Astronomical Data Analysis Software and Systems XX
ASP Conference Series, Vol. 442
Ian N. Evans, Alberto Accomazzi, Douglas J. Mink, and Arnold H. Rots, eds.

What Python Can Do for Astronomy

Perry Greenfield

Space Telescope Science Institute, 3700 San Martin Dr., Baltimore, MD 21218, USA

Abstract. Python continues to see increased use in astronomy, both by institutions developing software for new instruments, telescopes and missions, and by astronomers for use with analyzing their data. I will describe the advances in Python-based tools for astronomy in the past few years and the current effort to bring those tools together into a common repository. What are the most serious challenges facing the use of Python in astronomy and how should these be addressed? Finally, I'll discuss the non-technical contributions that Python can bring to astronomy and illustrate with an example and suggestions.

1. Suitability of Python for Astronomy

The positive aspects of Python as a language or its associated libraries are generally well known and will not be detailed at length here. It is a very accessible language for casual programmers, particularly astronomers, while also being a very powerful language. This enables many of the same tools to be shared by both researchers and for production quality programs. Python has clearly become the scripting language of choice for nearly all new large astronomy projects, and is gaining wider use among astronomers as well. While many consider it to be only a scripting language, there is also increasing usage of Python as an applications language.

The major downside of Python is its speed. Primitive operations are many tens of times slower than corresponding operations in low-level compiled languages such as C, C++, or Fortran. But this is not nearly as serious an obstacle as it may appear at first glance. In practical applications, relatively small fractions of the code need to be efficient. And where it needs to be efficient, there are often high level operations available that run efficiently in Python because they perform a lot of work underneath in C. For example, array operations in Python are efficiently carried out by the numpy package so long as operations are performed on large parts of the array with numpy primitives. And when no language libraries or constructs are available, it is not difficult to call code written in C (or other languages) to perform operations that must be done efficiently at a low level.

Another issue viewed a liability is Python's Global Interpreter Lock. This prevents multiple Python threads from running simultaneously. Thus, multi-threaded applications do not exploit the now common multi-core processors. The Python community generally frowns on the use of threads for such performance gains (for more reasons than performance) and generally suggests the use of multiple processes for parallel pro-

cessing. In any event, the efficient use multiple processors with Python is a topic that has many in the community investigating the best ways to do so.

In our view, the ease of installation is probably our greatest concern. Since the tools needed to process and visualize data in Python have many third-party dependencies, each with its own peculiar means of installation, installations can consist of several independent steps, each having possible snags leading to more difficulties than most users wish to deal with. This is an area that we are investing significant effort that will be described briefly later.

The suitability of Python as an applications language has been enabled by great progress in the past decade in the improvement of the tools that permit it to be used effectively to manipulate astronomical data. Both numpy and matplotlib now generally surpass the capabilities that are present in IDL for array manipulation and 2-d plotting and in some areas far surpass the equivalent capabilities. Likewise the same is true for PyFITS. No other language has the equivalent functionality to PyRAF. These combined with such tools as ipython, mayavi, and scipy give Python a wealth of basic tools to analyze and visualize data. STScI is pleased to have played an important part for many of these.

Nevertheless, there is no question that Python does not yet have the equivalent functionality of IRAF or IDL in the astronomical applications area, or in specialized astronomical libraries. This is clearly an area for improvement. It is worth noting that the past two years have seen a blossoming of activity in contributed astronomical packages such as:

APLy Astronomical plotting utilities. `http://aplpy.sourceforge.net/`

Asciitable Read and write ascii tables.
`http://cxc.harvard.edu/contrib/asciitable/`

astlib Astronomy utilities. `http://astlib.sourceforge.net/`

astronomical utilities `http://www.astro.washington.edu/users/rowen/`

astropysics Astrophysics utilities. `http://packages.python.org/Astropysics/`

ATpy generic Table module. `http://atpy.sourceforge.net/`

cosmology calculator `http://www.astro.ucla.edu/~wright/CC.python`

cosmolopy Cosmology tools. `http://roban.github.com/CosmoloPy/`

IDLSave Module to read IDL save files. `http://idlsave.sourceforge.net`

Kapteyn package Plotting, coordinate, tables utilities.
`http://www.astro.rug.nl/software/kapteyn/`

Pandora `http://cosmos.iasf-milano.inaf.it/pandora/software.html`

PyEphem Ephemeris tools. `http://rhodesmill.org/pyephem/`

pygalkin Tools for spectral cubes. `http://code.google.com/p/pygalkin/`

Python-Montage Python interface to Montage.
`http://astrofrog.github.com/python-montage`

