[1] These require data structures beyond simple arrays and use code full of logic and integer manipulations that is quite different from what tools like Lapack are good at. In addition, even today's numerical work has needs beyond hardware floating point, as many current problems of interest require integrated access to arbitrary precision integers, rationals, and floating-point numbers, often in combination with symbolic manipulation.

The Python tool stack doesn't attempt to replace the many critical codes and algorithms with versions written in Python. Rather, the approach is to expose those codes through Python wrappers while providing rich data structures and programming paradigms to tackle problems not easily managed with high-performance computing's traditional data structures. Although Python isn't unique in possessing rich data structures (C++ has them as well), it's particularly good at

1521-9615/11/$26.00 © 2011 IEEE
Copublished by the IEEE CS and the AIP

Fernando Pérez
University of California, Berkeley
Brian E. Granger
California Polytechnic State University, San Luis Obispo
John D. Hunter
TradeLink Securities

interoperating with multiple languages' structures and with a syntax accessible to scientists who aren't programmers.

Computing's role in science and engineering has also exploded beyond strictly algorithmic needs. A range of factors has driven this transformation including the Internet, ever-improving hardware, online collaboration tools, the rise of "data driven" science, and open source software development models. As an integral part of their research efforts, scientists today develop computational tools to tackle tasks as varied as

- accessing data and results over a network, often with custom protocols;
- exposing codes and data via Web applications;
- interacting with databases;
- integrating components written in multiple programming languages, and integrating tools and libraries from multiple scientific disciplines;
- providing graphical user interfaces (GUIs) to scientific codes;
- working with data in a wide range of formats, including binary, XML, JavaScript Object Notation (JSON), and Hierarchical Data Format (HDF5);
- controlling and interfacing with a range of hardware devices;
- parallelizing programs to run on GPUs, clusters, multicore CPUs, and supercomputers; and
- collaborating with large, geographically distributed teams of scientists, programmers, and engineers on computationally based research, software, and data.

Originally, computational resources (cycles, memory, and storage) were scarce, and human development time was comparatively cheap. Over time, this balance inevitably shifted; ease of use and development have become increasingly important considerations, as evidenced by the rise in popularity of high-level computing environments.

First, the 1980s brought the rise of systems—such as Matlab and IDL (the Interactive Data Language)—that effectively encapsulated Fortran libraries behind a friendly user interface with access to rich functionality, interactive exploration, and immediate data visualization. Later, systems with symbolic capabilities and arbitrary precision numerical support, such as Mathematica and Maple, became extremely popular. Such systems made entirely new kinds of computations possible that couldn't be tackled with a reasonable programming effort in plain Fortran or C.

All such systems provide interactive exploratory environments that make language features and libraries immediately available to scientists who can use them to explore a problem domain. This contrasts with the classical edit/compile/run cycle of C or Fortran programming, which typically requires separate computation and post-processing/visualization steps.

An interactive environment offers much more flexibility in using the computer as an exploration tool; the code evolves as scientists better understand a problem by performing small, incremental computations that lead to a complete program.

The challenges of modern scientific computing that we have discussed are hard to address purely with languages such as Fortran and C/C++. Indeed, they require computational expertise different from the scientist's typical training in writing fast `for` loops, and some problems aren't merely technical but have social aspects as well, such as developing tools collaboratively over the Internet.

Although the high-level environments we've mentioned have proven extremely useful, the carefully conceived Python programming language combined with a flexible stack of components provides a comprehensive ecosystem in which even better solutions can be constructed. Building free and open source codes for scientific problems is also consistent with the research goals of transparency and reproducibility.

Python: A General Purpose Programming Language

Most tools that we've mentioned so far, from Fortran to Mathematica, have one thing in common: they were designed primarily with mathematical and technical computing in mind. Fortran has arrays and mathematical functions defined as first-class entities in the language; Matlab has native syntax to express common linear algebra operations; and Mathematica symbols are all valid entities without declarations, so expressions can involve arbitrary symbol sets.

