unyt: Handle, manipulate, and convert data with units in Python

24 May 2018

Summary

Software that processes data with physical units or that models real world quantities with physical units must have some way of managing units. This might be as simple as the convention that all floating point numbers are understood to be in the same physical unit system (for example, the SI MKS units system). While simple approaches like this do work in practice, they also are fraught with possible error, both by programmers modifying the code who unintentionally misinterpret the units, and by users of the software who must take care to supply data in the correct units or who need to infer the units of data returned by the software.

The `unyt` library is designed both to aid quick calculations at an interactive python prompt and to be tightly integrated into a larger Python application or library. To aid quick calculations, the top-level `unyt` namespace ships with a large number of predefined units and physical constants to aid setting up quick calculations without needing to look up unit data or the value of a physical constant. Using the `unyt` library as an interactive calculation aid only requires knowledge of basic Python syntax and awareness of a few of the methods of the `unyt_array` class - for example, the `unyt_array.to()` method to convert data to a different unit. As the complexity of the usage increases, `unyt` provides a number of optional features to aid these cases, including custom unit registries containing both predefined physical units as well as user-defined units, built-in output to disk via the pickle protocol and to HDF5 files using the h5py library (Collette 2013), and round-trip conversions to create units compatible with other popular Python unit libraries.

Physical units in the `unyt` class are defined in terms of the dimensions of the unit, a string representation, and a floating point scaling to the MKS unit system. Rather than implementing algebra for unit expressions, we rely on the `SymPy` symbolic algebra library (Meurer et al. 2017) to handle symbolic algebraic manipulation. The `unyt.Unit` object can represent arbitrary units

formed out of the seven base dimensions in the SI unit system: time, length, mass, temperature, luminance, electric current, and amount of a substance. In addition, `unyt` supports forming quantities defined in other unit systems - in particular CGS Gaussian units common in astrophysics as well as geometrized "natural" units common in relativistic calculations. In addition, `unyt` ships with a number of other useful predefined unit systems based, including imperial units, Planck units, a unit system for calculations in the solar system, and a galactic unit system.

In addition to the `unyt.Unit` class, `unyt` also provides a two subclasses of the NumPy (Oliphant 2006) ndarray (Walt, Colbert, and Varoquaux 2011), `unyt.unyt_array` and `unyt.unyt_quantity` to represent arrays and scalars with units attached, respectively. In addition, `unyt` provides a `unyt.UnitRegistry` class to allow custom systems of units, for example to track the internal unit system used in a simulation. These subclasses are tightly integrated with the NumPy ufunc system, which ensures that algebraic calculations that include data with units automatically check to make sure the units are consistent, and allow automatic converting of the final answer of a calculation into a convenient unit.

We direct users interested in usage examples and a guide for integrating `unyt` into an exiting Python installation to the unyt documentation at hosted at http://unyt.readthedocs.io/en/latest/.

Comparison with `Pint` and `astropy.units`

The scientific python ecosystem has a long history of efforts to develop a library to handle unit conversions and enforce unit consistency. For a relatively recent review of these efforts, see (Bekolay 2013). While we won't exhaustively cover extant Python libraries for handling units in this paper, we will focus on Pint (Grecco 2018) and `astropy.units` (The Astropy Collaboration et al. 2018), which both a provide a robust implementation of an array container with units and are commonly used in research software projects. At time of writing a GitHub search for `import astropy.units` returns approximately 10,500 results and a search for `import pint` returns approximately 1,500 results.

While `unyt` provides functionality that overlaps with `astropy.units` and Pint, there are important differences which we elaborate on below. In addition, it's worth noting that all three codebases had origins at roughly the same time period. In the case of `unyt`, it originated via the `dimensionful` library (Stark 2012) in 2012. A few years later, the `dimensionful` was elaborated on and improved to become `yt.units`, the unit system for the `yt` library (Turk et al. 2011) at a `yt` developer workshop in 2013 and was subsequently released as part of `yt 3.0` in 2014. Similarly, Pint initially began development in 2012 according to the git repository logs, and `astropy.units` was added in 2012 and was released as part

of `astropy 0.2` in 2013, although the initial implementation was adapted from the `pynbody` library (Pontzen et al. 2013), which started in 2010 according to the git repository logs. That is to say, all three libraries began roughly at the same time and are examples in many ways of convergent evolution in software. We have decided to repackage and improve `yt.units` in the form of `unyt` to both make it easier to work on and improve the unit system and encourage use of the unit system for scientific python users who do not want to install a heavy-weight dependency like `yt`.

