Astronomy & Astrophysics manuscript no. astropy
June 3, 2013

Astropy: A Community Python Package for Astronomy

The Astropy Collaboration, Thomas P. Robitaille[1], Erik J. Tollerud[2], Perry Greenfield[3], Michael Droettboom[3], Erik Bray[3], Tom Aldcroft[4], Matt Davis[3], Adam Ginsburg[5], Adrian M. Price-Whelan[6], Wolfgang Kerzendorf[7], Alexander Conley[5], Neil Crighton[1], Kyle Barbary[8], Demitri Muna[9], Henry Ferguson[3], Frederic Grollier, Madhura M. Parikh[10], Prasanth H. Nair[11], Hans M. Günther[4], Christoph Deil[12], Julien Woillez[13], Simon Conseil[14], Roban Kramer[15], James Turner[16], Leo Singer[17], Ryan Fox[11], Benjamin A. Weaver[18], Victor Zabalza[12], Zachary Edwards[19], K. Azalee Bostroem[3], Doug Burke[4], Andy Casey[20], Steve Crawford[21], Nadia Dencheva[3], Justin Ely[3], Tim Jenness[22], Kathleen Labrie[23], Pey Lian Lim[3], Francesco Pierfederici[3], Andrew Pontzen[24], Andy Ptak[25], Brian Refsdal, Mathieu Servillat[26], and Ole Streicher[27]

[1] Max-Planck-Institut für Astronomie, Königstuhl 17, Heidelberg 69117, Germany
[2] Astronomy Department, Yale University, P.O. Box 208101, New Haven, CT 06510, USA
[3] Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218
[4] Harvard-Smithsonian Center for Astrophysics, 60 Garden Street, Cambridge, MA, 02138, USA
[5] Center for Astrophysics and Space Astronomy, University of Colorado, Boulder, CO 80309, USA
[6] Department of Astronomy, Columbia University, Pupin Hall, 550W 120th St., New York, NY 10027, USA
[7] Department of Astronomy and Astrophysics, University of Toronto, 50 Saint George Street, Toronto, ON M5S3H4, Canada
[8] Argonne National Laboratory, High Energy Physics Division, 9700 South Cass Avenue, Argonne, IL 60439, USA
[9] Department of Astronomy, Ohio State University, Columbus, OH 43210, USA
[10] S.V.National Institute of Technology, Surat., India
[11] Independent developer
[12] Max-Planck-Institut für Kernphysik, P.O. Box 103980, 69029 Heidelberg, Germany
[13] European Southern Observatory, Karl-Schwarzschild-Str. 2, 85748, Garching bei München, Germany
[14] Laboratoire d'Astrophysique de Marseille, OAMP, Université Aix-Marseille et CNRS, Marseille, France
[15] ETH Zürich, Institute for Astronomy, Wolfgang-Pauli-Strasse 27, Building HIT, Floor J, CH-8093 Zurich, Switzerland
[16] Gemini Observatory, Casilla 603, La Serena, Chile
[17] LIGO Laboratory, California Institute of Technology, 1200 E. California Blvd., Pasadena, CA, 91125, USA
[18] Center for Cosmology and Particle Physics, New York, University, New York, NY 10003, USA
[19] Department of Physics and Astronomy, Louisiana State University, Nicholson Hall, Baton Rouge, LA 70803, USA
[20] Research School of Astronomy and Astrophysics, Australian National University, Mount Stromlo Observatory
[21] SAAO, P.O. Box 9, Observatory 7935, Cape Town, South Africa
[22] Joint Astronomy Centre, 660 North A'ohoku Place, Hilo, HI 96720, USA
[23] Gemini Observatory, 670 N. A'ohoku Place, Hilo, Hawaii 96720, USA
[24] Oxford Astrophysics, Denys Wilkinson Building, Keble Road, Oxford OX1 3RH, UK
[25] NASA Goddard Space Flight Center, X-ray Astrophysics Lab Code 662, Greenbelt, MD 20771, USA
[26] Laboratoire AIM, CEA Saclay, Bat. 709, 91191 Gif-sur-Yvette, France
[27] Leibniz Institute for Astrophysics Potsdam (AIP)

Preprint online version: June 3, 2013

ABSTRACT

We present the first public version of the open-source and community-developed Python package, Astropy. This package provides core astronomy-related functionality to the community, including support for domain-specific file formats such as Flexible Image Transport System (FITS) files, Virtual Observatory (VO) tables, and common ASCII table formats, unit and physical quantity conversions, physical constants specific to astronomy, celestial coordinate and time transformations, world coordinate system (WCS) support, generalized containers for representing gridded as well as tabular data, and a framework for cosmological transformations and conversions. Significant functionality is under active development, such as a model fitting framework, VO client and server tools, and aperture and point spread function (PSF) photometry tools. The core development team is actively making additions and enhancements to the current code base, and we encourage anyone interested to participate in the development of future Astropy versions.

Key words. Keywords should be given

1. Introduction

The Python programming language[1] has been one of the fastest-growing programming language in the astron-

[1] `http://www.python.org`

omy community in the last decade. While there have been a number of efforts to develop Python packages for astronomy-specific functionality, these efforts have been fragmented, and several dozens of packages have been developed across the community with little or no coordination. This has led to duplication and a lack of homogeneity across packages, making it difficult for users to install all the required packages needed in an astronomer's toolkit. Because a number of these packages depend on individual or small groups of developers, packages are sometimes no longer maintained, or simply become unavailable, which is detrimental to long-term research and reproducibility.