This design focus offers important benefits: many everyday mathematical tasks can be performed with a compact and familiar syntax; the tool's abstractions can map well to many scientific domains; and there's often considerable out-of-the-box functionality for doing nontrivial scientific computing. Writing sophisticated user interface applications that integrate numerics with visualization—while talking to databases and handling custom user input—is possible but difficult in some custom scientific languages (such

```
def qsort(L):
    if not L:  return L # exit recursion if input is empty
    pivot, rest  = L[0], L[1:]
    less_than  = [ x  for x in rest if x < pivot ]
    greater_eq = [ x for x in rest  if x >= pivot ]
    return qsort(less_than) + [pivot] + qsort(greater_eq)
```

Figure 1. Implementing the basic QuickSort algorithm in Python. QuickSort is one of the simplest sorting algorithms to achieve O (n log n) complexity with a divide-and-conquer approach. Although tedious to implement in C or Fortran; in Python, these six lines suffice.

as Matlab) and practically impossible in others (such as R).

Background and Overview

In contrast, Python was designed as a general-purpose language. Python's creator, Guido van Rossum, has written a detailed series of articles on the language's history (see http://python-history.blogspot.com). Python was born in the early 1990s as a general-purpose "glue" language at the CWI (the Centrum voor Wiskunde and Informatica), the birthplace of ALGOL 68 (Algorithmic Language 68). Prior to this, van Rossum had worked on the ABC programming language, developed at the CWI as a teaching language that emphasized clarity. Although the ABC project was ultimately shut down, van Rossum took many lessons from it when he started to write Python as a tool for use in multimedia and operating systems research projects. He wanted Python to be high-level enough to be easy to read and write, while also—in sharp contrast to Java and key to our purposes—offering direct access to low-level capabilities, as well as offering easy portability and a well-defined error model based on exceptions.

Python is a dynamically typed language with a rich set of native types. Its number hierarchy includes native arbitrary-length integers, hardware-precision floating-point and complex numbers, and library support for rational numbers and arbitrary precision floating point. It also has powerful strings, variable-size lists, sets, and very flexible associative arrays called *dictionaries* in Python. These types give Python a rich vocabulary in which to express many complex algorithmic questions with clarity and efficiency. We can illustrate this expressive power with a fully functional implementation of the classic QuickSort.

QuickSort is one of the simplest sorting algorithms to achieve O (n log n) complexity with a divide-and-conquer approach. The basic algorithm without optimizations is easy to explain, but tedious to implement in C or Fortran. As Figure 1 shows, it can be implemented in Python in six lines. It's interesting to note how similar this code example is to the pseudo-code that expresses the algorithm on Wikipedia; such clarity and simplicity of expression have caused some to refer to Python code as "executable pseudo-code."

Furthermore, Python is an object-oriented language that lets users redefine the meaning of most operators for their own types; the expression `a + b**2` can thus call user-defined routines for addition and exponentiation on `a` and `b`. The language maps most syntax operators to specially named methods; for example, to implement exponentiation for a type, you write a method, `__pow__`, which is then called whenever the `**` operator is encountered. That Python is object oriented at its core helps users build complex codes that closely track the physical or mathematical entities they model; however, Python doesn't force this programming paradigm on users who prefer more familiar procedural approaches.

Python combines high-level flexibility, readability, and a well-defined interface with low-level capabilities, including an official C interface that lets you extend the language with C code and link to third-party libraries in C, C++, and Fortran. This generality is advantageous for modern scientific computing: it's a productive everyday environment that also lets you optimize performance-critical bottlenecks. Further, it provides the flexibility to build tools with a precise balance of low- and high-level features so you can appropriately choose between performance and ease of development or use.

This combination of semantic richness and flexibility makes Python well suited to solving many of the non-algorithmic computational issues we mentioned above, such as integrating with the Web, data formats, or low-level hardware. Python libraries—whether included in the language or from external projects—let you interface with Web servers, databases, and scientific storage formats such as HDF5 to process text and send data

```
In [1]: import urllib, matplotlib.mlab

In [2]: symbol = 'IBM'

In [3]: stock_url = ('http://ichart.finance.yahoo.com/table.csv?s=%s'+
   ...:              '&d=3&e=28&f=2010&g=d&a=0&b=2&c=1962&ignore=.csv') % symbol

In [4]: filename, headers = urllib.urlretrieve(stock_url)
```

Figure 2. Loading the `urllib` module for manipulating URLs from Python's standard library as well as the matplotlib package for data analysis and visualization. These components let us download stock trading data for IBM from the Internet using the Yahoo! historical financial data service, which provides files in comma-separated values (CSV) format typically read with spreadsheet software.

over raw network sockets, and so on. Such tasks often require a mix of fairly low-level machinery and high-level abstractions, such as the combination of managing binary data over raw sockets with objects to represent Web servers or open database connections.