Below we present a table comparing `unyt` with `astropy.units` and Pint. Estimates for lines of code in the library were generated using the `cloc` tool (Danial 2018); blank and comment lines are excluded from the estimate. Test coverage was estimated using the `coveralls` output for Pint and `astropy.units` and using the `codecov.io` output for `unyt`.

Library	`unyt`	`astropy.units`	Pint
Lines of code	5128	10163	8908
Lines of code excluding tests	3195	5504	4499
Test Coverage	99.91%	93.63%	77.44%

We offer lines of code as a very rough estimate for the "hackability" of the codebase. In general, smaller codebases with higher test coverage are have fewer defects (Lipow 1982; Koru, Zhang, and Liu 2007; Gopinath, Jensen, and Groce 2014). This comparison is somewhat unfair in favor of `unyt` in that `astropy.units` only depends on NumPy and Pint has no dependencies, while `unyt` depends on both `sympy` and NumPy. Much of the reduction in the size of the `unyt` library can be attributed to offloading the handling of algebra to `sympy` rather than needing to implement the algebra of unit symbols directly in `unyt`. For potential users who are wary of adding `sympy` as a dependency, that might argue in favor of using Pint in favor of `unyt`.

Astropy.units

The `astropy.units` subpackage provides a `PrefixUnit` class, a `Quantity` class that represents both scalar and array data with attached units, and a large number of predefined unit symbols. The preferred way to create `Quantity` instances is via multiplication with a `PrefixUnit` instance. Similar to `unyt`, the `Quantity` class is implemented via a subclass of the NumPy `ndarray` class. Indeed, in many ways the everyday usage patterns of `astropy.units` and `unyt` are similar, although `unyt` is not quite a drop-in replacement for `astropy.units` as there are some API differences. The main functional difference between `astropy.units` and `unyt` is that `astropy.units` is a subpackage of the larger `astropy` package. This means that depending on `astropy.units` requires depending on a large collection of astronomically focused software, including

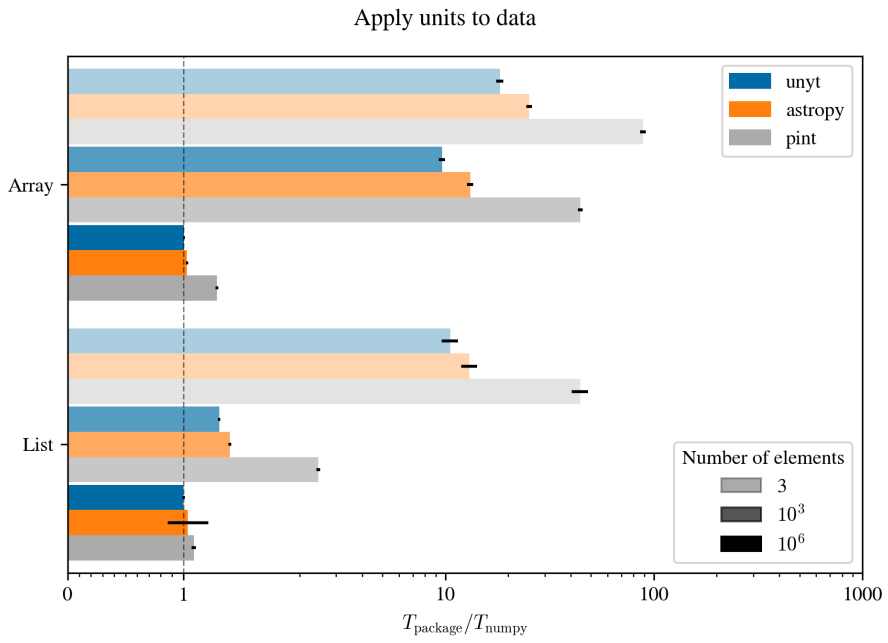

Figure 1: A benchmark comparing the time to apply units to lists and NumPy `ndarray` instances. Each test is shown for three different sizes of input data, including inputs with size 3, 1,000, and 1,000,000. The black lines at the top of the bars indicate the sample standard deviation. The T_{numpy} time is calculated by benchmarking the time to perform `np.asarray(data)` where `data` is either a `list` of an `ndarray`.

