# astroquery Documentation

***Release 0.3.7***

**Adam Ginsburg**

**Mar 29, 2018**

# Contents

This is the documentation for the Astroquery affiliated package of astropy.

Code and issue tracker are on GitHub.

# Part I

# Introduction

Astroquery is a set of tools for querying astronomical web forms and databases.

There are two other packages with complimentary functionality as Astroquery: astropy.vo is in the Astropy core and pyvo is an Astropy affiliated package. They are more oriented to general virtual observatory discovery and queries, whereas Astroquery has web service specific interfaces.

Check out the *A Gallery of Queries* for some nice examples.

# Part II

# Installation

The latest version of astroquery can be conda installed while the latest and development versions can be pip installed or be downloaded directly from GitHub.

CHAPTER 1

# Using pip

```
$ pip install astroquery
```

and the 'bleeding edge' master version:

```
$ pip install https://github.com/astropy/astroquery/archive/master.zip
```

# CHAPTER 2

## Using conda

It is also possible to install the latest astroquery with anaconda from the astropy channel:

```
$ conda install -c astropy astroquery
```

# CHAPTER 3

## Building from source

The development version can be obtained and installed from github:

```
$ # If you have a github account:
$ git clone git@github.com:astropy/astroquery.git
$ # If you do not:
$ git clone https://github.com/astropy/astroquery.git
$ cd astroquery
$ python setup.py install
```

# Part III

# Requirements

Astroquery works with Python 2.7 and 3.4 or later.

The following packages are required for astroquery installation & use:

- numpy >= 1.9
- astropy (>=1.0)
- requests
- keyring
- Beautiful Soup
- html5lib

and for running the tests:

- curl

The following packages are optional dependencies and are required for the full functionality of the `alma` module:

- APLpy
- pyregion

# Part IV

# Using astroquery

All astroquery modules are supposed to follow the same API. In its simplest form, the API involves queries based on coordinates or object names. Some simple examples, using SIMBAD:

```
>>> from astroquery.simbad import Simbad
>>> result_table = Simbad.query_object("m1")
>>> result_table.pprint(show_unit=True)
MAIN_ID      RA          DEC      RA_PREC ... COO_QUAL COO_WAVELENGTH     COO_BIBCODE
          "h:m:s"     "d:m:s"             ...
------- ----------- ----------- ------- ... -------- -------------- -------------------
  M   1 05 34 31.94 +22 00 52.2       6 ...        C              R 2011A&A...533A..10L
```

All query tools allow coordinate-based queries:

```
>>> from astropy import coordinates
>>> import astropy.units as u
>>> # works only for ICRS coordinates:
>>> c = coordinates.SkyCoord("05h35m17.3s -05d23m28s", frame='icrs')
>>> r = 5 * u.arcminute
>>> result_table = Simbad.query_region(c, radius=r)
>>> result_table.pprint(show_unit=True, max_width=80, max_lines=5)
  MAIN_ID         RA           DEC      ... COO_WAVELENGTH     COO_BIBCODE
               "h:m:s"      "d:m:s"     ...
------------ ------------ ------------ ... -------------- -------------------
       M  42    05 35 17.3    -05 23 28 ...                1981MNRAS.194..693L
         ...          ...          ... ...            ...                 ...
V* V2114 Ori 05 35 01.671 -05 26 36.30 ...              I 2003yCat.2246....0C
```

For additional guidance and examples, read the documentation for the individual services below.

# Part V

# Available Services

If you're new to Astroquery, a good place to start is the *A Gallery of Queries*:

CHAPTER 4

# A Gallery of Queries

A series of queries folks have performed for research or for kicks.

## 4.1 Example 1

This illustrates querying Vizier with specific keyword, and the use of `astropy.coordinates` to describe a query. Vizier's keywords can indicate wavelength & object type, although only object type is shown here.

```
>>> from astroquery.vizier import Vizier
>>> v = Vizier(keywords=['stars:white_dwarf'])
>>> from astropy import coordinates
>>> from astropy import units as u
>>> c = coordinates.SkyCoord(0,0,unit=('deg','deg'),frame='icrs')
>>> result = v.query_region(c, radius=2*u.deg)
>>> print(len(result))
31
>>> result[0].pprint()
   LP     Rem Name  RA1950   DE1950  Rmag l_Pmag Pmag u_Pmag spClass   pm    pmPA _RA.icrs _DE.icrs
-------- --- ---- -------- -------- ---- ------ ---- ------ ------- ------ ---- -------- --------
584-0063          00 03 23 +00 01.8 18.1        18.3              f  0.219   93 00 05 57 +00 18.7
643-0083          23 50 40 +00 33.4 15.9        17.0              k  0.197   93 23 53 14 +00 50.3
584-0030          23 54 05 -01 32.3 16.6        17.7              k  0.199  193 23 56 39 -01 15.4
```

## 4.2 Example 2

This illustrates adding new output fields to SIMBAD queries. Run `list_votable_fields` to get the full list of valid fields.

```
>>> from astroquery.simbad import Simbad
>>> s = Simbad()
```

(continues on next page)

(continued from previous page)

```
>>> # bibcodelist(date1-date2) lists the number of bibliography
>>> # items referring to each object over that date range
>>> s.add_votable_fields('bibcodelist(2003-2013)')
>>> r = s.query_object('m31')
>>> r.pprint()
MAIN_ID      RA          DEC       RA_PREC DEC_PREC COO_ERR_MAJA COO_ERR_MINA COO_ERR_ANGLE COO_QUAL COO_
↪WAVELENGTH     COO_BIBCODE     BIBLIST_2003_2013
------- ------------ ------------ ------- -------- ------------ ------------ ------------- -------- ----
↪---------- ------------------- -----------------
  M  31 00 42 44.330 +41 16 07.50       7        7          nan          nan             0        B    ␣
↪         I 2006AJ....131.1163S              3758
```

## 4.3 Example 3

This illustrates finding the spectral type of some particular star.

```
>>> from astroquery.simbad import Simbad
>>> customSimbad = Simbad()
>>> customSimbad.add_votable_fields('sptype')
>>> result = customSimbad.query_object('g her')
>>> result['MAIN_ID'][0]
'V* g Her'
>>> result['SP_TYPE'][0]
'M6III'
```

## 4.4 Example 4

```
>>> from astroquery.simbad import Simbad
>>> customSimbad = Simbad()
>>> # We've seen errors where ra_prec was NAN, but it's an int: that's a problem
>>> # this is a workaround we adapted
>>> customSimbad.add_votable_fields('ra(d)','dec(d)')
>>> customSimbad.remove_votable_fields('coordinates')
>>> from astropy import coordinates
>>> C = coordinates.SkyCoord(0,0,unit=('deg','deg'), frame='icrs')
>>> result = customSimbad.query_region(C, radius='2 degrees')
>>> result[:5].pprint()
    MAIN_ID        RA_d        DEC_d
 ------------- ----------- ------------
 ALFALFA 5-186  0.00000000   0.00000000
 ALFALFA 5-188  0.00000000   0.00000000
 ALFALFA 5-206  0.00000000   0.00000000
 ALFALFA 5-241  0.00000000   0.00000000
 ALFALFA 5-293  0.00000000   0.00000000
```

## 4.5 Example 5

This illustrates a simple usage of the open_exoplanet_catalogue module.

Finding the mass of a specific planet:

```
>>> from astroquery import open_exoplanet_catalogue as oec
>>> from astroquery.open_exoplanet_catalogue import findvalue
>>> cata = oec.get_catalogue()
>>> kepler68b = cata.find(".//planet[name='Kepler-68 b']")
>>> print(findvalue( kepler68b, 'mass'))
0.02105109
```

## 4.6 Example 6

Grab some data from ALMA, then analyze it using the Spectral Cube package after identifying some spectral lines in the data.

```
from astroquery.alma import Alma
from astroquery.splatalogue import Splatalogue
from astroquery.simbad import Simbad
from astropy import units as u
from astropy import constants
from spectral_cube import SpectralCube

m83table = Alma.query_object('M83', public=True)
m83urls = Alma.stage_data(m83table['Member ous id'])
# Sometimes there can be duplicates: avoid them with
# list(set())
m83files = Alma.download_and_extract_files(list(set(m83urls['URL'])))
m83files = m83files

Simbad.add_votable_fields('rvel')
m83simbad = Simbad.query_object('M83')
rvel = m83simbad['RVel_Rvel'][0]*u.Unit(m83simbad['RVel_Rvel'].unit)

for fn in m83files:
    if 'line' in fn:
        cube = SpectralCube.read(fn)
        # Convert frequencies to their rest frequencies
        frange = u.Quantity([cube.spectral_axis.min(),
                             cube.spectral_axis.max()]) * (1+rvel/constants.c)

        # Query the top 20 most common species in the frequency range of the
        # cube with an upper energy state <= 50K
        lines = Splatalogue.query_lines(frange[0], frange[1], top20='top20',
                                        energy_max=50, energy_type='eu_k',
                                        only_NRAO_recommended=True)
        lines.pprint()

        # Change the cube coordinate system to be in velocity with respect
        # to the rest frequency (in the M83 rest frame)
        rest_frequency = lines['Freq-GHz'][0]*u.GHz / (1+rvel/constants.c)
        vcube = cube.with_spectral_unit(u.km/u.s,
                                        rest_value=rest_frequency,
                                        velocity_convention='radio')

        # Write the cube with the specified line name
        fmt = "{Species}{Resolved QNs}"
        row = lines[0]
        linename = fmt.format(**dict(zip(row.colnames,row.data)))
```

(continues on next page)

(continued from previous page)

```
        vcube.write('M83_ALMA_{linename}.fits'.format(linename=linename))
```

## 4.7 Example 7

Find ALMA pointings that have been observed toward M83, then overplot the various fields-of view on a 2MASS image retrieved from SkyView. See http://nbviewer.ipython.org/gist/keflavich/19175791176e8d1fb204 for the notebook. There is an even more sophisticated version at http://nbviewer.ipython.org/gist/keflavich/bb12b772d6668cf9181a, which shows Orion KL in all observed bands.

```
# Querying ALMA archive for M83 pointings and plotting them on a 2MASS image

# In[2]:

from astroquery.alma import Alma
from astroquery.skyview import SkyView
import string
from astropy import units as u
import pylab as pl
import aplpy


# Retrieve M83 2MASS K-band image:

# In[3]:

m83_images = SkyView.get_images(position='M83', survey=['2MASS-K'], pixels=1500)


# Retrieve ALMA archive information *including* private data and non-science fields:
#

# In[4]:

m83 = Alma.query_object('M83', public=False, science=False)


# In[5]:

m83


# Parse components of the ALMA data.  Specifically, find the frequency support - the frequency range␣
↪covered - and convert that into a central frequency for beam radius estimation.

# In[6]:

def parse_frequency_support(frequency_support_str):
    supports = frequency_support_str.split("U")
    freq_ranges = [(float(sup.strip('[] ').split("..")[0]),
                    float(sup.strip('[] ').split("..")[1].split(',')[0].strip(string.letters)))
                   *u.Unit(sup.strip('[] ').split("..")[1].split(',')[0].strip(string.
↪punctuation+string.digits))
                   for sup in supports]
    return u.Quantity(freq_ranges)
```

(continues on next page)

(continued from previous page)

```
def approximate_primary_beam_sizes(frequency_support_str):
    freq_ranges = parse_frequency_support(frequency_support_str)
    beam_sizes = [(1.22*fr.mean().to(u.m, u.spectral())/(12*u.m)).to(u.arcsec,
                                                                    u.dimensionless_angles())
                  for fr in freq_ranges]
    return u.Quantity(beam_sizes)


# In[7]:

primary_beam_radii = [approximate_primary_beam_sizes(row['Frequency support']) for row in m83]


# Compute primary beam parameters for the public and private components of the data for plotting below.

# In[8]:

print("The bands used include: ",np.unique(m83['Band']))


# In[9]:

private_circle_parameters = [(row['RA'],row['Dec'],np.mean(rad).to(u.deg).value)
                             for row,rad in zip(m83, primary_beam_radii)
                             if row['Release date']!='' and row['Band']==3]
public_circle_parameters = [(row['RA'],row['Dec'],np.mean(rad).to(u.deg).value)
                             for row,rad in zip(m83, primary_beam_radii)
                             if row['Release date']=='' and row['Band']==3]
unique_private_circle_parameters = np.array(list(set(private_circle_parameters)))
unique_public_circle_parameters = np.array(list(set(public_circle_parameters)))

print("BAND 3")
print("PUBLIC:  Number of rows: {0}.  Unique pointings: {1}".format(len(m83), len(unique_public_circle_
↪parameters)))
print("PRIVATE: Number of rows: {0}.  Unique pointings: {1}".format(len(m83), len(unique_private_circle_
↪parameters)))

private_circle_parameters_band6 = [(row['RA'],row['Dec'],np.mean(rad).to(u.deg).value)
                             for row,rad in zip(m83, primary_beam_radii)
                             if row['Release date']!='' and row['Band']==6]
public_circle_parameters_band6 = [(row['RA'],row['Dec'],np.mean(rad).to(u.deg).value)
                             for row,rad in zip(m83, primary_beam_radii)
                             if row['Release date']=='' and row['Band']==6]


# Show all of the private observation pointings that have been acquired

# In[10]:

fig = aplpy.FITSFigure(m83_images[0])
fig.show_grayscale(stretch='arcsinh')
fig.show_circles(unique_private_circle_parameters[:,0],
                 unique_private_circle_parameters[:,1],
                 unique_private_circle_parameters[:,2],
                 color='r', alpha=0.2)
```

(continues on next page)

(continued from previous page)

```
# In principle, all of the pointings shown below should be downloadable from the archive:

# In[11]:

fig = aplpy.FITSFigure(m83_images[0])
fig.show_grayscale(stretch='arcsinh')
fig.show_circles(unique_public_circle_parameters[:,0],
                 unique_public_circle_parameters[:,1],
                 unique_public_circle_parameters[:,2],
                 color='b', alpha=0.2)


# Use pyregion to write the observed regions to disk.  Pyregion has a very awkward API; there is (in␣
↪principle) work in progress to improve that situation but for now one must do all this extra work.

# In[16]:

import pyregion
from pyregion.parser_helper import Shape
prv_regions = pyregion.ShapeList([Shape('circle',[x,y,r]) for x,y,r in private_circle_parameters])
pub_regions = pyregion.ShapeList([Shape('circle',[x,y,r]) for x,y,r in public_circle_parameters])
for r,(x,y,c) in zip(prv_regions+pub_regions,
                     np.vstack([private_circle_parameters,
                                public_circle_parameters])):
    r.coord_format = 'fk5'
    r.coord_list = [x,y,c]
    r.attr = ([], {'color': 'green',  'dash': '0 ',  'dashlist': '8 3 ',  'delete': '1 ',  'edit': '1 ',
                   'fixed': '0 ',  'font': '"helvetica 10 normal roman"',  'highlite': '1 ',
                   'include': '1 ',  'move': '1 ',  'select': '1 ',  'source': '1',  'text': '',
                   'width': '1 '})

prv_regions.write('M83_observed_regions_private_March2015.reg')
pub_regions.write('M83_observed_regions_public_March2015.reg')


# In[17]:

from astropy.io import fits


# In[18]:

prv_mask = fits.PrimaryHDU(prv_regions.get_mask(m83_images[0][0]).astype('int'),
                           header=m83_images[0][0].header)
pub_mask = fits.PrimaryHDU(pub_regions.get_mask(m83_images[0][0]).astype('int'),
                           header=m83_images[0][0].header)


# In[19]:

pub_mask.writeto('public_m83_almaobs_mask.fits', clobber=True)


# In[20]:

fig = aplpy.FITSFigure(m83_images[0])
```

(continues on next page)

(continued from previous page)

```
fig.show_grayscale(stretch='arcsinh')
fig.show_contour(prv_mask, levels=[0.5,1], colors=['r','r'])
fig.show_contour(pub_mask, levels=[0.5,1], colors=['b','b'])


# ## More advanced ##
#
# Now we create a 'hit mask' showing the relative depth of each observed field in each band

# In[21]:

hit_mask_band3_public = np.zeros_like(m83_images[0][0].data)
hit_mask_band3_private = np.zeros_like(m83_images[0][0].data)
hit_mask_band6_public = np.zeros_like(m83_images[0][0].data)
hit_mask_band6_private = np.zeros_like(m83_images[0][0].data)
from astropy import wcs
mywcs = wcs.WCS(m83_images[0][0].header)


# In[22]:

for row,rad in zip(m83, primary_beam_radii):
    shape = Shape('circle', (row['RA'], row['Dec'],np.mean(rad).to(u.deg).value))
    shape.coord_format = 'fk5'
    shape.coord_list = (row['RA'], row['Dec'],np.mean(rad).to(u.deg).value)
    shape.attr = ([], {'color': 'green',  'dash': '0 ',  'dashlist': '8 3 ',  'delete': '1 ',  'edit':
↪'1 ',
                   'fixed': '0 ',  'font': '"helvetica 10 normal roman"',  'highlite': '1 ',
                   'include': '1 ',  'move': '1 ',  'select': '1 ',  'source': '1',  'text': '',
                   'width': '1 '})
    if row['Release date']=='' and row['Band']==3:
        (xlo,xhi,ylo,yhi),mask = pyregion_subset(shape, hit_mask_band3_private, mywcs)
        hit_mask_band3_private[ylo:yhi,xlo:xhi] += row['Integration']*mask
    elif row['Release date'] and row['Band']==3:
        (xlo,xhi,ylo,yhi),mask = pyregion_subset(shape, hit_mask_band3_public, mywcs)
        hit_mask_band3_public[ylo:yhi,xlo:xhi] += row['Integration']*mask
    elif row['Release date'] and row['Band']==6:
        (xlo,xhi,ylo,yhi),mask = pyregion_subset(shape, hit_mask_band6_public, mywcs)
        hit_mask_band6_public[ylo:yhi,xlo:xhi] += row['Integration']*mask
    elif row['Release date']=='' and row['Band']==6:
        (xlo,xhi,ylo,yhi),mask = pyregion_subset(shape, hit_mask_band6_private, mywcs)
        hit_mask_band6_private[ylo:yhi,xlo:xhi] += row['Integration']*mask


# In[23]:

fig = aplpy.FITSFigure(m83_images[0])
fig.show_grayscale(stretch='arcsinh')
fig.show_contour(fits.PrimaryHDU(data=hit_mask_band3_public, header=m83_images[0][0].header),
                 levels=np.logspace(0,5,base=2, num=6), colors=['r']*6)
fig.show_contour(fits.PrimaryHDU(data=hit_mask_band3_private, header=m83_images[0][0].header),
                 levels=np.logspace(0,5,base=2, num=6), colors=['y']*6)
fig.show_contour(fits.PrimaryHDU(data=hit_mask_band6_public, header=m83_images[0][0].header),
                 levels=np.logspace(0,5,base=2, num=6), colors=['c']*6)
fig.show_contour(fits.PrimaryHDU(data=hit_mask_band6_private, header=m83_images[0][0].header),
                 levels=np.logspace(0,5,base=2, num=6), colors=['b']*6)
```

(continues on next page)

(continued from previous page)

```
# In[24]:

from astropy import wcs
import pyregion
from astropy import log

def pyregion_subset(region, data, mywcs):
    """
    Return a subset of an image (`data`) given a region.
    """
    shapelist = pyregion.ShapeList([region])
    if shapelist[0].coord_format not in ('physical','image'):
        # Requires astropy >0.4...
        # pixel_regions = shapelist.as_imagecoord(self.wcs.celestial.to_header())
        # convert the regions to image (pixel) coordinates
        celhdr = mywcs.sub([wcs.WCSSUB_CELESTIAL]).to_header()
        pixel_regions = shapelist.as_imagecoord(celhdr)
    else:
        # For this to work, we'd need to change the reference pixel after cropping.
        # Alternatively, we can just make the full-sized mask... todo....
        raise NotImplementedError("Can't use non-celestial coordinates with regions.")
        pixel_regions = shapelist

    # This is a hack to use mpl to determine the outer bounds of the regions
    # (but it's a legit hack - pyregion needs a major internal refactor
    # before we can approach this any other way, I think -AG)
    mpl_objs = pixel_regions.get_mpl_patches_texts()[0]

    # Find the minimal enclosing box containing all of the regions
    # (this will speed up the mask creation below)
    extent = mpl_objs[0].get_extents()
    xlo, ylo = extent.min
    xhi, yhi = extent.max
    all_extents = [obj.get_extents() for obj in mpl_objs]
    for ext in all_extents:
        xlo = xlo if xlo < ext.min[0] else ext.min[0]
        ylo = ylo if ylo < ext.min[1] else ext.min[1]
        xhi = xhi if xhi > ext.max[0] else ext.max[0]
        yhi = yhi if yhi > ext.max[1] else ext.max[1]

    log.debug("Region boundaries: ")
    log.debug("xlo={xlo}, ylo={ylo}, xhi={xhi}, yhi={yhi}".format(xlo=xlo,
                                                                 ylo=ylo,
                                                                 xhi=xhi,
                                                                 yhi=yhi))


    subwcs = mywcs[ylo:yhi, xlo:xhi]
    subhdr = subwcs.sub([wcs.WCSSUB_CELESTIAL]).to_header()
    subdata = data[ylo:yhi, xlo:xhi]

    mask = shapelist.get_mask(header=subhdr,
                              shape=subdata.shape)
    log.debug("Shapes: data={0}, subdata={2}, mask={1}".format(data.shape, mask.shape, subdata.shape))
    return (xlo,xhi,ylo,yhi),mask
```

## 4.8 Example 8

Retrieve data from a particular co-I or PI from the ESO archive

```
from astroquery.eso import Eso

# log in so you can get proprietary data
Eso.login('aginsburg')
# make sure you don't filter out anything
Eso.ROW_LIMIT = 1e6

# List all of your pi/co projects
all_pi_proj = Eso.query_instrument('apex', pi_coi='ginsburg')

# Have a look at the project IDs only
print(set(all_pi_proj['APEX Project ID']))
# set(['E-095.F-9802A-2015', 'E-095.C-0242A-2015', 'E-093.C-0144A-2014'])

# The full project name includes prefix and suffix
full_proj = 'E-095.F-9802A-2015'
proj_id = full_proj[2:-6]

# Then get the APEX quicklook "reduced" data
tbl = Eso.query_apex_quicklooks(prog_id=proj_id)

# and finally, download it
files = Eso.retrieve_data(tbl['Product ID'])

# then move the files to your local directory
# note that there is no .TAR suffix... not sure why this is
import shutil
for fn in files:
    shutil.move(fn+'.TAR','.')
```

The following modules have been completed using a common API:

CHAPTER 5

---

## SIMBAD Queries (astroquery.simbad)

---

## 5.1 Getting started

This module can be used to query the Simbad service. Presented below are examples that illustrate the different types of queries that can be formulated. If successful all the queries will return the results in a Table.

### 5.1.1 Query an Identifier

This is useful if you want to query a known identifier. For instance to query the messier object m1:

```
>>> from astroquery.simbad import Simbad
>>> result_table = Simbad.query_object("m1")
>>> print(result_table)

MAIN_ID      RA          DEC      RA_PREC DEC_PREC COO_ERR_MAJA COO_ERR_MINA COO_ERR_ANGLE COO_QUAL COO_
↪WAVELENGTH     COO_BIBCODE
------- ----------- ----------- ------- -------- ------------ ------------ ------------- -------- ------
↪--------- -------------------
M   1 05 34 31.94 +22 00 52.2       6        6          nan          nan             0        C      ␣
↪      R 2011A&A...533A..10L
```

Wildcards are supported. So for instance to query messier objects from 1 through 9:

```
>>> from astroquery.simbad import Simbad
>>> result_table = Simbad.query_object("m [1-9]", wildcard=True)
>>> print(result_table)

MAIN_ID      RA          DEC      RA_PREC DEC_PREC COO_ERR_MAJA COO_ERR_MINA COO_ERR_ANGLE COO_QUAL COO_
↪WAVELENGTH     COO_BIBCODE
------- ----------- ----------- ------- -------- ------------ ------------ ------------- -------- ------
↪--------- -------------------
M   1 05 34 31.94 +22 00 52.2       6        6          nan          nan             0        C      ␣
↪      R 2011A&A...533A..10L
```

(continues on next page)

(continued from previous page)

```
M   2 21 33 27.02 -00 49 23.7       6         6       100.000       100.000             0         C         ␣
↪     O 2010AJ....140.1830G
M   3 13 42 11.62 +28 22 38.2       6         6       200.000       200.000             0         C         ␣
↪     O 2010AJ....140.1830G
M   4 16 23 35.22 -26 31 32.7       6         6       400.000       400.000             0         C         ␣
↪     O 2010AJ....140.1830G
M   5 15 18 33.22 +02 04 51.7       6         6           nan           nan             0         C         ␣
↪     O 2010AJ....140.1830G
M   6    17 40 20    -32 15.2       4         4           nan           nan             0         E         ␣
↪     O 2009MNRAS.399.2146W
M   7    17 53 51    -34 47.6       4         4           nan           nan             0         E         ␣
↪     O 2009MNRAS.399.2146W
M   8    18 03 37    -24 23.2       4         4     18000.000     18000.000           179         E
M   9 17 19 11.78 -18 30 58.5       6         6           nan           nan             0         D         ␣
↪       2002MNRAS.332..441F
```

Wildcards are supported by other queries as well - where this is the case, examples are presented to this end. The wildcards that are supported and their usage across all these queries is the same. To see the available wildcards and their functions:

```
>>> from astroquery.simbad import Simbad
>>> Simbad.list_wildcards()

* : Any string of characters (including an empty one)

[^0-9] : Any (one) character not in the list.

? : Any character (exactly one character)

[abc] : Exactly one character taken in the list. Can also be defined by a range of characters: [A-Z]
```

### 5.1.2 Query a region

Queries that support a cone search with a specified radius - around an identifier or given coordinates are also supported. If an identifier is used then it will be resolved to coordinates using online name resolving services available in Astropy.

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

           MAIN_ID                 RA            DEC      RA_PREC DEC_PREC ... COO_ERR_ANGLE COO_QUAL␣
↪COO_WAVELENGTH     COO_BIBCODE
-------------------------- ------------- ------------- ------- --------- ... ------------- -------- -----
↪--------- -------------------
[VV2006c] J095534.0+043546 09 55 33.9854 +04 35 46.438       8        8 ...             0        B      ␣
↪        O 2009A&A...505..385A
...
```

When no radius is specified, the radius defaults to 20 arcmin. A radius may also be explicitly specified - it can be entered either as a string that is acceptable by `Angle` or by using the `Quantity` object:

```
>>> from astroquery.simbad import Simbad
>>> import astropy.units as u
>>> result_table = Simbad.query_region("m81", radius=0.1 * u.deg)
```

(continues on next page)

(continued from previous page)

```
>>> # another way to specify the radius.
>>> result_table = Simbad.query_region("m81", radius='0d6m0s')
>>> print(result_table)

        MAIN_ID               RA      ...     COO_BIBCODE
----------------------- ------------- ... -------------------
                  M  81 09 55 33.1730 ... 2004AJ....127.3587F
         [SPZ2011] ML2   09 55 32.97 ... 2011ApJ...735...26S
             [F88] X-5   09 55 33.32 ... 2001ApJ...554..202I
         [SPZ2011] 264  09 55 32.618 ... 2011ApJ...735...26S
         [SPZ2011] ML1   09 55 33.10 ... 2011ApJ...735...26S
         [SPZ2011] ML3   09 55 33.99 ... 2011ApJ...735...26S
         [SPZ2011] ML5   09 55 33.39 ... 2011ApJ...735...26S
         [SPZ2011] ML6   09 55 32.47 ... 2011ApJ...735...26S
                   ...           ... ...                 ...
         [MPC2001]   8   09 54 45.50 ... 2001A&A...379...90M
2MASS J09561112+6859003   09 56 11.13 ... 2003yCat.2246....0C
          [PR95] 50721  09 56 36.460 ...
               PSK  72    09 54 54.1 ...
               PSK 353    09 56 03.7 ...
          [BBC91] S02S    09 56 07.1 ...
               PSK 489   09 56 36.55 ... 2003AJ....126.1286L
               PSK   7    09 54 37.0 ...
```

If coordinates are used, then they should be entered using an `astropy.coordinates.SkyCoord` object.

```
>>> from astroquery.simbad import Simbad
>>> import astropy.coordinates as coord
>>> result_table = Simbad.query_region(coord.SkyCoord("05h35m17.3s -05h23m28s", frame='icrs'), radius=
↪'1d0m0s')
>>> print(result_table)

        MAIN_ID               RA      ...     COO_BIBCODE
----------------------- ------------- ... -------------------
             HD  38875 05 34 59.7297 ... 2007A&A...474..653V
          TYC 9390-799-1 05 33 58.2222 ... 1998A&A...335L..65H
          TYC 9390-646-1  05 35 02.830 ... 2000A&A...355L..27H
          TYC 9390-629-1  05 35 20.419 ... 2000A&A...355L..27H
          TYC 9390-857-1  05 30 58.989 ... 2000A&A...355L..27H
        TYC 9390-1171-1 05 37 35.9623 ... 1998A&A...335L..65H
          TYC 9390-654-1  05 35 27.395 ... 2000A&A...355L..27H
          TYC 9390-656-1  05 30 43.665 ... 2000A&A...355L..27H
                   ...           ... ...                 ...
          TYC 9373-779-1  05 11 57.788 ... 2000A&A...355L..27H
          TYC 9377-513-1 05 10 43.0669 ... 1998A&A...335L..65H
          TYC 9386-135-1  05 28 24.988 ... 2000A&A...355L..27H
        TYC 9390-1786-1  05 56 34.801 ... 2000A&A...355L..27H
2MASS J05493730-8141270   05 49 37.30 ... 2003yCat.2246....0C
          TYC 9390-157-1  05 35 55.233 ... 2000A&A...355L..27H
          PKS J0557-8122   05 57 26.80 ... 2003MNRAS.342.1117M
            PKS 0602-813    05 57 30.7 ...
```

```
>>> from astroquery.simbad import Simbad
>>> import astropy.coordinates as coord
>>> import astropy.units as u
>>> result_table = Simbad.query_region(coord.SkyCoord(31.0087, 14.0627,
```

(continues on next page)

(continued from previous page)

```
...                                    unit=(u.deg, u.deg), frame='galactic'),
...                                    radius='0d0m2s')
>>> print(result_table)

           MAIN_ID             RA       ... COO_WAVELENGTH     COO_BIBCODE
------------------- ------------- ... -------------- -------------------
NAME Barnard's star 17 57 48.4980 ...              O 2007A&A...474..653V
```

Two other options can also be specified - the epoch and the equinox. If these are not explicitly mentioned, then the epoch defaults to J2000 and the equinox to 2000.0. So here is a query with all the options utilized:

```
>>> from astroquery.simbad import Simbad
>>> import astropy.coordinates as coord
>>> import astropy.units as u
>>> result_table = Simbad.query_region(coord.SkyCoord(ra=11.70, dec=10.90,
...                                    unit=(u.deg, u.deg), frame='fk5'),
...                                    radius=0.5 * u.deg,
...                                    epoch='B1950',
...                                    equinox=1950)
>>> print(result_table)

       MAIN_ID               RA      ...     COO_BIBCODE
----------------------- ------------- ... -------------------
             PHL  6696       00 49.4 ...
            BD+10    97 00 49 25.4553 ... 2007A&A...474..653V
         TYC  607-238-1  00 48 53.302 ... 2000A&A...355L..27H
             PHL  2998       00 49.3 ...
2MASS J00492121+1121094  00 49 21.219 ... 2003yCat.2246....0C
        TYC  607-1135-1 00 48 46.5838 ... 1998A&A...335L..65H
2MASX J00495215+1118527  00 49 52.154 ... 2006AJ....131.1163S
            BD+10    98 00 50 03.4124 ... 1998A&A...335L..65H
                    ...           ... ...                 ...
         TYC  607-971-1 00 47 38.0430 ... 1998A&A...335L..65H
         TYC  607-793-1  00 50 35.545 ... 2000A&A...355L..27H
USNO-A2.0 0975-00169117  00 47 55.351 ... 2007ApJ...664...53A
         TYC  607-950-1  00 50 51.875 ... 2000A&A...355L..27H
            BD+10   100 00 51 15.0789 ... 1998A&A...335L..65H
          TYC  608-60-1  00 51 13.314 ... 2000A&A...355L..27H
         TYC  608-432-1  00 51 05.289 ... 2000A&A...355L..27H
         TYC  607-418-1  00 49 09.636 ... 2000A&A...355L..27H
```

### 5.1.3 Query a catalogue

Queries can also be formulated to return all the objects from a catalogue. For instance to query the ESO catalog:

```
>>> from astroquery.simbad import Simbad
>>> limitedSimbad = Simbad()
>>> limitedSimbad.ROW_LIMIT = 6
>>> result_table = limitedSimbad.query_catalog('eso')
>>> print(result_table)

MAIN_ID              RA       ... COO_WAVELENGTH     COO_BIBCODE
----------------------- ------------ ... -------------- -------------------
2MASS J08300740-4325465  08 30 07.41 ...              I 2003yCat.2246....0C
NGC  2573 01 41 35.091 ...              I 2006AJ....131.1163S
```

(continues on next page)

(continued from previous page)

```
ESO   1-2   05 04 36.8 ...              1982ESO...C......0L
ESO   1-3 05 22 36.509 ...            I 2006AJ....131.1163S
ESO   1-4 07 49 28.813 ...            I 2006AJ....131.1163S
ESO   1-5 08 53 05.006 ...            I 2006AJ....131.1163S
```

### 5.1.4 Query a bibcode

This retrieves the reference corresponding to a bibcode.

```
>>> from astroquery.simbad import Simbad
>>> result_table = Simbad.query_bibcode('2005A&A.430.165F')
>>> print(result_table)

References
--------------------------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------------------------
↪----------------
2005A&A...430..165F  --  ?
Astron. Astrophys., 430, 165-186 (2005)
FAMAEY B., JORISSEN A., LURI X., MAYOR M., UDRY S., DEJONGHE H. and TURON C.
Local kinematics of K and M giants from CORAVEL/Hipparcos/Tycho-2 data. Revisiting the concept of ␣
↪superclusters.
Files: (abstract)
Notes: <Available at CDS: tablea1.dat notes.dat>
```

Wildcards can be used in these queries as well. So to retrieve all the bibcodes from a given journal in a given year:

```
>>> from astroquery.simbad import Simbad
>>> result_table = Simbad.query_bibcode('2013A&A*', wildcard=True)
>>> print(result_table)

References
--------------------------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------
2013A&A...549A...1G  --  ?
Astron. Astrophys., 549A, 1-1 (2013)
GENTILE M., COURBIN F. and MEYLAN G.
Interpolating point spread function anisotropy.
Files: (abstract) (no object)

2013A&A...549A...2L  --  ?
Astron. Astrophys., 549A, 2-2 (2013)
LEE B.-C., HAN I. and PARK M.-G.
Planetary companions orbiting M giants HD 208527 and HD 220074.
Files: (abstract)

2013A&A...549A...3C  --  ?
Astron. Astrophys., 549A, 3-3 (2013)
COCCATO L., MORELLI L., PIZZELLA A., CORSINI E.M., BUSON L.M. and DALLA BONTA E.
Spectroscopic evidence of distinct stellar populations in the  counter-rotating stellar disks of NGC␣
↪3593 and NGC 4550.
Files: (abstract)
```

(continues on next page)

(continued from previous page)

```
2013A&A...549A...4S  --  ?
Astron. Astrophys., 549A, 4-4 (2013)
SCHAERER D., DE BARROS S. and SKLIAS P.
Properties of z ~ 3-6 Lyman break galaxies. I. Testing star formation  histories and the SFR-mass␣
↪relation with ALMA and near-IR spectroscopy.
Files: (abstract)

2013A&A...549A...5R  --  ?
Astron. Astrophys., 549A, 5-5 (2013)
RYGL K.L.J., WYROWSKI F., SCHULLER F. and MENTEN K.M.
Initial phases of massive star formation in high infrared extinction  clouds. II. Infall and onset of␣
↪star formation.
Files: (abstract)

2013A&A...549A...6K  --  ?
Astron. Astrophys., 549A, 6-6 (2013)
KAMINSKI T., SCHMIDT M.R. and MENTEN K.M.
Aluminium oxide in the optical spectrum of VY Canis Majoris.
Files: (abstract)
```

### 5.1.5 Query object identifiers

These queries can be used to retrieve all of the names (identifiers) associated with an object.