Motivated by these issues, the Astropy project was started in 2011[2] out of a desire to bring together developers across the field of astronomy in order to coordinate the development of a common set of Python tools for astronomers in order to simplify the landscape of available packages. The project has grown rapidly, and to date, over 180 people are signed up to the development mailing list for the Astropy project[3].

One of the primary aims of the Astropy *project* is to develop a core `astropy` *package* that covers much of the astronomy-specific functionality needed by researchers, complementing more general scientific packages such as NumPy (Oliphant 2006; Van Der Walt et al. 2011) and SciPy (Jones et al. 2001), which are invaluable for numerical array-based calculations and more general scientific algorithms (e.g. interpolation, integration, clustering). In addition, the Astropy project includes work on more specialized Python packages (which we call *affiliated packages*) that are not included in the core package for various reasons: for some the functionality is in early stages of development and is not robust; the license is not compatible with Astropy; the package includes large files; or the functionality is mature, but too domain-specific to be included in a core package.

The driving interface design philosophy behind the core package is that code using `astropy` should result in concise and easily readable code, even by those new to Python. Typical operations should appear in code similar to how they would appear if expressed in spoken or written language. Such an interface results in code that is less likely to contain errors and is easily readable, enabling astronomers to focus more of their effort on their science objectives rather than interpreting obscure function or variable names or otherwise spending time trying to understand the interface.

In this paper, we present the first public release (v0.2) of the `astropy` package. We provide an overview of the current capabilities (§2), our development workflow (§3), and planned functionality (§4). This paper is not intended to provide a detailed documentation for the package (which is available online[4]), but is rather intended to give an overview of the functionality and design.

[2] Following discussions on the `astropy` mailing list at `http://mail.scipy.org/mailman/listinfo/astropy`

[3] `https://groups.google.com/forum/?fromgroups#!forum/astropy-dev`

[4] `http://docs.astropy.org`

Fig. 1. Quantity conversion using the `astropy.units` sub-package.

```
Define a quantity from scalars and units:
>>> from astropy import units as u
>>> 15.1 * u.m / u.s
<Quantity 15.1 m / s>

Convert a distance:
>>> (1.15e13 * u.km).to(u.pc)
<Quantity 0.372689618289 pc>

Make use of the unit equivalencies:
>>> e = 130. * u.eV
>>> e.to(u.Angstrom, equivalencies=u.spectral())
<Quantity 95.37245609234003 Angstrom>

Combine quantities:
>>> x = 1.4e11*u.km / (0.7*u.Myr) / (4.1e11*u.s)
<Quantity 0.487804878049 km / (Myr s)>

Convert to SI units:
>>> x.si
<Quantity 1.54576038117e-11 m / s2>

Convert to CGS units:
>>> x.cgs
<Quantity 1.54576038117e-09 Gal>
```

2. Capabilities

This section provides a broad overview of the capabilities of the different `astropy` sub-packages, which covers units and unit conversions (§2.1), absolute dates and times (§2.2), celestial coordinates (§2.3), tabular and gridded data (§2.4), common astronomical file formats (§2.5), world coordinate system transformations (§2.6), and cosmological utilities (§2.7). We have illustrated each section with simple and concise code examples, but for more details and examples, we refer the reader to the online documentation[4].

2.1. Units, Quantities, and Physical Constants

The `astropy.units` sub-package provides support for physical units. It originates from code in the `pynbody` package[5], but has been significantly enhanced in behavior and implementation. This sub-package can be used to attach units to scalars and arrays, convert from one set of units to another, define custom units, define equivalencies for units that are not strictly the same (such as wavelength and frequency), and decompose units into base units. Unit definitions are included in both the International System of Units (SI) and the Centimeter-gram-second (CGS) systems, as well as a number of astronomy- and astrophysics-specific units.

2.1.1. Units

The `astropy.units` sub-package defines a `Unit` class to represent base units, which can be manipulated without attaching them to values, for example to determine the conversion factor from one set of units to another. Users can also define their own units, either as standalone base units

[5] `https://code.google.com/p/pynbody/`

or by composing other units together. It is also possible to decompose units into their base units, or alternatively search for higher-level units that are identical.

This sub-package includes the concept of "equivalencies" in units, which is intended to be used where there exists an equation that provides a relationship between two different physical quantities such that providing either one uniquely defines the other. A standard astronomical example is the relationships between the frequency, wavelength and energy of a photon - it is common practice to treat such units as equivalent even though they are not strictly comparable. This sub-package supports defining such relationships, and it is one of the areas in which it distinguishes itself from most other unit handling software. For example, wavelength and frequency are not normally convertible, but by passing an equivalency list, the conversion is supported (see Figure 1). Equivalencies are also included for flux densities, and users can easily implement their own equivalencies.