a substantial amount of compiled C code. For users who are not astronomers or do not need the observational astronomy capabilities provided by `astropy`, depending on all of `astropy` just to use `astropy.units` may be a tough sell.

Pint

The Pint package provides a somewhat different API compared with `unyt` and `astropy.units`. Rather than making units immediately importable from the Pint namespace, instead Pint requires users to instantiate a `UnitRegistry` instance (unrelated to the `unyt.UnitRegistry` class), which in turn has `Unit` instances as attributes. Just like with `unyt` and `astropy.units`, creating a `Quantity` instance requires multiplying an array or scalar by a `Unit` instance. Exposing the `UnitRegistry` directly to all users like this does force users of the library to think about which system of units they are working with, which may be beneficial in some cases, however it also means that users have a bit of extra cognitive overhead they need to deal with every time the use Pint.

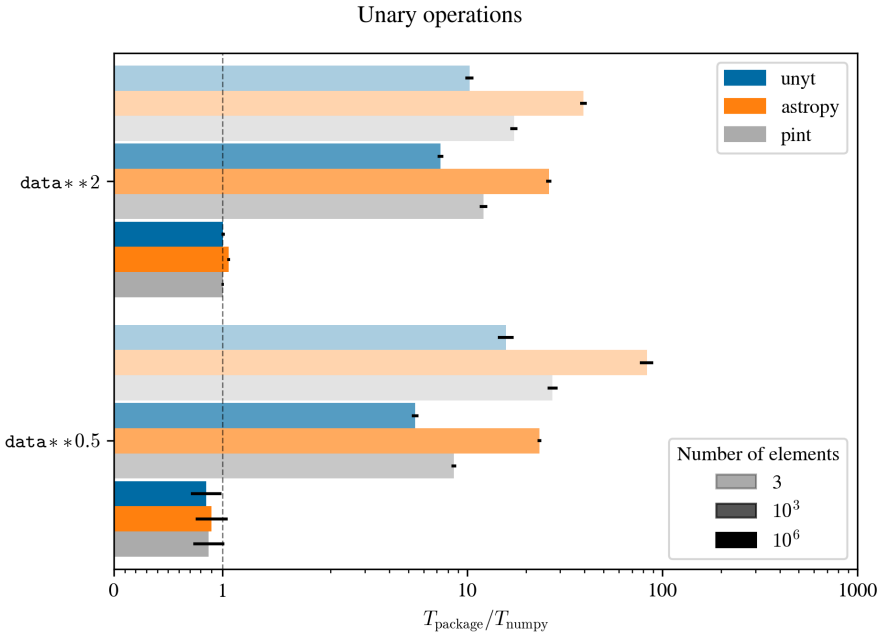

Figure 2: A benchmark comparing the time to square an array and to take the square root of an array. See Figure 1 for a detailed explanation of the plot style.

In addition, the `Quantity` class provided by Pint is not a subclass of NumPy's ndarray. Instead, it is a wrapper around an internal `ndarray` buffer. This

somewhat simplifies the implementation of Pint by avoiding the somewhat arcane process for creating an ndarray subclass, although the Pint `Quantity` class must also be careful to emulate the full NumPy `ndarray` API so that it can be a drop-in replacement for `ndarray`.