```
>>> from astroquery.simbad import Simbad
>>> result_table = Simbad.query_objectids("Polaris")
>>> print(result_table)
           ID
-----------------------
           NAME Polaris
        NAME North Star
          NAME Lodestar
               PLX  299
                SBC9 76
              *    1 UMi
              * alf UMi
          AAVSO 0122+88
            ADS  1477 A
             AG+89    4
            BD+88     8
      CCDM J02319+8915A
        CSI+88      8  1
               FK5  907
               GC  2243
             GCRV  1037
                    ...
              PPM   431
               ROT  3491
              SAO   308
              SBC7   51
             SKY#  3738
               TD1  835
         TYC 4628-237-1
            UBV   21589
           UBV M   8201
```

(continues on next page)

(continued from previous page)

```
            V* alf UMi
           PLX  299.00
   WDS J02318+8916Aa,Ab
          ADS   1477 AP
           ** WRH    39
       WDS J02318+8916A
          ** STF    93A
2MASS J02314822+8915503
```

### 5.1.6 Query a bibobj

These queries can be used to retrieve all the objects that are contained in the article specified by the bibcode:

```
>>> from astroquery.simbad import Simbad
>>> result_table = Simbad.query_bibobj('2006AJ....131.1163S')
>>> print(result_table)

       MAIN_ID             RA          DEC      RA_PREC DEC_PREC ... COO_ERR_MINA COO_ERR_ANGLE COO_
↪QUAL COO_WAVELENGTH     COO_BIBCODE
                        "h:m:s"      "d:m:s"                     ...     mas           deg
----------------------- ------------ ------------ ------- -------- ... ------------ ------------- ------
↪-- -------------- -------------------
                     M  32 00 42 41.825 +40 51 54.61       7        7 ...           --             0     ␣
↪ B              I 2006AJ....131.1163S
                     M  31 00 42 44.330 +41 16 07.50       7        7 ...           --             0     ␣
↪ B              I 2006AJ....131.1163S
                  NAME SMC    00 52 38.0    -72 48 01       5        5 ...           --             0     ␣
↪ D              O 2003A&A...412...45P
        Cl Melotte   22      03 47 00     +24 07.0       4        4 ...           --             0     ␣
↪ E              O 2009MNRAS.399.2146W
2MASX J04504846-7531580 04 50 48.462 -75 31 58.08       7        7 ...           --             0     ␣
↪ B              I 2006AJ....131.1163S
                  NAME LMC    05 23 34.6    -69 45 22       5        5 ...           --             0     ␣
↪ D              O 2003A&A...412...45P
      NAME Lockman Hole   10 45 00.0    +58 00 00       5        5 ...           --             0     ␣
↪ E                2011ApJ...734...99H
        NAME Gal Center  17 45 40.04  -29 00 28.1       6        6 ...           --             0     ␣
↪ E
```

### 5.1.7 Query based on any criteria

Anything done in SIMBAD's criteria interface can be done via astroquery. See that link for details of how these queries are formed.

```
>>> from astroquery.simbad import Simbad
>>> result = Simbad.query_criteria('region(box, GAL, 0 +0, 3d 1d)', otype='SNR')
>>> print(result)
       MAIN_ID              RA          DEC      RA_PREC DEC_PREC COO_ERR_MAJA COO_ERR_MINA COO_ERR_ANGLE␣
↪COO_QUAL COO_WAVELENGTH     COO_BIBCODE
--------------------- ----------- ----------- ------- -------- ------------ ------------ ------------- -
↪------- -------------- -------------------
  EQ J174702.6-282733  17 47 02.6   -28 27 33       5        5          nan          nan             0 ␣
↪      D                2002ApJ...565.1017S
```

(continues on next page)

(continued from previous page)

```
[L92] 174535.0-280410  17 48 44.4   -28 05 06       5        5     3000.000     3000.000            0 ␣
↪       D
           [GWC93] 19  17 42 04.9   -30 04 04       5        5     3000.000     3000.000            1 ␣
↪       D
      SNR G359.1-00.2    17 43 29     -29 45.9      4        4          nan          nan            0 ␣
↪       E                2000AJ....119..207L
      SNR G000.1-00.2  17 48 42.5   -28 09 11       5        5          nan          nan            0 ␣
↪       D                2008ApJS..177..255L
      SNR G359.9-00.9     17 45.8       -29 03      3        3          nan          nan            0
      SNR G359.4-00.1    17 44 37     -29 27.2      4        4    18000.000    18000.000            1 ␣
↪       E
           NAME SGR D    17 48 42     -28 01.4      4        4    18000.000    18000.000            0 ␣
↪       E
      SNR G359.1-00.5    17 45 25     -29 57.9      4        4    18000.000    18000.000            1 ␣
↪       E
       NAME SGR D SNR     17 48.7       -28 07      3        3          nan          nan            0 ␣
↪       E
    Suzaku J1747-2824    17 47 00     -28 24.5      4        4          nan          nan            0 ␣
↪       E                2007ApJ...666..934C
      SNR G000.4+00.2 17 46 27.65 -28 36 05.6       6        6      300.000      300.000            1 ␣
↪       D
      SNR G001.4-00.1  17 49 28.1   -27 47 45       5        5          nan          nan            0 ␣
↪       D                1999ApJ...527..172Y
     GAL 000.61+00.01     17 47.0       -28 25      3        3          nan          nan            0 ␣
↪       D
      SNR G000.9+00.1     17 47.3       -28 09      3        3          nan          nan            0 ␣
↪       E              R 2009BASI...37...45G
      SNR G000.3+00.0  17 46 14.9   -28 37 15       5        5     3000.000     3000.000            1 ␣
↪       D
      SNR G001.0-00.1     17 48.5       -28 09      3        3          nan          nan            0 ␣
↪       E              R 2009BASI...37...45G
      NAME SGR A EAST    17 45 47     -29 00.2      4        4    18000.000    18000.000            1 ␣
↪       E
```

## 5.2 Customizing the default settings

There may be times when you wish to change the defaults that have been set for the Simbad queries.

### 5.2.1 Changing the row limit

To fetch all the rows in the result, the row limit must be set to 0. However for some queries, results are likely to be very large, in such cases it may be best to limit the rows to a smaller number. If you want to do this only for the current python session then:

```
>>> from astroquery.simbad import Simbad
>>> Simbad.ROW_LIMIT = 15 # now any query fetches at most 15 rows
```

If you would like to make your choice persistent, then you can do this by modifying the setting in the Astroquery configuration file.

### 5.2.2 Changing the timeout

The timeout is the time limit in seconds for establishing connection with the Simbad server and by default it is set to 100 seconds. You may want to modify this - again you can do this at run-time if you want to adjust it only for the current session. To make it persistent, you must modify the setting in the Astroquery configuration file.

```
>>> from astroquery.simbad import Simbad
>>> Simbad.TIMEOUT = 60 # sets the timeout to 60s
```

### 5.2.3 Specifying which VOTable fields to include in the result

The VOTable fields that are currently returned in the result are set to `main_id` and `coordinates`. However you can specify other fields that you also want to be fetched in the result. To see the list of the fields:

```
 >>> from astroquery.simbad import Simbad
 >>> Simbad.list_votable_fields()

          col0                   col1              col2
------------------------ -------------------- --------------
      bibcodelist(y1-y2) fluxdata(filtername)       plx_qual
                     cel                 gcrv             pm
                    cl.g                  gen     pm_bibcode
                coo(opt)                   gj   pm_err_angle
             coo_bibcode                 hbet    pm_err_maja
           coo_err_angle                hbet1    pm_err_mina
            coo_err_maja                 hgam        pm_qual
```

The above shows just a small snippet of the table that is returned and has all the fields sorted lexicographically column-wise. For more information on a particular field:

```
>>> from astroquery.simbad import Simbad
>>> Simbad.get_field_description('ra_prec')

right ascension precision code (0:1/10deg, ..., 8: 1/1000 arcsec)
```

To set additional fields to be returned in the VOTable:

```
>>> from astroquery.simbad import Simbad
>>> customSimbad = Simbad()

# see which fields are currently set

>>> customSimbad.get_votable_fields()

['main_id', 'coordinates']

# To set other fields

>>> customSimbad.add_votable_fields('mk', 'rot', 'bibcodelist(1800-2014)')
>>> customSimbad.get_votable_fields()

['main_id', 'coordinates', 'mk', 'rot', 'bibcodelist(1800-2014')]
```

You can also remove a field you have set or `astroquery.simbad.SimbadClass.reset_votable_fields()`. Continuing from the above example:

```
>>> customSimbad.remove_votable_fields('mk', 'coordinates')
>>> customSimbad.get_votable_fields()

['main_id', 'rot', 'bibcodelist(1800-2014)']

# reset back to defaults

>>> customSimbad.reset_votable_fields()
>>> customSimbad.get_votable_fields()

['main_id', 'coordinates']
```

### 5.2.4 Specifying the format of the included VOTable fields

The output for several of the VOTable fields can be formatted in many different ways described in the help page of the SIMBAD query interface (see Sect. 4.3 of this page). As an example, the epoch and equinox for the Right Ascension and Declination can be specified as follows (e.g. epoch of J2017.5 and equinox of 2000):

```
>>> customSimbad.add_votable_fields('ra(2;A;ICRS;J2017.5;2000)', 'dec(2;D;ICRS;2017.5;2000)')
>>> customSimbad.remove_votable_fields('coordinates')
>>> customSimbad.query_object("HD189733")
<Table masked=True length=1>
 MAIN_ID  RA_2_A_ICRS_J2017_5_2000 DEC_2_D_ICRS_2017_5_2000
                  "h:m:s"                  "d:m:s"
  object           str13                    str13
--------- ------------------------ ------------------------
HD 189733            20 00 43.7107            +22 42 39.064
```

## 5.3 Reference/API

### 5.3.1 astroquery.simbad Package

#### SIMBAD Query Tool

The SIMBAD query tool creates a script query that returns VOtable XML data that is then parsed into a SimbadResult object. This object then parses the data and returns a table parsed with `astropy.io.votable.parse`.

#### Classes

| | |
|---|---|
| `SimbadClass`() | The class for querying the Simbad web service. |
| `Conf` | Configuration parameters for `astroquery.simbad`. |

#### SimbadClass

**class** astroquery.simbad.**SimbadClass**

Bases: `astroquery.query.BaseQuery`

The class for querying the Simbad web service.

Note that SIMBAD suggests submitting no more than 6 queries per second; if you submit more than that, your

IP may be temporarily blacklisted (http://simbad.u-strasbg.fr/simbad/sim-help?Page=sim-url)

#### Attributes Summary

| |
|---|
| ROW_LIMIT |
| SIMBAD_URL |
| TIMEOUT |
| WILDCARDS |

#### Methods Summary

| | |
|---|---|
| add_votable_fields(*args) | Sets fields to be fetched in the VOTable. |
| get_field_description(field_name) | Displays a description of the VOTable field. |
| get_votable_fields() | Display votable fields |
| list_votable_fields() | Lists all the fields that can be fetched for a VOTable. |
| list_wildcards() | Displays the available wildcards that may be used in Simbad queries and their usage. |
| query_bibcode(bibcode[, wildcard, verbose, ...]) | Queries the references corresponding to a given bibcode, and returns the results in a Table. |
| query_bibcode_async(bibcode[, wildcard, ...]) | Serves the same function as query_bibcode, but only collects the response from the Simbad server and returns. |
| query_bibobj(bibcode[, verbose, ...]) | Query all the objects that are contained in the article specified by the bibcode, and return results as a Table. |
| query_bibobj_async(bibcode[, cache, ...]) | Serves the same function as query_bibobj, but only collects the response from the Simbad server and returns. |
| query_catalog(catalog[, verbose, cache, ...]) | Queries a whole catalog. |
| query_catalog_async(catalog[, cache, ...]) | Serves the same function as query_catalog, but only collects the response from the Simbad server and returns. |
| query_criteria(*args, **kwargs) | Query SIMBAD based on any criteria. |
| query_criteria_async(*args, **kwargs) | Query SIMBAD based on any criteria. |
| query_object(object_name[, wildcard, ...]) | Queries Simbad for the given object and returns the result as a Table. |
| query_object_async(object_name[, wildcard, ...]) | Serves the same function as query_object, but only collects the response from the Simbad server and returns. |
| query_objectids(object_name[, verbose, ...]) | Query Simbad with an object name, and return a table of all names associated with that object in a Table. |
| query_objectids_async(object_name[, cache, ...]) | Serves the same function as query_objectids, but only collects the response from the Simbad server and returns. |
| query_objects(object_names[, wildcard, ...]) | Queries Simbad for the specified list of objects and returns the results as a Table. |
| query_objects_async(object_names[, ...]) | Same as query_objects, but only collects the response from the Simbad server and returns. |
| query_region(*args, **kwargs) | Queries the service and returns a table object. |

Continued on next page

Table 3 – continued from previous page

| | |
|---|---|
| query_region_async(coordinates[, radius, . . . ]) | Serves the same function as query_region, but only collects the response from the Simbad server and returns. |
| remove_votable_fields(*args, **kwargs) | Removes the specified field names from SimbadClass._VOTABLE_FIELDS |
| reset_votable_fields() | resets VOTABLE_FIELDS to defaults |

### Attributes Documentation

**ROW_LIMIT = 0**

**SIMBAD_URL = 'http://simbad.u-strasbg.fr/simbad/sim-script'**

**TIMEOUT = 60**

**WILDCARDS = {'*': 'Any string of characters (including an empty one)', '?': 'Any character (exactly one**

### Methods Documentation

**add_votable_fields**(**args*)

Sets fields to be fetched in the VOTable. Must be one of those listed by list_votable_fields.

**Parameters**

**list of field_names**

**get_field_description**(*field_name*)

Displays a description of the VOTable field.

**Parameters**

**field_name** : str

the name of the field to describe. Must be one of those listed by list_votable_fields.

#### Examples

```
>>> from astroquery.simbad import Simbad
>>> Simbad.get_field_description('main_id')
main identifier of an astronomical object. It is the same as id(1)
>>> Simbad.get_field_description('bibcodelist(y1-y2)')
number of references. The parameter is optional and limit the count to
the references between the years y1 and y2
```

**get_votable_fields**()

Display votable fields

#### Examples

```
>>> from astroquery.simbad import Simbad
>>> Simbad.get_votable_fields()
['main_id', 'coordinates']
```

**list_votable_fields()**
Lists all the fields that can be fetched for a VOTable.

### Examples

```
>>> from astroquery.simbad import Simbad
>>> Simbad.list_votable_fields()
--NOTES--...
```

**list_wildcards()**
Displays the available wildcards that may be used in Simbad queries and their usage.

### Examples

```
>>> from astroquery.simbad import Simbad
>>> Simbad.list_wildcards()
* : Any string of characters (including an empty one)...
```

[^0-9] : Any (one) character not in the list.

? : Any character (exactly one character)

**[abc]**
[Exactly one character taken in the list.] Can also be defined by a range of characters: [A-Z]

**query_bibcode**(*bibcode*, *wildcard=False*, *verbose=False*, *cache=True*, *get_query_payload=False*)
Queries the references corresponding to a given bibcode, and returns the results in a `Table`. Wildcards may be used to specify bibcodes.

**Parameters**
**bibcode** : str

the bibcode of the article

**wildcard** : boolean, optional

When it is set to `True` it implies that the object is specified with wildcards. Defaults to `False`.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**
**table** : `Table`

Query results table

**query_bibcode_async**(*bibcode*, *wildcard=False*, *cache=True*, *get_query_payload=False*)
Serves the same function as `query_bibcode`, but only collects the response from the Simbad server and returns.

**Parameters**
**bibcode** : str

the bibcode of the article

**wildcard** : boolean, optional

When it is set to `True` it implies that the object is specified with wildcards. Defaults to `False`.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**response** : `requests.Response`

Response of the query from the server.

**query_bibobj**(*bibcode*, *verbose=False*, *get_query_payload=False*)

Query all the objects that are contained in the article specified by the bibcode, and return results as a `Table`.

**Parameters**

**bibcode** : str

the bibcode of the article

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**table** : `Table`

Query results table

**query_bibobj_async**(*bibcode*, *cache=True*, *get_query_payload=False*)

Serves the same function as `query_bibobj`, but only collects the response from the Simbad server and returns.

**Parameters**

**bibcode** : str

the bibcode of the article

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**response** : `requests.Response`

Response of the query from the server.

**query_catalog**(*catalog*, *verbose=False*, *cache=True*, *get_query_payload=False*)

Queries a whole catalog.

Results may be very large -number of rows should be controlled by configuring `SimbadClass.ROW_LIMIT`.

**Parameters**

**catalog** : str

the name of the catalog.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**table** : `Table`

Query results table

**query_catalog_async**(*catalog*, *cache=True*, *get_query_payload=False*)

Serves the same function as `query_catalog`, but only collects the response from the Simbad server and returns.

**Parameters**

**catalog** : str

the name of the catalog.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**response** : `requests.Response`

Response of the query from the server.

**query_criteria**(**args*, ***kwargs*)

Query SIMBAD based on any criteria.

**Parameters**

**args:**

String arguments passed directly to SIMBAD's script (e.g., 'region(box, GAL, 10.5 -10.5, 0.5d 0.5d)')

**kwargs:**

Keyword / value pairs passed to SIMBAD's script engine (e.g., {'otype':'SNR'} will be rendered as otype=SNR)

**Returns**

**table** : `Table`

Query results table

**query_criteria_async**(**args*, ***kwargs*)

Query SIMBAD based on any criteria.

**Parameters**

**args:**

String arguments passed directly to SIMBAD's script (e.g., 'region(box, GAL, 10.5 -10.5, 0.5d 0.5d)')

**kwargs:**

Keyword / value pairs passed to SIMBAD's script engine (e.g., {'otype':'SNR'} will be rendered as otype=SNR)

**cache** : bool

Cache the query?

**Returns**

**response** : `requests.Response`

Response of the query from the server

**query_object**(*object_name*, *wildcard=False*, *verbose=False*, *get_query_payload=False*)

Queries Simbad for the given object and returns the result as a `Table`. Object names may also be specified with wildcard. See examples below.

**Parameters**

**object_name** : str

name of object to be queried

**wildcard** : boolean, optional

When it is set to `True` it implies that the object is specified with wildcards. Defaults to `False`.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**table** : `Table`

Query results table

**query_object_async**(*object_name*, *wildcard=False*, *cache=True*, *get_query_payload=False*)

Serves the same function as `query_object`, but only collects the response from the Simbad server and returns.

**Parameters**

**object_name** : str

name of object to be queried

**wildcard** : boolean, optional

When it is set to `True` it implies that the object is specified with wildcards. Defaults to `False`.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**response** : `requests.Response`

Response of the query from the server

**query_objectids**(*object_name*, *verbose=False*, *cache=True*, *get_query_payload=False*)

Query Simbad with an object name, and return a table of all names associated with that object in a `Table`.

**Parameters**

**object_name** : str

name of object to be queried

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**table** : `Table`

Query results table

**query_objectids_async**(*object_name*, *cache=True*, *get_query_payload=False*)

Serves the same function as `query_objectids`, but only collects the response from the Simbad server and returns.

**Parameters**

**object_name** : str

name of object to be queried

**Returns**

**response** : `requests.Response`

Response of the query from the server.

**query_objects**(*object_names*, *wildcard=False*, *verbose=False*, *get_query_payload=False*)

Queries Simbad for the specified list of objects and returns the results as a `Table`. Object names may be specified with wildcards if desired.

**Parameters**

**object_names** : sequence of strs

names of objects to be queried

**wildcard** : boolean, optional

When `True`, the names may have wildcards in them. Defaults to `False`.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**table** : `Table`

Query results table

**query_objects_async**(*object_names*, *wildcard=False*, *cache=True*, *get_query_payload=False*)

Same as `query_objects`, but only collects the response from the Simbad server and returns.

**Parameters**

**object_names** : sequence of strs

names of objects to be queried

**wildcard** : boolean, optional

When `True`, the names may have wildcards in them. Defaults to `False`.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**response** : `requests.Response`

Response of the query from the server

**query_region**(**args*, ***kwargs*)

Queries the service and returns a table object.

Serves the same function as `query_region`, but only collects the response from the Simbad server and returns.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

the identifier or coordinates around which to query.

**radius** : str or `Quantity`, optional

the radius of the region. If missing, set to default value of 2 arcmin.

**equinox** : float, optional

the equinox of the coordinates. If missing set to default 2000.0.

**epoch** : str, optional

the epoch of the input coordinates. Must be specified as [J|B] <epoch>. If missing, set to default J2000.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**table** : A `Table` object.

**query_region_async**(*coordinates*, *radius=<Quantity 2. arcmin>*, *equinox=2000.0*, *epoch='J2000'*, *cache=True*, *get_query_payload=False*)

Serves the same function as `query_region`, but only collects the response from the Simbad server and returns.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

the identifier or coordinates around which to query.

**radius** : str or `Quantity`, optional

the radius of the region. If missing, set to default value of 2 arcmin.

**equinox** : float, optional

the equinox of the coordinates. If missing set to default 2000.0.

**epoch** : str, optional

the epoch of the input coordinates. Must be specified as [J|B] <epoch>. If missing, set to default J2000.

**get_query_payload** : bool, optional

When set to `True` the method returns the HTTP request parameters. Defaults to `False`.

**Returns**

**response** : `requests.Response`

Response of the query from the server.

**remove_votable_fields**(**args*, ***kwargs*)

Removes the specified field names from `SimbadClass._VOTABLE_FIELDS`

**Parameters**

**list of field_names to be removed**

**strip_params: bool**

If true, strip the specified keywords before removing them: e.g., ra(foo) would remove ra(bar) if this is True

**reset_votable_fields**()

resets VOTABLE_FIELDS to defaults

### Conf

**class** astroquery.simbad.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.simbad`.

#### Attributes Summary

| | |
|---|---|
| `row_limit` | Maximum number of rows that will be fetched from the result. |
| `server` | Name of the SIMBAD mirror to use. |
| `timeout` | Time limit for connecting to Simbad server. |

#### Attributes Documentation

**row_limit**

Maximum number of rows that will be fetched from the result.

**server**

Name of the SIMBAD mirror to use.

**timeout**

Time limit for connecting to Simbad server.

CHAPTER 6

# VizieR Queries (astroquery.vizier)

## 6.1 Getting started

This is a python interface for querying the VizieR web service. This supports querying an object as well as querying a region around the target. For region queries, the region dimensions may be specified either for a box or as a radius. Similar to the VizieR web interface, the queries may be further constrained by specifying a choice of catalogs, keywords as well as filters on individual columns before retrieving the results.

### 6.1.1 Table Discover

If you want to search for a set of tables, e.g. based on author name or other keywords, the find_catalogs() tool can be used:

```
>>> from astroquery.vizier import Vizier
>>> catalog_list = Vizier.find_catalogs('Kang W51')
>>> print({k:v.description for k,v in catalog_list.items()})
{'J/ApJS/191/232': 'CO survey of W51 molecular cloud (Bieging+, 2010)',
 'J/ApJ/706/83': 'Embedded YSO candidates in W51 (Kang+, 2009)'}
```

From this result, you could either get any of these as a complete catalog or query them for individual objects or regions.

### 6.1.2 Get a whole catalog

If you know the name of the catalog you wish to retrieve, e.g. from doing a find_catalogs() search as above, you can then grab the complete contents of those catalogs:

```
>>> catalogs = Vizier.get_catalogs(catalog_list.keys())
>>> print(catalogs)
TableList with 3 tables:
   '0:J/ApJ/706/83/ysos' with 22 column(s) and 50 row(s)
```

(continues on next page)

(continued from previous page)

```
   '1:J/ApJS/191/232/table1' with 13 column(s) and 50 row(s)
   '2:J/ApJS/191/232/map' with 2 column(s) and 2 row(s)
```

Note that the row limit is set to 50 by default, so if you want to get a truly complete catalog, you need to change that:

```
>>> Vizier.ROW_LIMIT = -1
>>> catalogs = Vizier.get_catalogs(catalog_list.keys())
>>> print(catalogs)
TableList with 3 tables:
   '0:J/ApJ/706/83/ysos' with 22 column(s) and 737 row(s)
   '1:J/ApJS/191/232/table1' with 13 column(s) and 218 row(s)
   '2:J/ApJS/191/232/map' with 2 column(s) and 2 row(s)
>>> Vizier.ROW_LIMIT = 50
```

### 6.1.3 Query an object

For instance to query Sirius across all catalogs:

```
>>> from astroquery.vizier import Vizier
>>> result = Vizier.query_object("sirius")
>>> print(result)
TableList with 275 tables:
   '0:METAobj' with 5 column(s) and 5 row(s)
   '1:ReadMeObj' with 5 column(s) and 5 row(s)
   '2:I/34/greenw2a' with 16 column(s) and 1 row(s)
   ...
```

All the results are returned as a `TableList` object. This is a container for `Table` objects. It is basically an extension to `OrderedDict` for storing a `Table` against its name.

To access an individual table from the `TableList` object:

```
>>> interesting_table = result['IX/10A/cor_ros']
>>> print(interesting_table)
     _1RXS       Rank        sourceID       RAJ2000  DEJ2000  Sep
                                              deg      deg    arcs
---------------- ---- --------------------- -------- -------- ----
J064509.3-164241    2 1RXH J064509.2-164242 101.2885 -16.7119    2
J064509.3-164241   14 1RXP J0645 8.4-164302 101.2854 -16.7174   24
J064509.3-164241   20 1RXH J064515.7-164402 101.3156 -16.7339  123
```

To do some common processing to all the tables in the returned `TableList` object, do just what you would do for a python dictionary:

```
>>> for table_name in result.keys():
...     table = result[table_name]
...     # table is now an `astropy.table.Table` object
...     # some code to apply on table
```

### 6.1.4 Query a region

To query a region either the coordinates or the object name around which to query should be specified along with the value for the radius (or height/width for a box) of the region. For instance to query a large region around the quasar 3C 273:

```
>>> from astroquery.vizier import Vizier
>>> from astropy.coordinates import Angle
>>> result = Vizier.query_region("3C 273", radius=Angle(0.1, "deg"), catalog='GSC')
```

Note that the radius may also be specified as a string in the format expected by `Angle`. So the above query may also be written as:

```
>>> result = Vizier.query_region("3C 273", radius="0d6m0s", catalog='GSC')
```

Or using angular units and quantities from `astropy.units`:

```
>>> import astropy.units as u
>>> result = Vizier.query_region("3C 273", radius=0.1*u.deg, catalog='GSC')
```

To see the result:

```
>>> print(result)
TableList with 3 tables:
   '0:I/254/out' with 10 column(s) and 17 row(s)
   '1:I/271/out' with 11 column(s) and 50 row(s)
   '2:I/305/out' with 11 column(s) and 50 row(s)
```

As mentioned earlier, the region may also be mentioned by specifying the height and width of a box. If only one of the height or width is mentioned, then the region is treated to be a square having sides equal to the specified dimension.

```
>>> from astroquery.vizier import Vizier
>>> import astropy.units as u
>>> import astropy.coordinates as coord
>>> result = Vizier.query_region(coord.SkyCoord(ra=299.590, dec=35.201,
...                                             unit=(u.deg, u.deg),
...                                             frame='icrs'),
...                         width="30m",
...                         catalog=["NOMAD", "UCAC"])
>>> print(result)
TableList with 3 tables:
   '0:I/297/out' with 19 column(s) and 50 row(s)
   '1:I/289/out' with 13 column(s) and 50 row(s)
   '2:I/322A/out' with 24 column(s) and 50 row(s)
```

One more thing to note in the above example is that the coordinates may be specified by using the appropriate coordinate object from `astropy.coordinates`. Especially for ICRS coordinates, some support also exists for directly passing a properly formatted string as the coordinate. Finally the `catalog` keyword argument may be passed in either `query_object()` or `query_region()` methods. This may be a string (if only a single catalog) or a list of strings otherwise.

### 6.1.5 Specifying keywords, output columns and constraints on columns

To specify keywords on which to search as well as conditions on the output columns, an instance of the `VizierClass` class specifying these must be first created. All further queries may then be performed on this instance rather than on the Vizier class.

```
>>> v = Vizier(columns=['_RAJ2000', '_DEJ2000','B-V', 'Vmag', 'Plx'],
...            column_filters={"Vmag":">10"}, keywords=["optical", "xry"])

WARNING: xry : No such keyword [astroquery.vizier.core]
```

Note that whenever an unknown keyword is specified, a warning is emitted and that keyword is discarded from further consideration. The behavior for searching with these keywords is the same as defined for the web interface (for details see here). Now we call the different query methods on this Vizier instance:

```
>>> result = v.query_object("HD 226868", catalog=["NOMAD", "UCAC"])
>>> print(result)
TableList with 3 tables:
    '0:I/297/out' with 3 column(s) and 50 row(s)
    '1:I/289/out' with 3 column(s) and 18 row(s)
    '2:I/322A/out' with 3 column(s) and 10 row(s)

>>> print(result['I/322A/out'])
 _RAJ2000    _DEJ2000    Vmag
   deg         deg       mag
---------- ---------- ------
299.572419  35.194234 15.986
299.580291  35.176889 13.274
299.582571  35.185225 14.863
299.594172  35.179995 14.690
299.601402  35.198108 14.644
299.617669  35.186999 14.394
299.561498  35.201693 15.687
299.570217  35.225663 14.878
299.601081  35.233338 13.170
299.617995  35.205864 13.946
```

When specifying the columns of the query, sorting of the returned table can be requested by adding + (or - for reverse sorting order) in front of the column name. In the following example, the standard ("*") columns and the calculated distance column ("_r") of the 2MASS catalog (II/246) are queried, 20 arcsec around HD 226868. The result is sorted in increasing distance, as requested with the "+" in front of "_r".

```
>>> v = Vizier(columns=["*", "+_r"], catalog="II/246")
>>> result = v.query_region("HD 226868", radius="20s")
>>> print(result[0])
  _r    RAJ2000     DEJ2000        _2MASS       Jmag   ... Bflg Cflg Xflg Aflg
 arcs     deg         deg                        mag   ...
------ ---------- ---------- ---------------- ------ ... ---- ---- ---- ----
 0.134 299.590280  35.201599 19582166+3512057  6.872 ...  111  000    0    0
10.141 299.587491  35.203217 19582099+3512115 10.285 ...  111  c00    0    0
11.163 299.588599  35.198849 19582126+3511558 13.111 ...  002  00c    0    0
12.289 299.586356  35.200542 19582072+3512019 14.553 ...  111  ccc    0    0
17.688 299.586254  35.197994 19582070+3511527 16.413 ...  100  c00    0    0
```

Note: The special column "*" requests just the default columns of a catalog; "**" would request all the columns.

### 6.1.6 Query with table

A Table can also be used to specify the coordinates in a region query *if* it contains the columns _RAJ2000 and _DEJ2000. The following example starts by looking for AGNs in the Veron & Cety catalog with a Vmag between 10.0 and 11.0. Based on the result of this first query, guide stars with a Kmag brighter than 9.0 are looked for, with a separation between 2 and 30 arcsec. The column _q in the guide table is a 1-based index to the agn table (not the 0-based python convention).