There are multiple string representations for units used in the astronomy community. The FITS Standard Group (2008) defines a unit standard, as well as both the Centre de Données astronomiques de Strasbourg (CDS) (Ochsenbein 2000) and NASA/Goddard's Office of Guest Investigator Programs (OGIP) (George & Angelini 1995). In addition, the International Virtual Observatory Alliance (IVOA) has a forthcoming VOUnit standard (Derriere et al. 2012) in an attempt to resolve some of these differences. Rather than choose one of these, `astropy.units` supports all of these standards[6], and allows the user to select the appropriate one when reading and writing unit string definitions to and from external file formats.

2.1.2. Quantities and Physical Constants

While the previous section described the use of the units package to manipulate the units themselves, a more common use-case is to attach the units to quantities, and use them together in expressions. The `astropy.units` package allows units to be attached to Python scalars, or NumPy arrays, producing `Quantity` objects. These objects support arithmetic with other numbers and `Quantity` objects and preserve their units. For multiplication and division, the resulting object will retain all units used in the expression. The final object can then be converted to a specified set of units or decomposed, effectively canceling and combining any equivalent units and returning a `Quantity` object in some set of base units. This is demonstrated in Figure 1.

Using the `.to()` method, `Quantity` objects can easily be converted to different units. The units must either be dimensionally equivalent, or users should pass equivalencies through the `equivalencies` argument (c.f. §2.1.1).

The `Quantity` objects are used to define a number of useful astronomical constants included with `astropy`, each with an associated unit (where applicable) and additional metadata describing their provenance and uncertainties. These can be used along with `Quantity` objects to provide a convenient framework for computing any quantity in astronomy. Figure 2 includes a simple example that shows how the gravitational force between two bodies can be calculated in Newtons using physical quantities and user-specified quantities.

[6] OGIP support is forthcoming at the time of this writing.

Fig. 2. Using the `astropy.constants` sub-package.

```
Access physical constants:
>>> from astropy import constants as c
>>> print c.G
  Name   = Gravitational constant
  Value  = 6.67384e-11
  Error  = 8e-15
  Units  = m3 / (kg s2)
  Reference = CODATA 2010

Combine quantities and constants:
>>> F = (c.G * (3 * c.M_sun) * (2 * u.kg) /
...      (1.5 * u.au) ** 2)
>>> F.to(u.N)
<Quantity 0.01581795428812989 N>
```

2.2. Time

The `astropy.time` package provides functionality for manipulating times and dates. Specific emphasis is placed on supporting time scales (e.g. UTC, TAI, UT1) and time formats/representations (e.g. JD, MJD, ISO 8601) that are used in astronomy. Examples of using this sub-package are provided in Figure 3.

The most common way to use `astropy.time` is to create a `Time` object by supplying one or more input time values as well as the time format or representation and time scale of those values. The input time(s) can either be a single scalar like `"2010-01-01 00:00:00"` or `2455348.5` or a sequence of such values; the format or representation specifies how to interpret the input values, such as ISO or JD or Unix time; and the scale specifies the time standard used for the values, such as Coordinated Universal Time (UTC), Terrestial Time (TT), or International Atomic Time (TAI). The full list of available time scales are listed in Table 1. Many of these formats and scales are used within astronomy, and it is especially important to treat the different time scales properly when converting between celestial coordinate systems. To facilitate this, the `Time` class makes the conversion to a different format such as Julian Date straightforward, as well as the conversion to a different time scale, for instance from UTC to TT.

This package is based on a derived version of the Standards of Fundamental Astronomy (SOFA) time and calendar library[7] (Wallace 1996). Leveraging the robust and well-tested SOFA routines ensures that the fundamental time scale conversions are being computed correctly. An important feature of the SOFA time library which is supported by `astropy.time` is that each time is represented as a pair of double-precision (64-bit) floating-point values, which enables extremely high precision time computations. Using two 64-bit floating-point values allows users to represent times with a dynamic range of 30 orders of magnitude, for instance times accurate to better than a nanosecond over timescales of tens of Gyr. All time scale conversions are done by vectorized versions of the SOFA routines using Cython[8] (Behnel et al. 2011), a Python package that makes it easy to use C code in Python.

[7] `http://www.iausofa.org/`
[8] `http://www.cython.org/`

Table 1. Supported time systems for `astropy.time`

Scale	Description
TAI	International Atomic Time
TCB	Barycentric Coordinate Time
TCG	Geocentric Coordinate Time
TDB	Barycentric Dynamical Time
TT	Terrestrial Time
UT1	Universal Time
UTC	Coordinated Universal Time

Fig. 3. Time representation and conversion using the `astropy.time` sub-package.

```
Parse a date/time in ISO format and on the UTC scale
>>> from astropy.time import Time
>>> t = Time('2010-06-01 00:00:00',
             format='iso', scale='utc')
>>> t
<Time object: scale='utc' format='iso'
              vals=2010-06-01 00:00:00.000>

Access the time in Julian Date format
>>> t.jd
2455348.5

Access the time in year:day:time format
>>> t.yday
'2010:152:00:00:00.000'

Convert time to the TT scale
>>> t.tt
<Time object: scale='tt' format='iso'
              vals=2010-06-01 00:01:06.184>

Find the julian date in the TT scale
>>> t.tt.jd
2455348.5007660184
```

2.3. Celestial Coordinates

An essential element of any astronomy workflow is the manipulation, parsing, and conversion of astronomical coordinates. This functionality is provided in Astropy by the `astropy.coordinates` sub-package. The aim of this package is to provide a common application programming interface (API) for Python astronomy packages that use coordinates, and to relieve users from having to (re)implement extremely common utilities. To achieve this, it leverages important lessons learned from existing Python coordinates packages such as `kapteyn`, `pyast`, `pyephem` (Rhodes 2011), and `astropysics` (Tollerud 2012) that this package aims to supplant.