Numerics and Data

Given its rich native types and a flexible object model, Python lets users not only easily express many complex, high-level tasks concisely, but also offers a good platform for developing more specialized objects that are directly suited to scientific work. The NumPy library[2]—which Stéfan van der Walt and his colleagues describe in "The NumPy Array: A Structure for Efficient Numerical Computation" on page 22—provides a remarkably sophisticated multidimensional array object that can be accessed by low-level Fortran or C code for speed and provides a much richer Python-level interface.

Here, we begin to see the benefit of layering scientific tools on top of Python's general foundations: today's NumPy arrays have evolved from fairly closely mimicking *n*-dimensional Fortran arrays to being objects with rich semantics that include sophisticated broadcasting behavior and compound data types that make them suitable for database-like processing of complex datasets. As a simple example, we'll do a brief analysis with real financial data freely available on the Internet.

As Figure 2 shows, working in the IPython interactive environment,[3] we load the `urllib` module for manipulating URLs from Python's standard library as well as the matplotlib[4] package for data analysis and visualization. These components let us download stock trading data for IBM from the Internet using the Yahoo! historical financial data service, which provides files in comma-separated values (CSV) format typically read with spreadsheet software.

Next, we convert the CSV file to a database-like NumPy array using matplotlib's `csv2rec()` function and then answer a few simple questions about the IBM stock. The `stock` variable is a NumPy array, but rather than being a simple 2D array of numbers, NumPy arrays can support named fields that correspond to the different CSV file columns. The array contains fields such as date, closing price, and trading volume, each with its own data type. The `stock` variable is thus functionally equivalent to an array of C structs, but much easier to work with. We can then compute the daily trading volume in dollars as the product of the volume (in number of shares) and the closing price (in dollars), and find both the peak trading volume and the day on which it occurred (see Figure 3).

We can now find which days were above 90 percent of the maximum dollar volume. With these days, we can compute a Boolean mask to use as an indexing object to extract values from the daily volume array and the original stock price array's date field (See Figure 4).

Finally, we produce the plot in Figure 5 using just a few lines of code (omitted here for conciseness) and the matplotlib[4] library.

These simple operations show Python's flexibility and ease of exploration; using them, we can combine Python's own general-purpose modules (such as `urllib`), with libraries written for and by scientists, such as NumPy and matplotlib. And, while this example focuses on numerical operations with NumPy arrays, arbitrary precision numerics and symbolic tools also exist. Finally, because Python is an open source language, the extension path can be taken to an ultimate conclusion by going beyond Python itself.

Where NumPy has added powerful arrays that can contain all numerical types supported by modern compilers, the Sage system (www.sagemath.org)

```
In [5]: stock = matplotlib.mlab.csv2rec(filename)

In [6]: daily_vol = stock.volume*stock.close

In [7]: peak_vol = int(daily_vol.max()/1e6) # in Millions of dollars

In [8]: peak_date = stock.date[daily_vol.argmax()]

In [9]: print 'Peak volume $%s Million, on %s' % (peak_vol, peak_date)
Peak volume: $7771 Million, on 1997-01-22
```

Figure 3. Computing the daily trading volume in dollars as the product of the volume (in number of shares) and the closing price (in dollars) lets us find both the peak trading volume and the day on which it occurred.

```
In [10]: mask = daily_vol > 0.9+daily_vol.max()

In [11]: print 'Dates greater than 90% max volume', stock.date[mask]
Dates greater than 90% max volume [1999-04-22 1997-01-22]

In [12]: print 'Volume for these dates: $', daily_vol[mask]/1e6, 'Million'
Volume for these dates: $ [7618.5261  7771.1984] Million
```

Figure 4. Using a Boolean mask to find the days with the largest trading volumes and then printing the dates and volumes for those days.