```
>>> agn = Vizier(catalog="VII/258/vv10",
...              columns=['*', '_RAJ2000', '_DEJ2000']).query_constraints(Vmag="10.0..11.0")[0]
>>> print(agn)
```

(continues on next page)

(continued from previous page)

```
_RAJ2000 _DEJ2000  Cl  nR      Name      ...  Sp  n_Vmag  Vmag  B-V  r_z
  deg       deg                          ...                mag   mag
-------- -------- --- --- ------------- ... ---- ------ ----- ----- ----
 10.6846  41.2694   Q              M 31 ...   S2         10.57  1.08 1936
 60.2779 -16.1108   Q     NPM1G-16.0168 ...            R 10.16    --  988
 27.2387   5.9067   A   *      NGC  676 ...   S2         10.50    -- 1034
 40.6696  -0.0131   A          NGC 1068 ...  S1h         10.83  0.87   58
139.7596  26.2697   A          NGC 2824 ...   S?         10.88    -- 2528
147.5921  72.2792   A          NGC 2985 ... S1.9         10.61  0.76 1033
173.1442  53.0678   A          NGC 3718 ...  S3b         10.61  0.74 1033
184.9608  29.6139   A         UGC  7377 ...   S3         10.47  0.99 2500
185.0287  29.2808   A          NGC 4278 ...  S3b         10.87  0.98 1033
186.4537  33.5467   A          NGC 4395 ... S1.8         10.27  0.53 1033
192.7196  41.1194   A          NGC 4736 ...    S         10.85  0.85 1032
208.3612  40.2831   A          NGC 5353 ...   S?       R 10.91    --  368

>>> guide = Vizier(catalog="II/246", column_filters={"Kmag":"<9.0"}).query_region(agn, radius="30s",
↪inner_radius="2s")[0]
>>> guide.pprint()
 _q  RAJ2000    DEJ2000        _2MASS       Jmag  ... Rflg Bflg Cflg Xflg Aflg
       deg        deg                       mag   ...
--- ---------- ---------- ---------------- ------ ... ---- ---- ---- ---- ----
  1  10.686015  41.269630 00424464+4116106  9.399 ...  020  020  0c0    2    0
  1  10.685657  41.269550 00424455+4116103 10.773 ...  200  200  c00    2    0
  1  10.685837  41.270599 00424460+4116141  9.880 ...  020  020  0c0    2    0
  1  10.683263  41.267456 00424398+4116028 12.136 ...  200  100  c00    2    0
  1  10.683465  41.269676 00424403+4116108 11.507 ...  200  100  c00    2    0
  3  27.238636   5.906066 01485727+0554218  8.961 ...  112  111  000    0    0
  4  40.669277  -0.014225 02424062-0000512 11.795 ...  200  100  c00    2    0
  4  40.668802  -0.013064 02424051-0000470 11.849 ...  200  100  c00    2    0
  4  40.669219  -0.012236 02424061-0000440 12.276 ...  200  100  c00    2    0
  4  40.670761  -0.012208 02424098-0000439 12.119 ...  200  100  c00    2    0
  4  40.670177  -0.012830 02424084-0000461 11.381 ...  200  100  c00    2    0
 11 192.721982  41.121040 12505327+4107157 10.822 ...  200  100  c00    2    0
 11 192.721179  41.120201 12505308+4107127  9.306 ...  222  111  000    2    0
```

## 6.2 Reference/API

### 6.2.1 astroquery.vizier Package

#### VizieR Query Tool

**Author**
Julien Woillez (jwoillez@gmail.com)

This package is for querying the VizieR service, primarily hosted at: http://vizier.u-strasbg.fr

Note: If the access to catalogues with VizieR was helpful for your research work, the following acknowledgment would be appreciated:

```
This research has made use of the VizieR catalogue access tool, CDS,
Strasbourg, France.  The original description of the VizieR service was
published in A&AS 143, 23
```

### Classes

| | |
|---|---|
| Conf | Configuration parameters for astroquery.vizier. |

### VizierClass

**class** astroquery.vizier.**VizierClass**(*columns=['*'], column_filters={}, catalog=None, keywords=None, ucd='', timeout=60, vizier_server='vizier.u-strasbg.fr', row_limit=50*)

Bases: astroquery.query.BaseQuery

**Parameters**

**columns** : list

List of strings

**column_filters** : dict

**catalog** : str or None

**keywords** : str or None

**ucd** : string

"Unified Content Description" column descriptions. Specifying these will select only catalogs that have columns matching the column descriptions defined on the Vizier web pages. See http://vizier.u-strasbg.fr/vizier/vizHelp/1.htx#ucd and http://cds.u-strasbg.fr/w/doc/UCD/

#### Attributes Summary

| | |
|---|---|
| catalog | The default catalog to search. |
| column_filters | Filters to run on the individual columns. |
| columns | Columns to include. |
| keywords | The set of keywords to filter the Vizier search |
| ucd | UCD criteria: see http://vizier.u-strasbg.fr/vizier/vizHelp/1.htx#ucd |
| valid_keywords | |

#### Methods Summary

| | |
|---|---|
| find_catalogs(keywords[, include_obsolete, …]) | Search Vizier for catalogs based on a set of keywords, e.g. |
| get_catalogs(*args, **kwargs) | Queries the service and returns a table object. |
| get_catalogs_async(catalog[, verbose, …]) | Query the Vizier service for a specific catalog |
| query_constraints(*args, **kwargs) | Queries the service and returns a table object. |
| query_constraints_async([catalog, …]) | Send a query to Vizier in which you specify constraints with keyword/value pairs. |
| query_object(*args, **kwargs) | Queries the service and returns a table object. |

Continued on next page

Table 3 – continued from previous page

| | |
|---|---|
| query_object_async(object_name[, catalog, . . . ]) | Serves the same purpose as query_object but only returns the HTTP response rather than the parsed result. |
| query_region(*args, **kwargs) | Queries the service and returns a table object. |
| query_region_async(coordinates[, radius, . . . ]) | Serves the same purpose as query_region but only returns the HTTP response rather than the parsed result. |

### Attributes Documentation

**catalog**
The default catalog to search. If left empty, will search all catalogs.

**column_filters**
Filters to run on the individual columns. See the Vizier website for details.

**columns**
Columns to include. The special keyword 'all' will return ALL columns from ALL retrieved tables.

**keywords**
The set of keywords to filter the Vizier search

**ucd**
UCD criteria: see http://vizier.u-strasbg.fr/vizier/vizHelp/1.htx#ucd

#### Examples

```
>>> Vizier.ucd = '(spect.dopplerVeloc*|phys.veloc*)'
```

**valid_keywords**

### Methods Documentation

**find_catalogs**(*keywords*, *include_obsolete=False*, *verbose=False*, *max_catalogs=None*, *return_type='votable'*)
Search Vizier for catalogs based on a set of keywords, e.g. author name

**Parameters**
**keywords** : list or string

List of keywords, or space-separated set of keywords. From Vizier: "names or words of title of catalog. The words are and'ed, i.e. only the catalogues characterized by all the words are selected."

**include_obsolete** : bool, optional

If set to True, catalogs marked obsolete will also be returned.

**max_catalogs** : int or None

The maximum number of catalogs to return. If None, all catalogs will be returned.

**Returns**
**resource_dict** : dict

Dictionary of the "Resource" name and the VOTable resource object. "Resources" are generally publications; one publication may contain many tables.

### Examples

```
>>> from astroquery.vizier import Vizier
>>> catalog_list = Vizier.find_catalogs('Kang W51')
>>> print(catalog_list)
{u'J/ApJ/706/83': <astropy.io.votable.tree.Resource at 0x108d4d490>,
 u'J/ApJS/191/232': <astropy.io.votable.tree.Resource at 0x108d50490>}
>>> print({k:v.description for k,v in catalog_list.items()})
{u'J/ApJ/706/83': u'Embedded YSO candidates in W51 (Kang+, 2009)',
 u'J/ApJS/191/232': u'CO survey of W51 molecular cloud (Bieging+, 2010)'}
```

**get_catalogs**(**args*, ***kwargs*)

Queries the service and returns a table object.

Query the Vizier service for a specific catalog

**Parameters**

**catalog** : str or list, optional

The catalog(s) that will be retrieved

**Returns**

**table** : A `Table` object.

**get_catalogs_async**(*catalog*, *verbose=False*, *return_type='votable'*, *get_query_payload=False*)

Query the Vizier service for a specific catalog

**Parameters**

**catalog** : str or list, optional

The catalog(s) that will be retrieved

**Returns**

**response** : `Response`

Returned if asynchronous method used

**query_constraints**(**args*, ***kwargs*)

Queries the service and returns a table object.

Send a query to Vizier in which you specify constraints with keyword/value pairs.

See the vizier constraints page for details.

**Parameters**

**catalog** : str or list, optional

The catalog(s) which must be searched for this identifier. If not specified, all matching catalogs will be searched.

**kwargs** : dict

Any key/value pairs besides "catalog" will be parsed as additional column filters.

**Returns**

**table** : A `Table` object.

### Examples

```
>>> from astroquery.vizier import Vizier
>>> # note that glon/glat constraints here *must* be floats
>>> result = Vizier.query_constraints(catalog='J/ApJ/723/492/table1',
...                                   GLON='>49.0 & <51.0', GLAT='<0')
>>> result[result.keys()[0]].pprint()
    GRSMC      GLON   GLAT   VLSR  ... RD09 _RA.icrs _DE.icrs
------------- ------ ------ ------ ... ---- -------- --------
G049.49-00.41  49.49  -0.41  56.90 ... RD09   290.95    14.50
G049.39-00.26  49.39  -0.26  50.94 ... RD09   290.77    14.48
G049.44-00.06  49.44  -0.06  62.00 ... RD09   290.61    14.62
G049.04-00.31  49.04  -0.31  66.25 ... RD09   290.64    14.15
G049.74-00.56  49.74  -0.56  67.95 ... RD09   291.21    14.65
G050.39-00.41  50.39  -0.41  41.17 ... RD09   291.39    15.29
G050.24-00.61  50.24  -0.61  41.17 ... RD09   291.50    15.06
G050.94-00.61  50.94  -0.61  40.32 ... RD09   291.85    15.68
G049.99-00.16  49.99  -0.16  46.27 ... RD09   290.97    15.06
G049.44-00.06  49.44  -0.06  46.27 ... RD09   290.61    14.62
G049.54-00.01  49.54  -0.01  56.05 ... RD09   290.61    14.73
G049.74-00.01  49.74  -0.01  48.39 ... RD09   290.71    14.91
G049.54-00.91  49.54  -0.91  43.29 ... RD09   291.43    14.31
G049.04-00.46  49.04  -0.46  58.60 ... RD09   290.78    14.08
G049.09-00.06  49.09  -0.06  46.69 ... RD09   290.44    14.31
G050.84-00.11  50.84  -0.11  50.52 ... RD09   291.34    15.83
G050.89-00.11  50.89  -0.11  59.45 ... RD09   291.37    15.87
G050.44-00.41  50.44  -0.41  64.12 ... RD09   291.42    15.34
G050.84-00.76  50.84  -0.76  61.15 ... RD09   291.94    15.52
G050.29-00.46  50.29  -0.46  14.81 ... RD09   291.39    15.18
```

**query_constraints_async**(*catalog=None*, *return_type='votable'*, *cache=True*, ***kwargs*)

Send a query to Vizier in which you specify constraints with keyword/value pairs.

See the vizier constraints page for details.

**Parameters**

**catalog** : str or list, optional

The catalog(s) which must be searched for this identifier. If not specified, all matching catalogs will be searched.

**kwargs** : dict

Any key/value pairs besides "catalog" will be parsed as additional column filters.

**Returns**

**response** : `requests.Response`

The response of the HTTP request.

### Examples

```
>>> from astroquery.vizier import Vizier
>>> # note that glon/glat constraints here *must* be floats
>>> result = Vizier.query_constraints(catalog='J/ApJ/723/492/table1',
...                                   GLON='>49.0 & <51.0', GLAT='<0')
>>> result[result.keys()[0]].pprint()
    GRSMC      GLON   GLAT   VLSR  ... RD09 _RA.icrs _DE.icrs
------------- ------ ------ ------ ... ---- -------- --------
G049.49-00.41  49.49  -0.41  56.90 ... RD09   290.95    14.50
```

(continues on next page)

(continued from previous page)

```
G049.39-00.26  49.39  -0.26  50.94 ... RD09   290.77    14.48
G049.44-00.06  49.44  -0.06  62.00 ... RD09   290.61    14.62
G049.04-00.31  49.04  -0.31  66.25 ... RD09   290.64    14.15
G049.74-00.56  49.74  -0.56  67.95 ... RD09   291.21    14.65
G050.39-00.41  50.39  -0.41  41.17 ... RD09   291.39    15.29
G050.24-00.61  50.24  -0.61  41.17 ... RD09   291.50    15.06
G050.94-00.61  50.94  -0.61  40.32 ... RD09   291.85    15.68
G049.99-00.16  49.99  -0.16  46.27 ... RD09   290.97    15.06
G049.44-00.06  49.44  -0.06  46.27 ... RD09   290.61    14.62
G049.54-00.01  49.54  -0.01  56.05 ... RD09   290.61    14.73
G049.74-00.01  49.74  -0.01  48.39 ... RD09   290.71    14.91
G049.54-00.91  49.54  -0.91  43.29 ... RD09   291.43    14.31
G049.04-00.46  49.04  -0.46  58.60 ... RD09   290.78    14.08
G049.09-00.06  49.09  -0.06  46.69 ... RD09   290.44    14.31
G050.84-00.11  50.84  -0.11  50.52 ... RD09   291.34    15.83
G050.89-00.11  50.89  -0.11  59.45 ... RD09   291.37    15.87
G050.44-00.41  50.44  -0.41  64.12 ... RD09   291.42    15.34
G050.84-00.76  50.84  -0.76  61.15 ... RD09   291.94    15.52
G050.29-00.46  50.29  -0.46  14.81 ... RD09   291.39    15.18
```

**query_object**(**args*, ***kwargs*)

Queries the service and returns a table object.

Serves the same purpose as `query_object` but only returns the HTTP response rather than the parsed result.

**Parameters**

**object_name** : str

The name of the identifier.

**catalog** : str or list, optional

The catalog(s) which must be searched for this identifier. If not specified, all matching catalogs will be searched.

**radius** : `Quantity` or None

A degree-equivalent radius (optional).

**coordinate_system** : str or None

If the object name is given as a coordinate, you *should* use `query_region`, but you can specify a coordinate frame here instead (today, J2000, B1975, B1950, B1900, B1875, B1855, Galactic, Supergal., Ecl.J2000, )

**Returns**

**table** : A `Table` object.

**query_object_async**(*object_name*, *catalog=None*, *radius=None*, *coordinate_frame=None*, *get_query_payload=False*, *return_type='votable'*, *cache=True*)

Serves the same purpose as `query_object` but only returns the HTTP response rather than the parsed result.

**Parameters**

**object_name** : str

The name of the identifier.

**catalog** : str or list, optional

The catalog(s) which must be searched for this identifier. If not specified, all matching catalogs will be searched.

**radius** : `Quantity` or None

A degree-equivalent radius (optional).

**coordinate_system** : str or None

If the object name is given as a coordinate, you *should* use `query_region`, but you can specify a coordinate frame here instead (today, J2000, B1975, B1950, B1900, B1875, B1855, Galactic, Supergal., Ecl.J2000, )

**Returns**

**response** : `Response`

The response of the HTTP request.

**query_region**(**args*, ***kwargs*)

Queries the service and returns a table object.

Serves the same purpose as `query_region` but only returns the HTTP response rather than the parsed result.

**Parameters**

**coordinates** : str, `astropy.coordinates` object, or `Table`

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as a string. If a table is used, each of its rows will be queried, as long as it contains two columns named `_RAJ2000` and `_DEJ2000` with proper angular units.

**radius** : convertible to `Angle`

The radius of the circular region to query.

**inner_radius** : convertible to `Angle`

When set in addition to `radius`, the queried region becomes annular, with outer radius `radius` and inner radius `inner_radius`.

**width** : convertible to `Angle`

The width of the square region to query.

**height** : convertible to `Angle`

When set in addition to `width`, the queried region becomes rectangular, with the specified `width` and `height`.

**catalog** : str or list, optional

The catalog(s) which must be searched for this identifier. If not specified, all matching catalogs will be searched.

**Returns**

**table** : A `Table` object.

**query_region_async**(*coordinates*, *radius=None*, *inner_radius=None*, *width=None*, *height=None*, *catalog=None*, *get_query_payload=False*, *cache=True*, *return_type='votable'*)

Serves the same purpose as `query_region` but only returns the HTTP response rather than the parsed result.

**Parameters**

**coordinates** : str, `astropy.coordinates` object, or `Table`

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate astropy.coordinates object. ICRS coordinates may also be entered as a string. If a table is used, each of its rows will be queried, as long as it contains two columns named _RAJ2000 and _DEJ2000 with proper angular units.

**radius** : convertible to Angle

The radius of the circular region to query.

**inner_radius** : convertible to Angle

When set in addition to radius, the queried region becomes annular, with outer radius radius and inner radius inner_radius.

**width** : convertible to Angle

The width of the square region to query.

**height** : convertible to Angle

When set in addition to width, the queried region becomes rectangular, with the specified width and height.

**catalog** : str or list, optional

The catalog(s) which must be searched for this identifier. If not specified, all matching catalogs will be searched.

**Returns**

**response** : requests.Response

The response of the HTTP request.

## Conf

**class** astroquery.vizier.**Conf**

Bases: astropy.config.ConfigNamespace

Configuration parameters for astroquery.vizier.

### Attributes Summary

| | |
|---|---|
| row_limit | Maximum number of rows that will be fetched from the result (set to -1 for unlimited). |
| server | Name of the VizieR mirror to use. |
| timeout | Default timeout for connecting to server |

### Attributes Documentation

**row_limit**

Maximum number of rows that will be fetched from the result (set to -1 for unlimited).

**server**

Name of the VizieR mirror to use.

**timeout**

Default timeout for connecting to server

CHAPTER 7

# ESASky Queries (astroquery.esasky)

## 7.1 Getting started

This is a python interface for querying the ESASky web service. This supports querying an object as well as querying a region around the target. For region queries, the region dimensions may be specified as a radius. The queries may be further constrained by specifying a choice of catalogs or missions. Documentation on the ESASky web service can be found here.

### 7.1.1 Get the available catalog names

If you know the names of all the available catalogs you can use `list_catalogs()`:

```
>>> catalog_list = ESASky.list_catalogs()
>>> print(catalog_list)
['INTEGRAL', 'CHANDRA', 'XMM-EPIC', 'XMM-OM', 'XMM-SLEW', 'Tycho-2',
'Gaia DR1 TGAS', 'Gaia DR1', 'Hipparcos-2', 'HSC', 'Herschel-HPPSC-070',
'Herschel-HPPSC-100', 'Herschel-HPPSC-160', 'Herschel-SPSC-250',
'Herschel-SPSC-350', 'Herschel-SPSC-500', 'Planck-PGCC2',
'Planck-PCCS2E', 'Planck-PCCS2-HFI', 'Planck-PCCS2-LFI', 'Planck-PSZ']
```

### 7.1.2 Get the available maps mission names

If you know the names of all the available maps missions you can use `list_maps()`:

```
>>> maps_list = ESASky.list_maps()
>>> print(maps_list)
['INTEGRAL', 'XMM-EPIC', 'CHANDRA', 'SUZAKU', 'XMM-OM-OPTICAL',
 'XMM-OM-UV', 'HST', 'Herschel', 'ISO']
```

### 7.1.3 Query an object

There are two query objects methods in this module `query_object_catalogs()` and `query_object_maps()`. They both work in almost the same way except that one has catalogs as input and output and the other one has mission names and observations as input and output.

For catalogs, the query returns a maximum of 10000 sources per mission by default. However, this can be modified by the row_limit parameter. You can set the parameter to -1, which will result in the maximum number of sources (currently 100 000). To account for observation errors, this method will search for any sources within 5 arcsec from the object.

For instance to query an object around M51 in the integral catalog:

```
>>> from astroquery.esasky import ESASky
>>> result = ESASky.query_object_catalogs("M51", "integral")
```

Note that the catalog may also be specified as a list. So the above query may also be written as:

```
>>> result = ESASky.query_object_catalogs("M51", ["integral", "XMM-OM"])
```

To search in all available catalogs you can write `"all"` instead of a catalog name. The same thing will happen if you don't write any catalog name.

```
>>> result = ESASky.query_object_catalogs("M51", "all")
>>> result = ESASky.query_object_catalogs("M51")
```

To see the result:

```
>>> print(result)
TableList with 3 tables:
    '0:HSC' with 8 column(s) and 135 row(s)
    '1:XMM-EPIC' with 4 column(s) and 2 row(s)
    '2:XMM-OM' with 12 column(s) and 3 row(s)
```

All the results are returned as a `astroquery.utils.TableList` object. This is a container for `Table` objects. It is basically an extension to `collections.OrderedDict` for storing a `Table` against its name.

To access an individual table from the `astroquery.utils.TableList` object

```
>>> interesting_table = result['PLANCK-PCCS2-HFI']
>>> print(interesting_table)
          name              ra [1]        dec [1]
----------------------- ------------- -------------
PCCS2 217 G104.83+68.55 202.485459453 47.2001843799
```

To do some common processing to all the tables in the returned `astroquery.utils.TableList` object, do just what you would do for a python dictionary:

```
>>> for table_name in result:
...     table = result[table_name]
...     # table is now an `astropy.table.Table` object
...     # some code to apply on table
```

As mentioned earlier, `astroquery.esasky.ESASkyClass.query_object_maps()` works extremely similar. It will return all maps that contain the chosen object or coordinate. To execute the same command as above you write this:

```
>>> result = ESASky.query_object_maps("M51", "all")
```

The parameters are interchangeable in the same way as in `query_object_catalogs()`.

### 7.1.4 Query a region

The region queries work in a similar way as query_object, except that you must choose a radius as well. There are two query region methods in this module `astroquery.esasky.ESASkyClass.query_region_catalogs()` and `astroquery.esasky.ESASkyClass.query_region_maps()`. The row_limit parameter can be set to choose the maximum number of row to be selected. If this parameter is not set, the method will return the first 10000 sources. You can set the parameter to -1, which will result in the maximum number of sources (currently 100 000).

To query a region either the coordinates or the object name around which to query should be specified along with the value for the radius of the region. For instance to query region around M51 in the integral catalog:

```
>>> from astroquery.esasky import ESASky
>>> import astropy.units as u
>>> result = ESASky.query_region_catalogs("M51", 10 * u.arcmin, "integral")
```

Note that the catalog may also be specified as a list. So the above query may also be written as:

```
>>> result = ESASky.query_region_catalogs("M51", 10 * u.arcmin, ["integral", "XMM-OM"])
```

To search in all available catalogs you can write `"all"` instead of a catalog name. The same thing will happen if you don't write any catalog name.

```
>>> result = ESASky.query_region_catalogs("M51", 10 * u.arcmin, "all")
>>> result = ESASky.query_region_catalogs("M51", 10 * u.arcmin)
```

In the same manner, the radius can be specified with either a string or any `Quantity`

```
>>> result = ESASKY.query_region_catalogs("M51", "10 arcmin")
```

To see the result:

```
>>> print(result)
TableList with 4 tables:
    '0:XMM-EPIC' with 4 column(s) and 3 row(s)
    '1:HSC' with 8 column(s) and 10000 row(s)
    '2:XMM-OM' with 12 column(s) and 220 row(s)
    '3:PLANCK-PCCS2-HFI' with 8 column(s) and 1 row(s)
```

As mentioned earlier, `query_region_maps()` works extremely similar. To execute the same command as above you write this:

```
>>> result = ESASky.query_region_maps("M51", 10 * u.arcmin, "all")
```

The parameters are interchangeable in the same way as in `query_region_catalogs()`.

### 7.1.5 Get images

You can fetch images around the specified target or coordinates. When a target name is used rather than the coordinates, this will be resolved to coordinates using astropy name resolving methods that utilize online services like SESAME. Coordinates may be entered using the suitable object from `astropy.coordinates`.

The method returns a `dict` to separate the different missions. All mission except Herschel returns a list of `HDUList`. For Herschel each item in the list is a dictionary where the used filter is the key and the HDUList is the value.

```
>>> from astroquery.esasky import ESASky
>>> images = ESASky.get_images("m51", radius="20 arcmin",
...                            missions=['Herschel', 'XMM-EPIC'])

Starting download of HERSCHEL data. (12 files)
Downloading Observation ID: 1342183910 from http://archives.esac.esa.int/hsa/aio/jsp/standaloneproduct.
↪jsp?RETRIEVAL_TYPE=STANDALONE&OBSERVATION.OBSERVATION_ID=1342183910&OBSERVING_MODE.OBSERVING_MODE_
↪NAME=PacsPhoto&INSTRUMENT.INSTRUMENT_NAME=PACS [Done]
Downloading Observation ID: 1342183907 from http://archives.esac.esa.int/hsa/aio/jsp/standaloneproduct.
↪jsp?RETRIEVAL_TYPE=STANDALONE&OBSERVATION.OBSERVATION_ID=1342183907&OBSERVING_MODE.OBSERVING_MODE_
↪NAME=PacsPhoto&INSTRUMENT.INSTRUMENT_NAME=PACS [Done]
...

>>> print(images)
{
'HERSCHEL': [{'70': [HDUList], '160': [HDUList]}, {'70': [HDUList], '160': [HDUList]}, ...],
'XMM-EPIC' : [HDUList], HDUList], HDUList], HDUList], ...]
...
}
```

Note that the fits files also are stored to disk. By default they are saved to the working directory but the location can be chosen by the download_dir parameter:

```
>>> images = ESASky.get_images("m51", radius="20 arcmin",
...                            missions=['Herschel', 'XMM-EPIC'],
...                            download_dir="/home/user/esasky")
```

### 7.1.6 Get maps

You can also fetch images using `astroquery.esasky.ESASkyClass.get_maps()`. It works exactly as `astroquery.esasky.ESASkyClass.get_images()` except that it takes a `TableList` instead of position, radius and missions.

```
>>> table_list = ESASky.query_region_maps("m51", radius="20 arcmin",
...                                       missions=['Herschel', 'XMM-EPIC'])
>>> images = ESASky.get_maps(table_list, download_dir="/home/user/esasky")
```

This example is equivalent to:

```
>>> images = ESASky.get_images("m51", radius="20 arcmin",
...                            missions=['Herschel', 'XMM-EPIC'],
...                            download_dir="/home/user/esasky")
```

## 7.2 Reference/API

### 7.2.1 astroquery.esasky Package

#### Classes

| | |
|---|---|
| `ESASkyClass`() | |
| `Conf` | Configuration parameters for `astroquery.esasky`. |

### ESASkyClass

**class** astroquery.esasky.**ESASkyClass**

Bases: astroquery.query.BaseQuery

#### Attributes Summary

| | |
|---|---|
| DEFAULT_ROW_LIMIT | |
| TIMEOUT | |
| URLbase | |

#### Methods Summary

| | |
|---|---|
| get_images(position[, radius, missions, ...]) | This method gets the fits files available for the selected position and mission and downloads all maps to the the selected folder. |
| get_maps(query_table_list[, missions, ...]) | This method takes the dictionary of missions and metadata as returned by query_region_maps and downloads all maps to the selected folder. |
| list_catalogs() | Get a list of the mission names of the available catalogs in ESASky |
| list_maps() | Get a list of the mission names of the available observations in ESASky |
| query_object_catalogs(position[, catalogs, ...]) | This method queries a chosen object or coordinate for all available catalogs and returns a TableList with all the found catalogs metadata for the chosen missions and object. |
| query_object_maps(position[, missions, ...]) | This method queries a chosen object or coordinate for all available maps which have observation data on the chosen position. |
| query_region_catalogs(position, radius[, ...]) | This method queries a chosen region for all available catalogs and returns a TableList with all the found catalogs metadata for the chosen missions and region. |
| query_region_maps(position, radius[, ...]) | This method queries a chosen region for all available maps and returns a TableList with all the found maps metadata for the chosen missions and region. |

#### Attributes Documentation

**DEFAULT_ROW_LIMIT = 10000**

**TIMEOUT = 1000**

**URLbase = 'http://sky.esa.int/esasky-tap'**

### Methods Documentation

**get_images**(*position*, *radius='0 arcmin'*, *missions='all'*, *download_dir='Maps'*, *cache=True*)

This method gets the fits files available for the selected position and mission and downloads all maps to the the selected folder. The method returns a dictionary which is divided by mission. All mission except Herschel returns a list of HDULists. For Herschel each item in the list is a dictionary where the used filter is the key and the HDUList is the value.

**Parameters**

**position** : str or `astropy.coordinates` object

Can either be a string of the location, eg 'M51', or the coordinates of the object.

**radius** : str or `Quantity`, optional

The radius of a region. Defaults to 0.

**missions** : string or list, optional

Can be either a specific mission or a list of missions (all mission names are found in list_missions()) or 'all' to search in all missions. Defaults to 'all'.

**download_dir** : string, optional

The folder where all downloaded maps should be stored. Defaults to a folder called 'Maps' in the current working directory.

**cache** : bool, optional

When set to True the method will use a cache located at .astropy/astroquery/cache. Defaults to True.

**Returns**

**maps** : `dict`

All mission except Herschel returns a list of HDULists. For Herschel each item in the list is a dictionary where the used filter is the key and the HDUList is the value. It is structured in a dictionary like this: dict: { 'HERSCHEL': [{'70': [HDUList], '160': [HDUList]}, {'70': [HDUList], '160': [HDUList]}, …], 'HST':[[HDUList], [HDUList], [HDUList], [HDUList], [HDUList], …], 'XMM-EPIC' : [[HDUList], [HDUList], [HDUList], [HDUList], …] … }

#### Examples

get_images("m101", "14'", "all")

**get_maps**(*query_table_list*, *missions='all'*, *download_dir='Maps'*, *cache=True*)

This method takes the dictionary of missions and metadata as returned by query_region_maps and downloads all maps to the selected folder. The method returns a dictionary which is divided by mission. All mission except Herschel returns a list of HDULists. For Herschel each item in the list is a dictionary where the used filter is the key and the HDUList is the value.

**Parameters**

**query_table_list** : `TableList`

A TableList with all the missions wanted and their respective metadata. Usually the return value of query_region_maps.

**missions** : string or list, optional

Can be either a specific mission or a list of missions (all mission names are found in list_missions()) or 'all' to search in all missions. Defaults to 'all'.

**download_dir** : string, optional

The folder where all downloaded maps should be stored. Defaults to a folder called 'Maps' in the current working directory.

**cache** : bool, optional

When set to True the method will use a cache located at .astropy/astroquery/cache. Defaults to True.

**Returns**

**maps** : `dict`

All mission except Herschel returns a list of HDULists. For Herschel each item in the list is a dictionary where the used filter is the key and the HDUList is the value. It is structured in a dictionary like this: dict: { 'HERSCHEL': [{'70': [HDUList], '160': [HDUList]}, {'70': [HDUList], '160': [HDUList]}, ...], 'HST':[[HDUList], [HDUList], [HDUList], [HDUList], [HDUList], ...], 'XMM-EPIC' : [[HDUList], [HDUList], [HDUList], [HDUList], ...] ... }

### Examples

get_maps(query_region_catalogs("m101", "14'", "all"))

**list_catalogs**()

Get a list of the mission names of the available catalogs in ESASky

**list_maps**()

Get a list of the mission names of the available observations in ESASky

**query_object_catalogs**(*position*, *catalogs='all'*, *row_limit=10000*, *get_query_payload=False*, *cache=True*)

This method queries a chosen object or coordinate for all available catalogs and returns a TableList with all the found catalogs metadata for the chosen missions and object. To account for errors in telescope position, the method will look for any sources within a radius of 5 arcsec of the chosen position.

**Parameters**

**position** : str or `astropy.coordinates` object

Can either be a string of the location, eg 'M51', or the coordinates of the object.

**catalogs** : string or list, optional

Can be either a specific catalog or a list of catalogs (all catalog names are found in list_catalogs()) or 'all' to search in all catalogs. Defaults to 'all'.

**row_limit** : int, optional

Determines how many rows that will be fetched from the database for each mission. Can be -1 to select maximum (currently 100 000). Defaults to 10000.

**get_query_payload** : bool, optional

When set to True the method returns the HTTP request parameters. Defaults to False.

**cache** : bool, optional

When set to True the method will use a cache located at .astropy/astroquery/cache. Defaults to True.

**Returns**

**——-**

**table_list** : TableList

Each mission returns a Table with the metadata of the catalogs available for the chosen mission and object. It is structured in a TableList like this: TableList with 8 tables: '0:Gaia DR1 TGA' with 8 column(s) and 25 row(s) '1:HSC' with 8 column(s) and 75 row(s)

### Examples

query_object_catalogs("m101", "all")

query_object_catalogs("265.05, 69.0", "Gaia DR1 TGA") query_object_catalogs("265.05, 69.0", ["Gaia DR1 TGA", "HSC"])

**query_object_maps**(*position*, *missions='all'*, *get_query_payload=False*, *cache=True*)

This method queries a chosen object or coordinate for all available maps which have observation data on the chosen position. It returns a TableList with all the found maps metadata for the chosen missions and object.

**Parameters**

**position** : str or astropy.coordinates object

Can either be a string of the location, eg 'M51', or the coordinates of the object.

**missions** : string or list, optional

Can be either a specific mission or a list of missions (all mission names are found in list_missions()) or 'all' to search in all missions. Defaults to 'all'.

**get_query_payload** : bool, optional

When set to True the method returns the HTTP request parameters. Defaults to False.

**cache** : bool, optional

When set to True the method will use a cache located at .astropy/astroquery/cache. Defaults to True.

**Returns**

**table_list** : TableList

Each mission returns a Table with the metadata and observations available for the chosen missions and object. It is structured in a TableList like this: TableList with 8 tables: '0:HERSCHEL' with 8 column(s) and 25 row(s) '1:HST' with 8 column(s) and 735 row(s)

### Examples

query_object_maps("m101", "all")

query_object_maps("265.05, 69.0", "Herschel") query_object_maps("265.05, 69.0", ["Herschel", "HST"])

**query_region_catalogs**(*position*, *radius*, *catalogs='all'*, *row_limit=10000*, *get_query_payload=False*, *cache=True*)

This method queries a chosen region for all available catalogs and returns a TableList with all the found catalogs metadata for the chosen missions and region.

**Parameters**

**position** : str or `astropy.coordinates` object

Can either be a string of the location, eg 'M51', or the coordinates of the object.

**radius** : str or `Quantity`

The radius of a region.

**catalogs** : string or list, optional

Can be either a specific catalog or a list of catalogs (all catalog names are found in list_catalogs()) or 'all' to search in all catalogs. Defaults to 'all'.

**row_limit** : int, optional

Determines how many rows that will be fetched from the database for each mission. Can be -1 to select maximum (currently 100 000). Defaults to 10000.

**get_query_payload** : bool, optional

When set to True the method returns the HTTP request parameters. Defaults to False.

**cache** : bool, optional

When set to True the method will use a cache located at .astropy/astroquery/cache. Defaults to True.

**Returns**

**table_list** : `TableList`

Each mission returns a `Table` with the metadata of the catalogs available for the chosen mission and region. It is structured in a TableList like this: TableList with 8 tables: '0:Gaia DR1 TGA' with 8 column(s) and 25 row(s) '1:HSC' with 8 column(s) and 75 row(s)

### Examples

query_region_catalogs("m101", "14'", "all")

import astropy.units as u query_region_catalogs("265.05, 69.0", 14*u.arcmin, "Gaia DR1 TGA") query_region_catalogs("265.05, 69.0", 14*u.arcmin, ["Gaia DR1 TGA", "HSC"])

**query_region_maps**(*position*, *radius*, *missions='all'*, *get_query_payload=False*, *cache=True*)

This method queries a chosen region for all available maps and returns a TableList with all the found maps metadata for the chosen missions and region.

**Parameters**

**position** : str or `astropy.coordinates` object

Can either be a string of the location, eg 'M51', or the coordinates of the object.

**radius** : str or `Quantity`

The radius of a region.

**missions** : string or list, optional

Can be either a specific mission or a list of missions (all mission names are found in list_missions()) or 'all' to search in all missions. Defaults to 'all'.

**get_query_payload** : bool, optional

When set to True the method returns the HTTP request parameters. Defaults to False.

**cache** : bool, optional

When set to True the method will use a cache located at .astropy/astroquery/cache. Defaults to True.

**Returns**

**table_list** : TableList

Each mission returns a Table with the metadata and observations available for the chosen missions and region. It is structured in a TableList like this: TableList with 8 tables: '0:HERSCHEL' with 8 column(s) and 25 row(s) '1:HST' with 8 column(s) and 735 row(s)

### Examples

query_region_maps("m101", "14'", "all")

import astropy.units as u query_region_maps("265.05, 69.0", 14*u.arcmin, "Herschel") query_region_maps("265.05, 69.0", ["Herschel", "HST"])

## Conf

**class** astroquery.esasky.**Conf**

Bases: astropy.config.ConfigNamespace

Configuration parameters for astroquery.esasky.

### Attributes Summary

| | |
|---|---|
| row_limit | Maximum number of rows returned (set to -1 for unlimited). |
| timeout | Time limit for connecting to template_module server. |
| urlBase | ESASky base URL |

### Attributes Documentation

**row_limit**

Maximum number of rows returned (set to -1 for unlimited).

**timeout**

Time limit for connecting to template_module server.

**urlBase**

ESASky base URL

# CHAPTER 8

## IRSA Dust Extinction Service Queries (astroquery.irsa_dust)

### 8.1 Getting started

This module can be used to query the IRSA Dust Extinction Service.

#### 8.1.1 Fetch images

Retrieve the image cut-outs for the specified object name or coordinates. The images fetched in the FITS format and the result is returned as a list of HDUList objects. For all image queries, the radius may be optionally specified. If missing the radius defaults to 5 degrees. Note that radius may be specified in any appropriate unit, however it must fall in the range of 2 to 37.5 degrees.

```
>>> from astroquery.irsa_dust import IrsaDust
>>> image_list = IrsaDust.get_images("m81")

Downloading http://irsa.ipac.caltech.edu//workspace/TMP_pdTImE_1525/DUST/m81.v0001/p414Dust.fits
|============================================| 331k/331k (100.00%)        15s
Downloading http://irsa.ipac.caltech.edu//workspace/TMP_pdTImE_1525/DUST/m81.v0001/p414i100.fits
|============================================| 331k/331k (100.00%)        13s
Downloading http://irsa.ipac.caltech.edu//workspace/TMP_pdTImE_1525/DUST/m81.v0001/p414temp.fits
|============================================| 331k/331k (100.00%)        05s

>>> image_list

[[<astropy.io.fits.hdu.image.PrimaryHDU at 0x39b8610>],
[<astropy.io.fits.hdu.image.PrimaryHDU at 0x39b8bd0>],
[<astropy.io.fits.hdu.image.PrimaryHDU at 0x39bd8d0>]]
```

Image queries return cutouts for 3 images - E(B-V) reddening, 100 micron intensity, and dust temperature maps. If only the image of a particular type is required, then this may be specified by using the `image_type` keyword argument to the `get_images()` method. It can take on one of the three values `ebv`, `100um` and `temperature`, corresponding to each of the 3 kinds of images:

```
>>> from astroquery.irsa_dust import IrsaDust
>>> import astropy.units as u
>>> image = IrsaDust.get_images("m81", image_type="100um",
...                                 radius=2*u.deg)

Downloading http://irsa.ipac.caltech.edu//workspace/TMP_007Vob_24557/DUST/m81.v0001/p414i100.fits
|===========================================| 149k/149k (100.00%)        02s


>>> image

[[<astropy.io.fits.hdu.image.PrimaryHDU at 0x3a5a650>]]
```

The image types that are available can also be listed out any time:

```
>>> from astroquery.irsa_dust import IrsaDust
>>> IrsaDust.list_image_types()

['ebv', 'temperature', '100um']
```

The target may also be specified via coordinates passed as strings. Examples of acceptable coordinate strings can be found on this IRSA DUST coordinates description page.

```
>>> from astroquery.irsa_dust import IrsaDust
>>> import astropy.coordinates as coord
>>> import astropy.units as u
>>> image_list = IrsaDust.get_images("17h44m34s -27d59m13s", radius=2.0 * u.deg)

Downloading http://irsa.ipac.caltech.edu//workspace/TMP_46IWzq_9460/DUST/17h44m34s_-27d59m13s.v0001/
↪p118Dust.fits
|==============================|  57k/ 57k (100.00%)        00s
Downloading http://irsa.ipac.caltech.edu//workspace/TMP_46IWzq_9460/DUST/17h44m34s_-27d59m13s.v0001/
↪p118i100.fits
|==============================|  57k/ 57k (100.00%)        00s
Downloading http://irsa.ipac.caltech.edu//workspace/TMP_46IWzq_9460/DUST/17h44m34s_-27d59m13s.v0001/
↪p118temp.fits
|==============================|  57k/ 57k (100.00%)        00s
```

A list having the download links for the FITS image may also be fetched, rather than the actual images, via the `get_image_list()` method. This also supports the `image_type` argument, in the same way as described for `get_images()`.