Python scripts for astronomy
`http://www.atnf.csiro.au/people/Enno.Middelberg/python/python.html`

pytpm Python interface to the TPM library. `http://phn.github.com/pytpm`

pywcsgrid2 Astronomical image grid overlay and labeling tools.
`http://leejjoon.github.com/pywcsgrid2`

sampy Python SAMP implementation.
`http://cosmos.iasf-milano.inaf.it/pandora/sampy.thml`

Most of these have been contributed by individuals, not institutions. In addition to these we have significant Python contributions from larger institutional efforts that have been around awhile (e.g., STScI, NRAO, Chandra/SAO, ESO/ST-ECF). There is good reason to believe we will see much improvement in this area over the next few years.

2. Future STScI Community-Oriented Efforts for Python

We have three main areas of effort planned for the immediate future.

2.1. Addressing Installation Ease

The most important is developing an easy-to-install release of AURA software. We have been working with Gemini (and particularly with James Turner) over the past year to package all AURA distributed analysis and reduction software into a "one-step" install. Such a "unified release" will contain Python, IRAF/STSDAS/TABLES, stsci_python, Gemini packages, and all needed dependencies, as well as other third party libraries and tools that may be needed by common add ons. Our goal is that this should support all popular platforms, e.g., Mac OS X and Linux (but given the multiplicity of Linux variants, universal support is unlikely). It should be understood that there is no universal, easy-to-use, foolproof solution to the installation issues of software distribution, nor is one expected in the near future.

The distribution we are planning will allow users to update various software components, most easily with end-level libraries and applications (updating core items such as Python or numpy may require updating the many tools depending on them and therefore may be impractical for most users). Making such a distribution will allow us to start using the many tools available in packages such as scipy or mayavi that we previously have avoided because of installation issues. Users will also be able to install additional packages on top of this distribution (Python packages most easily). Nevertheless, the more the core distribution is updated or modified, the greater likelihood problems may be encountered. We are hoping to balance flexibility with ease of installation and provide users with the choices between the two. We expect to make an early version of this "unified release" available in early 2011.

Part of this distribution system is a nightly testing framework that ensures that new versions of all software components continue to work properly together to ensure its consistency. Such nightly testing highlights potentially disruptive changes in packages from all contributing institutions. This feedback will encourage greater interdependencies between the institutions than permitted by a traditional release cycle where fixing such changes becomes much more difficult given the frequency of such releases.

2.2. Replacing IRAF Functionality

In order to support the JWST mission and reduce dependencies on IRAF we are considering an effort to begin replacing important core IRAF functionalities. This is likely to focus first on spectral data calibration and processing capabilities, then on still useful STSDAS packages such as isophote and restore, and finally on generic image processing tools.

2.3. Fostering Community Python Contributions

We have set up a code repository at `http://astrolib.org`. The repository is intended to help merge the various community contributions into a more integrated and coherent whole. By putting different packages in the same repository it should help drive such packages to a more consistent documentation system and style (based on sphinx, the new standard system of the Python community), and more consistent interfaces to make their interoperability easier. Finally, it should make regular regression testing easier since there will be an existing framework running on many platforms for contributors to take advantage of.

3. What Else Python Can Contribute

Python has more to offer than just a language or libraries. The first Python conference I attended was about 12 years ago. It was small as such conferences go, with only about 90 attendees. Yet it made a great impression on me because I saw a significantly different culture at work than I had seen in the astronomical software community.

3.1. The Python Culture and Mindset

There are aspects of the Python developer culture that are fairly common in many other open source projects, and aspects that aren't. The characterizations that I'll describe are of course generalizations, and one must understand that these are all a matter of degree rather than absolutes. The culture I'll attempt to describe is primarily that of those that contribute to the language itself, its standard libraries, and to a somewhat lesser extent, the heavily used libraries in the community that aren't part of the standard library, and not of the wider community that uses Python.

I'll first note the aspects that are more peculiar to the Python community. These include:

- A steady temperament that is resistant to fads.
- A judicious restraint to adding new features to the language.
- But at the same time, recognizing that evolution is necessary.
- While backward compatibility is important, it isn't an absolute requirement.
- A good balance between practicality and purity.
- Great helpfulness to newcomers in the community.

There isn't much doubt that these reflect the characteristics of the language's originator, Guido van Rossum. The real success of Python goes well beyond the technical merits of the language itself. It is much more due to the community that has grown up to support the language. Without that community the language would not have been as successful as it has been. Guido van Rossum's great contribution has been in creating a large and productive community that leverages the advantages of the language he created.