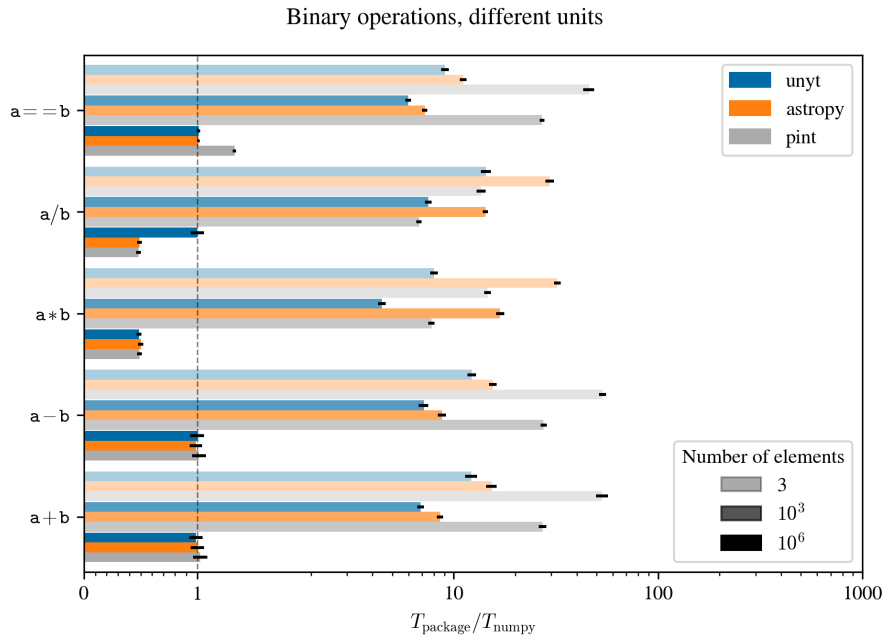

Figure 3: A benchmark comparing the time to perform various binary arithmetic operations on input operans that have different but dimensionallty compatible units. See Figure 1 for a detailed explanation of the plot style.

Finally, in carefully comparing the output of scripts using Pint, `astropy.units`, and `unyt`, we found that in-place operations making use of a NumPy ufunc will unexpectedly strip units. For example, if `a` and `b` are Pint `Quantity` instances, `np.add(a, b, out=out))` will operate on `a` and `b` as if neither have units attached. Interestingly, without the `out` keyword, Pint does get the correct answer, so it's possible that this is a bug in Pint which we have reported upstream (see https://github.com/hgrecco/pint/issues/644).

Performance Comparison

Checking units will always add some overhead over using hard-coded unit conversion factors. Thus a library that is entrusted with checking units in an application should incur the minimum possible overhead to avoid triggering

performance regressions after integrating unit checking into an application. Optimally, a unit library will add zero overhead regardless of the size of the array. In practice that is not the case for any of the three libraries under consideration, and there is a minimum array size above which the overhead of doing a mathematical operation exceeds the overhead of checking units. It is thus worth benchmarking unit libraries in a fair manner, comparing with the same operation implemented using plain NumPy.

Here we present such a benchmark. We made use of the `perf` (Stinner 2018) Python benchmarking tool, which not only provides facilities for establishing the statistical significance of a benchmark run, but also can tune a linux system to turn off operating system and hardware features like CPU throttling that might introduce variance in a benchmark. We made use of a Dell Latitude E7270 laptop equipped with an Intel i5-6300U CPU clocked at 2.4 Ghz. The testing environment was based on `Python 3.6.3` and had `NumPy 1.14.2`, `sympy 1.1.1`, `fastcache 1.0.2`, `Astropy 3.0.1`, and `Pint 0.8.1` installed. `fastcache` (Brady 2017) is an optional dependency of `sympy` that provides an optimized LRU cache implemented in C that can substantially speed up `sympy`.