```
>>> from astroquery.irsa_dust import IrsaDust
>>> import astropy.coordinates as coord
>>> import astropy.units as u
>>> coo = coord.SkyCoord(34.5565*u.deg, 54.2321*u.deg, frame='galactic')
>>> image_urls = IrsaDust.get_image_list(coo)
>>> image_urls

['http://irsa.ipac.caltech.edu//workspace/TMP_gB3awn_6492/DUST/34.5565_54.2321_gal.v0001/p292Dust.fits',
'http://irsa.ipac.caltech.edu//workspace/TMP_gB3awn_6492/DUST/34.5565_54.2321_gal.v0001/p292i100.fits',
'http://irsa.ipac.caltech.edu//workspace/TMP_gB3awn_6492/DUST/34.5565_54.2321_gal.v0001/p292temp.fits']
```

### 8.1.2 Fetching the extinction table

This fetches the extinction table as a `Table`. The input parameters are the same as in the queries discussed above, namely the target string and optionally a radius value:

```
>>> from astroquery.irsa_dust import IrsaDust
>>> import astropy.coordinates as coord
>>> import astropy.units as u
>>> # "22h57m57.5s +26d09m00.09s Equatorial B1950"
>>> coo = coord.SkyCoord("22h57m57.5s +26d09m00.09s", frame='fk4')
>>> table = IrsaDust.get_extinction_table(coo)

Downloading http://irsa.ipac.caltech.edu//workspace/TMP_wuevFn_3781/DUST/345.094229457703_26.
↪418650782801027.v0001/extinction.tbl
|=========================================================================================| 4.4k/4.4k␣
↪(100.00%)         0s

>>> print(table)

Filter_name  LamEff A_over_E_B_V_SandF A_SandF A_over_E_B_V_SFD A_SFD
            microns                      mags                    mags
----------- ------- ------------------ ------- ---------------- -----
     CTIO U  0.3734              4.107   0.229            4.968 0.277
     CTIO B  0.4309              3.641   0.203            4.325 0.241
     CTIO V  0.5517              2.682   0.149             3.24 0.181
     CTIO R   0.652              2.119   0.118            2.634 0.147
     CTIO I  0.8007              1.516   0.084            1.962 0.109
   DSS-II g  0.4621              3.381   0.188            3.907 0.218
   DSS-II r  0.6546              2.088   0.116            2.649 0.148
   DSS-II i  0.8111              1.487   0.083            1.893 0.105
     SDSS u  0.3587              4.239   0.236            5.155 0.287
     SDSS g  0.4717              3.303   0.184            3.793 0.211
     SDSS r  0.6165              2.285   0.127            2.751 0.153
     SDSS i  0.7476              1.698   0.095            2.086 0.116
     SDSS z  0.8923              1.263    0.07            1.479 0.082
    UKIRT J   1.248              0.709   0.039            0.902  0.05
    UKIRT H   1.659              0.449   0.025            0.576 0.032
    UKIRT K    2.19              0.302   0.017            0.367  0.02
    2MASS J    1.23              0.723    0.04            0.937 0.052
    2MASS H    1.64               0.46   0.026            0.591 0.033
   2MASS Ks    2.16               0.31   0.017            0.382 0.021
     IRAC-1    3.52              0.178    0.01             0.22 0.012
     IRAC-2    4.46              0.148   0.008            0.183  0.01
     IRAC-3    5.66               0.13   0.007            0.162 0.009
     IRAC-4    7.68              0.122   0.007            0.151 0.008
     WISE-1    3.32              0.189   0.011            0.234 0.013
     WISE-2    4.57              0.146   0.008             0.18  0.01
```

### 8.1.3 Get other query details

This fetches in a `Table` other additional details that may be returned in the query results. For instance additional details in the three sections - `ebv`, `100um` and `temperature` as mentioned earlier and an additional section `location` may be fetched using the `section` keyword argument. If on the other hand, `section` is missing then the complete table with all the four sections will be returned.

```
>>> from astroquery.irsa_dust import IrsaDust
>>> table = IrsaDust.get_query_table('2MASXJ23045666+1219223') # get the whole table
>>> print(table)

    RA      Dec    coord sys regSize ... temp mean temp std temp max temp min
   deg      deg                 deg
--------- --------- ---------- ------- ... --------- -------- -------- --------
346.23608 12.32286 equ J2000     5.0 ...   17.0721   0.0345  17.1189  17.0152

# fetch only one section of the table

>>> table = IrsaDust.get_query_table('2MASXJ23045666+1219223',
...                                  section='ebv')
>>> print(table)

    ext desc     ... ext SFD min
                 ...
---------------- ... -----------
E(B-V) Reddening ...      0.1099
```

## 8.2 Reference/API

### 8.2.1 astroquery.irsa_dust Package

#### IRSA Galactic Dust Reddening and Extinction Query Tool

**Revision History**

Refactored using common API as a part of Google Summer of Code 2013.

**Originally contributed by**
David Shiga (dshiga.dev@gmail.com)

#### Classes

| | |
|---|---|
| IrsaDustClass() | |
| Conf | Configuration parameters for astroquery.irsa_dust. |

#### IrsaDustClass

**class** astroquery.irsa_dust.**IrsaDustClass**
Bases: astroquery.query.BaseQuery

##### Attributes Summary

| | |
|---|---|
| DUST_SERVICE_URL | |

Continued on next page

Table 2 – continued from previous page

| |
|---|
| TIMEOUT |
| image_type_to_section |

### Methods Summary

| | |
|---|---|
| extract_image_urls(raw_xml[, image_type]) | Extracts the image URLs from the query results and returns these as a list. |
| get_extinction_table(coordinate[, radius, ...]) | Query function that fetches the extinction table from the query result. |
| get_extinction_table_async(coordinate[, ...]) | A query function similar to astroquery.irsa_dust.IrsaDustClass.get_extinction_table but returns a file-handler to the remote files rather than downloading it. |
| get_image_list(coordinate[, radius, ...]) | Query function that performs coordinate-based query and returns a list of URLs to the Irsa-Dust images. |
| get_images(coordinate[, radius, image_type, ...]) | A query function that performs a coordinate-based query to acquire Irsa-Dust images. |
| get_images_async(coordinate[, radius, ...]) | A query function similar to astroquery.irsa_dust.IrsaDustClass.get_images but returns file-handlers to the remote files rather than downloading them. |
| get_query_table(coordinate[, radius, ...]) | Create and return an Table representing the query response(s). |
| list_image_types() | Returns a list of image_types available in the Irsa Dust query results |

### Attributes Documentation

**DUST_SERVICE_URL = 'http://irsa.ipac.caltech.edu/cgi-bin/DUST/nph-dust'**

**TIMEOUT = 30**

**image_type_to_section = {'100um': 'e', 'ebv': 'r', 'temperature': 't'}**

### Methods Documentation

**extract_image_urls**(*raw_xml*, *image_type=None*)

Extracts the image URLs from the query results and returns these as a list. If section is missing or 'all' returns all the URLs, otherwise returns URL corresponding to the section specified ('emission', 'reddening', 'temperature').

**Parameters**

**raw_xml** : str

XML response returned by the query as a string

**image_type** : str, optional

When missing returns for all the images. Otherwise returns only for image of the specified type which must be one of 'temperature', 'ebv', '100um'. Defaults to None.

**Returns**

**url_list** : list

list of URLs to images extracted from query results.

**get_extinction_table**(*coordinate*, *radius=None*, *timeout=30*, *show_progress=True*)

Query function that fetches the extinction table from the query result.

**Parameters**

**coordinate** : str

Can be either the name of an object or a coordinate string If a name, must be resolvable by NED, SIMBAD, 2MASS, or SWAS. Examples of acceptable coordinate strings, can be found here.

**radius** : str / `Quantity`, optional

The size of the region to include in the dust query, in radian, degree or hour as per format specified by `Angle` or `Quantity`. Defaults to 5 degrees.

**timeout** : int, optional

Time limit for establishing successful connection with remote server. Defaults to `TIMEOUT`.

**Returns**

**table** : `Table`

**get_extinction_table_async**(*coordinate*, *radius=None*, *timeout=30*, *show_progress=True*)

A query function similar to `astroquery.irsa_dust.IrsaDustClass.get_extinction_table` but returns a file-handler to the remote files rather than downloading it. Useful for asynchronous queries so that the actual download may be performed later.

**Parameters**

**coordinate** : str

Can be either the name of an object or a coordinate string If a name, must be resolvable by NED, SIMBAD, 2MASS, or SWAS. Examples of acceptable coordinate strings, can be found here.

**radius** : str, optional

The size of the region to include in the dust query, in radian, degree or hour as per format specified by `Angle`. Defaults to 5 degrees.

**timeout** : int, optional

Time limit for establishing successful connection with remote server. Defaults to `TIMEOUT`.

**Returns**

**result** : A context manager that yields a file like readable object.

**get_image_list**(*coordinate*, *radius=None*, *image_type=None*, *timeout=30*)

Query function that performs coordinate-based query and returns a list of URLs to the Irsa-Dust images.

**Parameters**

**coordinate** : str

Can be either the name of an object or a coordinate string If a name, must be resolvable by NED, SIMBAD, 2MASS, or SWAS. Examples of acceptable coordinate strings, can be found here.

**radius** : str / `Quantity`, optional

The size of the region to include in the dust query, in radian, degree or hour as per format specified by `Angle` or `Quantity`. Defaults to 5 degrees.

**image_type** : str, optional

When missing returns for all the images. Otherwise returns only for image of the specified type which must be one of `'temperature'`, `'ebv'`, `'100um'`. Defaults to `None`.

**timeout** : int, optional

Time limit for establishing successful connection with remote server. Defaults to `TIMEOUT`.

**get_query_payload** : bool

If `True` then returns the dictionary of query parameters, posted to remote server. Defaults to `False`.

**Returns**

**url_list** : list

A list of URLs to the FITS images corresponding to the queried object.

**get_images**(*coordinate*, *radius=None*, *image_type=None*, *timeout=30*, *get_query_payload=False*, *show_progress=True*)

A query function that performs a coordinate-based query to acquire Irsa-Dust images.

**Parameters**

**coordinate** : str

Can be either the name of an object or a coordinate string If a name, must be resolvable by NED, SIMBAD, 2MASS, or SWAS. Examples of acceptable coordinate strings, can be found here.

**radius** : str / `Quantity`, optional

The size of the region to include in the dust query, in radian, degree or hour as per format specified by `Angle` or `Quantity`. Defaults to 5 degrees.

**image_type** : str, optional

When missing returns for all the images. Otherwise returns only for image of the specified type which must be one of `'temperature'`, `'ebv'`, `'100um'`. Defaults to `None`.

**timeout** : int, optional

Time limit for establishing successful connection with remote server. Defaults to `TIMEOUT`.

**get_query_payload** : bool, optional

If `True` then returns the dictionary of query parameters, posted to remote server. Defaults to `False`.

**Returns**

A list of `HDUList` objects

**get_images_async**(*coordinate*, *radius=None*, *image_type=None*, *timeout=30*, *get_query_payload=False*, *show_progress=True*)

A query function similar to `astroquery.irsa_dust.IrsaDustClass.get_images` but returns file-handlers to the remote files rather than downloading them. Useful for asynchronous queries so that the actual download may be performed later.

**Parameters**

**coordinate** : str

Can be either the name of an object or a coordinate string If a name, must be resolvable by NED, SIMBAD, 2MASS, or SWAS. Examples of acceptable coordinate strings, can be found here.

**radius** : str / `Quantity`, optional

The size of the region to include in the dust query, in radian, degree or hour as per format specified by `Angle` or `Quantity`. Defaults to 5 degrees.

**image_type** : str, optional

When missing returns for all the images. Otherwise returns only for image of the specified type which must be one of `'temperature'`, `'ebv'`, `'100um'`. Defaults to `None`.

**timeout** : int, optional

Time limit for establishing successful connection with remote server. Defaults to `TIMEOUT`.

**get_query_payload** : bool, optional

If `True` then returns the dictionary of query parameters, posted to remote server. Defaults to `False`.

**Returns**

**list** : list

A list of context-managers that yield readable file-like objects.

**get_query_table**(*coordinate*, *radius=None*, *section=None*, *timeout=30*, *url='http://irsa.ipac.caltech.edu/cgi-bin/DUST/nph-dust'*)

Create and return an `Table` representing the query response(s).

When `section` is missing, returns the full table. When a section is specified (`'location'`, `'temperature'`, `'ebv'`, or `'100um'`), only that portion of the table is returned.

**Parameters**

**coordinate** : str

Can be either the name of an object or a coordinate string If a name, must be resolvable by NED, SIMBAD, 2MASS, or SWAS. Examples of acceptable coordinate strings, can be found here.

**radius** : str / `Quantity`, optional

The size of the region to include in the dust query, in radian, degree or hour as per format specified by `Angle` or `Quantity`. Defaults to 5 degrees.

**section** : str, optional

When missing, all the sections of the query result are returned. Otherwise only the specified section (`'ebv'`, `'100um'`, `'temperature'`, `'location'`) is returned. Defaults to `None`.

**timeout** : int, optional

Time limit for establishing successful connection with remote server. Defaults to `TIMEOUT`.

**url** : str, optional

Only provided for debugging. Should generally not be assigned. Defaults to `DUST_SERVICE_URL`.

**Returns**

**table** : `Table`

Table representing the query results, (all or as per specified).

**list_image_types**()

Returns a list of image_types available in the Irsa Dust query results

## Conf

**class** astroquery.irsa_dust.**Conf**

Bases: astropy.config.ConfigNamespace

Configuration parameters for astroquery.irsa_dust.

### Attributes Summary

| | |
|---|---|
| server | Name of the irsa_dust server to use. |
| timeout | Default timeout for connecting to server. |

### Attributes Documentation

**server**

Name of the irsa_dust server to use.

**timeout**

Default timeout for connecting to server.

CHAPTER 9

# NED Queries (astroquery.ned)

## 9.1 Getting Started

This module can be used to query the Ned web service. All queries other than image and spectra queries return results in a `Table`. Image and spectra queries on the other hand return the results as a list of `HDUList` objects. Below are some working examples that illustrate common use cases.

### 9.1.1 Query an object

This may be used to query the object *by name* from the NED service. For instance if you want to query NGC 224

```
>>> from astroquery.ned import Ned
>>> result_table = Ned.query_object("NGC 224")
>>> print(result_table) # an astropy.table.Table

 No. Object Name  RA(deg)   ... Redshift Points Diameter Points Associations
--- ----------- ---------- ... --------------- --------------- ------------
  1 MESSIER 031   10.68479 ...              26               7            2
```

### 9.1.2 Query a region

These queries may be used for querying a region around a named object or coordinates (i.e *near name* and *near position* queries). The radius of the region should be specified in degrees or equivalent units. An easy way to do this is to use an `Quantity` object to specify the radius and units. The radius may also be specified as a string in which case it will be parsed using `Angle`. If no radius is specified, it defaults to 1 arcmin. Another optional parameter is the equinox if coordinates are specified. By default this is J2000.0 but can also be set to B1950.0.

```
>>> from astroquery.ned import Ned
>>> import astropy.units as u
>>> result_table = Ned.query_region("3c 273", radius=0.05 * u.deg)
```

(continues on next page)

(continued from previous page)

```
>>> print(result_table)

No.       Object Name        ... Diameter Points Associations
--- ------------------------ ... --------------- ------------
  1    3C 273:[PWC2011] 3640 ...               0            0
  2    3C 273:[PWC2011] 3592 ...               0            0
  3    3C 273:[PWC2011] 3593 ...               0            0
  4    3C 273:[PWC2011] 3577 ...               0            0
  5 SDSS J122856.35+020325.3 ...               3            0
  6    3C 273:[PWC2011] 3553 ...               0            0
  7    3C 273:[PWC2011] 3544 ...               0            0
  8    3C 273:[PWC2011] 3521 ...               0            0
...                      ... ...             ...          ...
346    3C 273:[PWC2011] 2370 ...               0            0
347 SDSS J122917.00+020436.3 ...               4            0
348    3C 273:[PWC2011] 2338 ...               0            0
349    3C 273:[PWC2011] 2349 ...               0            0
350 SDSS J122917.52+020301.5 ...               4            0
351    3C 273:[PWC2011] 2326 ...               0            0
352 SDSS J122917.72+020356.8 ...               3            0
353 SDSS J122918.38+020323.4 ...               4            0
```

Instead of using the name, the target may also be specified via coordinates. Any of the coordinate systems available in `astropy.coordinates` may be used (ICRS, Galactic, FK4, FK5). Note also the use of the equinox keyword argument:

```
>>> from astroquery.ned import Ned
>>> import astropy.units as u
>>> from astropy import coordinates
>>> co = coordinates.SkyCoord(ra=56.38, dec=38.43,
...                           unit=(u.deg, u.deg), frame='fk4')
>>> result_table = Ned.query_region(co, radius=0.1 * u.deg, equinox='B1950.0')
>>> print(result_table)

No.       Object Name       ... Diameter Points Associations
--- ----------------------- ... --------------- ------------
  1 2MASX J03514350+3841573 ...               2            0
  2 2MASX J03514563+3839573 ...               2            0
  3     NVSS J035158+384747 ...               0            0
  4 2MASX J03521115+3849288 ...               2            0
  5 2MASX J03521844+3840179 ...               2            0
```

### Query in the IAU format

The IAU format for coordinates may also be used for querying purposes. Additional parameters that can be specified for these queries is the reference frame of the coordinates. The reference frame defaults to `Equatorial`. But it can also take the values `Ecliptic`, `Galactic` and `SuperGalactic`. The equinox can also be explicitly chosen (same as in region queries). It defaults to `B1950` but again it may be set to `J2000.0`. Note that Ned report results by searching in a 15 arcmin radius around the specified target.

```
>>> from astroquery.ned import Ned
>>> result_table = Ned.query_region_iau('1234-423', frame='SuperGalactic', equinox='J2000.0')
>>> print(result_table)

    No.       Object Name        RA(deg)    ... Diameter Points Associations
--- ----------------------- ---------- ... --------------- ------------
```

(continues on next page)

(continued from previous page)

```
  1    SUMSS J123651-423554  189.21425 ...               0            0
  2    SUMSS J123658-423457    189.245 ...               0            0
  3    SUMSS J123711-424119  189.29663 ...               0            0
  4 2MASX J12373141-4239342  189.38083 ...               2            0
  5 2MASX J12373567-4239122  189.39908 ...               2            0
```

### Query a reference code for objects

These queries can be used to retrieve all objects that appear in the specified 19 digit reference code. These are similar to the `query_bibobj()` queries.

```
>>> from astroquery.ned import Ned
>>> result_table = Ned.query_refcode('1997A&A...323...31K')
>>> print(result_table)

    No.        Object Name        RA(deg)    ... Diameter Points Associations
--- ----------------------- ---------- ... --------------- ------------
  1                NGC 0262   12.19642 ...               8            0
  2                NGC 0449    19.0302 ...               7            0
  3                NGC 0591   23.38028 ...               7            0
  4               UGC 01214   25.99084 ...               7            0
  5 2MASX J01500266-0725482   27.51124 ...               2            0
  6             MESSIER 077   40.66963 ...               8            0
  7                MRK 0599   41.94759 ...               6            0
  8                MRK 1058   42.46596 ...               4            0
...                     ...        ... ...             ...          ...
 30                NGC 5643  218.16977 ...              18            0
 31            SBS 1439+537   220.1672 ...               2            3
 32                MRK 1388  222.65772 ...               6            0
 33 2MASX J20232535+1131352  305.85577 ...               2            0
 34               UGC 12149  340.28163 ...               8            0
 35                MRK 0522  345.07954 ...               4            0
 36                NGC 7674  351.98635 ...               8            0
```

### Image and Spectra Queries

The image queries return a list of `HDUList` objects for the specified name. For instance:

```
>>> from astroquery.ned import Ned
>>> images = Ned.get_images("m1")

Downloading http://ned.ipac.caltech.edu/dss1B2/Bb/MESSIER_001:I:103aE:dss1.fits.gz
|===========================================|  32k/ 32k (100.00%)        00s
Downloading http://ned.ipac.caltech.edu/img5/1995RXCD3.T...0000C/p083n22a:I:0.1-2.4keV:cop1995.fits.gz
|===========================================|  52k/ 52k (100.00%)        01s
Downloading http://ned.ipac.caltech.edu/img5/1996RXCD6.T...0000C/p083n22a:I:0.1-2.4keV:cps1996.fits.gz
|===========================================|  96k/ 96k (100.00%)        03s
Downloading http://ned.ipac.caltech.edu/img5/1995RXCD3.T...0000C/p084n22a:I:0.1-2.4keV:cop1995.fits.gz
|===========================================|  52k/ 52k (100.00%)        01s
Downloading http://ned.ipac.caltech.edu/img5/1998RXCD8.T...0000C/h083n22a:I:0.1-2.4keV:cps1998.fits.gz
|===========================================|  35k/ 35k (100.00%)        00s

>>> images # may be used to do further processing on individual cutouts
```

(continues on next page)

(continued from previous page)

```
[[<astropy.io.fits.hdu.image.PrimaryHDU at 0x4311890>],
[<astropy.io.fits.hdu.image.PrimaryHDU at 0x432b350>],
[<astropy.io.fits.hdu.image.PrimaryHDU at 0x3e9c5d0>],
[<astropy.io.fits.hdu.image.PrimaryHDU at 0x4339790>],
[<astropy.io.fits.hdu.image.PrimaryHDU at 0x433dd90>]]
```

To get the URLs of the downloadable FITS images:

```
>>> from astroquery.ned import Ned
>>> image_list = Ned.get_image_list("m1")
>>> image_list

['http://ned.ipac.caltech.edu/dss1B2/Bb/MESSIER_001:I:103aE:dss1.fits.gz',
 'http://ned.ipac.caltech.edu/img5/1995RXCD3.T...0000C/p083n22a:I:0.1-2.4keV:cop1995.fits.gz',
 'http://ned.ipac.caltech.edu/img5/1996RXCD6.T...0000C/p083n22a:I:0.1-2.4keV:cps1996.fits.gz',
 'http://ned.ipac.caltech.edu/img5/1995RXCD3.T...0000C/p084n22a:I:0.1-2.4keV:cop1995.fits.gz',
 'http://ned.ipac.caltech.edu/img5/1998RXCD8.T...0000C/h083n22a:I:0.1-2.4keV:cps1998.fits.gz']
```

Spectra can also be fetched in the same way:

```
>>> from astroquery.ned import Ned
>>> spectra = Ned.get_spectra('3c 273')

Downloading http://ned.ipac.caltech.edu/spc1/2009A+A...495.1033B/3C_273:S:B:bcc2009.fits.gz
|===========================================| 7.8k/7.8k (100.00%)        00s
Downloading http://ned.ipac.caltech.edu/spc1/1992ApJS...80..109B/PG_1226+023:S:B_V:bg1992.fits.gz
|===========================================| 5.0k/5.0k (100.00%)        00s
Downloading http://ned.ipac.caltech.edu/spc1/2009A+A...495.1033B/3C_273:S:RI:bcc2009.fits.gz
|===========================================| 9.4k/9.4k (100.00%)        00s

>>> spectra

[[<astropy.io.fits.hdu.image.PrimaryHDU at 0x41b4190>],
[<astropy.io.fits.hdu.image.PrimaryHDU at 0x41b0990>],
[<astropy.io.fits.hdu.image.PrimaryHDU at 0x430a450>]]
```

Similarly the list of URLs for spectra of a particular object may be fetched:

```
>>> from astroquery.ned import Ned
>>> image_list = Ned.get_image_list("3c 273", item='spectra')
>>> image_list

['http://ned.ipac.caltech.edu/spc1/2009A+A...495.1033B/3C_273:S:B:bcc2009.fits.gz',
'http://ned.ipac.caltech.edu/spc1/1992ApJS...80..109B/PG_1226+023:S:B_V:bg1992.fits.gz',
'http://ned.ipac.caltech.edu/spc1/2009A+A...495.1033B/3C_273:S:RI:bcc2009.fits.gz']
```

### 9.1.3 Fetching other data tables for an object

Several other data tables for an object may be fetched via the `get_table()` queries. These take a keyword argument `table`, which may be set to one of `photometry`, `diameters`, `redshifts`, `references` or `object-notes`. For instance the `table=photometry` will fetch all the relevant photometric data for the specified object. We look at a simple example:

```
>>> from astroquery.ned import Ned
>>> result_table = Ned.get_table("3C 273", table='positions')
>>> print(result_table)

 No.       RA              DEC        ... Published Frame  Published Frequence Mode ␣
↪    Qualifiers
--- -------------- -------------- ... --------------- ------------------------- ------------------------
↪-------------------------------------
 0 12h29m06.6997s +02d03m08.598s ...
 1 12h29m06.6997s +02d03m08.598s ...             ICR Multiple line measurement ␣
↪                   From new, raw data
 2  12h29m06.699s  +02d03m08.59s ...             ICR    Broad-band measurement ␣
↪                   From new, raw data
 3   12h29m06.64s   +02d03m09.0s ...             FK4    Broad-band measurement From reprocessed raw␣
↪data; Corrected for containing sources
 4   12h29m06.79s   +02d03m08.0s ...             FK5    Broad-band measurement  From new, raw data;␣
↪Systematic errors in RA and Dec corrected
 5   12h29m06.05s   +02d02m57.1s ...             FK4    Broad-band measurement ␣
↪                   From new, raw data
 6   12h29m05.60s   +02d03m09.0s ...             FK5    Broad-band measurement ␣
↪                   From new, raw data
 7    12h29m04.5s     +02d03m03s ...                    Broad-band measurement ␣
↪                   From new, raw data
 8   12h29m07.55s   +02d03m02.3s ...             FK4    Broad-band measurement ␣
↪            From reprocessed raw data
 9   12h29m06.05s   +02d03m11.3s ...             FK4    Broad-band measurement ␣
↪                   From new, raw data
10    12h29m06.5s     +02d02m53s ...             FK4    Broad-band measurement ␣
↪                   From new, raw data
11    12h29m06.5s     +02d02m52s ...             FK4    Broad-band measurement ␣
↪            From reprocessed raw data
```

## 9.2 Reference/API

### 9.2.1 astroquery.ned Package

#### NED Query Tool

Module containing a series of functions that execute queries to the NASA Extragalactic Database (NED):

**Revision History**

Refactored using common API as a part of Google Summer of Code 2013.

**Originally contributed by**

11. Willett, Jun 2011

**Acknowledgements**

**Based off Adam Ginsburg's Splatalogue search routine:**
http://code.google.com/p/agpy/source/browse/trunk/agpy/query_splatalogue.py

> Service URLs to acquire the VO Tables are taken from Mazzarella et al. (2007) The National Virtual Observatory: Tools and Techniques for Astronomical Research, ASP Conference Series, Vol. 382., p.165

### Classes

| | |
|---|---|
| NedClass() | Class for querying the NED (NASA/IPAC Extragalactic Database) system |
| Conf | Configuration parameters for astroquery.ned. |

### NedClass

**class** astroquery.ned.**NedClass**

Bases: astroquery.query.BaseQuery

Class for querying the NED (NASA/IPAC Extragalactic Database) system

http://ned.ipac.caltech.edu/

#### Attributes Summary

| |
|---|
| ALL_SKY_URL |
| BASE_URL |
| DATA_SEARCH_URL |
| IMG_DATA_URL |
| OBJ_SEARCH_URL |
| PHOTOMETRY_OUT |
| SPECTRA_URL |
| TIMEOUT |

#### Methods Summary

| | |
|---|---|
| extract_image_urls(html_in) | Helper function that uses regexps to extract the image urls from the given HTML. |
| get_image_list(object_name[, item, …]) | Helper function that returns a list of urls from which to download the FITS images. |
| get_images(object_name[, get_query_payload, …]) | Query function to fetch FITS images for a given identifier. |
| get_images_async(object_name[, …]) | Serves the same purpose as get_images but returns file-handlers to the remote files rather than downloading them. |
| get_spectra(object_name[, …]) | Query function to fetch FITS files of spectra for a given identifier. |
| get_spectra_async(object_name[, …]) | Serves the same purpose as get_spectra but returns file-handlers to the remote files rather than downloading them. |

Continued on next page

Table 3 – continued from previous page

| | |
|---|---|
| `get_table`(object_name[, table, ...]) | Fetches the specified data table for the object from NED and returns it as an `astropy.table.Table`. |
| `get_table_async`(object_name[, table, ...]) | Serves the same purpose as `query_region` but returns the raw HTTP response rather than the `astropy.table.Table` object. |
| `query_object`(object_name[, ...]) | Queries objects by name from the NED Service and returns the Main Source Table. |
| `query_object_async`(object_name[, ...]) | Serves the same purpose as `query_object` but returns the raw HTTP response rather than the `astropy.table.Table` object. |
| `query_refcode`(refcode[, get_query_payload, ...]) | Used to retrieve all objects contained in a particular reference. |
| `query_refcode_async`(refcode[, get_query_payload]) | Serves the same purpose as `query_region` but returns the raw HTTP response rather than the `astropy.table.Table` object. |
| `query_region`(coordinates[, radius, equinox, ...]) | Used to query a region around a known identifier or given coordinates. |
| `query_region_async`(coordinates[, radius, ...]) | Serves the same purpose as `query_region` but returns the raw HTTP response rather than the `astropy.table.Table` object. |
| `query_region_iau`(iau_name[, frame, equinox, ...]) | Used to query the Ned service via the IAU name. |
| `query_region_iau_async`(iau_name[, frame, ...]) | Serves the same purpose as `query_region_iau` but returns the raw HTTP response rather than the `astropy.table.Table` object. |

### Attributes Documentation

**ALL_SKY_URL = 'http://ned.ipac.caltech.edu/cgi-bin/nph-allsky'**

**BASE_URL = 'http://ned.ipac.caltech.edu/cgi-bin/'**

**DATA_SEARCH_URL = 'http://ned.ipac.caltech.edu/cgi-bin/nph-datasearch'**

**IMG_DATA_URL = 'http://ned.ipac.caltech.edu/cgi-bin/imgdata'**

**OBJ_SEARCH_URL = 'http://ned.ipac.caltech.edu/cgi-bin/nph-objsearch'**

**PHOTOMETRY_OUT = {1: Options(display_name='Data as Published and Homogenized (mJy)', cgi_name='bot'), 2:**

**SPECTRA_URL = 'http://ned.ipac.caltech.edu/cgi-bin/NEDspectra'**

**TIMEOUT = 60**

### Methods Documentation

**extract_image_urls**(*html_in*)

Helper function that uses regexps to extract the image urls from the given HTML.

**Parameters**

**html_in** : str

source from which the urls are to be extracted

**get_image_list**(*object_name*, *item='image'*, *get_query_payload=False*)

Helper function that returns a list of urls from which to download the FITS images.

**Parameters**

**object_name** : str

name of the identifier to query.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**item** : str, optional

Can be either 'image' or 'spectra'. Defaults to 'image'. Required to decide the right URL to query.

**Returns**

list of image urls

**get_images**(*object_name*, *get_query_payload=False*, *show_progress=True*)

Query function to fetch FITS images for a given identifier.

**Parameters**

**object_name** : str

name of the identifier to query.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**Returns**

A list of `HDUList` objects

**get_images_async**(*object_name*, *get_query_payload=False*, *show_progress=True*)

Serves the same purpose as `get_images` but returns file-handlers to the remote files rather than downloading them.

**Parameters**

**object_name** : str

name of the identifier to query.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**Returns**

A list of context-managers that yield readable file-like objects

**get_spectra**(*object_name*, *get_query_payload=False*, *show_progress=True*)

Query function to fetch FITS files of spectra for a given identifier.

**Parameters**

**object_name** : str

name of the identifier to query.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**Returns**
A list of `HDUList` objects

**get_spectra_async**(*object_name*, *get_query_payload=False*, *show_progress=True*)
Serves the same purpose as `get_spectra` but returns file-handlers to the remote files rather than downloading them.

**Parameters**
**object_name** : str

name of the identifier to query.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**Returns**
A list of context-managers that yield readable file-like objects

**get_table**(*object_name*, *table='photometry'*, *get_query_payload=False*, *verbose=False*, ***kwargs*)
Fetches the specified data table for the object from NED and returns it as an `astropy.table.Table`.

**Parameters**
**object_name** : str

name of the identifier to query.

**table** : str, optional

Must be one of ['photometry'|'positions'|'diameters'|'redshifts'|'references'|'object_notes']. Specifies the type of data-table that must be fetched for the given object. Defaults to 'photometry'.

**output_table_format** : int, [optional for photometry]

specifies the format of the output table. Must be 1, 2 or 3. Defaults to 1. These options stand for: (1) Data as Published and Homogenized (mJy) (2) Data as Published (3) Homogenized Units (mJy)

**from_year** : int, [optional for references]

4 digit year from which to get the references. Defaults to 1800

**to_year** : int, [optional for references]

4 digit year upto which to fetch the references. Defaults to the current year.

**extended_search** : bool, [optional for references]

If set to `True`, returns all objects beginning with the same identifier name. Defaults to `False`.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**verbose** : bool, optional.

When set to `True` displays warnings if the returned VOTable does not conform to the standard. Defaults to `False`.

**Returns**
**result** : `astropy.table.Table`

The result of the query as an `astropy.table.Table` object.

**get_table_async**(*object_name*, *table='photometry'*, *get_query_payload=False*, ***kwargs*)
Serves the same purpose as `query_region` but returns the raw HTTP response rather than the `astropy.table.Table` object.

**Parameters**

**object_name** : str

name of the identifier to query.

**table** : str, optional

Must be one of ['photometry'|'positions'|'diameters'|'redshifts'|'references'|'object_notes']. Specifies the type of data-table that must be fetched for the given object. Defaults to 'photometry'.

**from_year** : int, [optional for references]

4 digit year from which to get the references. Defaults to 1800

**to_year** : int, [optional for references]

4 digit year upto which to fetch the references. Defaults to the current year.

**extended_search** : bool, [optional for references]

If set to `True`, returns all objects beginning with the same identifier name. Defaults to `False`.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service.

**query_object**(*object_name*, *get_query_payload=False*, *verbose=False*)
Queries objects by name from the NED Service and returns the Main Source Table.

**Parameters**

**object_name** : str

name of the identifier to query.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**verbose** : bool, optional.

When set to `True` displays warnings if the returned VOTable does not conform to the standard. Defaults to `False`.

**Returns**

**result** : `astropy.table.Table`

The result of the query as an `astropy.table.Table` object.

**query_object_async**(*object_name*, *get_query_payload=False*)
Serves the same purpose as `query_object` but returns the raw HTTP response rather than the `astropy.table.Table` object.

**Parameters**

**object_name** : str

name of the identifier to query.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service

**query_refcode**(*refcode*, *get_query_payload=False*, *verbose=False*)

Used to retrieve all objects contained in a particular reference. Equivalent to by refcode queries of the web interface.

**Parameters**

**refcode** : str

19 digit reference code. Example: 1997A&A…323…31K.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**verbose** : bool, optional.

When set to `True` displays warnings if the returned VOTable does not conform to the standard. Defaults to `False`.

**Returns**

**result** : `astropy.table.Table`

The result of the query as an `astropy.table.Table` object.

**query_refcode_async**(*refcode*, *get_query_payload=False*)

Serves the same purpose as `query_region` but returns the raw HTTP response rather than the `astropy.table.Table` object.

**Parameters**

**refcode** : str

19 digit reference code. Example: 1997A&A…323…31K.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service.

**query_region**(*coordinates*, *radius=<Quantity 1. arcmin>*, *equinox='J2000.0'*, *get_query_payload=False*, *verbose=False*)

Used to query a region around a known identifier or given coordinates. Equivalent to the near position and near name queries from the Ned web interface.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 1 arcmin.

**equinox** : str, optional

The equinox may be either J2000.0 or B1950.0. Defaults to J2000.0

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**verbose** : bool, optional.

When set to `True` displays warnings if the returned VOTable does not conform to the standard. Defaults to `False`.

**Returns**

**result** : `astropy.table.Table`

The result of the query as an `astropy.table.Table` object.

**query_region_async**(*coordinates*, *radius=<Quantity 1. arcmin>*, *equinox='J2000.0'*, *get_query_payload=False*)

Serves the same purpose as `query_region` but returns the raw HTTP response rather than the `astropy.table.Table` object.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 1 arcmin.

**equinox** : str, optional

The equinox may be either J2000.0 or B1950.0. Defaults to J2000.0

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service

**query_region_iau**(*iau_name*, *frame='Equatorial'*, *equinox='B1950.0'*, *get_query_payload=False*, *verbose=False*)

Used to query the Ned service via the IAU name. Equivalent to the IAU format queries of the Web interface.

**Parameters**

**iau_name** : str

IAU coordinate-based name of target on which search is centered. Definition of IAU coordinates at http://cdsweb.u-strasbg.fr/Dic/iau-spec.html.

**frame** : str, optional

May be one of 'Equatorial', 'Ecliptic', 'Galactic', 'SuperGalactic'. Defaults to 'Equatorial'.

**equinox** : str, optional

The equinox may be one of J2000.0 or B1950.0. Defaults to B1950.0

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**verbose** : bool, optional.

When set to `True` displays warnings if the returned VOTable does not conform to the standard. Defaults to `False`.

**Returns**

**result** : `astropy.table.Table`

The result of the query as an `astropy.table.Table` object.

**query_region_iau_async**(*iau_name*, *frame='Equatorial'*, *equinox='B1950.0'*, *get_query_payload=False*)

Serves the same purpose as `query_region_iau` but returns the raw HTTP response rather than the `astropy.table.Table` object.

**Parameters**

**iau_name** : str

IAU coordinate-based name of target on which search is centered. Definition of IAU coordinates at http://cdsweb.u-strasbg.fr/Dic/iau-spec.html.

**frame** : str, optional

May be one of 'Equatorial', 'Ecliptic', 'Galactic', 'SuperGalactic'. Defaults to 'Equatorial'.

**equinox** : str, optional

The equinox may be one of J2000.0 or B1950.0. Defaults to B1950.0

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service.

## Conf

**class** astroquery.ned.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.ned`.

### Attributes Summary

| | |
|---|---|
| correct_redshift | The correct redshift for NED queries, see comments above. |
| hubble_constant | The correct redshift for NED queries may be chosen by specifying numbers 1, 2, 3 and 4, having the following meanings: (1) To the Reference Frame defined by the 3K CMB (2) To the Reference Frame defined by the Virgo Infall only (3) To the Reference Frame defined by the (Virgo + GA) only (4) To the Reference Frame defined by the (Virgo + GA + Shapley) |
| output_coordinate_frame | Frame in which to display the coordinates in the output. |
| output_equinox | Equinox for the output coordinates. |
| server | Name of the NED server to use. |
| sort_output_by | Display output sorted by this criteria. |
| timeout | Time limit for connecting to NED server. |

### Attributes Documentation

**correct_redshift**
The correct redshift for NED queries, see comments above.

**hubble_constant**
The correct redshift for NED queries may be chosen by specifying numbers 1, 2, 3 and 4, having the following meanings: (1) To the Reference Frame defined by the 3K CMB (2) To the Reference Frame defined by the Virgo Infall only (3) To the Reference Frame defined by the (Virgo + GA) only (4) To the Reference Frame defined by the (Virgo + GA + Shapley)

**output_coordinate_frame**
Frame in which to display the coordinates in the output.

**output_equinox**
Equinox for the output coordinates.

**server**
Name of the NED server to use.

**sort_output_by**
Display output sorted by this criteria.

**timeout**
Time limit for connecting to NED server.

CHAPTER 10

# Splatalogue Queries (astroquery.splatalogue)

## 10.1 Getting Started

This module provides an interface to the Splatalogue web service It returns tables of spectral lines with features that you can specify by the same means generally available on the Splatalogue website.

## 10.2 Examples

An example ipynb from an interactive tutorial session at NRAO in April 2014

### 10.2.1 Searching for Lines

In the Splatalogue web interface, you select "species" of interest using the left side menu seen in the query interface You can access the line list:

```
>>> from astroquery.splatalogue import Splatalogue
>>> line_ids = Splatalogue.get_species_ids()
```

This will return the complete Splatalogue chemical species list, including all isotopologues, etc. To search within this list for a particular species, you can use regular expressions:

```
>>> CO_containing_species = Splatalogue.get_species_ids('CO')
>>> len(CO_containing_species)
91
>>> just_CO = Splatalogue.get_species_ids(' CO ') # note the spaces
>>> len(just_CO)
4
>>> just_CO  # includes different vibrationally excited states
{u'02812 CO v = 0 - Carbon Monoxide': u'204',
 u'02813 CO v = 1 - Carbon Monoxide': u'990',
```

(continues on next page)

(continued from previous page)

```
 u'02814 CO v = 2 - Carbon Monoxide': u'991',
 u'02815 CO v = 3 - Carbon Monoxide': u'1343'}
>>> carbon_monoxide = Splatalogue.get_species_ids('Carbon Monoxide')
>>> len(carbon_monoxide) # includes isotopologues
13
>>> carbon_monoxide
>>>
{u'02812 CO v = 0 - Carbon Monoxide': u'204',
 u'02813 CO v = 1 - Carbon Monoxide': u'990',
 u'02814 CO v = 2 - Carbon Monoxide': u'991',
 u'02815 CO v = 3 - Carbon Monoxide': u'1343',
 u'02816 CO+ - Carbon Monoxide Ion': u'709',
 u'02910 13CO v = 0 - Carbon Monoxide': u'4',
 u'02911 13CO v = 1 - Carbon Monoxide': u'992',
 u'02912 13CO v = 2 - Carbon Monoxide': u'993',
 u'02913 C17O - Carbon Monoxide': u'226',
 u'03004 14CO - Carbon Monoxide': u'778',
 u'03005 C18O - Carbon Monoxide': u'245',
 u'03006 13C17O - Carbon Monoxide': u'264',
 u'03101 13C18O - Carbon Monoxide': u'14'}
>>> atomic_weight_88 = Splatalogue.get_species_ids('^088')
>>> atomic_weight_88
{u'08801 SiC5 - ': u'265',
 u'08802 CH3C6H - Methyltriacetylene': u'388',
 u'08803 C6O - Hexacarbon monoxide': u'585'}
```

The returned items are dictionaries, but they are also searchable.

```
>>> carbon_monoxide.find(' 13')  # note leading space
{u'02910 13CO v = 0 - Carbon Monoxide': u'4',
 u'02911 13CO v = 1 - Carbon Monoxide': u'992',
 u'02912 13CO v = 2 - Carbon Monoxide': u'993',
 u'03006 13C17O - Carbon Monoxide': u'264',
 u'03101 13C18O - Carbon Monoxide': u'14'}
```

### 10.2.2 Querying Splatalogue: Getting Line Information

Unlike most astroquery tools, the Splatalogue tool closely resembles the online interface. In principle, we can make a higher level wrapper, but it is not obvious what other parameters one might want to query on (whereas with catalogs, you almost always need a sky-position based query tool).

Any feature you can change on the Splatalogue web form can be modified in the `query_lines()` tool.

For any Splatalogue query, you *must* specify a minimum/maximum frequency. However, you can do it with astropy units, so wavelengths are OK too.

```
>>> from astropy import units as u
>>> CO1to0 = Splatalogue.query_lines(115.271*u.GHz,115.273*u.GHz)
>>> CO1to0.pprint()
      Species         Chemical Name   Freq-GHz ... E<sub>U</sub> (K) Linelist
------------------- --------------- ---------- ... ----------------- --------
              COv=0 Carbon Monoxide         -- ...           5.53211     CDMS
              COv=0 Carbon Monoxide         -- ...           5.53211      JPL
              COv=0 Carbon Monoxide   115.2712 ...               0.0    Lovas
              COv=0 Carbon Monoxide   115.2712 ...           5.53211    SLAIM
```

(continues on next page)

(continued from previous page)

```
        CH3CHOvt=1    Acetaldehyde 115.27182 ...         223.65667    SLAIM
        CH3CHOvt=1    Acetaldehyde        -- ...         223.65581      JPL
CH3O13CHO(TopModel)  Methyl Formate  115.2728 ...         272.75041 TopModel
```

Querying just by frequency isn't particularly effective; a nicer approach is to use both frequency and chemical name. If you can remember that CO 2-1 is approximately in the 1 mm band, but you don't know its exact frequency (after all, why else would you be using splatalogue?), this query works:

```
>>> CO2to1 = Splatalogue.query_lines(1*u.mm, 2*u.mm, chemical_name=" CO ")
>>> CO2to1.pprint()
Species  Chemical Name   Freq-GHz ... E<sub>U</sub> (K) Linelist
------- --------------- --------- ... ----------------- --------
  COv=1 Carbon Monoxide        -- ...        3100.11628     CDMS
  COv=1 Carbon Monoxide 228.43911 ...        3100.11758    SLAIM
  COv=0 Carbon Monoxide        -- ...          16.59608     CDMS
  COv=0 Carbon Monoxide        -- ...          16.59608      JPL
  COv=0 Carbon Monoxide   230.538 ...               0.0    Lovas
  COv=0 Carbon Monoxide   230.538 ...          16.59608    SLAIM
```

Of course, there's some noise in there: both the vibrationally excited line and a whole lot of different line lists. Start by thinning out the line lists used:

```
>>> CO2to1 = Splatalogue.query_lines(1*u.mm, 2*u.mm, chemical_name=" CO ",only_NRAO_recommended=True)
>>> CO2to1.pprint()
Species  Chemical Name   Freq-GHz ... E<sub>U</sub> (K) Linelist
------- --------------- --------- ... ----------------- --------
  COv=1 Carbon Monoxide 228.43911 ...        3100.11758    SLAIM
  COv=0 Carbon Monoxide   230.538 ...          16.59608    SLAIM
```

Then get rid of the vibrationally excited line by setting an energy upper limit in Kelvin:

```
>>> CO2to1 = Splatalogue.query_lines(1*u.mm, 2*u.mm, chemical_name=" CO ",
...                                  only_NRAO_recommended=True,
...                                  energy_max=50, energy_type='eu_k')
>>> CO2to1.pprint()
Species  Chemical Name  Freq-GHz ... E<sub>U</sub> (K) Linelist
------- --------------- -------- ... ----------------- --------
  COv=0 Carbon Monoxide  230.538 ...          16.59608    SLAIM
```

### 10.2.3 A note on recombination lines

Radio recombination lines are included in the splatalogue catalog under the names "Hydrogen Recombination Line", "Helium Recombination Line", and "Carbon Recombination Line". If you want to search specifically for the alpha, beta, delta, gamma, epsilon, or zeta lines, you need to use the unicode character for these symbols (H$\alpha$, H$\beta$, H$\gamma$, H$\delta$, H$\epsilon$, H$\zeta$), even though they will show up as `α` in the ASCII table. For example:

```
>>> Splatalogue.query_lines(84*u.GHz, 115*u.GHz, chemical_name='Hα')
<Table masked=True length=4>
Species         Chemical Name          Freq-GHz Freq Err Meas Freq-GHz Meas Freq Err ... Lovas/AST␣
↪Intensity E_L (cm^-1) E_L (K) E_U (cm^-1) E_U (K) Linelist
  str8              str27              float64   int64       int64         int64     ...       int64   ␣
↪      float64    float64    float64    float64   str6
-------- --------------------------- --------- -------- ------------- ------------- ... ----------------
↪--- ----------- ------- ----------- ------- --------
```

(continues on next page)

(continued from previous page)

```
H&alpha; Hydrogen Recombination Line  85.68839        0            --            -- ...                 ␣
↪ --          0.0      0.0          0.0      0.0    Recomb
H&alpha; Hydrogen Recombination Line  92.03443        0            --            -- ...                 ␣
↪ --          0.0      0.0          0.0      0.0    Recomb
H&alpha; Hydrogen Recombination Line  99.02295        0            --            -- ...                 ␣
↪ --          0.0      0.0          0.0      0.0    Recomb
H&alpha; Hydrogen Recombination Line 106.73736        0            --            -- ...                 ␣
↪ --          0.0      0.0          0.0      0.0    Recomb
```

You could also search by specifying the line list

```
>>> Splatalogue.query_lines(84*u.GHz, 85*u.GHz, line_lists=['Recomb'])
<Table masked=True length=3>
 Species         Chemical Name        Freq-GHz Freq Err Meas Freq-GHz Meas Freq Err ... Lovas/AST␣
↪Intensity E_L (cm^-1) E_L (K) E_U (cm^-1) E_U (K) Linelist
   str9              str27             float64    int64       int64         int64     ...         int64    ␣
↪      float64    float64    float64    float64    str6
--------- -------------------------- -------- -------- ------------- ------------- ... ----------------
↪--- ----------- ------- ----------- ------- --------
 H&gamma; Hydrogen Recombination Line 84.91439        0            --            -- ...                 ␣
↪ --          0.0      0.0          0.0      0.0    Recomb
He&gamma;   Helium Recombination Line   84.949        0            --            -- ...                 ␣
↪ --          0.0      0.0          0.0      0.0    Recomb
 C&gamma;   Carbon Recombination Line 84.95676        0            --            -- ...                 ␣
↪ --          0.0      0.0          0.0      0.0    Recomb
```

### 10.2.4 Cleaning Up the Returned Data

Depending on what sub-field you work in, you may be interested in fine-tuning splatalogue queries to return only a subset of the columns and lines on a regular basis. For example, if you want data returned preferentially in units of K rather than inverse cm, you're interested in low-energy lines, and you want your data sorted by energy, you can use an approach like this:

```
>>> S = Splatalogue(energy_max=500,
...    energy_type='eu_k',energy_levels=['el4'],
...    line_strengths=['ls4'],
...    only_NRAO_recommended=True,noHFS=True)
>>> def trimmed_query(*args,**kwargs):
...     columns = ('Species','Chemical Name','Resolved QNs','Freq-GHz',
...                'Meas Freq-GHz','Log<sub>10</sub> (A<sub>ij</sub>)',
...                'E_U (K)')
...     table = S.query_lines(*args, **kwargs)[columns]
...     table.rename_column('Log<sub>10</sub> (A<sub>ij</sub>)','log10(Aij)')
...     table.rename_column('E_U (K)','EU_K')
...     table.rename_column('Resolved QNs','QNs')
...     table.sort('EU_K')
...     return table
>>> trimmed_query(1*u.GHz,30*u.GHz,
...     chemical_name='(H2.*Formaldehyde)|( HDCO )',
...     energy_max=50).pprint()
Species Chemical Name      QNs      Freq-GHz Meas Freq-GHz log10(Aij)   EU_K
------- ------------- ------------- -------- ------------- ---------- --------
   HDCO  Formaldehyde 1(1,0)-1(1,1)       --       5.34614   -8.31616 11.18287
 H2C18O  Formaldehyde 1(1,0)-1(1,1)   4.3888        4.3888   -8.22052 15.30187
 H213CO  Formaldehyde 1(1,0)-1(1,1)       --       4.59309   -8.51332 15.34693
```

(continues on next page)

(continued from previous page)

```
  H2CO  Formaldehyde 1(1,0)-1(1,1)  4.82966         --   -8.44801 15.39497
  HDCO  Formaldehyde 2(1,1)-2(1,2)       --     16.03787   -7.36194 17.62746
H2C18O  Formaldehyde 2(1,1)-2(1,2) 13.16596     13.16596   -6.86839 22.17455
H213CO  Formaldehyde 2(1,1)-2(1,2)       --      13.7788   -7.55919 22.38424
  H2CO  Formaldehyde 2(1,1)-2(1,2) 14.48848         --   -7.49383 22.61771
H2C18O  Formaldehyde 3(1,2)-3(1,3) 26.33012     26.33014   -6.03008 32.48204
H213CO  Formaldehyde 3(1,2)-3(1,3)       --     27.55567   -6.95712  32.9381
  H2CO  Formaldehyde 3(1,2)-3(1,3)       --      28.9748   -6.89179 33.44949
```

## 10.3 Reference/API

### 10.3.1 astroquery.splatalogue Package

#### Splatalogue Catalog Query Tool

**Author**
Adam Ginsburg (adam.g.ginsburg@gmail.com)

**Originally contributed by**
Magnus Vilhelm Persson (magnusp@vilhelm.nu)

#### Classes

| | |
|---|---|
| SplatalogueClass(**kwargs) | Initialize a Splatalogue query class with default arguments set. |
| Conf | Configuration parameters for astroquery.splatalogue. |

#### SplatalogueClass

**class** astroquery.splatalogue.**SplatalogueClass**(***kwargs*)

Bases: astroquery.query.BaseQuery

Initialize a Splatalogue query class with default arguments set. Frequency specification is required for *every* query, but any default keyword arguments (see query_lines) can be overridden here.

##### Attributes Summary

| |
|---|
| ALL_LINE_LISTS |
| FREQUENCY_BANDS |
| LINES_LIMIT |
| QUERY_URL |
| SLAP_URL |
| TIMEOUT |
| TOP20_LIST |
| versions |

### Methods Summary

| | |
|---|---|
| get_fixed_table([columns]) | Convenience function to get the table with html column names made human readable. |
| get_species_ids([restr, reflags]) | Get a dictionary of "species" IDs, where species refers to the molecule name, mass, and chemical composition. |
| query_lines(*args, **kwargs) | Queries the service and returns a table object. |
| query_lines_async([min_frequency, ...]) | The Splatalogue service returns lines with rest frequencies in the range [min_frequency, max_frequency]. |
| set_default_options(**kwargs) | Modify the default options. |

### Attributes Documentation

**ALL_LINE_LISTS = ('Lovas', 'SLAIM', 'JPL', 'CDMS', 'ToyoMA', 'OSU', 'Recomb', 'Lisa', 'RFI')**

**FREQUENCY_BANDS = {'alma10': 'ALMA Band 10 (787-950 GHz)', 'alma3': 'ALMA Band 3 (84-116 GHz)', 'alma4':**

**LINES_LIMIT = 1000**

**QUERY_URL = 'http://www.cv.nrao.edu/php/splat/c_export.php'**

**SLAP_URL = 'http://find.nrao.edu/splata-slap/slap'**

**TIMEOUT = 60**

**TOP20_LIST = ('comet', 'planet', 'top20', 'ism_hotcore', 'ism_darkcloud', 'ism_diffusecloud')**

**versions = ('v1.0', 'v2.0', 'v3.0', 'vall')**

### Methods Documentation

**get_fixed_table**(*columns=None*)

Convenience function to get the table with html column names made human readable. It returns only the columns identified with the `columns` keyword. See the source for the defaults.

**get_species_ids**(*restr=None*, *reflags=0*)

Get a dictionary of "species" IDs, where species refers to the molecule name, mass, and chemical composition.

**Parameters**

**restr** : str

String to compile into an re, if specified. Searches table for species whose names match

**reflags** : int

Flags to pass to `re`.

### Examples

```
>>> import re
>>> import pprint # unfortunate hack required for documentation testing
>>> rslt = Splatalogue.get_species_ids('Formaldehyde')
>>> pprint.pprint(rslt)
{'03023 H2CO - Formaldehyde': '194',
 '03106 H213CO - Formaldehyde': '324',
 '03107 HDCO - Formaldehyde': '109',
 '03108 H2C17O - Formaldehyde': '982',
 '03202 H2C18O - Formaldehyde': '155',
 '03203 D2CO - Formaldehyde': '94',
 '03204 HD13CO - Formaldehyde': '1219',
 '03301 D213CO - Formaldehyde': '1220',
 '03315 HDC18O - Formaldehyde': '21141',
 '0348 D2C18O - Formaldehyde': '21140'}
>>> rslt = Splatalogue.get_species_ids('H2CO')
>>> pprint.pprint(rslt)
{'03023 H2CO - Formaldehyde': '194',
 '03109 H2COH+ - Hydroxymethylium ion': '224',
 '04406 c-H2COCH2 - Ethylene Oxide': '21',
 '07510 H2NCH2COOH - I v=0 - Glycine': '389',
 '07511 H2NCH2COOH - I v=1 - Glycine': '1312',
 '07512 H2NCH2COOH - I v=2 - Glycine': '1313',
 '07513 H2NCH2COOH - II v=0 - Glycine': '262',
 '07514 H2NCH2COOH - II v=1 - Glycine': '1314',
 '07515 H2NCH2COOH - II v=2 - Glycine': '1315',
 '07517 NH2CO2CH3 v=0 - Methyl Carbamate': '1334',
 '07518 NH2CO2CH3 v=1 - Methyl Carbamate': '1335',
 '08902 CH3CHNH2COOH - I - α-Alanine': '1321',
 '08903 CH3CHNH2COOH - II - α-Alanine': '1322'}
>>> # note the whitespace, preventing H2CO within other
>>> # more complex molecules
>>> Splatalogue.get_species_ids(' H2CO ')
{'03023 H2CO - Formaldehyde': '194'}
>>> Splatalogue.get_species_ids(' h2co ', re.IGNORECASE)
{'03023 H2CO - Formaldehyde': '194'}
```

**query_lines**(**args*, ***kwargs*)

Queries the service and returns a table object.

The Splatalogue service returns lines with rest frequencies in the range [min_frequency, max_frequency].

**Parameters**

**min_frequency** : `astropy.units`

Minimum frequency (or any spectral() equivalent)

**max_frequency** : `astropy.units`

Maximum frequency (or any spectral() equivalent)

**band** : str

The observing band. If it is not 'any', it overrides minfreq/maxfreq.

**top20: str**

One of `'comet'`, `'planet'`, `'top20'`, `'ism_hotcore'`, `'ism_darkcloud'`, `'ism_diffusecloud'`. Overrides chemical_name

**chemical_name** : str

Name of the chemical to search for. Treated as a regular expression. An empty set (‘’, (), [], {}) will match *any* species. Examples:

`'H2CO'` - 13 species have H2CO somewhere in their formula.

**`'Formaldehyde'` - There are 8 isotopologues of Formaldehyde**
(e.g., H213CO).

`'formaldehyde'` - Thioformaldehyde,Cyanoformaldehyde.

**`'formaldehyde',chem_re_flags=re.I` - Formaldehyde,thioformaldehyde,**
and Cyanoformaldehyde.

**`' H2CO '` - Just 1 species, H2CO. The spaces prevent including**
others.

**parse_chemistry_locally** : bool

Attempt to determine the species ID #’s locally before sending the query? This will prevent queries that have no matching species. It also performs a more flexible regular expression match to the species IDs. See the examples in `get_species_ids`

**chem_re_flags** : int

See the `re` module

**energy_min** : `None` or float

Energy range to include. See energy_type

**energy_max** : `None` or float

Energy range to include. See energy_type

**energy_type** : `'el_cm1'`, `'eu_cm1'`, `'eu_k'`, `'el_k'`

Type of energy to restrict. L/U for lower/upper state energy, cm/K for *inverse* cm, i.e. wavenumber, or K for Kelvin

**intensity_lower_limit** : `None` or float

Lower limit on the intensity. See intensity_type

**intensity_type** : `None` or `'sij'`, `'cdms_jpl'`, `'aij'`

The type of intensity on which to place a lower limit

**transition** : str

e.g. 1-0

**version** : `'v1.0'`, `'v2.0'`, `'v3.0'` or `'vall'`

Data version

**exclude** : list

Types of lines to exclude. Default is: (`'potential'`, `'atmospheric'`, `'probable'`) Can also exclude `'known'`. To exclude nothing, use ‘none’, not the python object None, since the latter is meant to indicate ‘leave as default’

**only_NRAO_recommended** : bool

Show only NRAO recommended species?

**line_lists** : list

Options: Lovas, SLAIM, JPL, CDMS, ToyoMA, OSU, Recomb, Lisa, RFI

**line_strengths** : list

- CDMS/JPL Intensity : ls1
- Sij : ls3
- Aij : ls4
- Lovas/AST : ls5

**energy_levels** : list

- E_lower (cm^-1) : el1
- E_lower (K) : el2
- E_upper (cm^-1) : el3
- E_upper (K) : el4

**export** : bool

Set up arguments for the export server (as opposed to the HTML server)?

**export_limit** : int

Maximum number of lines in output file

**noHFS** : bool

No HFS Display

**displayHFS** : bool

Display HFS Intensity

**show_unres_qn** : bool

Display Unresolved Quantum Numbers

**show_upper_degeneracy** : bool

Display Upper State Degeneracy

**show_molecule_tag** : bool

Display Molecule Tag

**show_qn_code** : bool

Display Quantum Number Code

**show_lovas_labref** : bool

Display Lab Ref

**show_lovas_obsref** : bool

Display Obs Ref

**show_orderedfreq_only** : bool

Display Ordered Frequency ONLY

**show_nrao_recommended** : bool

Display NRAO Recommended Frequencies

**Returns**

**table** : A `Table` object.

**query_lines_async**(*min_frequency=None*, *max_frequency=None*, *cache=True*, ***kwargs*)

The Splatalogue service returns lines with rest frequencies in the range [min_frequency, max_frequency].

**Parameters**

**min_frequency** : `astropy.units`

Minimum frequency (or any spectral() equivalent)

**max_frequency** : `astropy.units`

Maximum frequency (or any spectral() equivalent)

**band** : str

The observing band. If it is not 'any', it overrides minfreq/maxfreq.

**top20: str**

One of `'comet'`, `'planet'`, `'top20'`, `'ism_hotcore'`, `'ism_darkcloud'`, `'ism_diffusecloud'`. Overrides chemical_name

**chemical_name** : str

Name of the chemical to search for. Treated as a regular expression. An empty set ('', (), [], {}) will match *any* species. Examples:

`'H2CO'` - 13 species have H2CO somewhere in their formula.

**`'Formaldehyde'` - There are 8 isotopologues of Formaldehyde**
(e.g., H213CO).

`'formaldehyde'` - Thioformaldehyde,Cyanoformaldehyde.

**`'formaldehyde',chem_re_flags=re.I` - Formaldehyde,thioformaldehyde,**
and Cyanoformaldehyde.

**`' H2CO '` - Just 1 species, H2CO. The spaces prevent including**
others.

**parse_chemistry_locally** : bool

Attempt to determine the species ID #'s locally before sending the query? This will prevent queries that have no matching species. It also performs a more flexible regular expression match to the species IDs. See the examples in `get_species_ids`

**chem_re_flags** : int

See the `re` module

**energy_min** : `None` or float

Energy range to include. See energy_type

**energy_max** : `None` or float

Energy range to include. See energy_type

**energy_type** : `'el_cm1'`, `'eu_cm1'`, `'eu_k'`, `'el_k'`

Type of energy to restrict. L/U for lower/upper state energy, cm/K for *inverse* cm, i.e. wavenumber, or K for Kelvin

**intensity_lower_limit** : `None` or float

Lower limit on the intensity. See intensity_type

**intensity_type** : `None` or `'sij'`, `'cdms_jpl'`, `'aij'`

The type of intensity on which to place a lower limit

**transition** : str

e.g. 1-0

**version** : `'v1.0'`, `'v2.0'`, `'v3.0'` or `'vall'`

Data version

**exclude** : list

Types of lines to exclude. Default is: (`'potential'`, `'atmospheric'`, `'probable'`) Can also exclude `'known'`. To exclude nothing, use 'none', not the python object None, since the latter is meant to indicate 'leave as default'

**only_NRAO_recommended** : bool

Show only NRAO recommended species?

**line_lists** : list

Options: Lovas, SLAIM, JPL, CDMS, ToyoMA, OSU, Recomb, Lisa, RFI

**line_strengths** : list

- CDMS/JPL Intensity : ls1
- Sij : ls3
- Aij : ls4
- Lovas/AST : ls5

**energy_levels** : list

- E_lower (cm^-1) : el1
- E_lower (K) : el2
- E_upper (cm^-1) : el3
- E_upper (K) : el4

**export** : bool

Set up arguments for the export server (as opposed to the HTML server)?

**export_limit** : int

Maximum number of lines in output file

**noHFS** : bool

No HFS Display

**displayHFS** : bool

Display HFS Intensity

**show_unres_qn** : bool

Display Unresolved Quantum Numbers

**show_upper_degeneracy** : bool

Display Upper State Degeneracy

**show_molecule_tag** : bool

Display Molecule Tag

**show_qn_code** : bool

Display Quantum Number Code

**show_lovas_labref** : bool

Display Lab Ref

**show_lovas_obsref** : bool

Display Obs Ref

**show_orderedfreq_only** : bool

Display Ordered Frequency ONLY

**show_nrao_recommended** : bool

Display NRAO Recommended Frequencies

**Returns**

**response** : `requests.Response`

The response of the HTTP request.

**`set_default_options`**(***kwargs*)

Modify the default options. See `query_lines`

## Conf

**class** `astroquery.splatalogue.`**`Conf`**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.splatalogue`.

### Attributes Summary

| | |
|---|---|
| `lines_limit` | Limit to number of lines exported. |
| `query_url` | Splatalogue web interface URL. |
| `slap_url` | Splatalogue SLAP interface URL (not used). |
| `timeout` | Time limit for connecting to Splatalogue server. |

### Attributes Documentation

**`lines_limit`**

Limit to number of lines exported.

**`query_url`**

Splatalogue web interface URL.

**`slap_url`**

Splatalogue SLAP interface URL (not used).

**`timeout`**

Time limit for connecting to Splatalogue server.

CHAPTER 11

---

# Vamdc Queries (astroquery.vamdc)

---

## 11.1 Getting Started

The astroquery vamdc interface requires vamdclib. The documentation is sparse to nonexistant, but installation is straightforward:

```
pip install https://github.com/keflavich/vamdclib/archive/master.zip
```

This is the personal fork of the astroquery maintainer that includes astropy's setup helpers on top of the vamdclib infrastructure. If the infrastructure is merged into the main vamdclib library, we'll change these instructions.

## 11.2 Examples

If you want to compute the partition function, you can do so using a combination of astroquery and the vamdclib tools:

```
.. code-block:: python
```

```
>>> from astroquery.vamdc import Vamdc
>>> ch3oh = Vamdc.query_molecule('CH3OH')
>>> from vamdclib import specmodel
>>> partition_func = specmodel.calculate_partitionfunction(ch3oh.data['States'],
                                                           temperature=100)
>>> print(partition_func)
{'XCDMS-149': 1185.5304044622881}
```

## 11.3 Reference/API

### 11.3.1 astroquery.vamdc Package

VAMDC molecular line database

#### Classes

| | |
|---|---|
| `VamdcClass`([doimport]) | |
| `Conf` | Configuration parameters for `astroquery.vamdc`. |

#### VamdcClass

**class** astroquery.vamdc.**VamdcClass**(*doimport=True*)

Bases: `astroquery.query.BaseQuery`

##### Attributes Summary

| | |
|---|---|
| `CACHE_LOCATION` | |
| `TIMEOUT` | |
| `species_lookuptable` | As a property, you can't turn off caching.... |

##### Methods Summary

| | |
|---|---|
| `query_molecule`(molecule_name[, ...]) | Query for the VAMDC data for a specific molecule |

##### Attributes Documentation

**CACHE_LOCATION = '/home/docs/.astropy/cache/astroquery/vamdc'**

**TIMEOUT = 60**

**species_lookuptable**

As a property, you can't turn off caching....

##### Methods Documentation

**query_molecule**(*molecule_name*, *chem_re_flags=0*, *cache=True*)

Query for the VAMDC data for a specific molecule

**Parameters**

**molecule_name: str**

The common name (including unicode characters) or the ordinary molecular formula (e.g., CH3OH for Methanol) of the molecule.

**chem_re_flags: int**

The re (regular expression) flags for comparison of the molecule name with the lookuptable keys

**cache: bool**

Use the astroquery cache to store/recover the result

**Returns**

result: `vamdclib.request.Result`

A vamdclib Result object that has a data attribute. The result object has dictionary-like entries but has more functionality built on top of that

## Conf

**class** `astroquery.vamdc.`**`Conf`**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.vamdc`.

### Attributes Summary

| | |
|---|---|
| `cache_location` | |
| `timeout` | Timeout in seconds |

### Attributes Documentation

**`cache_location = '/home/docs/.astropy/cache/astroquery/vamdc'`**

**`timeout`**

Timeout in seconds

CHAPTER 12

# IRSA Image Server program interface (IBE) Queries (`astroquery.ibe`)

This module can has methods to perform different types of queries on the catalogs present in the IRSA Image Server program interface (IBE), which currently provides access to the 2MASS, WISE, and PTF image archives. In addition to supporting the standard query methods `query_region()` and `query_region_async()`, there are also methods to query the available missions (`list_missions()`), datasets (`list_datasets()`), tables (`list_tables()`), and columns (`get_columns()`).

## 12.1 Reference/API

### 12.1.1 astroquery.ibe Package

#### IRSA Image Server program interface (IBE) Query Tool

This module contains various methods for querying the IRSA Image Server program interface (IBE).

#### Classes

| | |
|---|---|
| `IbeClass`() | |
| `Conf` | Configuration parameters for `astroquery.ibe`. |

#### IbeClass

**class** astroquery.ibe.**IbeClass**

Bases: `astroquery.query.BaseQuery`

##### Attributes Summary

| |
|---|
| DATASET |
| MISSION |
| TABLE |
| TIMEOUT |
| URL |

### Methods Summary

| | |
|---|---|
| get_columns([mission, dataset, table]) | Get the schema for a given table. |
| list_datasets([mission, cache]) | For a given mission, list the available datasets |
| list_missions([cache]) | Return a list of the available missions |
| list_tables([mission, dataset, cache]) | For a given mission and dataset (see list_missions, list_datasets), return the list of valid table names to query. |
| query_region([coordinate, where, mission, . . . ]) | For certain missions, this function can be used to search for image and catalog files based on a point, a box (bounded by great circles) and/or an SQL-like where clause. |
| query_region_async([coordinate, where, . . . ]) | For certain missions, this function can be used to search for image and catalog files based on a point, a box (bounded by great circles) and/or an SQL-like where clause. |
| query_region_sia([coordinate, mission, . . . ]) | Query using simple image access protocol. |
| show_docs([mission, dataset, table]) | Open the documentation for a given table in a web browser. |

### Attributes Documentation

**DATASET = 'images'**

**MISSION = 'ptf'**

**TABLE = 'level1'**

**TIMEOUT = 60**

**URL = 'http://irsa.ipac.caltech.edu/ibe/'**

### Methods Documentation

**get_columns**(*mission=None*, *dataset=None*, *table=None*)

Get the schema for a given table.

**Parameters**

**mission** : str

The mission to be used (if not the default mission).

**dataset** : str

The dataset to be used (if not the default dataset).

**table** : str

The table to be queried (if not the default table).

**Returns**

**table** : `Table`

A table containing a description of the columns

`list_datasets`(*mission=None*, *cache=True*)

For a given mission, list the available datasets

**Parameters**

**mission** : str

A mission name. Must be one of the valid missions from `list_missions`. Defaults to the configured Mission

**cache** : bool

Cache the query result

**Returns**

**datasets** : list

A list of dataset names

`list_missions`(*cache=True*)

Return a list of the available missions

**Parameters**

**cache** : bool

Cache the query result

`list_tables`(*mission=None*, *dataset=None*, *cache=True*)

For a given mission and dataset (see `list_missions`, `list_datasets`), return the list of valid table names to query.

**Parameters**

**mission** : str

A mission name. Must be one of the valid missions from `list_missions`. Defaults to the configured Mission

**dataset** : str

A dataset name. Must be one of the valid dataset from `list_datsets(mission)`. Defaults to the configured Dataset

**cache** : bool

Cache the query result

**Returns**

**tables** : list

A list of table names

**query_region**(*coordinate=None*, *where=None*, *mission=None*, *dataset=None*, *table=None*, *columns=None*, *width=None*, *height=None*, *intersect='OVERLAPS'*, *most_centered=False*)

For certain missions, this function can be used to search for image and catalog files based on a point, a box (bounded by great circles) and/or an SQL-like `where` clause.

If `coordinates` is specified, then the optional `width` and `height` arguments control the width and height of the search box. If neither `width` nor `height` are provided, then the search area is a point. If only one of `width` or `height` are specified, then the search area is a square with that side length centered at the coordinate.

**Parameters**

**coordinate** : str, `astropy.coordinates` object

Gives the position of the center of the box if performing a box search. If it is a string, then it must be a valid argument to `SkyCoord`. Required if `where` is absent.

**where** : str

SQL-like query string. Required if `coordinates` is absent.

**mission** : str

The mission to be used (if not the default mission).

**dataset** : str

The dataset to be used (if not the default dataset).

**table** : str

The table to be queried (if not the default table).

**columns** : str, list

A space-separated string or a list of strings of the names of the columns to return.

**width** : str or `Quantity` object

Width of the search box if `coordinates` is present.

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used.

**height** : str, `Quantity` object

Height of the search box if `coordinates` is present.

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used.

**intersect** : `'COVERS'`, `'ENCLOSED'`, `'CENTER'`, `'OVERLAPS'`

Spatial relationship between search box and image footprint.

`'COVERS'`: X must completely contain S. Equivalent to `'CENTER'` and `'OVERLAPS'` if S is a point.

`'ENCLOSED'`: S must completely contain X. If S is a point, the query will always return an empty image table.

`'CENTER'`: X must contain the center of S. If S is a point, this is equivalent to `'COVERS'` and `'OVERLAPS'`.

`'OVERLAPS'`: The intersection of S and X is non-empty. If S is a point, this is equivalent to `'CENTER'` and `'COVERS'`.

**most_centered** : bool

If True, then only the most centered image is returned.

**Returns**

**table** : `Table`

A table containing the results of the query

**query_region_async**(*coordinate=None*, *where=None*, *mission=None*, *dataset=None*, *table=None*, *columns=None*, *width=None*, *height=None*, *action='search'*, *intersect='OVERLAPS'*, *most_centered=False*)

For certain missions, this function can be used to search for image and catalog files based on a point, a box (bounded by great circles) and/or an SQL-like `where` clause.

If `coordinates` is specified, then the optional `width` and `height` arguments control the width and height of the search box. If neither `width` nor `height` are provided, then the search area is a point. If only one of `width` or `height` are specified, then the search area is a square with that side length centered at the coordinate.

**Parameters**

**coordinate** : str, `astropy.coordinates` object

Gives the position of the center of the box if performing a box search. If it is a string, then it must be a valid argument to `SkyCoord`. Required if `where` is absent.

**where** : str

SQL-like query string. Required if `coordinates` is absent.

**mission** : str

The mission to be used (if not the default mission).

**dataset** : str

The dataset to be used (if not the default dataset).

**table** : str

The table to be queried (if not the default table).

**columns** : str, list

A space-separated string or a list of strings of the names of the columns to return.

**width** : str or `Quantity` object

Width of the search box if `coordinates` is present.

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used.

**height** : str, `Quantity` object

Height of the search box if `coordinates` is present.

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used.

**intersect** : `'COVERS'`, `'ENCLOSED'`, `'CENTER'`, `'OVERLAPS'`

Spatial relationship between search box and image footprint.

`'COVERS'`: X must completely contain S. Equivalent to `'CENTER'` and `'OVERLAPS'` if S is a point.

'ENCLOSED': S must completely contain X. If S is a point, the query will always return an empty image table.

'CENTER': X must contain the center of S. If S is a point, this is equivalent to 'COVERS' and 'OVERLAPS'.

'OVERLAPS': The intersection of S and X is non-empty. If S is a point, this is equivalent to 'CENTER' and 'COVERS'.

**most_centered** : bool

If True, then only the most centered image is returned.

**action** : 'search', 'data', or 'sia'

The action to perform at the server. The default is 'search', which returns a table of the available data. 'data' requires advanced path construction that is not yet supported. 'sia' provides access to the 'simple image access' IVOA protocol

**Returns**

**response** : Response

The HTTP response returned from the service

**query_region_sia**(*coordinate=None*, *mission=None*, *dataset=None*, *table=None*, *width=None*, *height=None*, *intersect='OVERLAPS'*, *most_centered=False*)

Query using simple image access protocol. See query_region for details. The returned table will include a list of URLs.

**show_docs**(*mission=None*, *dataset=None*, *table=None*)

Open the documentation for a given table in a web browser.

**Parameters**

**mission** : str

The mission to be used (if not the default mission).

**dataset** : str

The dataset to be used (if not the default dataset).

**table** : str

The table to be queried (if not the default table).

## Conf

**class** astroquery.ibe.**Conf**

Bases: astropy.config.ConfigNamespace

Configuration parameters for astroquery.ibe.

### Attributes Summary

| | |
|---|---|
| dataset | Default data set. |
| mission | Default mission. |
| server | Name of the IBE server to use. |
| table | Default table. |
| timeout | Time limit for connecting to the IRSA server. |

### Attributes Documentation

**dataset**
Default data set. See, for example, http://irsa.ipac.caltech.edu/ibe/search/ptf for options.

**mission**
Default mission. See, for example, http://irsa.ipac.caltech.edu/ibe/search/ for options.

**server**
Name of the IBE server to use.

**table**
Default table. See, for example, http://irsa.ipac.caltech.edu/ibe/search/ptf/images for options.

**timeout**
Time limit for connecting to the IRSA server.

CHAPTER 13

# IRSA Queries (astroquery.irsa)

## 13.1 Getting started

This module can has methods to perform different types of queries on the catalogs present in the IRSA general catalog service. All queries can be performed by calling `query_region()`, with different keyword arguments. There are 4 different region queries that are supported: `Cone`, `Box`, `Polygon` and `All-Sky`. All successful queries return the results in a `Table`. We now look at some examples.

### 13.1.1 Available catalogs

All region queries require a `catalog` keyword argument, which is the name of the catalog in the IRSA database, on which the query must be performed. To take a look at all the available catalogs:

```
>>> from astroquery.irsa import Irsa
>>> Irsa.list_catalogs()

{'a1763t2': 'Abell 1763 Source Catalog',
 'a1763t3': 'Abell 1763 MIPS 70 micron Catalog',
 'acs_iphot_sep07': 'COSMOS ACS I-band photometry catalog September 2007',
 'akari_fis': 'Akari/FIS Bright Source Catalogue',
 'akari_irc': 'Akari/IRC Point Source Catalogue',
 'astsight': 'IRAS Minor Planet Survey',
 ...
 ...
 'xmm_cat_s05': "SWIRE XMM_LSS Region Spring '05 Spitzer Catalog"}
```

This returns a dictionary of catalog names with their description. If you would rather just print out this information:

```
>>> from astroquery.irsa import Irsa
>>> Irsa.print_catalogs()

wise_allsky_2band_p1bm_frm      WISE Post-Cryo Single Exposure (L1b) Image Inventory Table
```

(continues on next page)

(continued from previous page)

```
wise_allsky_4band_p3as_psr      WISE All-Sky Reject Table
cosmos_morph_col_1              COSMOS Zamojski Morphology Catalog v1.0
wise_prelim_p3al_lod            WISE Preliminary Release Atlas Inventory Table (Superseded)
com_pccs1_100                   Planck PCCS 100GHz Catalog
swire_lhisod                    SWIRE Lockman Hole ISOCAM Deep Field Catalog
...
...
sdwfs_ch1_epoch3                SDWFS Aug '09 DR1.1 IRAC 3.6um-Selected 3x30sec Coadd, epoch 3 (Feb '08)
```

## 13.1.2 Performing a cone search

A cone search query is performed by setting the `spatial` keyword to `Cone`. The target name or the coordinates of the search center must also be specified. The radius for the cone search may also be specified - if this is missing, it defaults to a value of 10 arcsec. The radius may be specified in any appropriate unit using a `Quantity` object. It may also be entered as a string that is parsable by `Angle`.

```
>>> from astroquery.irsa import Irsa
>>> import astropy.units as u
>>> table = Irsa.query_region("m31", catalog="fp_psc", spatial="Cone",
...                           radius=2 * u.arcmin)
>>> print(table)

      ra     dec       clon         clat      err_maj ...  j_h   h_k    j_k    id
------- ------- ------------ ------------ ------- ... ----- ------ ------ ---
 10.685  41.248 00h42m44.45s 41d14m52.56s    0.14 ... 1.792 -0.821  0.971   0
 10.697  41.275 00h42m47.39s 41d16m30.25s    0.13 ...    --     --     --   1
 10.673  41.254 00h42m41.63s 41d15m15.66s    0.26 ...    --  1.433     --   2
 10.671  41.267 00h42m41.10s 41d15m59.97s    0.17 ...    --     --     --   3
 10.684  41.290 00h42m44.11s 41d17m24.99s    0.19 ... 0.261 -1.484 -1.223   4
 10.692  41.290 00h42m46.08s 41d17m24.99s    0.18 ...    --     --  0.433   5
 10.716  41.260 00h42m51.77s 41d15m36.31s    0.13 ...  0.65     --     --   6
 10.650  41.286 00h42m35.96s 41d17m08.48s    0.41 ... 1.205     --     --   7
    ...     ...          ...          ...     ... ...   ...    ...    ... ...
 10.686  41.271 00h42m44.60s 41d16m14.16s    0.13 ...    --     --     -- 768
 10.694  41.277 00h42m46.55s 41d16m36.13s    0.27 ...    --     --     -- 769
 10.690  41.277 00h42m45.71s 41d16m36.54s    0.15 ...    --     --     -- 770
 10.679  41.281 00h42m42.88s 41d16m51.62s    0.43 ...    --     --     -- 771
 10.689  41.237 00h42m45.26s 41d14m13.32s    0.22 ...    --     --     -- 772
 10.661  41.274 00h42m38.53s 41d16m24.76s    0.18 ...    --     --     -- 773
 10.653  41.281 00h42m36.78s 41d16m52.98s    0.17 ...    --  0.795     -- 774
```

The coordinates of the center may be specified rather than using the target name. The coordinates can be specified using the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered directly as a string, as specified by `astropy.coordinates`:

```
>>> from astroquery.irsa import Irsa
>>> import astropy.coordinates as coord
>>> table = Irsa.query_region(coord.SkyCoord(121.1743,
...                           -21.5733, unit=(u.deg,u.deg),
...                           frame='galactic'),
...                           catalog='fp_psc', radius='0d2m0s')
>>> print(table)
```

### 13.1.3 Performing a box search

The box queries have a syntax similar to the cone queries. In this case the `spatial` keyword argument must be set to `Box`. Also the width of the box region is required. The width may be specified in the same way as the radius for cone search queries, above - so it may be set using the appropriate `Quantity` object or a string parsable by `Angle`.

```
>>> from astroquery.irsa import Irsa
>>> import astropy.units as u
>>> table = Irsa.query_region("00h42m44.330s +41d16m07.50s",
...                           catalog='fp_psc', spatial='Box',
...                           width=5 * u.arcsec)

WARNING: Coordinate string is being interpreted as an ICRS
coordinate. [astroquery.irsa.core]

>>> print(table)

     ra     dec       clon         clat      err_maj ...  j_h   h_k   j_k   id
------- ------- ------------ ------------ ------- ... ----- ----- ----- ---
 10.685  41.269 00h42m44.34s 41d16m08.53s    0.08 ... 0.785 0.193 0.978   0
```

Note that in this case we directly passed ICRS coordinates as a string to the `query_region()`.

### 13.1.4 Queries over a polygon

Polygon queries can be performed by setting `spatial='Polygon'`. The search center is optional in this case. One additional parameter that must be set for these queries is `polygon`. This is a list of coordinate pairs that define a convex polygon. The coordinates may be specified as usual by using the appropriate `astropy.coordinates` object (Again ICRS coordinates may be directly passed as properly formatted strings). In addition to using a list of `astropy.coordinates` objects, one additional convenient means of specifying the coordinates is also available - Coordinates may also be entered as a list of tuples, each tuple containing the ra and dec values in degrees. Each of these options is illustrated below:

```
>>> from astroquery.irsa import Irsa
>>> from astropy import coordinates
>>> table = Irsa.query_region("m31", catalog="fp_psc", spatial="Polygon",
... polygon=[coordinates.SkyCoord(ra=10.1, dec=10.1, unit=(u.deg, u.deg), frame='icrs'),
...          coordinates.SkyCoord(ra=10.0, dec=10.1, unit=(u.deg, u.deg), frame='icrs'),
...          coordinates.SkyCoord(ra=10.0, dec=10.0, unit=(u.deg, u.deg), frame='icrs')
...         ])
>>> print(table)

     ra     dec       clon         clat      err_maj ...  j_h   h_k   j_k   id
------- ------- ------------ ------------ ------- ... ----- ----- ----- ---
 10.016  10.099 00h40m03.77s 10d05m57.22s     0.1 ... 0.602 0.154 0.756   0
 10.031  10.063 00h40m07.44s 10d03m47.10s    0.19 ... 0.809 0.291   1.1   1
 10.037  10.060 00h40m08.83s 10d03m37.00s    0.11 ... 0.468 0.372  0.84   2
 10.060  10.085 00h40m14.39s 10d05m07.60s    0.23 ... 0.697 0.273  0.97   3
 10.016  10.038 00h40m03.80s 10d02m17.02s    0.09 ... 0.552 0.313 0.865   4
 10.011  10.094 00h40m02.68s 10d05m38.05s    0.23 ... 0.378 0.602  0.98   5
 10.006  10.018 00h40m01.33s 10d01m06.24s    0.16 ... 0.662 0.566 1.228   6
```

Another way to specify the polygon is directly as a list of tuples - each tuple is an ra, dec pair expressed in degrees:

```
>>> from astroquery.irsa import Irsa
>>> table = Irsa.query_region("m31", catalog="fp_psc", spatial="Polygon",
```

(continues on next page)

(continued from previous page)

```
... polygon = [(10.1, 10.1), (10.0, 10.1), (10.0, 10.0)])
>>> print(table)

     ra      dec        clon         clat      err_maj ...  j_h   h_k   j_k   id
------- ------- ------------ ------------ ------- ... ----- ----- ----- ---
 10.016  10.099 00h40m03.77s 10d05m57.22s     0.1 ... 0.602 0.154 0.756   0
 10.031  10.063 00h40m07.44s 10d03m47.10s    0.19 ... 0.809 0.291   1.1   1
 10.037  10.060 00h40m08.83s 10d03m37.00s    0.11 ... 0.468 0.372  0.84   2
 10.060  10.085 00h40m14.39s 10d05m07.60s    0.23 ... 0.697 0.273  0.97   3
 10.016  10.038 00h40m03.80s 10d02m17.02s    0.09 ... 0.552 0.313 0.865   4
 10.011  10.094 00h40m02.68s 10d05m38.05s    0.23 ... 0.378 0.602  0.98   5
 10.006  10.018 00h40m01.33s 10d01m06.24s    0.16 ... 0.662 0.566 1.228   6
```

### 13.1.5 Other Configurations

By default the maximum number of rows that is fetched is set to 500. However, this option may be changed by changing the astroquery configuration file. To change the setting only for the ongoing python session, you could also do:

```
>>> from astroquery.irsa import Irsa
>>> Irsa.ROW_LIMIT = 1000 # value of new row limit here.
```

## 13.2 Reference/API

### 13.2.1 astroquery.irsa Package

#### IRSA Query Tool

This module contains various methods for querying the IRSA Catalog Query Service(CatQuery).

#### Classes

| | |
|---|---|
| IrsaClass() | |
| Conf | Configuration parameters for astroquery.irsa. |

#### IrsaClass

**class** astroquery.irsa.**IrsaClass**

Bases: astroquery.query.BaseQuery

##### Attributes Summary

| |
|---|
| GATOR_LIST_URL |
| IRSA_URL |
| ROW_LIMIT |

Continued on next page

Table 2 – continued from previous page

| |
|---|
| TIMEOUT |

#### Methods Summary

| | |
|---|---|
| list_catalogs() | Return a dictionary of the catalogs in the IRSA Gator tool. |
| print_catalogs() | Display a table of the catalogs in the IRSA Gator tool. |
| query_region([coordinates, catalog, ...]) | This function can be used to perform either cone, box, polygon or all-sky search in the catalogs hosted by the NASA/IPAC Infrared Science Archive (IRSA). |
| query_region_async([coordinates, catalog, ...]) | This function serves the same purpose as query_region(), but returns the raw HTTP response rather than the results in a Table. |

#### Attributes Documentation

**GATOR_LIST_URL = 'https://irsa.ipac.caltech.edu/cgi-bin/Gator/nph-scan'**

**IRSA_URL = 'https://irsa.ipac.caltech.edu/cgi-bin/Gator/nph-query'**

**ROW_LIMIT = 500**

**TIMEOUT = 60**

#### Methods Documentation

**list_catalogs**()

Return a dictionary of the catalogs in the IRSA Gator tool.

**Returns**

**catalogs** : dict

A dictionary of catalogs where the key indicates the catalog name to be used in query functions, and the value is the verbose description of the catalog.

**print_catalogs**()

Display a table of the catalogs in the IRSA Gator tool.

**query_region**(*coordinates=None*, *catalog=None*, *spatial='Cone'*, *radius=<Quantity 10. arcsec>*, *width=None*, *polygon=None*, *get_query_payload=False*, *verbose=False*)

This function can be used to perform either cone, box, polygon or all-sky search in the catalogs hosted by the NASA/IPAC Infrared Science Archive (IRSA).

**Parameters**

**coordinates** : str, astropy.coordinates object

Gives the position of the center of the cone or box if performing a cone or box search. The string can give coordinates in various coordinate systems, or the name of a source that will be resolved on the server (see here for more details). Required if spatial is 'Cone' or 'Box'. Optional if spatial is 'Polygon'.

**catalog** : str

The catalog to be used. To list the available catalogs, use `print_catalogs()`.

**spatial** : str

Type of spatial query: `'Cone'`, `'Box'`, `'Polygon'`, and `'All-Sky'`. If missing then defaults to `'Cone'`.

**radius** : str or `Quantity` object, [optional for spatial is `'Cone'`]

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 10 arcsec.

**width** : str, `Quantity` object [Required for spatial is `'Polygon'`.]

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used.

**polygon** : list, [Required for spatial is `'Polygon'`]

A list of `(ra, dec)` pairs (as tuples), in decimal degrees, outlining the polygon to search in. It can also be a list of `astropy.coordinates` object or strings that can be parsed by `astropy.coordinates.ICRS`.

**get_query_payload** : bool, optional

If `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**verbose** : bool, optional.

If `True` then displays warnings when the returned VOTable does not conform to the standard. Defaults to `False`.

**Returns**

**table** : `Table`

A table containing the results of the query

**query_region_async**(*coordinates=None*, *catalog=None*, *spatial='Cone'*, *radius=<Quantity 10. arcsec>*, *width=None*, *polygon=None*, *get_query_payload=False*)

This function serves the same purpose as `query_region()`, but returns the raw HTTP response rather than the results in a `Table`.

**Parameters**

**coordinates** : str, `astropy.coordinates` object

Gives the position of the center of the cone or box if performing a cone or box search. The string can give coordinates in various coordinate systems, or the name of a source that will be resolved on the server (see here for more details). Required if spatial is `'Cone'` or `'Box'`. Optional if spatial is `'Polygon'`.

**catalog** : str

The catalog to be used. To list the available catalogs, use `print_catalogs()`.

**spatial** : str

Type of spatial query: `'Cone'`, `'Box'`, `'Polygon'`, and `'All-Sky'`. If missing then defaults to `'Cone'`.

**radius** : str or `Quantity` object, [optional for spatial is `'Cone'`]

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 10 arcsec.

**width** : str, `Quantity` object [Required for spatial is `'Polygon'`.]

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used.

**polygon** : list, [Required for spatial is `'Polygon'`]

A list of `(ra, dec)` pairs (as tuples), in decimal degrees, outlining the polygon to search in. It can also be a list of `astropy.coordinates` object or strings that can be parsed by `astropy.coordinates.ICRS`.

**get_query_payload** : bool, optional

If `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service

## Conf

**class** astroquery.irsa.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.irsa`.

### Attributes Summary

| | |
|---|---|
| `gator_list_catalogs` | URL from which to list all the public catalogs in IRSA. |
| `row_limit` | Maximum number of rows to retrieve in result |
| `server` | Name of the IRSA mirror to use. |
| `timeout` | Time limit for connecting to the IRSA server. |

### Attributes Documentation

**gator_list_catalogs**

URL from which to list all the public catalogs in IRSA.

**row_limit**

Maximum number of rows to retrieve in result

**server**

Name of the IRSA mirror to use.

**timeout**

Time limit for connecting to the IRSA server.

CHAPTER 14

# UKIDSS Queries (astroquery.ukidss)

## 14.1 Getting started

This module allows searching catalogs and retrieving images from the UKIDSS web service. Some data can on UKIDSS can be accessed only after a valid login. For accessing such data a UKIDSS object must first be instantiated with valid credentials. On the other hand, to access the public data, the various query functions may just be called as class methods - i.e. no instantiation is required. Below are examples that illustrate both the means for accessing the data.

**Case 1 : Access only public data - No login**

```
>>> from astroquery.ukidss import Ukidss

# perform any query as a class method - no instantiation required

>>> images = Ukidss.get_images("m1")

Found 1 targets
Downloading http://surveys.roe.ac.uk/wsa/cgi-bin/getFImage.cgi?file=/disk05/wsa/ingest/fits/20071011_v1/
↪w20071011_01818_sf_st.fit&mfid=1737581&extNo=4&lx=1339&hx=1638&ly=1953&hy=2252&rf=0&flip=1&uniq=5348_
↪573_21_31156_1&xpos=150.7&ypos=150.3&band=K&ra=83.633083&dec=22.0145
|===========================================| 174k/174k (100.00%)        02s
```

**Case 2 : Login to access non-public data**

```
>>> from astroquery.ukidss import Ukidss

# Now first instantiate a Ukidss object with login credentials

>>> u_obj = Ukidss(username='xyz', password='secret',
...                community='your_community')

>>> # The prompt appears indicating successful login
```

Note that at login time you may also optionally set the database and the survey that you would like to query. By default the database is set to 'UKIDSSDR7PLUS' and the programme_id is set to 'all' - which includes all the surveys. A word of warning - for region queries you should explicitly set the programme_id to the survey you wish to query. Querying all surveys is permitted only for image queries.

```
>>> from astroquery.ukidss import Ukidss
>>> u_obj = Ukidss(username='xyz', password='secret',
...                community='your_community', database='UKIDSSDR8PLUS',
...                programme_id='GPS')
>>>
```

At any given time you may if you wish check your login status (continuing from the above example):

```
>>> u_obj.logged_in()

True
```

If you want to change your programme_id and database after you have already instantiated the object - say u_obj then you should do:

```
>>> u_obj.programme_id = 'new_id_here'
>>> u_obj.database = 'new_database_here'
```

The above examples mention programme_id that specifies the catalog or survey you wish to query. If you would like to get a list of the commonly available UKIDSS survey - either the abbreviations or the complete names, you can do so by using list_catalogs():

```
>>> from astroquery.ukidss import Ukidss
>>> Ukidss.list_catalogs()

['UDS', 'GCS', 'GPS', 'DXS', 'LAS']

>>> Ukidss.list_catalogs(style='long')

['Galactic Plane Survey',
 'Deep Extragalactic Survey',
 'Galactic Clusters Survey',
 'Ultra Deep Survey',
 'Large Area Survey']
```

Now we look at examples of the actual queries that can be performed.

### 14.1.1 Get images around a target

You can fetch images around the specified target or coordinates. When a target name is used rather than the coordinates, this will be resolved to coordinates using astropy name resolving methods that utilize online services like SESAME. Coordinates may be entered using the suitable object from astropy.coordinates. The images are returned as a list of HDUList objects.

```
>>> from astroquery.ukidss import Ukidss
>>> images = Ukidss.get_images("m1")

Found 1 targets
Downloading http://surveys.roe.ac.uk/wsa/cgi-bin/getFImage.cgi?file=/disk05/wsa/ingest/fits/20071011_v1/
↪w20071011_01818_sf_st.fit&mfid=1737581&extNo=4&lx=1339&hx=1638&ly=1953&hy=2252&rf=0&flip=1&uniq=142_
↪292_19_31199_1&xpos=150.7&ypos=150.3&band=K&ra=83.633083&dec=22.0145
```

(continues on next page)

(continued from previous page)

```
|===========================================| 174k/174k (100.00%)        07s

>>> print(images)

[[<astropy.io.fits.hdu.image.PrimaryHDU object at 0x40f8b10>, <astropy.io.fits.hdu.image.ImageHDU␣
↪object at 0x41026d0>]]
```

Note if you have logged in using the procedure described earlier and assuming that you already have a UkidssClass object u_obj instantiated:

```
>>> images = u_obj.get_images("m1")
```

There are several optional parameters that you can specify in image queries. For instance to specify the image size you should set the image_width and the image_height keyword arguments. If only the image_width is set then the image_height is taken to be the same as this width. By default the image_width is set to 1 arcmin. To set this to your desired value, you should enter it using a Quantity object with appropriate units or as a string that is parsable by Angle. Another parameter you may set is radius. This may be specified in the same way as the image_height and image_width with the radius keyword. By specifying this multi-frame FITS images will be retrieved. Note that in this case the image height and width parameters will no longer be effective.

```
>>> from astroquery.ukidss import Ukidss
>>> import astropy.units as u
>>> import astropy.coordinates as coord
>>> images = Ukidss.get_images(coord.SkyCoord(49.489, -0.27,
...                                           unit=(u.deg, u.deg),
...                                           frame='galactic'),
...                            image_width=5 * u.arcmin)

    Found 6 targets
Downloading http://surveys.roe.ac.uk/wsa/cgi-bin/getFImage.cgi?file=/disk24/wsa/ingest/fits/20060603_v1/
↪w20060603_01510_sf_st_two.fit&mfid=2514752&extNo=1&lx=862&hx=1460&ly=1539&hy=2137&rf=270&flip=1&
↪uniq=575_115_31_31555_1&xpos=300.1&ypos=299.7&band=J&ra=290.8256247&dec=14.56435
|===========================================| 518k/518k (100.00%)        06s
Downloading http://surveys.roe.ac.uk/wsa/cgi-bin/getFImage.cgi?file=/disk24/wsa/ingest/fits/20060603_v1/
↪w20060603_01510_sf_st.fit&mfid=966724&extNo=1&lx=862&hx=1460&ly=1539&hy=2137&rf=270&flip=1&uniq=575_
↪115_31_31555_2&xpos=300&ypos=299.8&band=J&ra=290.8256247&dec=14.56435
|===========================================| 517k/517k (100.00%)        06s
Downloading http://surveys.roe.ac.uk/wsa/cgi-bin/getFImage.cgi?file=/disk24/wsa/ingest/fits/20060603_v1/
↪w20060603_01544_sf_st_two.fit&mfid=2514753&extNo=1&lx=863&hx=1461&ly=1538&hy=2136&rf=270&flip=1&
↪uniq=575_115_31_31555_3&xpos=300&ypos=300&band=H&ra=290.8256247&dec=14.56435
|===========================================| 654k/654k (100.00%)        06s
Downloading http://surveys.roe.ac.uk/wsa/cgi-bin/getFImage.cgi?file=/disk24/wsa/ingest/fits/20060603_v1/
↪w20060603_01544_sf_st.fit&mfid=965662&extNo=1&lx=863&hx=1461&ly=1538&hy=2136&rf=270&flip=1&uniq=575_
↪115_31_31555_4&xpos=300&ypos=300.1&band=H&ra=290.8256247&dec=14.56435
|===========================================| 647k/647k (100.00%)        06s
Downloading http://surveys.roe.ac.uk/wsa/cgi-bin/getFImage.cgi?file=/disk24/wsa/ingest/fits/20060603_v1/
↪w20060603_01577_sf_st.fit&mfid=952046&extNo=1&lx=863&hx=1460&ly=1538&hy=2135&rf=270&flip=1&uniq=575_
↪115_31_31555_5&xpos=299.2&ypos=299.7&band=K&ra=290.8256247&dec=14.56435
|===========================================| 586k/586k (100.00%)        06s
Downloading http://surveys.roe.ac.uk/wsa/cgi-bin/getFImage.cgi?file=/disk24/wsa/ingest/fits/20060603_v1/
↪w20060603_01577_sf_st_two.fit&mfid=2514749&extNo=1&lx=863&hx=1460&ly=1538&hy=2135&rf=270&flip=1&
↪uniq=575_115_31_31555_6&xpos=299.5&ypos=299.3&band=K&ra=290.8256247&dec=14.56435
|===========================================| 587k/587k (100.00%)        04s
```

Again the query may be performed similarly with a log-in.

If you haven't logged-in then you could also specify the programme_id as a keyword argument. By default this is set to 'all'. But you can change it to a specific survey as mentioned earlier. The same goes for the database which is set by default to 'UKIDSSDR7PLUS'. Some more parameters you can set are the frame_type which may be one of

```
'stack' 'normal' 'interleave' 'deep_stack' 'confidence' 'difference'
'leavstack' 'all'
```

and the waveband that decides the color filter to download. This must be chosen from

```
'all' 'J' 'H' 'K' 'H2' 'Z' 'Y' 'Br'
```

Note that rather than fetching the actual images, you could also get the URLs of the downloadable images. To do this simply replace the call to get_images() by a call to get_image_list() with exactly the same parameters. Let us now see a complete example to illustrate these points.

```
>>> from astroquery.ukidss import Ukidss
>>> import astropy.units as u
>>> import astropy.coordinates as coord
>>> image_urls = Ukidss.get_image_list(coord.SkyCoord(ra=83.633083,
...          dec=22.0145, unit=(u.deg, u.deg), frame='icrs'),
...          frame_type='interleave',
...          programme_id="GCS", waveband="K", radius=20*u.arcmin)
>>> image_urls

    ['http://surveys.roe.ac.uk/wsa/cgi-bin/fits_download.cgi?file=/disk05/wsa/ingest/fits/20071011_v1/
↪w20071011_01802_sf.fit&MFID=1737551&rID=2544',
 'http://surveys.roe.ac.uk/wsa/cgi-bin/fits_download.cgi?file=/disk05/wsa/ingest/fits/20071011_v1/
↪w20071011_01802_sf_st.fit&MFID=1737553&rID=2544',
 'http://surveys.roe.ac.uk/wsa/cgi-bin/fits_download.cgi?file=/disk05/wsa/ingest/fits/20071011_v1/
↪w20071011_01806_sf.fit&MFID=1737559&rID=2544',
 'http://surveys.roe.ac.uk/wsa/cgi-bin/fits_download.cgi?file=/disk05/wsa/ingest/fits/20071011_v1/
↪w20071011_01818_sf_st.fit&MFID=1737581&rID=2544',
 'http://surveys.roe.ac.uk/wsa/cgi-bin/fits_download.cgi?file=/disk05/wsa/ingest/fits/20071011_v1/
↪w20071011_01818_sf.fit&MFID=1737579&rID=2544',
 'http://surveys.roe.ac.uk/wsa/cgi-bin/fits_download.cgi?file=/disk05/wsa/ingest/fits/20071011_v1/
↪w20071011_01822_sf.fit&MFID=1737587&rID=2544']
```

### 14.1.2 Query a region

Another type of query is to search a catalog for objects within a specified radius of a source. Again the source may be either a named identifier or it may be specified via coordinates. The radius may be specified as in the previous cases by using a Quantity or a string parsable via Angle. If this is missing, then it defaults to 1 arcmin. As before you may also mention the programme_id and the database. The query results are returned in a Table.

```
>>> from astroquery.ukidss import Ukidss
>>> import astropy.coordinates as coord
>>> import astropy.units as u
>>> table = Ukidss.query_region(coord.SkyCoord(10.625, -0.38,
...                                            unit=(u.deg, u.deg),
...                                            frame='galactic'),
...                             radius=6 * u.arcsec)

Downloading http://surveys.roe.ac.uk/wsa/tmp/tmp_sql/results1_4_45_58_24651.xml
|===========================================| 4.6k/4.6k (100.00%)        00s

>>> print(table)
```

(continues on next page)

(continued from previous page)

```
    sourceID     framesetID        RA      ... H2AperMag3Err     distance
------------ ------------ ------------- ... ------------- ---------------
438758381345 438086690175 272.615581037 ...  -9.99999e+08 0.0864656469768
438758381157 438086690175   272.6178395 ...  -9.99999e+08 0.0717893063941
438758381256 438086690175 272.616581639 ...  -9.99999e+08 0.0725261678319
438758399768 438086690175 272.618154346 ...  -9.99999e+08 0.0800819201056
438758399809 438086690175 272.617630209 ...  -9.99999e+08 0.0996951205267
438758399828 438086690175 272.617238438 ...  -9.99999e+08 0.0435480204348
438758399832 438086690175 272.617034836 ...  -9.99999e+08 0.0665854385804
438758414982 438086690175 272.616576986 ...  -9.99999e+08 0.0214102038115
```

## 14.2 Reference/API

### 14.2.1 astroquery.ukidss Package

#### UKIDSS Image and Catalog Query Tool

**Revision History**

Refactored using common API as a part of Google Summer of Code 2013.

**Originally contributed by**

Thomas Robitalle (thomas.robitaille@gmail.com)

Adam Ginsburg (adam.g.ginsburg@gmail.com)

#### Functions

| | |
|---|---|
| clean_catalog(ukidss_catalog[, clean_band, …]) | Attempt to remove 'bad' entries in a catalog. |

#### clean_catalog

astroquery.ukidss.**clean_catalog**(*ukidss_catalog*, *clean_band='K_1'*, *badclass=-9999*, *maxerrbits=41*, *minerrbits=0*, *maxpperrbits=60*)

Attempt to remove 'bad' entries in a catalog.

**Parameters**

**ukidss_catalog** : BinTableHDU

A FITS binary table instance from the UKIDSS survey.

**clean_band** : 'K_1', 'K_2', 'J', 'H'

The band to use for bad photometry flagging.

**badclass** : int

Class to exclude.

**minerrbits** : int

**maxerrbits** : int

Inside this range is the accepted number of error bits.

**maxpperrbits** : int

Exclude this type of error bit.

### Classes

| | |
|---|---|
| UkidssClass([username, password, community, ...]) | The UKIDSSQuery class. |
| Conf | Configuration parameters for astroquery.ukidss. |

### UkidssClass

**class** astroquery.ukidss.**UkidssClass**(*username=None*, *password=None*, *community=None*, *database='UKIDSSDR7PLUS'*, *programme_id='all'*)

Bases: astroquery.query.QueryWithLogin

The UKIDSSQuery class. Must instantiate this class in order to make any queries. Allows registered users to login, but defaults to using the public UKIDSS data sets.

#### Attributes Summary

| |
|---|
| ARCHIVE_URL |
| BASE_URL |
| IMAGE_URL |
| LOGIN_URL |
| REGION_URL |
| TIMEOUT |
| all_databases |
| filters |
| frame_types |
| ukidss_programmes_long |
| ukidss_programmes_short |

#### Methods Summary

| | |
|---|---|
| extract_urls(html_in) | Helper function that uses regexps to extract the image urls from the given HTML. |
| get_image_list(**kwargs) | |
| get_images(coordinates[, waveband, ...]) | Get an image around a target/ coordinates from UKIDSS catalog. |
| get_images_async(coordinates[, waveband, ...]) | Serves the same purpose as get_images but returns a list of file handlers to remote files. |
| list_catalogs([style]) | Returns a list of available catalogs in UKIDSS. |
| list_databases() | List the databases available from the UKIDSS WFCAM archive. |
| logged_in() | Determine whether currently logged in. |
| query_region(coordinates[, radius, ...]) | Used to query a region around a known identifier or given coordinates from the catalog. |

Continued on next page

Table 4 – continued from previous page

| | |
|---|---|
| query_region_async(coordinates[, radius, ...]) | Serves the same purpose as query_region. |

### Attributes Documentation

**ARCHIVE_URL = 'http://surveys.roe.ac.uk:8080/wsa/ImageList'**

**BASE_URL = 'http://surveys.roe.ac.uk:8080/wsa/'**

**IMAGE_URL = 'http://surveys.roe.ac.uk:8080/wsa/GetImage'**

**LOGIN_URL = 'http://surveys.roe.ac.uk:8080/wsa/DBLogin'**

**REGION_URL = 'http://surveys.roe.ac.uk:8080/wsa/WSASQL'**

**TIMEOUT = 30**

**all_databases = ('UKIDSSDR9PLUS', 'UKIDSSDR8PLUS', 'UKIDSSDR7PLUS', 'UKIDSSDR6PLUS', 'UKIDSSDR5PLUS', 'U**

**filters = {'Br': 7, 'H': 4, 'H2': 6, 'J': 3, 'K': 5, 'Y': 2, 'Z': 1, 'all': 'all'}**

**frame_types = {'all': 'all', 'confidence': 'conf', 'deep_stack': 'deep%stack', 'difference': 'diff', 'in**

**ukidss_programmes_long = {'Deep Extragalactic Survey': 104, 'Galactic Clusters Survey': 103, 'Galactic P**

**ukidss_programmes_short = {'DXS': 104, 'GCS': 103, 'GPS': 102, 'LAS': 101, 'UDS': 105}**

### Methods Documentation

**extract_urls**(*html_in*)

Helper function that uses regexps to extract the image urls from the given HTML.

**Parameters**

**html_in** : str

source from which the urls are to be extracted.

**Returns**

**links** : list

The list of URLS extracted from the input.

**get_image_list**(***kwargs*)

**get_images**(*coordinates*, *waveband='all'*, *frame_type='stack'*, *image_width=<Quantity 1. arcmin>*, *image_height=None*, *radius=None*, *database='UKIDSSDR7PLUS'*, *programme_id='all'*, *verbose=True*, *get_query_payload=False*, *show_progress=True*)

Get an image around a target/ coordinates from UKIDSS catalog.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**waveband** : str

The color filter to download. Must be one of `'all'`, `'J'`, `'H'`, `'K'`, `'H2'`, `'Z'`, `'Y'`, `'Br'`].

**frame_type** : str

The type of image. Must be one of `'stack'`, `'normal'`, `'interleave'`, `'deep_stack'`, `'confidence'`, `'difference'`, `'leavstack'`, `'all'`]

**image_width** : str or `Quantity` object, optional

The image size (along X). Cannot exceed 15 arcmin. If missing, defaults to 1 arcmin.

**image_height** : str or `Quantity` object, optional

The image size (along Y). Cannot exceed 90 arcmin. If missing, same as image_width.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. When missing only image around the given position rather than multi-frames are retrieved.

**programme_id** : str

The survey or programme in which to search for.

**database** : str

The UKIDSS database to use.

**verbose** : bool

Defaults to `True`. When `True` prints additional messages.

**get_query_payload** : bool, optional

If `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**Returns**

**list** : A list of `HDUList` objects.

**get_images_async**(*coordinates*, *waveband='all'*, *frame_type='stack'*, *image_width=<Quantity 1. arcmin>*, *image_height=None*, *radius=None*, *database='UKIDSSDR7PLUS'*, *programme_id='all'*, *verbose=True*, *get_query_payload=False*, *show_progress=True*)

Serves the same purpose as `get_images` but returns a list of file handlers to remote files.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**waveband** : str

The color filter to download. Must be one of 'all', 'J', 'H', 'K', 'H2', 'Z', 'Y', 'Br'].

**frame_type** : str

The type of image. Must be one of 'stack', 'normal', 'interleave', 'deep_stack', 'confidence', 'difference', 'leavstack', 'all']

**image_width** : str or Quantity object, optional

The image size (along X). Cannot exceed 15 arcmin. If missing, defaults to 1 arcmin.

**image_height** : str or Quantity object, optional

The image size (along Y). Cannot exceed 90 arcmin. If missing, same as image_width.

**radius** : str or Quantity object, optional

The string must be parsable by Angle. The appropriate Quantity object from astropy.units may also be used. When missing only image around the given position rather than multi-frames are retrieved.

**programme_id** : str

The survey or programme in which to search for. See list_catalogs.

**database** : str

The UKIDSS database to use.

**verbose** : bool

Defaults to True. When True prints additional messages.

**get_query_payload** : bool, optional

If True then returns the dictionary sent as the HTTP request. Defaults to False.

**Returns**

**list** : list

A list of context-managers that yield readable file-like objects.

**list_catalogs**(*style='short'*)

Returns a list of available catalogs in UKIDSS. These can be used as programme_id in queries.

**Parameters**

**style** : str, optional

Must be one of 'short', 'long'. Defaults to 'short'. Determines whether to print long names or abbreviations for catalogs.

**Returns**

**list** : list containing catalog name strings in long or short style.

**list_databases**()

List the databases available from the UKIDSS WFCAM archive.

**logged_in**()

Determine whether currently logged in.

**query_region**(*coordinates, radius=<Quantity 1. arcmin>, programme_id='GPS', database='UKIDSSDR7PLUS', verbose=False, get_query_payload=False, system='J2000'*)

Used to query a region around a known identifier or given coordinates from the catalog.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. When missing defaults to 1 arcmin. Cannot exceed 90 arcmin.

**programme_id** : str

The survey or programme in which to search for. See `list_catalogs`.

**database** : str

The UKIDSS database to use.

**verbose** : bool, optional.

When set to `True` displays warnings if the returned VOTable does not conform to the standard. Defaults to `False`.

**get_query_payload** : bool, optional

If `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**system** : 'J2000' or 'Galactic'

The system in which to perform the query. Can affect the output data columns.

**Returns**

**result** : `Table`

Query result table.

**query_region_async**(*coordinates*, *radius=<Quantity 1. arcmin>*, *programme_id='GPS'*, *database='UKIDSSDR7PLUS'*, *get_query_payload=False*, *system='J2000'*)

Serves the same purpose as `query_region`. But returns the raw HTTP response rather than the parsed result.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. When missing defaults to 1 arcmin. Cannot exceed 90 arcmin.

**programme_id** : str

The survey or programme in which to search for. See `list_catalogs`.

**database** : str

The UKIDSS database to use.

**get_query_payload** : bool, optional

If `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service.

## Conf

**class** `astroquery.ukidss.`**`Conf`**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.ukidss`.

### Attributes Summary

| | |
|---|---|
| `server` | Name of the UKIDSS mirror to use. |
| `timeout` | Time limit for connecting to UKIDSS server. |

### Attributes Documentation

**`server`**

Name of the UKIDSS mirror to use.

**`timeout`**

Time limit for connecting to UKIDSS server.

CHAPTER 15

# MAGPIS Queries (astroquery.magpis)

## 15.1 Getting started

This module may be used to fetch image cutouts in the FITS format from various MAGPIS surveys. The only required parameter is the target you wish to search for. This may be specified as a name - which is resolved online via astropy functions or as coordinates using any of the coordinate systems available in astropy.coordinates. The FITS image is returned as an HDUList object. Here is a sample query:

```
>>> from astroquery.magpis import Magpis
>>> from astropy import coordinates
>>> from astropy import units as u
>>> image = Magpis.get_images(coordinates.SkyCoord(10.5*u.deg, 0.0*u.deg,
...                                               frame='galactic'))
>>> image

[<astropy.io.fits.hdu.image.PrimaryHDU at 0x4008650>]
```

There are some other optional parameters that you may additionally specify. For instance the image size may be specified by setting the image_size parameter. It defaults to 1 arcmin, but may be set to any other value using the appropriate Quantity object.

You may also specify the MAGPIS survey from which to fetch the cutout via the keyword survey. To know the list of valid surveys:

```
>>> from astroquery.magpis import Magpis
>>> Magpis.list_surveys()

   ['gps6epoch3',
    'gps6epoch4',
    'gps20',
    'gps20new',
    'gps90',
    'gpsmsx',
    'gpsmsx2',
```

(continues on next page)

(continued from previous page)

```
    'gpsglimpse36',
    'gpsglimpse45',
    'gpsglimpse58',
    'gpsglimpse80',
    'mipsgal',
    'bolocam']
```

The default survey used is 'bolocam'. Here is a query setting these optional parameters as well.

```
>>> from astroquery.magpis import Magpis
>>> import astropy.units as u
>>> import astropy.coordinates as coord
>>> image = Magpis.get_images(coordinates.SkyCoord(10.5*u.deg, 0.0*u.deg,
...                                                 frame='galactic'),
...                                                 image_size=10*u.arcmin,
...                                                 survey='gps20new')
>>> image

[<astropy.io.fits.hdu.image.PrimaryHDU at 0x4013e10>]
```

## 15.2 Reference/API

### 15.2.1 astroquery.magpis Package

#### MAGPIS Image and Catalog Query Tool

**Revision History**

Refactored using common API as a part of Google Summer of Code 2013.

**Originally contributed by**
Adam Ginsburg (adam.g.ginsburg@gmail.com)

#### Classes

| | |
|---|---|
| MagpisClass() | |
| Conf | Configuration parameters for astroquery.magpis. |

#### MagpisClass

**class** astroquery.magpis.**MagpisClass**

Bases: astroquery.query.BaseQuery

##### Attributes Summary

| |
|---|
| TIMEOUT |
| URL |
| maximsize |
| surveys |

### Methods Summary

| | |
|---|---|
| get_images(coordinates[, image_size, ...]) | Fetches image cutouts from MAGPIS surveys. |
| get_images_async(coordinates[, image_size, ...]) | Fetches image cutouts from MAGPIS surveys. |
| list_surveys() | Return a list of surveys for MAGPIS |

### Attributes Documentation

**TIMEOUT = 60**

**URL = 'https://third.ucllnl.org/cgi-bin/gpscutout'**

**maximsize = 1024**

**surveys = ['gps6', 'gps6epoch2', 'gps6epoch3', 'gps6epoch4', 'gps20', 'gps20new', 'gps90', 'gpsmsx', 'gp**

### Methods Documentation

**get_images**(*coordinates*, *image_size=<Quantity 1. arcmin>*, *survey='bolocam'*, *get_query_payload=False*)

Fetches image cutouts from MAGPIS surveys.

**Parameters**

**coordinates** : str or astropy.coordinates object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate astropy.coordinates object. ICRS coordinates may also be entered as strings as specified in the astropy.coordinates module.

**image_size** : str or Quantity object, optional

The string must be parsable by astropy.coordinates.Angle. The appropriate Quantity object from astropy.units may also be used. Specifies the symmetric size of the image. Defaults to 1 arcmin.

**survey** : str, optional

The MAGPIS survey you want to cut out. Defaults to 'bolocam'. The other surveys that can be used can be listed via list_surveys().

**maximsize** : int, optional

Specify the maximum image size (in pixels on each dimension) that will be returned. Max is 2048.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**Returns**

A list of `HDUList` objects

**get_images_async**(*coordinates*, *image_size=<Quantity 1. arcmin>*, *survey='bolocam'*, *get_query_payload=False*)

Fetches image cutouts from MAGPIS surveys.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**image_size** : str or `Quantity` object, optional

The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Specifies the symmetric size of the image. Defaults to 1 arcmin.

**survey** : str, optional

The MAGPIS survey you want to cut out. Defaults to 'bolocam'. The other surveys that can be used can be listed via `list_surveys()`.

**maximsize** : int, optional

Specify the maximum image size (in pixels on each dimension) that will be returned. Max is 2048.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service

**list_surveys**()

Return a list of surveys for MAGPIS

## Conf

**class** astroquery.magpis.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.magpis`.

### Attributes Summary

| | |
|---|---|
| `server` | Name of the MAGPIS server. |
| `timeout` | Time limit for connecting to MAGPIS server. |

### Attributes Documentation

**server**
Name of the MAGPIS server.

**timeout**
Time limit for connecting to MAGPIS server.

CHAPTER 16

# NRAO Queries (astroquery.nrao)

## 16.1 Getting started

This module supports fetching the table of observation summaries from the NRAO data archive. The results are returned in a Table. The service can be queried using the query_region(). The only required argument to this is the target around which to query. This may be specified either by using the identifier name directly - this is resolved via astropy functions using online services. The coordinates may also be specified directly using the appropriate coordinate system from astropy.coordinates. Here is a basic example:

```
>>> from astroquery.nrao import Nrao
>>> import astropy.coordinates as coord
>>> result_table = Nrao.query_region("04h33m11.1s 05d21m15.5s")
>>> print(result_table)

  Source     Project     Start Time Stop Time ...  RA DEC ARCH_FILE_ID
---------- ------------ ---------- --------- ... --- --- ------------
  0430+052  SRAM-public         --        -- ...  --  --    181927539
  0430+052  SRAM-public         --        -- ...  --  --    181927647
  0430+052  SRAM-public         --        -- ...  --  --    181927705
     3C120  BALI-public         --        -- ...  --  --    181927008
     3C120  BALI-public         --        -- ...  --  --    181927008
     3C120  BALI-public         --        -- ...  --  --    181927010
     3C120  BALI-public         --        -- ...  --  --    181927016
     3C120  BALI-public         --        -- ...  --  --    181927024
       ...          ...        ...       ... ... ... ...          ...
J0433+0521 13A-281-lock         --        -- ...  --  --    424632771
J0433+0521 13A-281-lock         --        -- ...  --  --    424632771
J0433+0521 13A-281-lock         --        -- ...  --  --    424632771
J0433+0521 13A-281-lock         --        -- ...  --  --    424632771
J0433+0521 13A-281-lock         --        -- ...  --  --    424632771
J0433+0521 13A-281-lock         --        -- ...  --  --    424632771
J0433+0521 13A-281-lock         --        -- ...  --  --    424632771
```

### 16.1.1 More detailed parameters

There are several other optional parameters that may also be specified. For instance the radius may be specified via Quantity object or a string acceptable be Angle. By default this is set to 1 degree. equinox may be set to 'J2000' or 'B1950' for equatorial systems, the default being 'J2000'. You can also specify the telescope from which to fetch the observations. This can be one of the following.

```
'gbt' 'all' 'historical_vla' 'vlba' 'jansky_vla'
```

Another parameter is the telescope_config. Valid values are

```
'all' 'A' 'AB' 'BnA' 'B' 'BC' 'CnB' 'C' 'CD' 'DnC' 'D'  'DA'
```

You may also specify the range of frequencies for the observation by specifying the freq_low and freq_up in appropriate units of frequency via astropy.units. The other optional parameters are the sub_array which may be set to 'all' or any value from 1 to 5. Finally you may also set the frequency bands for observation

```
'all' '4' 'P' 'L' 'S'  'C' 'X' 'U' 'K' 'Ka' 'Q' 'W'
```

Here's an example with all these optional parameters.

```
>>> from astroquery.nrao import Nrao
>>> import astropy.units as u
>>> import astropy.coordinates as coord
>>> result_table = Nrao.query_region(coord.SkyCoord(68.29625,
... 5.35431,  unit=(u.deg, u.deg), frame='icrs'), radius=2*u.arcmin,
... telescope='historical_vla', start_date='1985-06-30 18:16:49',
... end_date='1985-06-30 18:20:19', freq_low=1600*u.MHz, freq_up=1700*u.MHz,
... telescope_config='BC', sub_array=1)
>>> print(result_table)

Source      Project    Start Time Stop Time ...  RA DEC ARCH_FILE_ID
 -------- ------------- ---------- --------- ... --- --- ------------
 0430+052 AR0122-public         --        -- ...  --  --    181888822
 0430+052 AR0122-public         --        -- ...  --  --    181888822
```

## 16.2 Reference/API

### 16.2.1 astroquery.nrao Package

Module to query the NRAO Data Archive for observation summaries.

#### Classes

| | |
|---|---|
| NraoClass() | |
| Conf | Configuration parameters for astroquery.nrao. |

#### NraoClass

**class** astroquery.nrao.**NraoClass**

Bases: astroquery.query.QueryWithLogin

#### Attributes Summary

| | |
|---|---|
| DATA_URL | |
| TIMEOUT | |
| USERNAME | |
| obs_bands | |
| subarrays | |
| telescope_code | |
| telescope_config | |

#### Methods Summary

| | |
|---|---|
| query(*args, **kwargs) | Queries the service and returns a table object. |
| query_async([get_query_payload, cache, retry]) | Queries the NRAO data archive and fetches table of observation summaries. |
| query_region(*args, **kwargs) | Queries the service and returns a table object. |
| query_region_async(coordinates[, radius, …]) | Queries the NRAO data archive and fetches table of observation summaries. |

#### Attributes Documentation

**DATA_URL = 'https://archive.nrao.edu/archive/ArchiveQuery'**

**TIMEOUT = 60**

**USERNAME = ''**

**obs_bands = ['ALL', 'all', '4', 'P', 'L', 'S', 'C', 'X', 'U', 'K', 'Ka', 'Q', 'W']**

**subarrays = ['ALL', 1, 2, 3, 4, 5]**

**telescope_code = {'all': 'ALL', 'gbt': 'GBT', 'historical_vla': 'VLA', 'jansky_vla': 'EVLA', 'vlba': 'VI**

**telescope_config = ['ALL', 'A', 'AB', 'BnA', 'B', 'BC', 'CnB', 'C', 'CD', 'DnC', 'D', 'DA']**

#### Methods Documentation

**query**(**args*, ***kwargs*)

Queries the service and returns a table object.

Queries the NRAO data archive and fetches table of observation summaries.

**Parameters**

**coordinates** : str or astropy.coordinates object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate astropy.coordinates object. ICRS coordinates may also be entered as a string.

**radius** : str or Quantity object, optional

The string must be parsable by astropy.coordinates.Angle. The appropriate Quantity object may also be used. Defaults to 1 arcminute.

**equinox** : str, optional

One of 'J2000' or 'B1950'. Defaults to 'J2000'.

**telescope** : str, optional

The telescope that produced the data. Defaults to 'all'. Valid values are: ['gbt', 'all', 'historical_vla', 'vlba', 'jansky_vla']

**start_date** : str, optional

The starting date and time of the observations , e.g. 2010-06-21 14:20:30 Decimal seconds are not allowed. Defaults to None for no constraints.

**end_date** : str, optional

The ending date and time of the observations , e.g. 2010-06-21 14:20:30 Decimal seconds are not allowed. Defaults to None for no constraints.

**freq_low** : Quantity object, optional

The lower frequency of the observations in proper units of frequency via astropy.units. Defaults to None for no constraints.

**freq_up** : Quantity object, optional

The upper frequency of the observations in proper units of frequency via astropy.units. Defaults to None for no constraints.

**telescope_config** : str, optional

Select the telescope configuration (only valid for VLA array). Defaults to 'all'. Valid values are ['all', 'A', 'AB', 'BnA', 'B', 'BC', 'CnB', 'C', 'CD', 'DnC', 'D', 'DA']

**obs_band** : str, optional

The frequency bands for the observation. Defaults to 'all'. Valid values are ['all', '4', 'P', 'L', 'S', 'C', 'X', 'U', 'K', 'Ka', 'Q', 'W'].

**sub_array** : str, number, optional

VLA subarray designations, may be set to an integer from 1 to 5. Defaults to 'all'.

**project_code** : str, optional

A string indicating the project code. Examples:

```
* GBT: AGBT12A_055
* JVLA: 12A-256
```

**querytype** : str

The type of query to perform. "OBSSUMMARY" is the default, but it is only valid for VLA/VLBA observations. ARCHIVE will give the list of files available for download. OBSERVATION will provide full details of the sources observed and under what configurations.

**source_id** : str, optional

A source name (to be parsed by SIMBAD or NED)

**protocol** : 'VOTable-XML' or 'HTML'

The type of table to return. In theory, this should not matter, but in practice the different table types actually have different content. For `querytype='ARCHIVE'`, the protocol will be force to HTML because the archive doesn't support votable returns for archive queries.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**cache** : bool

Cache the query results

**retry** : bool or int

The number of times to retry querying the server if it doesn't raise an exception but returns a null result (this sort of behavior seems unique to the NRAO archive)

**Returns**

**table** : A `Table` object.

**query_async**(*get_query_payload=False*, *cache=True*, *retry=False*, ***kwargs*)

Queries the NRAO data archive and fetches table of observation summaries.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as a string.

**radius** : str or `Quantity` object, optional

The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object may also be used. Defaults to 1 arcminute.

**equinox** : str, optional

One of 'J2000' or 'B1950'. Defaults to 'J2000'.

**telescope** : str, optional

The telescope that produced the data. Defaults to 'all'. Valid values are: ['gbt', 'all', 'historical_vla', 'vlba', 'jansky_vla']

**start_date** : str, optional

The starting date and time of the observations , e.g. 2010-06-21 14:20:30 Decimal seconds are not allowed. Defaults to `None` for no constraints.

**end_date** : str, optional

The ending date and time of the observations , e.g. 2010-06-21 14:20:30 Decimal seconds are not allowed. Defaults to `None` for no constraints.

**freq_low** : `Quantity` object, optional

The lower frequency of the observations in proper units of frequency via `astropy.units`. Defaults to `None` for no constraints.

**freq_up** : `Quantity` object, optional

The upper frequency of the observations in proper units of frequency via `astropy.units`. Defaults to `None` for no constraints.

**telescope_config** : str, optional

Select the telescope configuration (only valid for VLA array). Defaults to 'all'. Valid values are ['all', 'A', 'AB', 'BnA', 'B', 'BC', 'CnB', 'C', 'CD', 'DnC', 'D', 'DA']

**obs_band** : str, optional

The frequency bands for the observation. Defaults to 'all'. Valid values are ['all', '4', 'P', 'L', 'S', 'C', 'X', 'U', 'K', 'Ka', 'Q', 'W'].

**sub_array** : str, number, optional

VLA subarray designations, may be set to an integer from 1 to 5. Defaults to 'all'.

**project_code** : str, optional

A string indicating the project code. Examples:

```
* GBT: AGBT12A_055
* JVLA: 12A-256
```

**querytype** : str

The type of query to perform. "OBSSUMMARY" is the default, but it is only valid for VLA/VLBA observations. ARCHIVE will give the list of files available for download. OBSERVATION will provide full details of the sources observed and under what configurations.

**source_id** : str, optional

A source name (to be parsed by SIMBAD or NED)

**protocol** : 'VOTable-XML' or 'HTML'

The type of table to return. In theory, this should not matter, but in practice the different table types actually have different content. For `querytype='ARCHIVE'`, the protocol will be force to HTML because the archive doesn't support votable returns for archive queries.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**cache** : bool

Cache the query results

**retry** : bool or int

The number of times to retry querying the server if it doesn't raise an exception but returns a null result (this sort of behavior seems unique to the NRAO archive)

**Returns**

**response** : `Response`

The HTTP response returned from the service.

**query_region**(**args*, ***kwargs*)

Queries the service and returns a table object.

Queries the NRAO data archive and fetches table of observation summaries.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as a string.

**radius** : str or `Quantity` object, optional

The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object may also be used. Defaults to 1 arcminute.

**equinox** : str, optional

One of 'J2000' or 'B1950'. Defaults to 'J2000'.

**telescope** : str, optional

The telescope that produced the data. Defaults to 'all'. Valid values are: ['gbt', 'all', 'historical_vla', 'vlba', 'jansky_vla']

**start_date** : str, optional

The starting date and time of the observations , e.g. 2010-06-21 14:20:30 Decimal seconds are not allowed. Defaults to `None` for no constraints.

**end_date** : str, optional

The ending date and time of the observations , e.g. 2010-06-21 14:20:30 Decimal seconds are not allowed. Defaults to `None` for no constraints.

**freq_low** : `Quantity` object, optional

The lower frequency of the observations in proper units of frequency via `astropy.units`. Defaults to `None` for no constraints.

**freq_up** : `Quantity` object, optional

The upper frequency of the observations in proper units of frequency via `astropy.units`. Defaults to `None` for no constraints.

**telescope_config** : str, optional

Select the telescope configuration (only valid for VLA array). Defaults to 'all'. Valid values are ['all', 'A', 'AB', 'BnA', 'B', 'BC', 'CnB', 'C', 'CD', 'DnC', 'D', 'DA']

**obs_band** : str, optional

The frequency bands for the observation. Defaults to 'all'. Valid values are ['all', '4', 'P', 'L', 'S', 'C', 'X', 'U', 'K', 'Ka', 'Q', 'W'].

**sub_array** : str, number, optional

VLA subarray designations, may be set to an integer from 1 to 5. Defaults to 'all'.

**project_code** : str, optional

A string indicating the project code. Examples:

```
* GBT: AGBT12A_055
* JVLA: 12A-256
```

**querytype** : str

The type of query to perform. "OBSSUMMARY" is the default, but it is only valid for VLA/VLBA observations. ARCHIVE will give the list of files available for download. OBSERVATION will provide full details of the sources observed and under what configurations.

**source_id** : str, optional

A source name (to be parsed by SIMBAD or NED)

**protocol** : 'VOTable-XML' or 'HTML'

The type of table to return. In theory, this should not matter, but in practice the different table types actually have different content. For `querytype='ARCHIVE'`, the protocol will be force to HTML because the archive doesn't support votable returns for archive queries.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**cache** : bool

Cache the query results

**retry** : bool or int

The number of times to retry querying the server if it doesn't raise an exception but returns a null result (this sort of behavior seems unique to the NRAO archive)

**Returns**

**table** : A `Table` object.

**query_region_async**(*coordinates*, *radius=<Quantity 1. arcmin>*, *equinox='J2000'*, *telescope='all'*, *start_date="*, *end_date="*, *freq_low=None*, *freq_up=None*, *telescope_config='all'*, *obs_band='all'*, *querytype='OBSSUMMARY'*, *sub_array='all'*, *project_code=None*, *protocol='VOTable-XML'*, *retry=False*, *get_query_payload=False*, *cache=True*)

Queries the NRAO data archive and fetches table of observation summaries.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as a string.

**radius** : str or `Quantity` object, optional

The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object may also be used. Defaults to 1 arcminute.

**equinox** : str, optional

One of 'J2000' or 'B1950'. Defaults to 'J2000'.

**telescope** : str, optional

The telescope that produced the data. Defaults to 'all'. Valid values are: ['gbt', 'all', 'historical_vla', 'vlba', 'jansky_vla']

**start_date** : str, optional

The starting date and time of the observations , e.g. 2010-06-21 14:20:30 Decimal seconds are not allowed. Defaults to `None` for no constraints.

**end_date** : str, optional

The ending date and time of the observations , e.g. 2010-06-21 14:20:30 Decimal seconds are not allowed. Defaults to `None` for no constraints.

**freq_low** : `Quantity` object, optional

The lower frequency of the observations in proper units of frequency via `astropy.units`. Defaults to `None` for no constraints.

**freq_up** : `Quantity` object, optional

The upper frequency of the observations in proper units of frequency via `astropy.units`. Defaults to `None` for no constraints.

**telescope_config** : str, optional

Select the telescope configuration (only valid for VLA array). Defaults to 'all'. Valid values are ['all', 'A', 'AB', 'BnA', 'B', 'BC', 'CnB', 'C', 'CD', 'DnC', 'D', 'DA']

**obs_band** : str, optional

The frequency bands for the observation. Defaults to 'all'. Valid values are ['all', '4', 'P', 'L', 'S', 'C', 'X', 'U', 'K', 'Ka', 'Q', 'W'].

**sub_array** : str, number, optional

VLA subarray designations, may be set to an integer from 1 to 5. Defaults to 'all'.

**project_code** : str, optional

A string indicating the project code. Examples:

```
* GBT: AGBT12A_055
* JVLA: 12A-256
```

**querytype** : str

The type of query to perform. "OBSSUMMARY" is the default, but it is only valid for VLA/VLBA observations. ARCHIVE will give the list of files available for download. OBSERVATION will provide full details of the sources observed and under what configurations.

**source_id** : str, optional

A source name (to be parsed by SIMBAD or NED)

**protocol** : 'VOTable-XML' or 'HTML'

The type of table to return. In theory, this should not matter, but in practice the different table types actually have different content. For `querytype='ARCHIVE'`, the protocol will be force to HTML because the archive doesn't support votable returns for archive queries.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`

**cache** : bool

Cache the query results

**retry** : bool or int

The number of times to retry querying the server if it doesn't raise an exception but returns a null result (this sort of behavior seems unique to the NRAO archive)

**Returns**

**response** : Response

The HTTP response returned from the service.

## Conf

class astroquery.nrao.**Conf**

Bases: astropy.config.ConfigNamespace

Configuration parameters for astroquery.nrao.

### Attributes Summary

| | |
|---|---|
| server | Name of the NRAO mirror to use. |
| timeout | Time limit for connecting to NRAO server. |
| username | Optional default username for ALMA archive. |

### Attributes Documentation

**server**
Name of the NRAO mirror to use.

**timeout**
Time limit for connecting to NRAO server.

**username**
Optional default username for ALMA archive.

# CHAPTER 17

## Besancon Queries (astroquery.besancon)

### 17.1 Getting started

The following example illustrates an Besancon catalog/image query. The API describes the relevant keyword arguments (of which there are many)

**Note:** These examples are improperly formatted because they are not safe to run generally! The Besancon servers are hosted by individuals and not mean to handle large or repeated requests, so we disable *all* automated testing.

A successful run should look something like this

```
>>> from astroquery.besancon import Besancon
>>> #besancon_model = Besancon.query(glon=10.5, glat=0.0, email='your@email.net')

Waiting 30s for model to finish (elapsed wait time 30s, total 32)
Loading page...
Waiting 30s for model to finish (elapsed wait time 60s, total 198)
Loading page...
Waiting 30s for model to finish (elapsed wait time 90s, total 362)
Loading page...
Waiting 30s for model to finish (elapsed wait time 120s, total 456)
Loading page...
Awaiting Besancon file...
Downloading ftp://sasftp.obs-besancon.fr/modele/modele2003/1407398212.212272.resu
|==========================================================================================================
↪2.5M/2.5M (100.00%)         0s

>>> # B.pprint()
 Dist  Mv   CL Typ  LTef logg Age Mass  J-K   J-H   V-K   H-K    K    [Fe/H]   l     b     Av   Mbol
----- ---- --- ---- ----- ---- --- ---- ----- ----- ------ ----- ------ ------ ----- ----- ----- ------
 0.87 12.5   5  7.5 3.515 4.99   5 0.24 0.966 0.649  5.408 0.318  17.44  -0.02 10.62 -0.38 0.637 10.844
 0.91 13.0   5  7.5 3.506 5.03   1 0.21 0.976 0.671  5.805 0.305 17.726   0.13 10.62 -0.38 0.669 11.312
```

(continues on next page)

(continued from previous page)

```
 0.97 18.5   5  7.9  3.39 5.32   7 0.08 0.804 0.634  8.634  0.17 20.518  -0.46 10.62 -0.38 0.716    0.0
 1.01 15.2   5  7.9 3.465 5.14   1 0.13 1.045 0.649  7.015 0.396 18.957   0.22 10.62 -0.38 0.748 13.353
 1.01 16.5   5  7.8 3.435 5.27   5 0.09 1.024 0.701  7.545 0.323 19.642  -0.09 10.62 -0.38 0.748    0.0
 1.03 17.0   5 7.85 3.424 5.29   1 0.09 1.133 0.701  8.132 0.432 19.631   0.07 10.62 -0.38 0.764    0.0
 1.09 13.5   5  7.6 3.497 5.05   7 0.18 0.995  0.69  5.829 0.305 18.629  -0.43 10.62 -0.38 0.812  11.78
 1.17 13.7   5 7.65 3.493  5.0   2 0.17 1.025  0.68  6.319 0.345 18.552    0.2 10.62 -0.38 0.876 11.948
 1.17 18.5   5  7.9  3.39 5.32   5 0.08 0.927  0.69  8.876 0.237 20.763  -0.27 10.62 -0.38 0.876    0.0
 1.25 20.0   5  7.9 3.353 5.36   3 0.08 1.533 0.883 10.202  0.65 21.215   0.15 10.62 -0.38 0.941    0.0
 1.29  9.3   5  7.1  3.58 4.74   2 0.56 0.997 0.777  4.592 0.219 16.263   0.21 10.62 -0.38 0.974  7.834
 1.29 13.5   6  9.0 3.853  8.0   7  0.6 0.349 0.283  1.779 0.066 23.266  -0.17 10.62 -0.38 0.974    0.0
 1.33  6.9   5  6.4 3.656 4.62   5 0.77 0.857  0.69  3.604 0.167 14.889   0.25 10.62 -0.38 1.006  5.795
 1.35  7.5   5  6.5 3.633 4.62   5  0.7 0.902 0.729  3.885 0.172  15.22   0.08 10.62 -0.38 1.023  6.225
  ...  ... ...  ...   ...  ... ...  ...   ...   ...    ...   ...    ...    ...   ...   ...   ...    ...
40.19 17.1   6  9.0 3.515 8.19   9  0.7 2.013 1.481 11.166 0.532 35.816  -1.99 10.62 -0.38  11.8    0.0
41.01 17.1   6  9.0 3.515 8.19   9  0.7 2.013 1.481 11.166 0.532 35.899  -2.14 10.62 -0.38  11.8    0.0
41.21 17.3   6  9.0 3.485 8.19   9  0.7 1.933 1.471 10.826 0.462 36.312  -1.36 10.62 -0.38  11.8    0.0
41.41 16.9   6  9.0 3.542 8.19   9  0.7 1.893 1.301 11.436 0.592 35.358  -0.92 10.62 -0.38  11.8    0.0
41.87 16.7   6  9.0 3.568 8.19   9  0.7 1.783 1.141 11.686 0.642 34.917  -1.79 10.62 -0.38  11.8    0.0
42.05 16.7   6  9.0 3.568 8.19   9  0.7 1.783 1.141 11.686 0.642 34.936  -2.06 10.62 -0.38  11.8    0.0
44.19 16.9   6  9.0 3.542 8.19   9  0.7 1.893 1.301 11.436 0.592 35.494  -3.04 10.62 -0.38  11.8    0.0
45.39 17.3   6  9.0 3.485 8.19   9  0.7 1.933 1.471 10.826 0.462 36.497  -1.28 10.62 -0.38  11.8    0.0
46.01 18.5   6  9.0 3.297 8.19   9  0.7 0.813 1.381  8.016 0.568 40.611  -2.01 10.62 -0.38  11.8    0.0
46.71 16.7   6  9.0 3.568 8.19   9  0.7 1.783 1.141 11.686 0.642 35.087  -2.59 10.62 -0.38  11.8    0.0
46.97 17.9   6  9.0 3.389 8.19   9  0.7 1.233 1.161  9.536 0.072 38.563  -1.42 10.62 -0.38  11.8    0.0
47.45 17.3   6  9.0 3.485 8.19   9  0.7 1.933 1.471 10.826 0.462 36.579  -2.25 10.62 -0.38  11.8    0.0
48.05  5.2   5 4.82 3.786 4.54   9 0.74  2.42 1.563 11.919 0.857 23.548  -1.45 10.62 -0.38  11.8   5.08
49.39 16.9   6  9.0 3.542 8.19   9  0.7 1.893 1.301 11.436 0.592 35.813  -1.19 10.62 -0.38  11.8    0.0
```

**Note:** These tests are commented out (and will fail) because I don't want to put unnecessary strain on the Besancon servers by running queries every time we test.

### 17.1.1 Reading a previously downloaded file

If you've downloaded a `.resu`, you can parse it with the custom parser in astroquery:

```
>>> from astroquery.besancon import parse_besancon_model_file
>>> tbl = parse_besancon_model_file('file.resu')
>>> tbl.pprint()
  Dist   Mv   CL Typ  LTef logg Age Mass  J-H   H-K   J-K   V-K     V    [Fe/H]   l     b     Av   Mbol
 ------ ---- --- --- ----- ---- --- ---- ----- ----- ----- ----- ------ ------ ----- ----- ----- ------
  0.091 10.2   5 7.2 3.559 4.85   7 0.48 0.601 0.223 0.824 4.175 15.133   0.02 10.62 -0.38 0.056  8.671
```

## 17.2 Reference/API

### 17.2.1 astroquery.besancon Package

#### Besancon Query Tool

A tool to query the Besancon model of the galaxy http://model.obs-besancon.fr/

**Author**
Adam Ginsburg (adam.g.ginsburg@gmail.com)

## Functions

| | |
|---|---|
| parse_besancon_model_file(filename) | Parse a besancon model from a file on disk |
| parse_besancon_model_string(bms) | Given an entire Besancon model result in *string* form, parse it into an Table. |

### parse_besancon_model_file

astroquery.besancon.**parse_besancon_model_file**(*filename*)
Parse a besancon model from a file on disk

### parse_besancon_model_string

astroquery.besancon.**parse_besancon_model_string**(*bms*)
Given an entire Besancon model result in *string* form, parse it into an Table.

## Classes

| | |
|---|---|
| BesanconClass([email]) | |
| Conf | Configuration parameters for astroquery.besancon. |

### BesanconClass

**class** astroquery.besancon.**BesanconClass**(*email=None*)
Bases: astroquery.query.BaseQuery

#### Attributes Summary

| |
|---|
| QUERY_URL |
| TIMEOUT |
| ping_delay |
| result_re |
| url_download |

#### Methods Summary

| | |
|---|---|
| get_besancon_model_file(filename[, verbose, …]) | Download a Besancon model from the website. |
| query(*args, **kwargs) | Queries the service and returns a table object. |
| query_async(*args, **kwargs) | Perform a query on the Besancon model of the galaxy. |

### Attributes Documentation

`QUERY_URL = 'http://model.obs-besancon.fr/modele_form.php'`

`TIMEOUT = 30.0`

`ping_delay = 30.0`

`result_re = re.compile('[0-9]{10}\\.[0-9]{6}\\.resu')`

`url_download = 'ftp://sasftp.obs-besancon.fr/modele/modele2003/'`

### Methods Documentation

`get_besancon_model_file`(*filename*, *verbose=True*, *timeout=5.0*)
Download a Besancon model from the website.

**Parameters**
**filename** : string

The besancon filename, with format ##########.######.resu

**verbose** : bool

Print details about the download process

**timeout** : float

Amount of time to wait after pinging the server to see if a file is present. Default 5s, which is probably reasonable.

`query`(**args*, ***kwargs*)
Queries the service and returns a table object.

Perform a query on the Besancon model of the galaxy.

http://model.obs-besancon.fr/

**Parameters**
**glon** : float

**glat** : float

Galactic latitude and longitude at the center

**email** : str

A valid e-mail address to send the report of completion to

**smallfield** : bool

Small field (True) or Large Field (False) LARGE FIELD NOT SUPPORTED YET

**extinction** : float

Extinction per kpc in A_V

**area** : float

Area in square degrees

**absmag_limits** : (float,float)

Absolute magnitude lower,upper limits

**colors_limits** : dict of (float,float)

**Should contain 4 elements listing color differences in the valid**
bands, e.g.: {"J-H":(99,-99),"H-K":(99,-99),"J-K":(99,-99),"V-K":(99,-99)}

**mag_limits = dict of (float,float)**

Lower and Upper magnitude difference limits for each magnitude band U B V R I J H K L

**clouds** : list of 2-tuples

Up to 25 line-of-sight clouds can be specified in pairs of (A_V, distance in pc)

**verbose** : bool

Print out extra error messages?

**kwargs** : dict

Can override any argument in the request if you know the name of the POST keyword.

**retrieve_file** : bool

If True, will try to retrieve the file every 30s until it shows up. Otherwise, just returns the filename (the job is still executed on the remote server, though)

**Returns**

**table** : A `Table` object.

**query_async**(**args*, ***kwargs*)

Perform a query on the Besancon model of the galaxy.

http://model.obs-besancon.fr/

**Parameters**

**glon** : float

**glat** : float

Galactic latitude and longitude at the center

**email** : str

A valid e-mail address to send the report of completion to

**smallfield** : bool

Small field (True) or Large Field (False) LARGE FIELD NOT SUPPORTED YET

**extinction** : float

Extinction per kpc in A_V

**area** : float

Area in square degrees

**absmag_limits** : (float,float)

Absolute magnitude lower,upper limits

**colors_limits** : dict of (float,float)

**Should contain 4 elements listing color differences in the valid**
bands, e.g.: {“J-H”:(99,-99),”H-K”:(99,-99),”J-K”:(99,-99),”V-K”:(99,-99)}

**mag_limits = dict of (float,float)**

Lower and Upper magnitude difference limits for each magnitude band U B V R I J H K L

**clouds** : list of 2-tuples

Up to 25 line-of-sight clouds can be specified in pairs of (A_V, distance in pc)

**verbose** : bool

Print out extra error messages?

**kwargs** : dict

Can override any argument in the request if you know the name of the POST keyword.

**retrieve_file** : bool

If True, will try to retrieve the file every 30s until it shows up. Otherwise, just returns the filename (the job is still executed on the remote server, though)

**Returns**

**response** : `requests.Response` object

The response of the HTTP request.

## Conf

**class** `astroquery.besancon.Conf`

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.besancon`.

### Attributes Summary

| | |
|---|---|
| `download_url` | Besancon download URL. |
| `model_form` | Besancon model form URL |
| `ping_delay` | Amount of time before pinging the Besancon server to see if the file is ready. |
| `timeout` | Timeout for Besancon query |

### Attributes Documentation

**download_url**

Besancon download URL. Changed to modele2003 in 2013.

**model_form**

Besancon model form URL

**ping_delay**

Amount of time before pinging the Besancon server to see if the file is ready. Minimum 30s.

**timeout**

Timeout for Besancon query

# CHAPTER 18

## NIST Queries (astroquery.nist)

Tool to query the NIST Atomic Lines database (http://physics.nist.gov/cgi-bin/ASD/lines1.pl).

### 18.1 Getting started

This is a relatively simple module that you may use to query spectra from NIST. All the results are returned as a Table. To do this you just need to specify the lower and the upper wavelength for the spectrum you want to fetch. These wavelengths must be specified as an appropriate Quantity object, for instance having units of nanometer, or angstrom or the like. For example, to use a lower wavelength value of 4000 Angstroms, you should use `` `4000 * u.AA` `` and if you want the same in nanometers, just use `` `400 * u.nm` ``. Of course there are several optional parameters you can also specify. For instance use the `linename` parameter to specify the spectrum you wish to fetch. By default this is set to “H I”, but you can set it to several other values like “Na;Mg”, etc. Lets now see a simple example.