The Python developer community is large and quite active. It has attracted many very talented and enthusiastic developers. The community is loosely and primarily self-organized. There is very little top-down control or coordination. Guido van Rossum still holds the final say on many decisions, where he chooses between various alternatives, and vetoes or approves various proposals (Guido has been awarded the title "Benevolent Dictator For Life" by the community). But he hasn't done much development himself on the core system over the past several years. This isn't to say that the development process is always orderly. It isn't. Often there are false starts, or strong disagreements on the best approach, leading to various competing alternatives that are tried before there is a clear outcome. Nevertheless, the community is quite cooperative and generally makes great progress. The process exhibits a lot of detailed technical communication. The software produced is usually of very high quality. Most of it is well out in the open, warts and all.

There are other interesting aspects of the developer community. Much of the philosophy is reflected in the "Zen of Python" (which can be displayed by typing "import this" at the python interpreter prompt). Their rapid engagement with different approaches appears to lead to quick and good evaluations of new technologies. Technologies that are viewed as having problems or which are overhyped are fairly quickly discarded or de-emphasized. Their weight in decision making is heavily weighted towards those that actually contribute effort (either in development or use of the new tools). They have little patience for those who want to tell others how to do things without contributing work themselves.

3.2. Contrasted with the Astronomical Software Development Community

On the other hand, much of the software development in the astronomical software community is fragmented, despite sharing many of the same needs. The organizations and individuals in that community usually are protective of their code and slow to share. Cooperative efforts, when they do occur, are committee- and consensus-driven, often lead to very feature-rich goals, and often fail to achieve them, taking a long time in the process. These cooperative efforts are, ironically given the nature of the science, more oriented towards talk and analysis than experimentation. Very often, such efforts are top-down and centrally planned. The community is slow to discard obsolete or failed technologies. In comparison to the Python developer community, the astronomical community appears quite slow moving and not nearly as productive. Is there anything that the astronomical software community can learn from the Python developer community?

3.3. Reasons for Difference

Before trying to answer that question it is important to try to understand the possible reasons for the differences between the two communities. I'll go through various possibilities starting with those I consider least important.

Difference in nature of work or funding
I do not consider this a serious reason. Many in the Python community have similar issues and yet are able to work around them.

Lack of Guidos
I suspect the lack of leaders like Guido is due more to our culture rather than our culture being due to the lack of leaders like Guido.

Lack of theme comedy troupe/enthusiasm
It does appear to me that the astronomical community takes itself more seriously than the Python community, and that there is more enthusiasm in the Python community. This may be due in part to software being secondary to astronomy, and thus getting less respect and correspondingly less enthusiasm (e.g., it isn't necessarily good to be seen as enjoying the software aspect too much as that may not be beneficial to one's career). Likewise, the seriousness may arise out the academic culture taking itself more seriously.

Academic culture
There are possibly a number of aspects to this that are significant. A couple were already mentioned for the previous item. Other elements of the academic culture that may be relevant are:

- Committee/consensus orientation to decision making.
- Territorial views towards software as being part of a competitive research or organizational advantage.
- A view of software issues as having final, long-term (and thus static) solutions much like physics or math problems.

Lack of external competition
This is perhaps the most important reason. The Python community is well aware that it is in competition with other languages. If they wish to see their language thrive, they know they must make good progress in comparison to its competitors. This is a strong motivating force for cooperation. If they don't they may see the language they favor fail to attract new users and eventually die, and with it, all the tools they have invested in it. The astronomical software community has no corresponding motive. If astronomical software is less efficient or more fragmented, astronomy will not cease, and correspondingly, the need for astronomical software will not end.

Lack of a Guido
The Python community has an advantage that when occasions arise in which there is no clear technical winner when multiple alternatives are presented, someone must make a decision. They have a clear leader who commands much respect and who can make that decision.

3.4. Different for Good Reasons

There are clear reasons why the astronomical software community has not developed a culture similar to that of the Python community. But does that mean the astronomical software community cannot learn from the beneficial aspects of the Python culture? I

do not believe that it is clear that it cannot. There does not appear to have been any attempt yet to try. Perhaps the forces that prevented it from forming naturally would prevent its adoption. But we don't really know that.

4. Case Study

To this point the comparisons made have been very high level and general. It is more interesting (and risky) to illustrate with a specific example. There are many examples that could be given, but I'll use one that I think is most important.

