Draft version February 6, 2018
Typeset using LaTeX **twocolumn** style in AASTeX61

ASTROQUERY: AN ASTRONOMICAL WEB-QUERYING PACKAGE IN PYTHON

Adam Ginsburg,[1] Brigitta Sipocz,[2] Brett M. Morris,[2] Thomas P. Robitaille,[3] Leo P. Singer,[4,5]
Christoph Deil,[6] Madhura Parikh,[7] Henrik Norman,[8,9] Magnus V. Persson,[10] Pey Lian Lim,[11] Julien Woillez,[12]
Ivan Valtchanov,[13] Erik J. Tollerud,[11] C. E. Brasseur,[11] Juan-Carlos Segovia,[9] and
the Astroquery collaboration, a subset of the astropy collaboration

[1] *Jansky fellow of the National Radio Astronomy Observatory, 1003 Lopezville Rd, Socorro, NM 87801 USA*
[2] *Astronomy Department, University of Washington, Seattle, WA 98195, USA*
[3] *Aperio Software Ltd., Headingley Enterprise and Arts Centre, Bennett Road, Leeds, LS6 3HN, United Kingdom*
[4] *Astroparticle Physics Laboratory, NASA Goddard Space Flight Center, Mail Code 661, Greenbelt, MD 20771, USA*
[5] *Joint Space-Science Institute, University of Maryland, College Park, MD 20742, USA*
[6] *Max-Planck-Institut für Kernphysik, Heidelberg, Germany*
[7] *Google Summer of Code*
[8] *Winter Way, Uppsala, Sweden*
[9] *ESAC Science Data Centre, European Space Agency, Madrid, Spain*
[10] *Department of Space, Earth and Environment, Chalmers University of Technology, Onsala Space Observatory, 439 92, Onsala, Sweden*
[11] *Space Telescope Science Institute, 3700 San Martin Dr, Baltimore, MD 21218, USA*
[12] *European Southern Observatory, Karl-Schwarzschild-Str. 2, 85748 Garching bei München, Germany*
[13] *European Space Astronomy Centre, European Space Agency, Madrid, Spain*

ABSTRACT

Astroquery is a collection of tools for requesting data from databases hosted on the internet, particularly those with web pages but without formal application program interfaces (APIs). These tools are based on the Python requests module, which is used to make HTTP requests, and astropy, which provides most of the data parsing functionality. Astroquery has received significant contributions from the broader astronomical community, including several significant contributions from telescope archives. [a)]

Corresponding author: Adam Ginsburg
aginsbur@nrao.edu; adam.g.ginsburg@gmail.com

[a)] The repository associated with this paper is: https://github.com/adamginsburg/astroquery-paper

1. INTRODUCTION

Sharing data is a critical component of astronomical research. Astronomy has historically been a leading field in data sharing, motivated at least in part by questions that cannot be answered with single instruments. In the past few decades, blind surveys have played a huge role in advancing our understanding of the universe, and these surveys have produced large reservoirs of data that astronomers regularly access.

Data sharing has taken on a variety of forms. The most prominent are the major observatory archives: MAST, NOAO, ESO, IPAC, CADC, CDS (hosting Vizier and SIMBAD), NRAO, CXC, HEASARC, and ESA are the main organizations hosting raw and processed data from ground and space based telescopes. These data archives also serve as the primary means for serving data to users when the data are taken in queue mode, i.e., when the data are taken while the observer is not on-site.

In addition to observatories and telescopes, individual surveys often share their full data sets. In some cases, these data sets are shared via the observatory that acquired them - for example, the all-sky data acquired with Planck, WMAP, and COBE delivered a variety of data products as part of the mission. Other surveys, particularly ground-based surveys, serve their own data. Examples include SDSS, 2MASS, UKIDSS, and likely many more in the near future.

Individual teams and small groups will often share their data. These services do not follow any particular standard and can be widely varied in the type and amount of data shared. Sometimes these data are shared via the archive systems (e.g., IRSA at IPAC hosts many individual survey data sets), while others use their own web hosting systems (e.g., MAGPIS).

Finally, there are other data types relevant to astronomy that are not served by the typical astronomical databases. Examples include molecular and atomic properties, such as those provided by Splatalogue and NIST.