The sub-package has been designed to present a natural Python interface for representing coordinates in computations, simplify input and output formatting, and allow straightforward transformation between coordinate systems. It also supports implementation of new or custom coordinate systems that work consistently with the built-in systems. A future design goal is to also seamlessly support arbitrarily large data sets.

To that end, Figure 4 shows some typical usage examples for `astropy.coordinates`. Coordinate objects are created using standard Python object instantiation via a Python class named after the coordinate system (e.g., `ICRSCoordinates`). Astronomical coordinates may be expressed in a myriad of ways: the classes support string, numeric, and tuple value specification through a sophisticated input parser. A design goal of the input parser is to be able to determine the angle value and unit from the input alone if a person can unambiguously determine them. For example, an astronomer seeing the input string "12h53m11.5123s" would understand the units to be in hours, minutes, and seconds, so this value is alone sufficient to pass to the angle initializer. This functionality is build around the `Angle` object, which can be instantiated and used on its own. It provides additional functionality like string formatting and mechanisms to specify the valid bounds of an angle.

Fig. 4. Celestial coordinate representation and conversion.

```
Parse coordinate string
>>> import astropy.coordinates as coords
>>> c = coords.ICRSCoordinates('00h42m44.3s +41d16m9s')

Access the RA/Dec values
>>> c.ra
<RA 10.68458 deg>
>>> c.dec
<Dec 41.26917 deg>
>>> c.ra.degrees
10.684583333333332
>>> c.ra.hms
(0.0, 42, 44.299999999999784)

Convert to Galactic coordinates
>>> c.galactic.l
<Angle 121.17431 deg>
>>> c.galactic.b
<Angle -21.57280 deg>

Create a separate object in Galactic coordinates
>>> g = c.transform_to(coords.GalacticCoordinates)
>>> g.l.format('degree', sep=':', precision=3)
'121:10:27.499'

Set the distance and view the cartesian coordinates
>>> from astropy import units as u
>>> c.distance = coords.Distance(770., u.kpc)
>>> c.x
568.7128882165681
>>> c.y
107.3009359688103
>>> c.z
507.8899092486349

Query SIMBAD to get coordinates from object names
>>> m = coords.ICRSCoordinates.from_name("M32")
>>> m
<ICRSCoordinates RA=10.67446 deg, Dec=40.86589 deg>

Two coordinates can be used to get distances
>>> m.distance = coords.Distance(765., u.kpc)
>>> m.separation_3d(c)
<Distance 7.36155 kpc>
```

The coordinate classes represent different coordinate systems, and provide most of the user-facing functionality for `astropy.coordinates`. The systems provide customized initializers and appropriate formatting and representation defaults. For some classes, they also contain added functionality specific to a subset of systems, such as code to precess a coordinate to a new equinox. The imple-

mented systems include a variety of equatorial coordinate systems (ICRS, FK4, and FK5), Galactic coordinates, and horizontal (Alt/Az) coordinates. Future versions of Astropy will include additional common systems, including ecliptic systems, supergalactic coordinates, and all necessary intermediate coordinate systems for the IAU 2000/2006 equatorial-to-horizontal mapping (e.g., Soffel et al. 2003; Kaplan 2005).

Another major feature of `astropy.coordinates` is the methodology for transforming between coordinate systems. Figure 4 illustrates the most basic use of this functionality to convert from ICRS to Galactic coordinates. Transformations are provided between all coordinate systems built into version 0.2 of Astropy, with the exception of conversions from celestial to horizontal coordinates. This is planned for the next major release, using the transformation architecture endorsed by the IAU 2000/2006 resolutions (see e.g., Soffel et al. 2003; Kaplan 2005).

A final significant feature of `astropy.coordinates` is support for line-of-sight distances. While the term "celestial coordinates" can be taken to refer to only on-sky angles, in `astropy.coordinates` a coordinate object is conceptually treated as a point in three dimensional space. Users have the option of specifying a line of sight distance to the object from the origin of the coordinate system (typically the origin is the Earth or solar system barycenter). These distances can be given in physical units or as redshifts. The `astropy.coordinates` package will in the latter case transparently make use of the cosmological calculations in `astropy.cosmology` (c.f. §2.7) for conversion to physical distances. Figure 4 illustrates an application of this information in the form of computing three-dimensional distances between two objects.

These features are made more useful by the design philosophy of `astropy.coordinates` that it should be easy for a user to add new systems with minimal bookkeeping. In fact, it is possible for a new contributor or user to add a new coordinate system with only a class definition and two transformation functions. Coordinate systems added in this manner are "first-class citizens" of the package, meaning that they have all the capabilities of the systems that are included in the package. An example of such a user-defined system is provided in the documentation[9], illustrating the definition of a coordinate system useful for a specific scientific task (Johnston & Price-Whelan 2013, in prep).