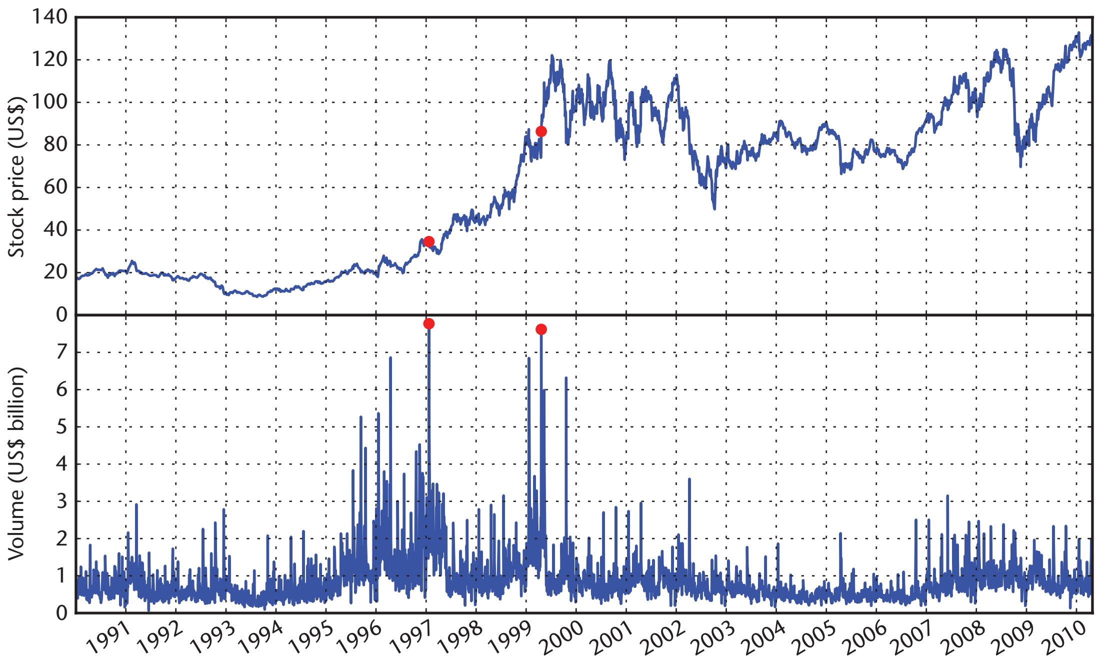

Figure 5. A matplotlib-based display of IBM stock prices, starting in 1990. Dates in red are the upper 10 percent of the trading volume, which we found using the example code.

actually extends the core language. Sage modifies the Python syntax to accept constructs that go beyond those valid in Python and redefines all of the language's basic numerical types. The resulting integers are based on new C-level multiprecision integers whose division produces rational numbers, and all floating-point operations are done using extended precision types instead of hardware floating point. This lets Sage design a language whose numerical behavior follows a strict mathematical model, while being largely compatible with Python itself. Sage's Web-browser-based notebook interface also provides a good example of how to write custom interfaces to the Python interpreter.

This level of flexibility is hard to achieve in languages such as Fortran/C/C++ or environments such as Matlab or Mathematica. Fortran arrays are fast, but nowhere as flexible or semantically rich as NumPy's, and doing symbolic computing in C++ is tedious at best. Building efficient low-level machinery for proprietary systems is often a difficult task that requires custom build systems, producing a result that might not be portable or easy to maintain as vendor tool and interface versions change beyond the scientist's control. Although Python also has portability issues—such as between multiple versions of Python or multiple versions of the Numpy application binary interface[5]—the fact that Python is free and open source helps ease such transitions. For example, researchers can run multiple versions of Python at the same time to support old production codes, which can be prohibitively expensive under a proprietary licensing model. Researchers can also contribute to porting codes forward or backward, which is largely done on the Internet in an open, collaborative fashion. Although this might entail significant work, the key point is that the users—who can always access the entire tool stack—retain control.

We've found clear benefits in having Python designed and maintained by an external community as a general-purpose language using the process of Python enhancement proposals (www.python.org/dev/peps/pep-0001). Not having what appears to be a fundamental type like the array rigidly defined in the language has actually allowed the scientific community to evolve its own array library over the years, fine-tuning it to become increasingly sophisticated. The scientific community maintains a dialog with the language designers, and the core Python team has been highly receptive to scientists' needs, letting the language develop specifically requested features for NumPy and its predecessors while also balancing the competing demands of other Python user communities. This dialog and the vetting of a solid language design group ensures that the language remains balanced and consistent for all constituencies, even if it means that scientists occasionally don't get a requested feature.