Astroquery arose from a desire to access these databases from the command line in a scriptable fashion. Script-based data access provides astronomers with the ability to make reproducible analysis scripts in which the data are acquired and processed into scientifically relevant results with minimal overhead.

In this paper, we provide an overview of the `astroquery` package. Section 2 describes the basic layout of the software and the shared API concept underlying all modules. Section 3 describes the development model.

2. THE SOFTWARE

Astroquery consists of a collection of modules that mostly share a similar interface, but are meant to be used independently. They are primarily based on a common framework that uses the Python `requests` package to perform HTTP requests to communicate with web services.

For new development, there is a `template` that lays out the basic framework of any new module. All modules are based on having a single core `class` that will have some number of `query_*` methods. The most common query methods are `query_region`, which usually provide a “cone search” functionality, i.e., they search for data within a circularly symmetric region projected on the sky.

An example using the SIMBAD interface is shown below (see http://astroquery.readthedocs.io/en/latest/simbad/simbad.html):

Example 1. Query SIMBAD for a region around M81

```
from astroquery.simbad import Simbad
result_table = Simbad.query_region("m81")
```

In this example, `Simbad` is an instance of the `astroquery.simbad.Sim` class. The returned result, stored in the variable `result_table`, is an astropy table.

While there is a common suggested API described in the `template` module, individual packages are not *required* to support this API because, for some, it is not possible. For example, the atomic and molecular databases refer to physical data that is not related to positions on the sky and therefore their Astroquery modules cannot include `query_region` methods.

2.1. *HTTP User-Agent*

Astroquery identifies itself to host services using the `HTTP User-Agent` header data. The format of the User-Agent string is `astroquery/{version} {requests_version}`, where `{version}` is a version number of the form #.#.# and `{requests_version}` is the corresponding version of the python requests module. This information can be used by data hosting services to determine how many users are accessing their service via astroquery and to assist in debugging if improper queries are being posted.

2.2. *The API*

The common API has a few features defined in the `template` module. Each service is expected to provide the following interfaces, assuming they are applicable:

- `query_region` - A function that accepts an astropy `SkyCoord` object representing a point on the sky plus a specification of the radius around which to search. The returned object is an astropy `Table`.

- `query_object` - A function that accepts the name of an object. This method relies on the service to resolve the object name. The returned object is an astropy `Table`.
- `get_images` - For services that provide image data, this function accepts an astropy `SkyCoord` and a radius to search for data that cover the specified target. The returned object should be a list of `astropy.io.fits.HDUList` objects.

Beyond these basic functions, there is a series of functions with the same names, but with the additional suffix `_async`. The `query_*_async` functions return a `requests.Response` object from the accessed website, providing developers with the ability to access the data in a stream or access only the response metadata (i.e., the `async` methods do not download the corresponding data, so they may be useful for collecting metadata for very large files). The `get_images_async` method returns `FileContainer` objects that similarly provide 'lazy' access to the data, but specifically for FITS files. Developers need only implement these `_async` functions because a wrapper tool exists to convert `_async` functions into their corresponding non-asynchronous versions.

2.3. *Layout of the base Query module and the QueryWithLogin module*

TODO: Julien, could you add something here?

2.4. *Testing*

Astroquery testing is somewhat different from most other packages in the Python ecosystem. While the tests are based on the astropy package-template and use pytest to run and check the outputs, the `astroquery` tests are split into *remote* and *non-remote*. The remote tests exactly replicate what a user would enter at the command line, but they are dependent on the stability of the remote services. Historically, we have found that it is quite rare for all of the Astroquery-supported services to be accessible simultaneously.

We therefore require that each module provide some tests that do not rely on having an internet connection. These tests rely on *monkeypatching*[1] to replace the remote requests, instead using locally available files to test the query mechanisms and the data parsers. Monkeypatching in the context of `pytest` results in code that is generally more difficult to understand than typical Python code, but a set of tests independent of the remote services is necessary.

The non-remote tests are run as part of the continuous integration for the project with each commit. The remote tests are run as part of a regularly-scheduled `cron` job. Running the remote tests infrequently also helps reduce the burden on the remote services.

[1] Monkeypatching is the dynamic replacement of attributes at runtime, i.e., changing what functions do after they are imported.