This flexibility is achieved in `astropy.coordinates` through the use of a transformation graph. All coordinate systems are represented as classes; internally, the transformation infrastructure keeps track of a network of coordinate systems and the transformations between them. When a coordinate object is to be transformed from one system into another, the package determines the shortest path on the transformation graph to the new system and applies the necessary sequence of transformations. Thus, implementing a new coordinate system only requires implementing one pair of transformations to and from a system that is already connected to the transformation graph. Once this pair is specified, `astropy.coordinates` can transform from that coordinate system to any other in the graph. This greatly simplifies implementation and mirrors the way most coordinate systems are defined. For example, Galactic coordinates are defined relative to FK4 coordinates rather than with respect to a set of other common coordinate systems (Blaauw et al. 1960; Reid & Brunthaler 2004).

[9] `http://docs.astropy.org/en/v0.2/coordinates/sgr-example.html#complete-code-for-example`

Fig. 5. Table input/output and manipulation using the `astropy.table` sub-package.

```
Create an empty table and add columns
>>> from astropy.table import Table, Column
>>> t = Table()
>>> t.add_column(Column(data=['a', 'b', 'c'],
...                     name='source'))
>>> t.add_column(Column(data=[1.2, 3.3, 5.3],
...                     name='flux'))
>>> print t
source flux
------ ----
     a  1.2
     b  3.3
     c  5.3

Read a table from a file
>>> t1 = Table.read('catalog.vot')
>>> t1 = Table.read('catalog.tbl', format='ipac')
>>> t1 = Table.read('catalog.cds', format='cds')

Select all rows from t1 where the flux column
is greater than 5
>>> t2 = t1[t1['flux'] > 5.0]

Manipulate columns
>>> t2.remove_column('J_mag')
>>> t2.rename_column('Source', 'sources')

Write a table to a file
>>> t2.write('new_catalog.hdf5')
>>> t2.write('new_catalog.rdb')
>>> t2.write('new_catalog.tex')
```

2.4. Tables and Gridded data

Tables and n-dimensional data arrays are the most common forms of data encountered in astronomy. The Python-community has various solution for tables, such as NumPy structured arrays or `DataFrame` objects in Pandas[10] to name only a couple. For n-dimensional data the NumPy `ndarray` is the most popular.

However, for use in astronomy all of these implementations lack some key features. The data that is stored in arrays and tables often contains vital metadata: the data is associated with units, and might also contain additional arrays that either mask or provide additional attributes to each cell. Furthermore, the data often has a parameter/value dataset attached with it. Finally, the data comes in a plethora of astronomy specific formats (FITS, specially formatted ASCII tables, etc.) which are not recognized by the pre-existing packages.

The `astropy.table` and `astropy.nddata` sub-packages contain classes (`Table` and `NDData`) that try to alleviate these problems. They allow users to represent astronomical data in the form of tables or n-dimensional gridded datasets, including all meta-data. Examples of usage of `astropy.table` are shown in Figure 5.

[10] `http://pandas.pydata.org`

The `Table` class provides a high-level wrapper to NumPy structured arrays, which are essentially arrays that have fields (or columns) with heterogeneous data types, and any number of rows. NumPy structured arrays are however difficult to manipulate or modify, so the `Table` class makes it easy for users to create a table from columns, add and remove columns or rows, and mask values from the table. Furthermore, tables can be easily read from and written to common file formats using the `Table.read` and `Table.write` methods. These methods are connected to sub-packages in `astropy.io` such as `astropy.io.ascii` (§2.5.2) and `astropy.io.votable` (§2.5.3), which allow ASCII and VO tables to be seamlessly read or written respectively.

In addition to providing easy manipulation and input or output of table objects, the `Table` class allows units to be specified for each column using the `astropy.units` framework, and also allows the `Table` object to contain arbitrary meta-data (stored in `Table.meta`).

Similarly, the `NDData` class provides a way to easily store n-dimensional array data and builds upon the NumPy `ndarray` class. The actual data is stored in an `ndarray`, which allows for easy compatibility with other scientific packages. In addition to keyword-value meta-data, the `NDData` class can store a boolean mask with the same dimensions as the data, several sets of flags (n-dimensional arrays that store attributes for each cell of the data array), uncertainties, units, and a transformation between array-index coordinate system and other coordinate systems (c.f. §2.6). In addition, the `NDData` class intends to provide methods to arithmetically combine the data in a meaningful way. `NDData` is not meant for direct user interaction but more for providing a framework for higher-level subclasses like a spectrum class or an astronomical image class. The `NDData` class currently does not have any built-in readers and writers.

2.5. File Formats

2.5.1. FITS

Support for reading and writing FITS files is provided by the `astropy.io.fits` sub-package, which at the time of writing is a direct port of the PyFITS[11] project (Barrett & Bridgman 1999). Users already familiar with PyFITS will therefore feel at home with this package. Although PyFITS will continue to be released as a separate product in the near term, the long term plan is to discontinue PyFITS releases in favor of Astropy. It is expected that direct support of PyFITS will end mid-2014, so users of PyFITS should plan to make suitable changes to support the eventual transition to Astropy.

Figure 6 shows an example of how to open an existing FITS file, access and modify the header and data, and write a new file back to disk. Because the interface is exactly the same as that of PyFITS, users may directly replace PyFITS with Astropy in existing code by changing import statements like `import pyfits` to `from astropy.io import fits as pyfits` without any additional code changes.