Languages, Environments, and Distributions

Until now, we've looked at Python mostly as a programming language. In practice, however, scientists use rich environments with lots of functionality, and integrating multiple tools is an attractive aspect of commercial technical computing systems.

Tools

As we now summarize, in the Python world, scientists have collaboratively developed an open ecosystem of foundational tools that provide the key functionality needed for everyday computing work.

Interactive, exploratory work. IPython[3] offers an end-user environment for interactive work, a component to embed in other systems to provide an interactive control interface, and an abstraction of these ideas over the network for interactive distributed and parallel computing. Customized versions of IPython are used in multiple systems—including Sage, which also provides a Web-based graphical notebook interface that lets users combine code, text, and graphics in a single environment.

Numerical arrays. All Python numerical codes today are based on the NumPy library, and the NumPy array is the basic data structure that virtually all libraries understand as a common data interchange object. NumPy arrays have a rich Python interface, but they also can be readily accessed from C, C++, and Fortran, making it easy to optimize performance bottlenecks or reuse existing libraries that have no knowledge of Python.

Data visualization. matplotlib is the most widely used library for high-quality plotting, with support for a wide array of 2D and 3D plot types, precise layout control, a built-in LaTeX typesetting engine for label equations, and publication-quality output in all major image formats.[4]

The Chaco library is often used for building graphical interactive interfaces that tightly couple 2D data visualization to user controls. For high-end data visualization in three dimensions, Mayavi[6] provides both a rich GUI to the powerful Visualization ToolKit (VTK) libraries and an easy to use Python library. As Prabhu Ramachandran and Gaël Varoquaux describe in "Mayavi: 3D Visualization of Scientific Data" on page 40, Mayavi wraps much of VTK's complexity in high-level objects that are easy to use for common tasks and directly support NumPy arrays. Finally, the VisIt (https://wci.llnl.gov/codes/visit) and ParaView (www.paraview.org) projects provide comprehensive visualization systems with parallel rendering support and rich feature sets that users can control and extend in Python.

Algorithms. The SciPy package (www.scipy.org) provides a powerful collection of tools, both by wrapping existing libraries such as Lapack, FFTPack (a Fortran subroutine library of fast Fourier transforms), and the Arnoldi Package (ARPack), and with new code written for SciPy itself. SciPy has support for dense and sparse linear algebra, signal processing, optimization, numerical integration and ordinary differential equations, special functions, statistics, data fitting, and much more.

The Scikits (http://scikits.appspot.com/scikits) project offers add-on packages for SciPy, developed by teams focused on a specific application domain; today there are Scikits for audio and image processing, time series analysis, statistical model description, environmental time series manipulation, machine learning, speech processing, and more. Also, many more projects are available on the Internet that provide specialized tools, including notable contributions from various US national laboratories such as Python bindings to the Portable, Extensible Toolkit for Scientific Computation (PETSc) and Trilinos solvers, and the Networkx[7] graph theory package.

Performance. Fortran codes (such as those included in SciPy) have been traditionally accessed via f2py, a tool that generates Python wrappers for Fortran libraries included with NumPy. As Stefan Behnel and his colleagues describe in "Cython: The Best of Both Worlds," on page 31, the Cython project (www.cython.org) can be described as an extension to the Python language with type annotations. Cython source code is processed by the Cython compiler; this produces pure C code that can then be compiled into a Python extension module (a `.so`, `.dylib`, or `.dll` dynamically linked library). Cython supports NumPy arrays and is an extremely powerful way to easily optimize existing Python code, as well as to link to C and C++ libraries (Sage uses Cython extensively in both ways). Recently, Cython has also added Fortran support.

The Simplified Wrapper Interface Generator (SWIG) system[8] can automatically wrap large C and C++ libraries after you manually write an initial set of interface descriptions. The `ctypes` module (included with Python itself) can call any dynamic library at runtime, making it possible to interface even with libraries that have no available source code; we know of many scientists who use this module to interface with hardware devices for which manufacturers only provide a Windows DLL and no source code.

Symbolic manipulation. Python doesn't have a native notion of symbols like languages such as Mathematica, but the Sympy package (www.sympy.org) offers basic symbolic objects and a growing collection of symbolic manipulation tools, from simple algebra to Gröbner bases.