4.1. Backward Compatibility in the Python Community

While the Python developer community views backward compatibility as important, it will break it when the benefits are seen to outweigh the costs. Some good examples of this are:

1. Python 3 seriously breaks backward compatibility. Yet the changes were made regardless since these were viewed as necessary in fixing some of Python's intrinsic problems or drawbacks.

2. The Python documentation system underwent major changes when a newer and more maintainable technology (sphinx) became available. It meant redoing all the standard documentation formatting, yet that was accomplished fairly quickly despite the work required. Most of the Python world has fallen in line with the new standard quickly, as well.

3. Transition from Numeric to numarray to numpy made various backward incompatible changes to the array manipulation tools (twice!). Despite the pain, what resulted is far superior to the original package.

Breaking backward compatibility obviously causes problems. On the other hand, it does allow the community to evolve their software to keep up with the rest of the software world. There are costs in not evolving that must be considered along with the costs of revising existing software to deal with incompatible changes.

4.2. Backward Compatibility in the Astronomical Software Community

The astronomical software community generally appears much more resistant to change, and much more unwilling to change standards even when there are great benefits to doing so.

One such example is with the FITS standard. The FITS standard has been a great success. But arguably too great. What is the basis for this shocking claim? The standard originated in the very early 1980s when the software (and hardware) environment was quite different. And even though extensions have been made to it, much of it still reflects very outdated limitations.

I argue that these limitations are making our software harder to develop, understand, and maintain. Furthermore, the data files being generated are less consistent than they could be due to the fragmentation of various approaches used to work around the limitations of the format.

A good example of how FITS limitations have impeded progress is the World Coordinate System standards for FITS. It took about 10 years for the basics aspects to become part of the standard once the effort started. There are aspects still in limbo 16 years after this effort started with no end in sight (i.e., FITS "Paper IV" that deals with distortion representations sees no prospects of near-term acceptance). The existing standard is complex yet inflexible and has arbitrary restrictions imposed by the keyword size restriction (e.g., how many polynomial terms are permitted). Nevertheless, even with the Paper IV additions, it is inadequate for HST data, where we must resort to non-standard representations to fully describe the WCS. The existing WCS approach illustrates the contortions that one must go through to adapt coordinate system transformations to the FITS standard.

There are much better ways to deal with such issues these days, yet the issue of updating or replacing the FITS standard is not really entertained in the community. Adopting or changing standards is so painful that it is a topic widely avoided. While the principle that "once FITS, always FITS" is a sensible requirement, many in the FITS community take that further in requiring any future FITS standard be accessible at some level (e.g., being able to interpret headers) by older FITS readers. Any such restrictions either make it nearly impossible to make significant improvements to the FITS capabilities, or make such improvements convoluted in order to fit the current header restrictions. There is no question that the FITS format is "Turing complete" in a data sense. It does not mean that it is sensible to layer enhancements on top of it.

VOTable was a potential chance to come up with a new standard, yet it was fatally flawed in its inability to deal with binary data efficiently. So now we have two inadequate data formats to support now. In some respects, we are even worse off for it.

Perhaps HDF5 is a suitable replacement. It is an existing standard, and has many features, but its drawback is that it is complex, making software re-implementations of supporting libraries expensive.

There are many possible alternatives that would be clear improvements. It isn't so much the issue of picking the perfect optimum, but rather settling on a solution that is a significant improvement for most cases. There are times to replace standards when they have outlived their usefulness (and many examples exist where standards have changed in the commercial world despite the high cost of switching standards). We are well overdue for such an improvement.

The cost of switching standards is likely overrated. Given the increasing growth of data, the amount of data in older formats will quickly become a minor component of the data available, and correspondingly, a small amount that needs conversion. Secondarily, all of the existing FITS data is on media that is comparatively non-permanent, due to the physical nature of the physical media itself (e.g., magnetic tapes) or the fact that the media may not have devices to read it indefinitely. If such data must be copied anyway to preserve it, inserting a conversion step in the copying process is likely an easy thing.

I don't believe the Python community would have found itself in a similar situation. They are much more aware of the changing nature of the software landscape and much more willing to deal with such issues more quickly. The astronomy community appears to prefer isolating itself more, resulting in a more static environment for themselves, but leading to greater discontinuities in the future with the external software world.

4.3. Conclusions

For those who use Python, look beyond the language and libraries and see what can be learned from the Python community itself. Even non-users of Python may find it beneficial to look at how its developer community works (or one of any number of other successful open source communities) for ideas of how to better collaborate with other projects. Finally, if you produce Python libraries that have potentially greater applicability within astronomy, consider hosting them on `http://astrolib.org`.