Becoming integrated with Astropy as the `astropy.io.fits` sub-package will greatly enhance future development on the existing PyFITS code base in several areas. First and perhaps foremost is integration with Astropy's `Table` interface which is much more flexible and powerful than PyFITS' current table interface. We will also be able to integrate Astropy's unit support in order to attach units to FITS table columns as well as header values that specify units in their comments in accordance with the FITS standard. Finally, as the PyWCS package has also been integrated into Astropy as `astropy.wcs` (c.f. §2.6) tighter association between data from FITS files and their world coordinate system (WCS) will be possible.

Fig. 6. Accessing data in FITS format.

```
Read in a FITS file from disk
>>> from astropy.io import fits
>>> hdus = fits.open('sample.fits')

Access the header of the first HDU:
>>> hdus[0].header
SIMPLE  =                    T
BITPIX  =                  -32
NAXIS   =                    3
NAXIS1  =                  200
NAXIS2  =                  200
NAXIS2  =                   10
EXTEND  =                    T
...

Access the shape of the data in the first HDU:
>>> hdus[0].data.shape
(10, 200, 200)

Update/add header keywords
>>> hdus[0].header['TELESCOP'] = 'Python'
>>> hdus[0].header['INSTRUME'] = 'Computer'

Multiply data by 1.2
>>> hdus[0].data *= 1.2

Write out to disk
>>> hdus.writeto('new_file.fits')
```

2.5.2. ASCII table formats

The `astropy.io.ascii` sub-package (formerly the standalone project `asciitable`[12]) provides the ability to read and write tabular data for a wide variety of ASCII-based formats. In addition to generic formats such as space-delimited, tab-delimited or comma-separated values, `astropy.io.ascii` provides classes for specialized table formats like CDS[13], IPAC[14], IRAF DAOphot, and LaTeX. There is also a flexible class for handling a wide variety of fixed-width table formats. Finally, this package is designed to be extensible, making it easy for users to define their own readers and writers for any other ASCII formats.

[11] `http://www.stsci.edu/institute/software_hardware/pyfits`

[12] `https://asciitable.readthedocs.org`

[13] `http://vizier.u-strasbg.fr/doc/catstd.htx`

[14] `http://irsa.ipac.caltech.edu/applications/DDGEN/Doc/ipac_tbl.html`

2.5.3. Virtual Observatory tables

The `astropy.io.votable` package (formerly the standalone project `vo.table`) provides full support for reading and writing VOTable format files versions 1.1, 1.2, and the proposed 1.3 (Oschenbein et al. 2004, 2009). It efficently stores the tables in memory as NumPy structured arrays. The file is read using streaming to avoid reading in the entire file at once and greatly reducing the memory footprint.

It is possible to convert any one of the tables in a VOTable file to a `Table` object (§2.4), where it can be edited and then written back to a VOTable file without any loss of data.

The VOTable standard is not strictly adhered to by all VOTable file writers in the wild. Therefore, `astropy.io.votable` provides a number of tricks and workarounds to support as many VOTable sources as possible, whenever the result would not be ambiguous. A validation tool (`volint`) is also provided that outputs recommendations to improve the standard compliance of a given file, as well as validate it against the official VOTable schema.

2.6. World Coordinate Systems

The `astropy.wcs` package contains utilities for managing World Coordinate System (WCS) transformations in FITS files. These transformations map the pixel locations in an image to their real-world units, such as their position on the celestial sphere. This library is specific to WCS as it relates to FITS as described in the FITS WCS papers (Greisen & Calabretta 2002; Calabretta & Greisen 2002; Greisen et al. 2006) and is distinct from a planned Astropy package that will handle WCS transformations in general, regardless of their representation.

This package is a wrapper around Mark Calabretta's `wcslib` (Calabretta 2013). Since all of the FITS header parsing is done using `wcslib`, it is assured the same behavior as the many other tools that use `wcslib`. On top of the basic FITS WCS support, it adds support for the Simple Imaging Polynomial (SIP) convention and table lookup distortions as defined in the draft WCS "Paper IV" (Calabretta et al. 2004). Each of these transformations can be used independently or together in a fixed pipeline.

The `astropy.wcs` package also serves as a useful FITS WCS validation tool, as it is able to report on many common mistakes or deviations from the standard in a given FITS file.

2.7. Cosmology

The `astropy.cosmology` sub-package contains classes for representing widely used cosmologies, and functions for calculating quantities that depend on a cosmological model. It also contains a framework for working with less frequently employed cosmologies that may be not be flat, or have a time-varying pressure to density ratio, w, for dark energy. The quantities that can be calculated are generally taken from those described by Hogg (1999). Some examples are the angular diameter distance, comoving distance, critical density, distance modulus, lookback time, luminosity distance, and Hubble parameter as a function of redshift.

The fundamental model for this sub-package is that any given cosmology is represented by a class. An instance of this class has attributes giving all the parameters required to uniquely specify the cosmology, such as the Hubble parameter, CMB temperature and the baryonic, cold dark matter, and dark energy densities at $z = 0$. One can then use methods of this class to perform calculations using these parameters.