Documentation. Properly documenting scientific codes is a critical issue that is often (and unfortunately) given insufficient attention. In Python, the powerful Sphinx system (sphinx.pocoo.org) makes it easy to create well-formatted documentation in HTML and PDF formats, including mathematical markup with LaTeX support, figures, and code examples that can be executed for validation as part of the build process.

The Python language itself is documented using Sphinx, and by now virtually all scientific projects have standardized on it as well; we therefore now have a set of uniform tools and extensions for producing documentation that we all benefit from. Although the responsibility of actually *writing* the documentation still falls on the code's authors, an efficient and flexible toolchain makes the process much easier. Indeed, we've already seen that this leads to projects having more and better quality documents.

The SciPy community has developed a Web-based system that allows anyone to easily contribute to NumPy and SciPy documentation. User contributions are peer-reviewed and improved until they meet the required quality standard for project incorporation. This system has greatly improved documentation quality and solved the common conundrum of having only the developers write documentation. It also offers a great way for newcomers to make contributions before they feel comfortable modifying the code. Because the Sphinx documentation system is written in Python itself, the scientific Python community has written several extensions. For example, users can embed source code including IPython constructs, NumPy, SciPy, and matplotlib directly into the document source; this ensures that example code, output, and figure generation are run at documentation build time.

Distributions

One of the main attractions of systems like Matlab and Mathematica is that—in addition to a programming language and libraries—they provide a complete working environment, or distribution, in a single download. Earlier, we described individual components for many tasks, but thanks to the hard work of numerous individuals, groups, and companies, there are now many distributions that provide this for the Python ecosystem, each with a different focus.

This is akin to how many Linux distributions work, where the various packagers (such as RedHat, Debian, and Ubuntu) integrate the vast collection of open source software into a coherent environment that users can install and update. Each of the collections described below includes a different set of tools, but all ship the basic ones we listed earlier:

- *Enthought Python Distribution* (EPD) is a large collection of packages available natively for OS X, various Unix systems, and Windows. The included software is all open source, but the bundled packaging, installer, and support are commercially licensed (similar to how RedHat Enterprise Linux works, for example). The 32-bit version is available for free download to academic users.
- *PythonXY* is a collection for Windows that emphasizes support for the Qt libraries for GUI development. PythonXY is distributed under the terms of the GNU General Public License version 3.
- *PyIMSL Studio* offers the basic set of tools along with the well-known IMSL (International Mathematics and Statistics Library) numerical libraries and their Python bindings, distributed under a commercial license from Visual Numerics.
- *Sage* is a complete environment that integrates most of the packages we discuss here (and many more) into a self-contained system for mathematical computing based on Python but with its own language extensions. Because Sage's GUI is via a Web browser, it doesn't ship matplotlib or Mayavi GUI components, which require GUI toolkits such as WxWidgets and Qt. Sage is distributed under the terms of the GNU General Public License version 2.

Finally, on modern Linux distributions, most of the individual open source Python libraries we've mentioned (including IPython, NumPy, SciPy, matplotlib, the Enthought tool suite, Cython, Sympy, and Networkx) are available for installation via the built-in package management system. Some distributions also include older releases of Sage, but we prefer to use a more up-to-date Sage version downloaded directly from the project's website.

Just a few years ago, getting a complete set of Python tools for scientific computing installed on a system was a daunting effort. Now, thanks to the efforts of these projects, the situation is drastically different. We've successfully used EPD for teaching multiple scientific computing workshops, where students at the high school, college, and graduate level have self-installed the platform with "one-click" installers. This has made an enormous difference in the ease with which newcomers can get started with the language. Users have several options, depending on their specific needs. Some might want the convenience of an integrated, single-click installer like those in EPD, Sage, or PythonXY, while others might need customized installations. Sage fully supports building the entire system from source from a single makefile, so that users can adapt the project to their local needs. In special environments with custom hardware or limited storage, users can forgo integrated downloads and opt to use only those libraries that they need. This freedom can be invaluable and is something that can't be done with large commercial systems that are only provided by the vendors for a limited number of architectures.

Getting Started

After installing one of the prepackaged distributions we described earlier, you can take advantage of the vast amount of documentation—both free and for pay, online and in print—to speed your way to using Python and the core tools discussed here.