2.5. *Other utilities*

There are several general-use utilities implemented as part of Astroquery, such as a bulk FITS file downloader and renamer and a download progressbar. There is also a schema system implemented to allow user-side parameter validation. So far, at the time of writing, this tool is only implemented in the ESO and Vizier modules, but it could be expanded to other modules to reduce the number of doomed-to-fail queries sent through Astroquery.

3. THE DEVELOPMENT MODEL

Astroquery is an `astropy` affiliated package and is a core part of the `astropy` ecosystem (Astropy Collaboration et al. 2013). It is a standalone project and will always remain independent of the `astropy` core package.

Astroquery has received contributions from 53 people as of June 2017. While the primary maintenance burden is shouldered by 2-3 people at any given time, most individual modules have been implemented independently by interested volunteers.

Some contributions have come as direct institutional support. The ESA GAIA and ESASky modules were provided by developers working for ESA. Meanwhile, the MAST and VO Cone Search query tools were added by developers at STSCI, with the latter moved over from `astropy.vo`.

Astroquery has received support from the Google Summer of Code program, with two students (coauthors Madhura Parikh and Simon Liedtke) from 2013-2017.

Anyone can contribute to Astroquery. The maintainers are committed to helping developers make new modules that meet the requirements of Astroquery.

3.1. *Versioning*

Unlike the core astropy module, astroquery exists in an 'always-unstable' state. Because astroquery relies on remote services' APIs to function, changes to those APIs may break astroquery without warning. For that reason, we sometimes have frequent or sudden version releases to accommodate remote changes.

Some services have been helpful about alerting the astroquery development team in advance when API changes are imminent. We are grateful for such advance warning and encourage other services to check in and file Issues on astroquery's issue tracker when significant API modifications are made.

4. DOCUMENTATION AND REFERENCES

4.1. *Online documentation*

The astroquery modules are documented online and can be accessed through readthedocs at https://astroquery.readthedocs.io/en/latest/. We include one detailed example of how to use astroquery in Appendix A, but interested users will find many more on the documentation page and in the example gallery (https://astroquery.readthedocs.io/en/latest/gallery.html).

4.2. *Other Documents*

Several authors have independently described how to use various `astroquery` modules, which is a helpful practice we encourage.

- https://www.cosmosim.org/cms/news/cosmosim-package-fo a worked example of downloading data from the cosmosim database, including logging in.
- https://arxiv.org/abs/1408.7026: a worked example of querying Vizier and SIMBAD to make a surface gravity - effective temperature plot for a star survey

5. SUMMARY

Astroquery is a toolkit for accessing remotely hosted data through python. It is part of the astropy affiliated package system. We have described its general layout and development model. We welcome any new contributions.

REFERENCES

Astropy Collaboration, Robitaille, T. P., Tollerud, E. J., et al. 2013, A&A, 558, A33

APPENDIX

A. EXAMPLE

In this appendix, we show an example of astroquery in action, highlighting the ability to use multiple modules and interact with astropy's table, coordinate, and unit tools. This example approximately reproduces Figure 1 of **?**, but with a different background. It can also be found on astropy's gallery page (http://astroquery.readthedocs.io/en/latest/gallery.html).

```
from astropy import coordinates, units as u, wcs
from astroquery.skyview import SkyView
from astroquery.vizier import Vizier
import pylab as pl

center = coordinates.SkyCoord.from_name('Orion KL')

# Grab an image from SkyView of the Orion KL nebula region
imglist = SkyView.get_images(position=center, survey='2MASS-J')

# the returned value is a list of images, but there is only one
img = imglist[0]

# 'img' is now a fits.HDUList object; the 0th entry is the image
mywcs = wcs.WCS(img[0].header)

fig = pl.figure(1)
fig.clf() # just in case one was open before
# use astropy's wcsaxes tool to create an RA/Dec image
ax = fig.add_axes([0.15, 0.1, 0.8, 0.8], projection=mywcs)
ax.set_xlabel("RA")
ax.set_ylabel("Dec")

ax.imshow(img[0].data, cmap='gray_r', interpolation='none', origin='lower',
         norm=pl.matplotlib.colors.LogNorm())


# retrieve a specific table from Vizier to overplot
tablelist = Vizier.query_region(center, radius=5*u.arcmin, catalog='J/ApJ/826/16/table1')
# again, the result is a list of tables, so we'll get the first one
result = tablelist[0]

# convert the ra/dec entries in the table to astropy coordinates
tbl_crds = coordinates.SkyCoord(result['RAJ2000'], result['DEJ2000'],
                             unit=(u.hour, u.deg), frame='fk5')

# we want this table too:
tablelist2 = Vizier(row_limit=10000).query_region(center, radius=5*u.arcmin, catalog='J/ApJ/540/236')
result2 = tablelist2[0]
tbl_crds2 = coordinates.SkyCoord(result2['RAJ2000'], result2['DEJ2000'],
                              unit=(u.hour, u.deg), frame='fk5')


# overplot the data in the image
ax.plot(tbl_crds.ra, tbl_crds.dec, 'o', transform=ax.get_transform('fk5'),
       mec='b', mfc='none')
ax.plot(tbl_crds2.ra, tbl_crds2.dec, 'o', transform=ax.get_transform('fk5'),
       mec='r', mfc='none')
# zoom in on the relevant region
```

```
ax.axis([100,200,100,200])
```