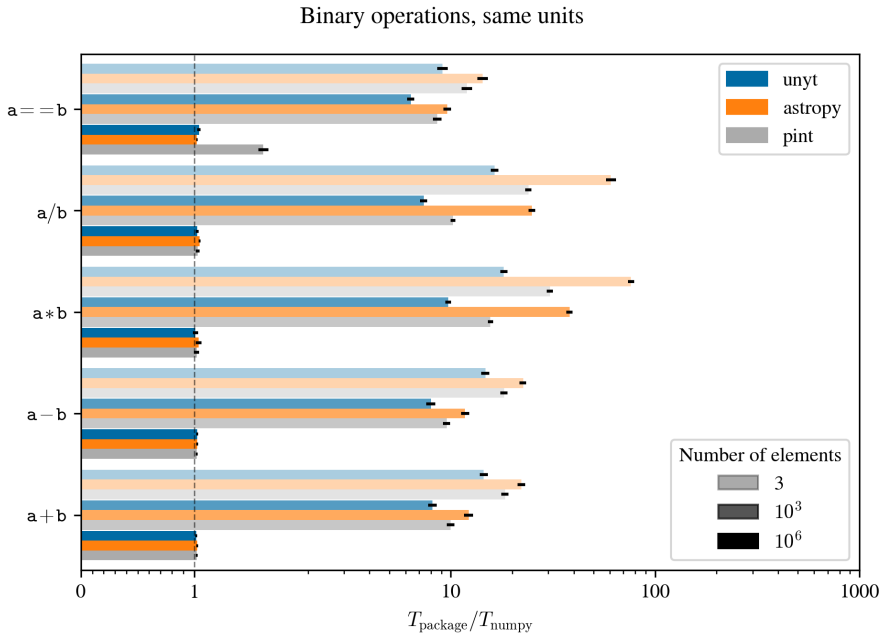

Figure 4: A benchmark comparing the time to perform various binary arithmetic operations on input operands that have the same units. See Figure 1 for a detailed explanation of the plot style.

For each of the benchmarks below, we show the ratio of the time to perform an

operation with one of `unyt`, Pint, and `astropy.units`, T_{package}, to the time it takes for NumPy to perform the equivalent operation, T_{numpy}. For example, for the comparison of the performance of `np.add(a, b)` where `a` and `b` have different units with the same dimension, the corresponding benchmark to generate `T_{\rm numpy}` would use the code `np.add(a, c*b)` where `a` and `b` would be `ndarray` instances and `c` would be the floating point conversion factor between the units of `a` and `b`. Much of the time in `T_{\rm package}` relative to `T_{\rm numpy}` is spent in the respective packages calculating the appropriate conversion factor `c`. Thus the comparisons below depict very directly the overhead for using a unit library over an equivalent operation that uses hard-coded unit-conversion factors.

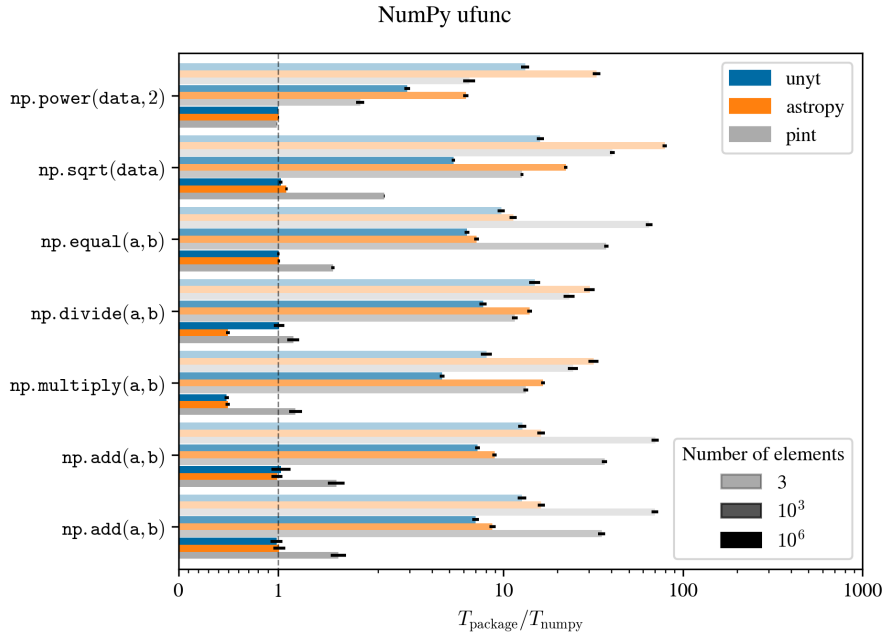

Figure 5: A benchmark comparing the overhead for computing various NumPy `ufunc` operations. The operands of all binary ufuncs have the same units. See Figure 1 for a detailed explanation of the plot style.