Fig. 7. Cosmology utilities.

```
Create a custom cosmology object
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3)
>>> cosmo
FlatLambdaCDM(H0=70, Om0=0.3, Ode0=0.7)

Compute the comoving volume to z=6.5 in cubic Mpc using
this cosmology
>>> cosmo.comoving_volume(6.5)
1074707289417.6837

Compute the age of the universe in Gyr using the
pre-defined WMAP 5-year and WMAP 9-year cosmologies
>>> from astropy.cosmology import WMAP5, WMAP9
>>> WMAP5.age(0)
13.723782349795023
>>> WMAP9.age(0)
13.768899510689097

Create a cosmology with a varying 'w'
>>> from astropy.cosmology import Flatw0waCDM
>>> cosmo = Flatw0waCDM(H0=70, Om0=0.3, w0=-1, wa=0.2)

Find the separation in proper kpc at z=4 corresponding to
10 arcsec in this cosmology compared to a WMAP9 cosmology
>>> cosmo.kpc_proper_per_arcmin(4) * 10 / 60.
68.87214405278925
>>> WMAP9.kpc_proper_per_arcmin(4) * 10 / 60.
71.21374615575363
```

Figure 7 shows how `FlatLambdaCDM` can be used to create an object representing a flat ΛCDM cosmology, and how the methods of this object can be called to calculate the comoving volume, age and transverse at a given redshift. Further calculations can be performed using the many methods of the cosmology object as described in the Astropy documentation. For users who are more comfortable using a procedural coding style, these methods are also available as functions that take a cosmology class instance as a keyword argument.

The sub-package provides several pre-defined cosmology instances corresponding to commonly used cosmological parameter sets. Currently parameters from the WMAP 5-year (Komatsu et al. 2009), 7-year (Komatsu et al. 2011) and 9-year results (Hinshaw et al. 2012) are included (`WMAP5`, `WMAP7` and `WMAP9`). There are several classes corresponding to non-flat cosmologies, and the most common dark energy models are supported: a cosmological constant, constant w, and $w(a) = w_0 + w_a(1 - a)$ (e.g. Linder 2003, here a is the scale factor). Figure 7 gives examples showing how to use the pre-defined cosmologies, and how to define a new cosmology with a time-varying dark energy $w(a)$. Any other arbitrary cosmology can be represented by sub-classing one of the basic cosmology classes.

All of the code in the sub-package is tested against the web-based cosmology calculator by Wright (2006) and two

other widely-used calculators[15,16]. In cases when these calculators are not precise enough to enable a meaningful comparison, the code is tested against calculations performed with MATHEMATICA.

3. Development Approach

A primary guiding philosophy of Astropy is that it is developed for and (at least in part) *by* the astronomy user community. This ensures the interface is designed with the workflow of working astronomers in mind. At the same time, it aims to make use of the expertise of software developers to design code that encourages good software practices such as a consistent and clean API, thorough documentation, and integrated testing. It is also dedicated to remaining open source, allowing input from all users and assisting wider adoption. Achieving these aims requires code collaboration between over 30 geographically-distributed developers, and here we describe our development workflow with the hope that it may be replicated by other astronomy software projects that are likely to have similar needs.

To enable this collaboration, we have made use of the GitHub[17] open source code hosting and development platform. The main repository for `astropy` is stored in a git[18] repository on GitHub, and any non-trivial changes are made via *pull requests*, which are a mechanism for submitting code for review by other developers prior to merging into the main code base. This workflow aids in increasing the quality, documentation and testing of the code to be included in `astropy`. Not all contributions are necessarily accepted - community consensus is needed for incorporating major new functionality in `astropy`, and any new feature has to be justified to avoid implementing features that are only useful to a minority of users, but may cause issues in the future.

At the time of writing, `astropy` includes several thousand tests, which are small units of code that check that functions, methods, and classes in `astropy` are behaving as expected, both in terms of scientific correctness and from a programming interface perspective. We make use of *continuous integration*, which is the process of running all the tests under various configurations (such as different versions of Python or NumPy, and on different platforms) in order to ensure that the package is held to the highest standard of stability. In particular, any change made via a pull request is subject to extensive testing before being merged into the core repository. For the latter, we make use of Travis[19], while for running more extensive tests across Linux, MacOS X, and Windows, we make use of Jenkins[20] (both are examples of continuous integration systems).

This development workflow has worked very well so far, allowing contributions by many developers, and blurring the line between developers and users. Indeed, users who encounter bugs and who know how to fix them can submit suggested changes. We have also implemented a feature that means that anyone reading the documentation at `http://docs.astropy.org` can suggest improvements to the documentation with just a few clicks in a web browser without any prior knowledge of the git version control system.

[15] `http://www.kempner.net/cosmic.php`
[16] `http://www.icosmos.co.uk/index.html`
[17] `http://www.github.com`
[18] `http://git-scm.com/`
[19] `https://travis-ci.org/`
[20] `http://jenkins-ci.org/`

4. Planned functionality

Development on the Astropy package is very active, and we are focusing on several areas for the next (v0.3) release (some of which have already been implemented in the publicly-available developer version):

- Seamlessly integrating the `Quantity` framework across all sub-packages
- Supporting more file formats for reading and writing `Table` and `NDData` objects
- Implementing a Virtual Observatory cone search tool
- Implementing a generalized model-fitting framework

In the longer term, we are already planning the following functionality:

- More Virtual Observatory tools
- A SAMP server/client (based on the SAMPy[21] package)
- Aperture and PSF photometry tools
- Spectroscopic analysis tools
- Generalized WCS transformations beyond the FITS WCS standard

and undoubtedly the core functionality will grow beyond this. In fact, the `astropy` package will likely remain a continuously-evolving package, and will never be considered 'complete' in the traditional sense.