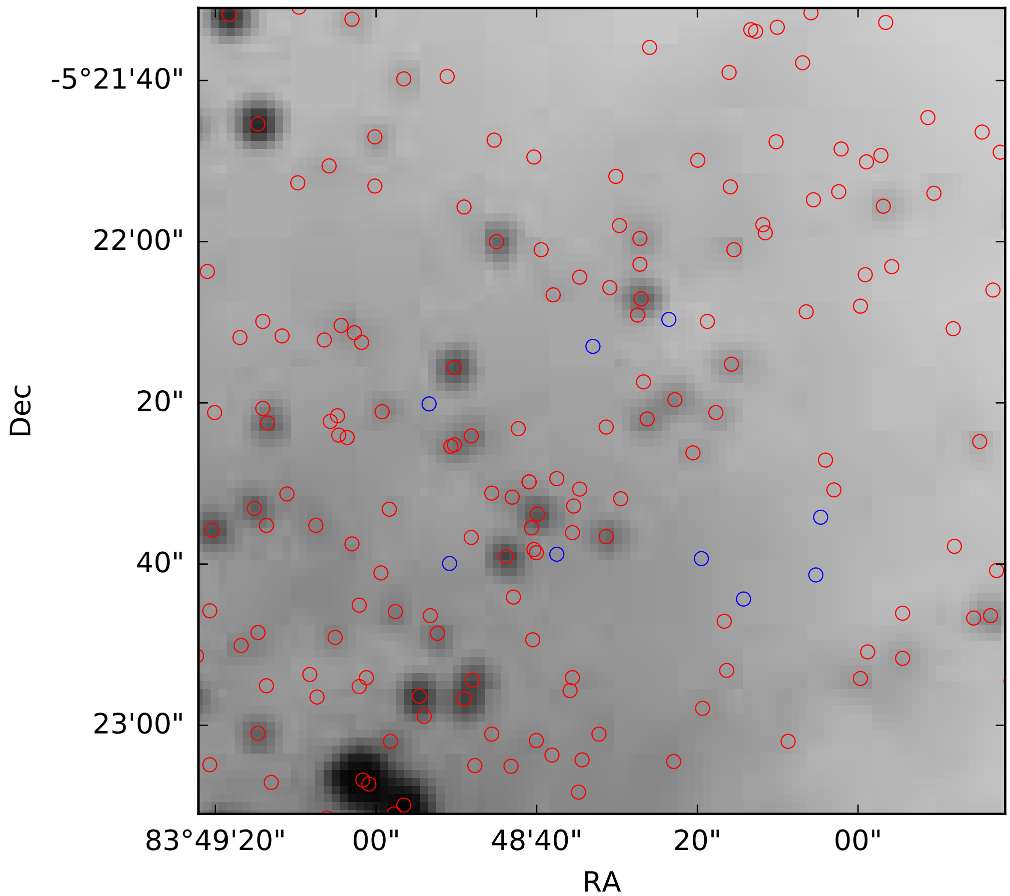

Figure 1. An example figure made using astroquery's `skyview` and `vizier` modules with astropy's `table`, `coordinates`, `units`, and `wcs` modules.