Applying units to data

In Figure 1 we plot the overhead for applying units to data, showing both Python lists and NumPy `ndarray` instances as the input to apply data to. Since all three libraries eventually convert input data to a NumPy `ndarray`, the comparison with array inputs more explicitly shows the overhead for *just* applying units to

data. When applying units to a list, all three libraries as well as NumPy need to first copy the contents of the list into a NumPy array or a subclass of `ndarray`. This explains why the overhead is systematically lower when starting with a list.

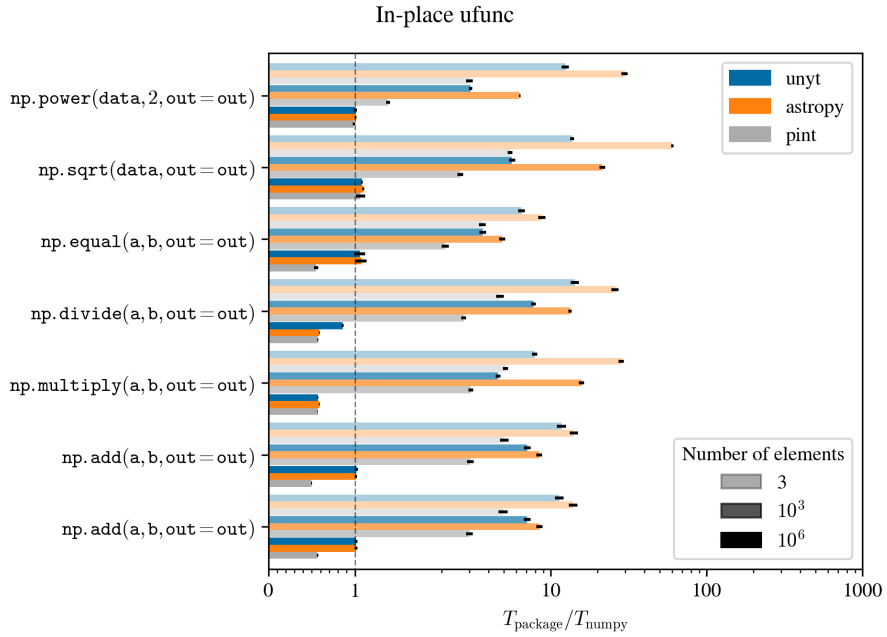

Figure 6: The same as Figure 5, but with in-place `ufunc` operations. See Figure 1 for a detailed explanation of the plot style.

In all cases, `unyt` either is fastest by a statistically significant margin, or ties with `astropy`. Even for large input arrays, Pint still has statistically significant overhead, while both `unyt` and `astropy.units` have negligible overhead once the input array size reaches 10^6 elements.

Unary arithmetical operations

Expressions involving powers of data with units, including integer and fractional powers, are very common in the physical sciences. It is therefore very important for a library that handles units to be able to track this case in a performant way. In Figure 2 we present a benchmark comparing Pint, `unyt`, and `astropy.units` for the squaring and square root operation. In all cases, `unyt` has the lowest overhead, with Pint coming in second, and `astropy.units` trailing. Note that the y-axis is plotted on a log scale, so `astropy` is as much as 4 times slower than `unyt` for these operations.