5. Summary

We have presented the first public release of the Astropy package (v0.2), a core Python package for astronomers. In this paper we have described the main functionality, which includes:

- Units and unit conversions (§2.1)
- Absolute dates and times (§2.2)
- Celestial coordinate systems (§2.3)
- Tabular and gridded data (§2.4)
- Support for common astronomical file formats (§2.5)
- World Coordinate System transformations (§2.6)
- Cosmological calculations (§2.7).

We also briefly described our development approach (§3), which has enabled an international collaboration of scientists and software developers to create and contribute to the package. We outlined our plans for the future (§4) which includes more interoperability of sub-packages, as well as new functionality.

We invite members of the community to join the effort by adopting the Astropy package for their own projects, reporting any issues, and whenever possible, developing new functionality.

[21] `http://pythonhosted.org/sampy/`

6. Acknowledgements

We would like to thank the NumPy, SciPy, IPython and Matplotlib communities for providing their packages which are invaluable to the development of Astropy. We thank the GitHub team for providing us with an excellent free development platform. We also are grateful to Read The Docs (`https://readthedocs.org/`), Shining Panda (`https://www.shiningpanda-ci.com/`), and Travis (`https://www.shiningpanda-ci.com/`) for providing free documentation hosting an testing respectively. Finally, we would like to thank all the `astropy` users that have provided feedback and submitted bug reports.

References

Barrett, P. E., & Bridgman, W. T. 1999, in Astronomical Society of the Pacific Conference Series, Vol. 172, Astronomical Data Analysis Software and Systems VIII, 483

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., & Smith, K. 2011, Computing in Science Engineering, 13, 31

Blaauw, A., Gum, C. S., Pawsey, J. L., & Westerhout, G. 1960, MNRAS, 121, 123

Calabretta, M. R. 2013, WCSLIB 4.17 and PGSBOX 4.17 Documentation

Calabretta, M. R., & Greisen, E. W. 2002, Astronomy & Astrophysics, 1077

Calabretta, M. R., Valdes, F. G., Greisen, E. W., & L., A. S. 2004, Unpublished draft

Derriere, S., Gray, N., Louys, M., McDowell, J., Ochsenbein, F., Osuna, P., Rino, B., & Salgado, J. 2012, Units in the VO, Version 1.0, ivoa proposed recommendation 20 august 2012 edn.

George, I., & Angelini, L. 1995, Specification of Physical Units within OGIP (Office of Guest Investigator Programs) FITS files

Greisen, E. W., & Calabretta, M. R. 2002, Astronomy & Astrophysics, 1061

Greisen, E. W., Calabretta, M. R., Valdes, F. G., & Allen, S. L. 2006, Astronomy & Astrophysics, 747

Group, F. W. 2008, Definition of the Flexible Image Transport System (FITS)

Hinshaw, G. et al. 2012, ArXiv e-prints, 1212.5226

Hogg, D. W. 1999, ArXiv Astrophysics e-prints, arXiv:astro-ph/9905116

Jones, E., Oliphant, T., & Peterson, P. 2001, http://www. scipy. org/

Kaplan, G. H. 2005, U.S. Naval Observatory Circulars, 179, arXiv:astro-ph/0602086

Komatsu, E. et al. 2009, ApJS, 180, 330, 0803.0547

——. 2011, ApJS, 192, 18, 1001.4538

Linder, E. V. 2003, Physical Review Letters, 90, 091301, arXiv:astro-ph/0208512

Ochsenbein, F. 2000, Astronomical Catalogues and Tables Adopted Standards, Version 2.0

Oliphant, T. 2006, A Guide to NumPy, Vol. 1 (Trelgol Publishing USA)

Oschenbein, F. et al. 2004, VOTable Format Definition, Version 1.1, International Virtual Observatory Alliance (IVOA)

——. 2009, VOTable Format Definition, Version 1.2, International Virtual Observatory Alliance (IVOA)

Reid, M. J., & Brunthaler, A. 2004, ApJ, 616, 872, arXiv:astro-ph/0408107

Rhodes, B. C. 2011, PyEphem: Astronomical Ephemeris for Python, 1112.014

Soffel, M. et al. 2003, AJ, 126, 2687, arXiv:astro-ph/0303376

Tollerud, E. 2012, Astropysics: Astrophysics utilities for python, 1207.007

Van Der Walt, S., Colbert, S., & Varoquaux, G. 2011, Computing in Science & Engineering, 13, 22

Wallace, P. T. 1996, in Astronomical Society of the Pacific Conference Series, Vol. 101, Astronomical Data Analysis Software and Systems V, ed. G. H. Jacoby & J. Barnes, 207

Wright, E. L. 2006, PASP, 118, 1711, arXiv:astro-ph/0609593