The official Python tutorial (http://docs.python.org/tutorial) provides a solid introduction to the language; for more depth, *The Python Essential Reference*[9] provides an excellent treatment of the language and standard library. The SciPy documentation portal (http://scipy.org/doc) provides links to reference and user documentation for Numpy and SciPy, and the additional documentation section (http://scipy.org/Additional_Documentation) offers links to many helpful tutorials and overviews, such as a guide to transitioning from Matlab to Python and Numpy.

In 2007, Travis Oliphant wrote an article in a previous *CiSE* special issue on Python detailing its strengths for scientific computing.[2] At that point, many of the core pieces of the puzzle were only beginning to solidify, the documentation was scattered and incomplete, the Sage system was in its infancy, and systems like Cython, Sympy, and the SciPy scikits infrastructure didn't yet exist. Today, although much work remains, we can safely say that the community has found a core set of effective tools and is busy developing new directions based on this foundation.

Python still lags behind projects such as R in offering a full library of statistical and time-series functionality, but rapid progress is being made on this front in pystatsmodels and other related projects. We now have in Python a rich environment based on a solid and well-designed language and with increasingly high-quality tools. This ecosystem, which emerged from a fruitful and open collaboration of academics and industry partners, lets scientists, engineers, and educators find precisely the parts they need. Having well-documented, open source computational tools will be increasingly important as computing becomes an integral component of all scientific research and engineering work; fortunately, all the projects we have listed here continue to grow, and important new areas are being tackled.

Python has now entered a phase where it's clearly a valid choice for high-level scientific code development, and its use is rapidly growing. This growth is occurring thanks to the dedicated work of many contributors throughout the past two decades; we hope that many more will follow in kind, finding creative ways to solve many problems that remain.

Acknowledgments

We thank Guido van Rossum for comments regarding Python's history, and Ariel Rokem, William Stein, Michael Trumpis, and Stéfan van der Walt for valuable feedback on the entire article. We also thank the anonymous reviewers for their careful reading and detailed comments. The US National Institutes of Health partially support Fernando Pérez's work under grant 5R01MH081909-02.

References

1. K. Asanovic et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, tech. report UCB/EECS-2006-183, Electrical Eng. and Computer Science Dept., Univ. California, Berkeley, 2006.
2. T. Oliphant, "Python for Scientific Computing," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 10–20.
3. F. Pérez and B.E. Granger, "IPython: A System for Interactive Scientific Computing," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 21–29.
4. J.D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 90–95.
5. T. Oliphant, *Guide to NumPy*, Tregol Publishing, 2006.
6. P. Ramachandran and G. Varoquaux, "Mayavi: Making 3D Data Visualization Reusable," *Proc. 7th Python in Science Conf.*, SciPy Community, 2008, pp. 51–56.
7. A.A. Hagberg, D.A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function Using NetworkX," *Proc. 7th Python in Science Conf.*, SciPy Community, 2008; http://conference.scipy.org/proceedings/SciPy2008/paper_2.
8. D.M. Beazley, "Automated Scientific Software Scripting with SWIG," *Future Generation Computing Systems*, vol. 19, no. 5, 2003, pp. 599–609.
9. D.M. Beazley, *Python Essential Reference*, 4th ed., Addison-Wesley, 2009.

***Fernando Pérez** is an associate researcher at the University of California, Berkeley's Helen Wills Neuroscience Institute. His research interests include developing methods and tools for analyzing neuroimaging data, using high-level languages for scientific computing, and exploring new approaches to distributed and parallel problems. He contributes to the scientific Python ecosystem as a developer and educator; he is the original author of the IPython interactive environment and leads its development in collaboration with Brian Granger. Pérez has a PhD in theoretical physics from the University of Colorado at Boulder. Contact him at Fernando.Perez@berkeley.edu.*

***Brian Granger** is an assistant professor in the Physics Department at Cal Poly State University, San Luis Obispo, California. His research interests include computational quantum mechanics and parallel and distributed computing in scientific computing. He leads the IPython development in collaboration with Fernando Pérez. Granger has a PhD in theoretical physics from the University of Colorado. Contact him at bgranger@calpoly.edu.*

***John Hunter** is senior quantitative analyst at TradeLink Securities. Hunter has a PhD in neurobiology at the University of Chicago, and is the author and lead developer of the matplotlib scientific visualization package. Contact him at jdh2358@gmail.com.*