Binary arithmetical operations

Binary operations form the core of arithmetic. It is vital for a library that handles unit manipulation to both transparently convert units when necessary and to ensure that expressions involving quantities with units are dimensionally consistent. In Figure 3 and 4 we present benchmarks for binary arithmetical expressions, both with input data that has the same units and with input data with different units but the same dimensions. In most cases, `unyt` has less overhead than both `astropy` and `Pint`, although there are a few anomalies that are worth explaining in more detail. For comparison operations, Pint exhibits a slowdown even on large input arrays. This is not present for other binary operations, so it's possible that this overhead might be eliminated with a code change in Pint. For multiplication on large arrays, all three libraries have measured overhead of ~ 0.5 that of the "equivalent" Numpy operation. That is because all three libraries produce a results with units given by the product of the input units, that is, there is no need to multiply the result of the multiplication operation by an additional constant, while a pure NumPy implementation would need to multiply the result by a constant to keep the units consistent. Finally, for division, both Pint and `astropy.units` exhibit the same behavior as for multiplication, and for similar reasons: the result of the division operation is output with units given by the ratio of the input units. On the other hand, `unyt` will automatically cancel the dimensionally compatible units in the ratio and return a result with dimensionless units.

NumPy `ufunc` performance

Lastly, In Figures 4 and 5, we present benchmarks of NumPy `ufunc` operations. A NumPy `ufunc` is a fast C implementation of a basic mathematical operation. This includes arithmetical operators as well as trigonometric and special functions. By using a `ufunc` directly, one bypasses the Python object protocol and short-circuits directly to the low-level NumPy math kernels. We show both directly using the NumPy `ufunc` operators (Figure 4) as well as using the same operators with a pre-allocated output array.

Acknowledgements

References

Bekolay, Trevor. 2013. "A Comprehensive Look at Representing Physical Quantities in Python." https://www.youtube.com/watch?v=N-edLdxiM40.

Brady, Peter. 2017. "Fastcache." *GitHub Repository.* https://github.com/pbrady/fastcache; GitHub.

Collette, Andrew. 2013. *Python and Hdf5.* O'Reilly.

Danial, Al. 2018. "Cloc." *GitHub Repository.* https://github.com/AlDanial/cloc; GitHub.

Gopinath, Rahul, Carlos Jensen, and Alex Groce. 2014. "Code Coverage for Suite Evaluation by Developers." In *Proceedings of the 36th International Conference on Software Engineering*, 72–82. ICSE 2014. New York, NY, USA: ACM. https://doi.org/10.1145/2568225.2568278.

Grecco, Hernan E. 2018. "Pint." *GitHub Repository.* https://github.com/hgrecco/pint; GitHub.

Koru, A. G., D. Zhang, and H. Liu. 2007. "Modeling the Effect of Size on Defect Proneness for Open-Source Software." In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, 10–10. https://doi.org/10.1109/PROMISE.2007.9.

Lipow, M. 1982. "Number of Faults Per Line of Code." *IEEE Trans. Softw. Eng.* 8 (4). Piscataway, NJ, USA: IEEE Press: 437–39. https://doi.org/10.1109/TSE.1982.235579.

Meurer, Aaron, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, et al. 2017. "SymPy: Symbolic Computing in Python." *PeerJ Computer Science* 3 (January): e103. https://doi.org/10.7717/peerj-cs.103.

Oliphant, Travis E. 2006. *A Guide to Numpy.* Trelgol Publishing.

Pontzen, A., R. Roškar, G. S. Stinson, R. Woods, D. M. Reed, J. Coles, and T. R. Quinn. 2013. "pynbody: Astrophysics Simulation Analysis for Python."

Stark, Casey W. 2012. "Dimensionful." *GitHub Repository.* https://github.com/caseywstark/dimensionful; GitHub.

Stinner, Victor. 2018. "Perf." *GitHub Repository.* https://github.com/vstinner/perf; GitHub.

The Astropy Collaboration, A. M. Price-Whelan, B. M. Sipőcz, H. M. Günther, P. L. Lim, S. M. Crawford, S. Conseil, et al. 2018. "The Astropy Project: Building an inclusive, open-science project and status of the v2.0 core package." *ArXiv E-Prints*, January.

Turk, M. J., B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. 2011. "yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data." *The Astrophysical Journal Supplement Series* 192 (January): 9. https://doi.org/10.1088/0067-0049/192/1/9.

Walt, S. van der, S. C. Colbert, and G. Varoquaux. 2011. "The Numpy Array: A Structure for Efficient Numerical Computation." *Computing in Science Engineering* 13 (2): 22–30. https://doi.org/10.1109/MCSE.2011.37.