```
>>> from astroquery.nist import Nist
>>> import astropy.units as u
>>> table = Nist.query(4000 * u.AA, 7000 * u.AA, linename="H I")
>>> print(table)

    Observed         Ritz       Rel.      Aki      Acc. ...      Lower level          Upper level      Type ␣
↪TP    Line
 ------------- ------------- ------ ---------- ---- ... ------------------- ------------------- ---- ---
↪-- --------
            -- 4102.85985516     --  4287700.0  AAA ...                  --                  --   -- ␣
↪--        --
            -- 4102.86191086     --   245010.0  AAA ... 2p      | 2P*  | 1/2 6s      | 2S   | 1/2  --␣
↪T8637       --
            --     4102.8632     --         --   -- ...            |      |             |      |   -- ␣
↪--       c57
 4102.86503481 4102.86503481     --         --   -- ... 2s      | 2S   | 1/2 6d      | 2D   | 5/2   E2 ␣
↪--   L11759
            -- 4102.86579134     --  2858300.0  AAA ...                  --                  --   -- ␣
↪--        --
 4102.86785074 4102.86785074     --         --   -- ... 2s      | 2S   | 1/2 6s      | 2S   | 1/2   M1 ␣
↪--   L11759
```

(continues on next page)

(continued from previous page)

```
            -- 4102.86807252     --  2858400.0  AAA ...                  --                   --   --   ␣
↪--         --
      4102.892     4102.8991  70000   973200.0  AAA ...     2      |      |      6      |      |   --␣
↪T8637 L7436c29
            --     4102.8922     --         --   -- ...              |      |             |      |   --   ␣
↪--        c58
            -- 4102.92068748     --  5145000.0  AAA ... 2p      | 2P*  | 3/2 6d      | 2D   | 5/2   --␣
↪T8637        --
            --     4102.9208     --         --   -- ...              |      |             |      |   --   ␣
↪--        c59
            -- 4102.92144772     --   857480.0  AAA ...                  --                   --   --   ␣
↪--         --
            -- 4102.92350348     --   490060.0  AAA ... 2p      | 2P*  | 3/2 6s      | 2S   | 1/2   --␣
↪T8637        --
            --   4341.647191     --  7854800.0  AAA ... 2p      | 2P*  | 1/2 5d      | 2D   | 3/2   --␣
↪T8637        --
            --     4341.6512     --         --   -- ...              |      |             |      |   --   ␣
↪--        c60
           ...           ...    ...        ...  ... ...                 ...                  ...  ...   .
↪..        ...
   6564.522552   6564.522555     -- 53877000.0  AAA ... 2p      | 2P*  | 1/2 3d      | 2D   | 3/2   --␣
↪T8637     L2752
            --      6564.527     --         --   -- ...              |      |             |      |   --   ␣
↪--        c67
            --      6564.535     --         --   -- ...              |      |             |      |   --   ␣
↪--        c68
   6564.537684   6564.537684     -- 22448000.0  AAA ... 2s      | 2S   | 1/2 3p      | 2P*  | 3/2   --␣
↪T8637 L6891c38
            --   6564.564672     --  2104600.0  AAA ... 2p      | 2P*  | 1/2 3s      | 2S   | 1/2   --␣
↪T8637        --
            --   6564.579878     --         --   -- ... 2s      | 2S   | 1/2 3s      | 2S   | 1/2   M1   ␣
↪--         --
            --      6564.583     --         --   -- ...              |      |             |      |   --   ␣
↪--        c66
   6564.584404   6564.584403     -- 22449000.0  AAA ... 2s      | 2S   | 1/2 3p      | 2P*  | 1/2   --␣
↪T8637 L6891c38
        6564.6      6564.632 500000 44101000.0  AAA ...     2      |      |      3      |      |   --␣
↪T8637 L7400c29
            --      6564.608     --         --   -- ...              |      |             |      |   --   ␣
↪--        c69
    6564.66464    6564.66466     -- 64651000.0  AAA ... 2p      | 2P*  | 3/2 3d      | 2D   | 5/2   --␣
↪T8637     L2752
            --     6564.6662     --         --   -- ...              |      |             |      |   --   ␣
↪--        c71
            --      6564.667     --         --   -- ...              |      |             |      |   --   ␣
↪--        c70
            --   6564.680232     -- 10775000.0  AAA ... 2p      | 2P*  | 3/2 3d      | 2D   | 3/2   --␣
↪T8637        --
            --   6564.722349     --  4209700.0  AAA ... 2p      | 2P*  | 3/2 3s      | 2S   | 1/2   --␣
↪T8637        --
```

Note that using a different unit will result in different output units in the `Observed` and `Ritz` columns.

There are several other optional parameters that you may also set. For instance you may set the `energy_level_unit` to any one of these values.

```
'R' 'Rydberg' 'rydberg' 'cm' 'cm-1' 'EV' 'eV' 'electronvolt' 'ev' 'invcm'
```

Similarly you can set the `output_order` to any one of 'wavelength' or 'multiplet'. A final parameter you may also set is the `wavelength_type` to one of 'vacuum' or 'vac+air'. Here is an example with all these parameters.

```
>>> from astroquery.nist import Nist
>>> table = Nist.query(4000 * u.nm, 7000 * u.nm, 'H I',
...                    energy_level_unit='eV', output_order='wavelength',
...                    wavelength_type='vacuum')
>>> print(table)

    Observed     Ritz     Rel.    Aki     ...     Upper level     Type   TP  Line
-------- ----------- ----- --------- ... ------------------- ---- ----- ----
      --    4020.871 (200)    5526.5 ...                  --    --    --   --
      --  4052.18664    -- 1238100.0 ... 5d      | 2D     | 3/2   -- T8637   --
      --  4052.19376    --  737160.0 ... 5p      | 2P*    | 3/2   -- T8637   --
      --  4052.22121    --  215030.0 ...                  --    --    --   --
      --  4052.23222    --  737210.0 ... 5p      | 2P*    | 1/2   -- T8637   --
      -- 4052.248747    -- 2412100.0 ...                  --    --    --   --
      --  4052.24892    -- 1485800.0 ...                  --    --    --   --
      --  4052.26147    --   18846.0 ... 5p      | 2P*    | 3/2   -- T8637   --
     ...         ...   ...       ... ...                 ...   ...   ...  ...
      --     5525.19 (150)    2470.9 ...                  --    --    --   --
      --    5711.464 (180)    3515.8 ...                  --    --    --   --
      --     5908.22 (540)   70652.0 ...     9       |      |   -- T8637   --
      --    5956.845 (210)    5156.2 ...                  --    --    --   --
      --    6291.918 (250)    7845.7 ...                  --    --    --   --
      --    6771.993 (300)   12503.0 ...     12      |      |   -- T8637   --
      --    6946.756    --    688.58 ...                  --    --    --   --
```

## 18.2 Reference/API

### 18.2.1 astroquery.nist Package

Fetches line spectra from the NIST Atomic Spectra Database.

#### Classes

| | |
|---|---|
| `NistClass`() | |
| `Conf` | Configuration parameters for `astroquery.nist`. |

#### NistClass

**class** astroquery.nist.**NistClass**

Bases: `astroquery.query.BaseQuery`

##### Attributes Summary

| |
|---|
| TIMEOUT |
| URL |
| energy_level_code |
| order_out_code |
| unit_code |
| wavelength_unit_code |

### Methods Summary

| | |
|---|---|
| query(*args, **kwargs) | Queries the service and returns a table object. |
| query_async(minwav, maxwav[, linename, …]) | Serves the same purpose as query but returns the raw HTTP response rather than a Table object. |

### Attributes Documentation

**TIMEOUT = 30**

**URL = 'http://physics.nist.gov/cgi-bin/ASD/lines1.pl'**

**energy_level_code = {'EV': 1, 'R': 2, 'Rydberg': 2, 'cm': 0, 'cm-1': 0, 'eV': 1, 'electronvolt': 1, 'ev**

**order_out_code = {'multiplet': 1, 'wavelength': 0}**

**unit_code = {'Angstrom': 0, 'nm': 1, 'um': 2}**

**wavelength_unit_code = {'vac+air': 4, 'vacuum': 3}**

### Methods Documentation

**query**(**args*, ***kwargs*)

Queries the service and returns a table object.

Serves the same purpose as query but returns the raw HTTP response rather than a Table object.

**Parameters**

**minwav** : astropy.units.Quantity object

The lower wavelength for the spectrum in appropriate units.

**maxwav** : astropy.units.Quantity object

The upper wavelength for the spectrum in appropriate units.

**linename** : str, optional

The spectrum to fetch. Defaults to "H I"

**energy_level_unit** : str, optional

The energy level units must be one of the following: 'R', 'Rydberg', 'rydberg', 'cm', 'cm-1', 'EV', 'eV', 'electronvolt', 'ev', 'invcm' Defaults to 'eV'.

**output_order** : str, optional

Decide ordering of output. Must be one of following: [‘wavelength’, ‘multiplet’]. Defaults to ‘wavelength’.

**wavelength_type** : str, optional

Must be one of ‘vacuum’ or ‘vac+air’. Defaults to ‘vacuum’.

**get_query_payload** : bool, optional

If true then returns the dictionary of query parameters, posted to remote server. Defaults to `False`.

**Returns**

**table** : A `Table` object.

**query_async**(*minwav*, *maxwav*, *linename='H I'*, *energy_level_unit='eV'*, *output_order='wavelength'*, *wavelength_type='vacuum'*, *get_query_payload=False*)

Serves the same purpose as `query` but returns the raw HTTP response rather than a `Table` object.

**Parameters**

**minwav** : `astropy.units.Quantity` object

The lower wavelength for the spectrum in appropriate units.

**maxwav** : `astropy.units.Quantity` object

The upper wavelength for the spectrum in appropriate units.

**linename** : str, optional

The spectrum to fetch. Defaults to “H I”

**energy_level_unit** : str, optional

The energy level units must be one of the following: ‘R’, ‘Rydberg’, ‘rydberg’, ‘cm’, ‘cm-1’, ‘EV’, ‘eV’, ‘electronvolt’, ‘ev’, ‘invcm’ Defaults to ‘eV’.

**output_order** : str, optional

Decide ordering of output. Must be one of following: [‘wavelength’, ‘multiplet’]. Defaults to ‘wavelength’.

**wavelength_type** : str, optional

Must be one of ‘vacuum’ or ‘vac+air’. Defaults to ‘vacuum’.

**get_query_payload** : bool, optional

If true then returns the dictionary of query parameters, posted to remote server. Defaults to `False`.

**Returns**

**response** : `requests.Response` object

The response of the HTTP request.

## Conf

**class** astroquery.nist.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.nist`.

#### Attributes Summary

| | |
|---|---|
| `server` | Name of the NIST URL to query. |
| `timeout` | Time limit for connecting to NIST server. |

#### Attributes Documentation

**server**
Name of the NIST URL to query.

**timeout**
Time limit for connecting to NIST server.

# CHAPTER 19

## NVAS Queries (astroquery.nvas)

### 19.1 Getting started

This module may be used to retrieve the NVAS VLA archive images. All images are returned as a list of HDUList objects. Images may be fetched by specifying directly an object name around which to search - in this case the name will be resolved to coordinates by astropy name resolving methods that use online services like SESAME. The search centre may also be entered as a coordinate using any coordinate system from astropy.coordinates. ICRS coordinates can also be entered directly as a string that conforms to the format specified by astropy.coordinates. Some other parameters you may optionally specify are the radius and the frequency band for which the image must be fetched. You can also specify the maximum allowable noise level in mJy via the max_rms keyword parameter. By default this is set to 10000 mJy

```
>>> from astroquery.nvas import Nvas
>>> import astropy.units as u
>>> images = Nvas.get_images("3c 273", radius=2*u.arcsec, band="K", max_rms=500)

1 images found.
Downloading http://www.vla.nvas.edu/astro/archive/pipeline/position/J122906.7+020308/22.4I0.37_TEST_
↪1995NOV15_1_352.U55.6S.imfits
|===========================================|  10M/ 10M (100.00%)     19m37s

>>> images

[[<astropy.io.fits.hdu.image.PrimaryHDU at 0x3376150>]]
```

The radius may be specified in any appropriate unit using a Quantity object. Apart from that it may also be entered as a string in a format parsable by Angle. The frequency bands are specified using the band keyword parameter. This defaults to a value of all - i.e all the bands. Here's a list of the valid values that this parameter can take.

```
"all", "L", "C", "X", "U", "K", "Q"
```

Let's look at an example that uses coordinates for specifying the search centre.

```
>>> from astroquery.nvas import Nvas
>>> import astropy.coordinates as coord
>>> import astropy.units as u
>>> images = Nvas.get_images(coord.SkyCoord(49.489, -0.37,
...                          unit=(u.deg, u.deg), frame='galactic'),
...                          band="K")
```

You may also fetch UVfits files rather than the IMfits files which is the default. To do this simply set the `get_uvfits` to `True`, in any of the query methods. You can also fetch the URLs to the downloadable images rather than the actual images themselves. To do this use the `get_image_list()` which takes in all the same arguments as `get_images()` above except for the `verbose` argument which isn't relevant in this case.

```
>>> from astroquery.nvas import Nvas
>>> import astropy.coordinates as coord
>>> import astropy.units as u
>>> image_urls = Nvas.get_image_list("05h34m31.94s 22d00m52.2s",
...                                  radius='0d0m0.6s', max_rms=500)

WARNING: Coordinate string is being interpreted as an ICRS
coordinate. [astroquery.utils.commons]

>>> image_urls

 ['http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.5+220114/1.51I4.12_T75_1986AUG12_1_
↪118.U3.06M.imfits',
 'http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.5+220114/1.51I3.92_T75_1986AUG20_1_
↪373.U2.85M.imfits',
 'http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.5+220114/4.89I1.22_T75_1986AUG12_1_84.
↪8U2.73M.imfits',
 'http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.9+220052/1.44I1.26_AH0336_1989FEB03_1_
↪197.U8.29M.imfits',
 'http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.9+220052/1.44I1.32_AH0336_1989FEB03_1_
↪263.U3.84M.imfits',
 'http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.9+220052/4.91I0.96_AH595_1996OCT14_1_
↪41.3U2.45M.imfits',
 'http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.9+220052/4.91I0.89_AH595_1996OCT11_1_
↪43.2U2.45M.imfits',
 'http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.9+220052/4.91I0.99_AH0595_1996OCT16_1_
↪66.4U2.55M.imfits',
 'http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.9+220052/8.46I2.18_AM503_1996FEB23_1_
↪243.U2.59M.imfits',
 'http://www.vla.nrao.edu/astro/archive/pipeline/position/J053431.9+220052/8.46I1.60_AM503_1996FEB01_1_
↪483.U2.59M.imfits']
```

## 19.2 Reference/API

### 19.2.1 astroquery.nvas Package

**Revision History**

Refactored using common API as a part of Google Summer of Code 2013.

**Originally contributed by**
Adam Ginsburg (adam.g.ginsburg@gmail.com)

### Classes

| | |
|---|---|
| NvasClass() | |
| Conf | Configuration parameters for astroquery.nvas. |

### NvasClass

**class** astroquery.nvas.**NvasClass**
Bases: astroquery.query.BaseQuery

#### Attributes Summary

| |
|---|
| TIMEOUT |
| URL |
| band_freqs |
| valid_bands |

#### Methods Summary

| | |
|---|---|
| extract_image_urls(html_in[, get_uvfits]) | Helper function that uses regexps to extract the image urls from the given HTML. |
| get_image_list(coordinates[, radius, . . . ]) | Function that returns a list of urls from which to download the FITS images. |
| get_images(coordinates[, radius, max_rms, . . . ]) | Get an image around a target/ coordinates from the NVAS image archive. |
| get_images_async(coordinates[, radius, . . . ]) | Serves the same purpose as get_images but returns a list of file handlers to remote files. |

#### Attributes Documentation

**TIMEOUT = 60**

**URL = 'https://webtest.aoc.nrao.edu/cgi-bin/lsjouwer/archive-pos.pl'**

**band_freqs = {'C': (4, 8), 'D': (110, 170), 'E': (60, 90), 'F': (90, 140), 'K': (18, 26.5), 'Ka': (26.5,**

**valid_bands = ['all', 'L', 'C', 'X', 'U', 'K', 'Q']**

### Methods Documentation

**extract_image_urls**(*html_in*, *get_uvfits=False*)
Helper function that uses regexps to extract the image urls from the given HTML.

**Parameters**
**html_in** : str

source from which the urls are to be extracted.

**get_uvfits** : bool, optional

Gets the UVfits files instead of the IMfits files when set to `True`. Defaults to `False`.

**Returns**
**image_urls** : list

The list of URLS extracted from the input.

**get_image_list**(*coordinates*, *radius=<Quantity 0.25 arcmin>*, *max_rms=10000*, *band='all'*, *get_uvfits=False*, *get_query_payload=False*)
Function that returns a list of urls from which to download the FITS images.

**Parameters**
**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 0.25 arcmin.

**max_rms** : float, optional

Maximum allowable noise level in the image (mJy). Defaults to 10000 mJy.

**band** : str, optional

The band of the image to fetch. Valid bands must be from ["all","L","C","X","U","K","Q"]. Defaults to 'all'

**get_uvfits** : bool, optional

Gets the UVfits files instead of the IMfits files when set to `True`. Defaults to `False`.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**Returns**
list of image urls

**get_images**(*coordinates*, *radius=<Quantity 0.25 arcmin>*, *max_rms=10000*, *band='all'*, *get_uvfits=False*, *verbose=True*, *get_query_payload=False*, *show_progress=True*)
Get an image around a target/ coordinates from the NVAS image archive.

**Parameters**
**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 0.25 arcmin.

**max_rms** : float, optional

Maximum allowable noise level in the image (mJy). Defaults to 10000 mJy.

**band** : str, optional

The band of the image to fetch. Valid bands must be from [“all”,”L”,”C”,”X”,”U”,”K”,”Q”]. Defaults to ‘all’

**get_uvfits** : bool, optional

Gets the UVfits files instead of the IMfits files when set to `True`. Defaults to `False`.

**verbose** : bool, optional

When `True` print out additional messages. Defaults to `True`.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**Returns**

A list of `HDUList` objects

**get_images_async**(*coordinates*, *radius=<Quantity 0.25 arcmin>*, *max_rms=10000*, *band='all'*, *get_uvfits=False*, *verbose=True*, *get_query_payload=False*, *show_progress=True*)

Serves the same purpose as `get_images` but returns a list of file handlers to remote files.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 0.25 arcmin.

**max_rms** : float, optional

Maximum allowable noise level in the image (mJy). Defaults to 10000 mJy.

**band** : str, optional

The band of the image to fetch. Valid bands must be from [“all”,”L”,”C”,”X”,”U”,”K”,”Q”]. Defaults to ‘all’

**get_uvfits** : bool, optional

Gets the UVfits files instead of the IMfits files when set to `True`. Defaults to `False`.

**verbose** : bool, optional

When `True` print out additional messages. Defaults to `True`.

**get_query_payload** : bool, optional

if set to `True` then returns the dictionary sent as the HTTP request. Defaults to `False`.

**Returns**

A list of context-managers that yield readable file-like objects

## Conf

**class** astroquery.nvas.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.nvas`.

### Attributes Summary

| | |
|---|---|
| `server` | Name of the NVAS mirror to use. |
| `timeout` | Time limit for connecting to NVAS server. |

### Attributes Documentation

**server**

Name of the NVAS mirror to use.

**timeout**

Time limit for connecting to NVAS server.

CHAPTER 20

# GAMA Queries (astroquery.gama)

## 20.1 Getting started

This module can be used to query the GAMA (Galaxy And Mass Assembly) survey, second data release (DR2). Currently queries must be formulated in SQL. If successful, the results are returned as a Table.

### 20.1.1 SQL Queries

This sends an SQL query, passed as a string, to the GAMA server and returns a Table. For example, to return basic information on the first 100 spectroscopic objects in the database:

```
>>> from astroquery.gama import GAMA
>>> result = GAMA.query_sql('SELECT * FROM SpecAll LIMIT 100')
Downloading http://www.gama-survey.org/dr2/query/../tmp/GAMA_VHI6pj.fits
|==========================================|  37k/ 37k (100.00%)        00s
>>> print(result)
      SPECID       SURVEY SURVEY_CODE     RA     ... DIST IS_SBEST IS_BEST
------------------ ------ ----------- --------- ... ---- -------- -------
131671727225700352   SDSS           1 132.16668 ...  0.1        1       1
131671727229894656   SDSS           1 132.17204 ... 0.13        1       1
131671727246671872   SDSS           1 132.24395 ... 0.13        1       1
131671727255060480   SDSS           1  132.1767 ... 0.06        1       1
131671727267643392   SDSS           1 132.63599 ... 0.05        1       1
131671727271837696   SDSS           1 132.85366 ... 0.02        1       1
131671727276032000   SDSS           1 132.70244 ... 0.03        1       1
131671727292809216   SDSS           1 132.19579 ... 0.12        1       1
131671727301197824   SDSS           1 132.57563 ...  0.0        1       1
131671727309586432   SDSS           1 133.01007 ... 0.06        1       1
131671727313780736   SDSS           1 132.76907 ... 0.04        1       1
131671727322169344   SDSS           1 132.81014 ... 0.03        1       1
131671727334752256   SDSS           1 132.85607 ... 0.02        1       1
131671727338946560   SDSS           1 132.90222 ... 0.04        1       1
131671727351529472   SDSS           1 133.00397 ... 0.05        1       1
```

(continues on next page)

(continued from previous page)

```
131671727355723776   SDSS          1 132.96032 ... 0.05       1       1
131671727359918080   SDSS          1 132.92164 ... 0.03       1       1
               ...    ...        ...       ... ...  ...     ...     ...
131671727791931392   SDSS          1 131.59537 ... 0.03       1       1
131671727796125696   SDSS          1 131.58167 ... 0.11       1       1
131671727800320000   SDSS          1 131.47693 ... 0.05       1       1
131671727804514304   SDSS          1 131.47471 ... 0.03       1       1
131671727808708608   SDSS          1 131.60197 ... 0.03       1       1
131671727825485824   SDSS          1 132.18426 ... 0.05       1       1
131671727833874432   SDSS          1  132.2593 ... 0.05       1       1
131671727838068736   SDSS          1  132.1901 ... 0.09       1       1
131671727854845952   SDSS          1 132.30575 ... 0.04       1       1
131671727859040256   SDSS          1   132.419 ... 0.04       1       1
131671727867428864   SDSS          1 132.29052 ... 0.15       1       1
131671727871623168   SDSS          1 132.37213 ... 0.01       1       1
131671727880011776   SDSS          1 132.36358 ...  0.1       1       1
131671727892594688   SDSS          1  132.3956 ... 0.05       1       1
131671727896788992   SDSS          1 131.89562 ... 0.15       1       1
131671727900983296   SDSS          1 131.85848 ... 0.05       1       1
131671727905177600   SDSS          1 132.12958 ... 0.09       0       0
```

## 20.2 Reference/API

### 20.2.1 astroquery.gama Package

#### gama

Access to GAMA (Galaxy And Mass Assembly) data, via the DR2 SQL query form. http://www.gama-survey.org/dr2/query/

**author**
James T. Allen <james.thomas.allen@gmail.com>

#### Classes

| | |
|---|---|
| GAMAClass() | TODO: document |

#### GAMAClass

**class** astroquery.gama.**GAMAClass**

Bases: astroquery.query.BaseQuery

TODO: document

##### Attributes Summary

| |
|---|
| request_url |
| timeout |

### Methods Summary

| | |
|---|---|
| query_sql(*args, **kwargs) | Queries the service and returns a table object. |
| query_sql_async(*args, **kwargs) | Query the GAMA database |

### Attributes Documentation

**request_url = 'http://www.gama-survey.org/dr3/query/'**

**timeout = 60**

### Methods Documentation

**query_sql**(**args*, ***kwargs*)

Queries the service and returns a table object.

Query the GAMA database

**Returns**

**table** : A Table object.

**query_sql_async**(**args*, ***kwargs*)

Query the GAMA database

**Returns**

**url** : The URL of the FITS file containing the results.

CHAPTER 21

# ESO Queries (astroquery.eso)

## 21.1 Getting started

This is a python interface for querying the ESO archive web service. For now, it supports the following:

- listing available instruments
- listing available surveys (phase 3)
- searching all instrument specific raw data: http://archive.eso.org/cms/eso-data/instrument-specific-query-forms.html
- searching data products (phase 3): http://archive.eso.org/wdb/wdb/adp/phase3_main/form
- downloading data by dataset identifiers: http://archive.eso.org/cms/eso-data/eso-data-direct-retrieval.html

## 21.2 Requirements

The following packages are required for the use of this module:

- keyring
- lxml
- requests >= 2.4.0

## 21.3 Authentication with ESO User Portal

Whereas querying the ESO database is fully open, accessing actual datasets requires authentication with the ESO User Portal (https://www.eso.org/sso/login). This authentication is performed directly with the provided `login()` command, as illustrated in the example below. This method uses your keyring to securely store the password in your operating system. As such you should have to enter your correct password only once, and later be able to use this package for automated interaction with the ESO archive.

```
>>> from astroquery.eso import Eso
>>> eso = Eso()
>>> # First example: TEST is not a valid username, it will fail
>>> eso.login("TEST")
TEST, enter your ESO password:

Authenticating TEST on www.eso.org...
Authentication failed!
>>> # Second example: pretend ICONDOR is a valid username
>>> eso.login("ICONDOR", store_password=True)
ICONDOR, enter your ESO password:

Authenticating ICONDOR on www.eso.org...
Authentication successful!
>>> # After the first login, your password has been stored
>>> eso.login("ICONDOR")
Authenticating ICONDOR on www.eso.org...
Authentication successful!
```

### 21.3.1 Automatic password

As shown above, your password can be stored by the keyring module, if you pass the argument `store_password=True` to `Eso.login()`. For security reason, storing the password is turned off by default.

MAKE SURE YOU TRUST THE MACHINE WHERE YOU USE THIS FUNCTIONALITY!!!

NB: You can delete your password later with the command `keyring.delete_password('astroquery:www.eso.org', 'username')`.

### 21.3.2 Automatic login

You can further automate the authentication process by configuring a default username. The astroquery configuration file, which can be found following the procedure detailed in astropy.config, needs to be edited by adding `username = ICONDOR` in the `[eso]` section.

When configured, the username in the `login()` method call can be omitted as follows:

```
>>> from astroquery.eso import Eso
>>> eso = Eso()
>>> eso.login()
ICONDOR, enter your ESO password:
```

NB: If an automatic login is configured, other Eso methods can log you in automatically when needed.

## 21.4 Query the ESO archive

### 21.4.1 Identifying available instrument-specific queries

The direct retrieval of datasets is better explained with a running example, continuing from the authentication example above. The first thing to do is to identify the instrument to query. The list of available instrument-specific queries can be obtiained with the `list_instruments()` method.

```
>>> eso.list_instruments()
['fors1', 'fors2', 'vimos', 'omegacam', 'hawki', 'isaac', 'naco', 'visir', 'vircam',
'apex', 'uves', 'giraffe', 'xshooter', 'muse, 'crires', 'kmos', 'sinfoni',
'amber', 'gravity', 'midi', 'pionier']
```

In the example above, 22 instruments are available, they correspond to the instruments listed on the following web page: http://archive.eso.org/cms/eso-data/instrument-specific-query-forms.html.

### 21.4.2 Inspecting available query options

Once an instrument is chosen, `midi` in our case, the query options for that instrument can be inspected by setting the `help=True` keyword of the `query_instrument()` method.

```
>>> eso.query_instrument('midi', help=True)
List of the column_filters parameters accepted by the midi instrument query.
The presence of a column in the result table can be controlled if prefixed with a [ ] checkbox.
The default columns in the result table are shown as already ticked: [x].

Target Information
------------------
    target:
    resolver: simbad (SIMBAD name), ned (NED name), none (OBJECT as specified by the observer)
    coord_sys: eq (Equatorial (FK5)), gal (Galactic)
    coord1:
    coord2:
    box:
    format: sexagesimal (Sexagesimal), decimal (Decimal)
[x] wdb_input_file:

Observation  and proposal parameters
-------------------------------------
[ ] night:
    stime:
    starttime: 00 (00 hrs [UT]), 01 (01 hrs [UT]), 02 (02 hrs [UT]), 03 (03 hrs [UT]), 04 (04 hrs [UT]),
↪ 05 (05 hrs [UT]), 06 (06 hrs [UT]), 07 (07 hrs [UT]), 08 (08 hrs [UT]), 09 (09 hrs [UT]), 10 (10 hrs␣
↪[UT]), 11 (11 hrs [UT]), 12 (12 hrs [UT]), 13 (13 hrs [UT]), 14 (14 hrs [UT]), 15 (15 hrs [UT]), 16␣
↪(16 hrs [UT]), 17 (17 hrs [UT]), 18 (18 hrs [UT]), 19 (19 hrs [UT]), 20 (20 hrs [UT]), 21 (21 hrs␣
↪[UT]), 22 (22 hrs [UT]), 23 (23 hrs [UT]), 24 (24 hrs [UT])
    etime:
    endtime: 00 (00 hrs [UT]), 01 (01 hrs [UT]), 02 (02 hrs [UT]), 03 (03 hrs [UT]), 04 (04 hrs [UT]),␣
↪05 (05 hrs [UT]), 06 (06 hrs [UT]), 07 (07 hrs [UT]), 08 (08 hrs [UT]), 09 (09 hrs [UT]), 10 (10 hrs␣
↪[UT]), 11 (11 hrs [UT]), 12 (12 hrs [UT]), 13 (13 hrs [UT]), 14 (14 hrs [UT]), 15 (15 hrs [UT]), 16␣
↪(16 hrs [UT]), 17 (17 hrs [UT]), 18 (18 hrs [UT]), 19 (19 hrs [UT]), 20 (20 hrs [UT]), 21 (21 hrs␣
↪[UT]), 22 (22 hrs [UT]), 23 (23 hrs [UT]), 24 (24 hrs [UT])
[x] prog_id:
[ ] prog_type: % (Any), 0 (Normal), 1 (GTO), 2 (DDT), 3 (ToO), 4 (Large), 5 (Short), 6 (Calibration)
[ ] obs_mode: % (All modes), s (Service), v (Visitor)
[ ] pi_coi:
    pi_coi_name: PI_only (as PI only), none (as PI or CoI)
[ ] prog_title:
...
```

Only the first two sections, of the parameters accepted by the `midi` instrument query, are shown in the example above: `Target Information` and `Observation and proposal parameters`.

As stated at the beginning of the help message, the parameters accepted by the query are given just before the first `:` sign (e.g. `target`, `resolver`, `stime`, `etime`…). When a parameter is prefixed by `[ ]`, the presence of the associated column in the query result can be controlled.

Note: the instrument query forms can be opened in your web browser directly using the open_form option of the query_instrument() method. This should also help with the identification of acceptable keywords.

### 21.4.3 Querying with constraints

It is now time to query the midi instrument for datasets. In the following example, observations of target NGC 4151 between 2007-01-01 and 2008-01-01 are searched, and the query is configured to return the observation date column.

```
>>> table = eso.query_instrument('midi', column_filters={'target':'NGC 4151', 'stime':'2007-01-01',
↪'etime':'2008-01-01'}, columns=['night'])
>>> print(len(table))
38
>>> print(table.columns)
<TableColumns names=('Object','RA', 'DEC','Target l b','DATE OBS','ProgId','DP.ID','OB.ID','OBS.TARG.
↪NAME','DPR.CATG','DPR.TYPE','DPR.TECH','INS.MODE','DIMM S-avg')>
>>> table.pprint(max_width=100)
         Object               Target Ra Dec          Target l b      ... INS.MODE  DIMM S-avg
----------------------- ----------------------- -------------------- ... -------- -----------
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.69 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.68 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.68 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.69 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.69 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.74 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.69 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.66 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.64 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.60 [0.01]
                NGC4151 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.59 [0.01]
                    ...                     ...                  ... ...      ...         ...
 TRACK,OBJECT,DISPERSED 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.70 [0.01]
 TRACK,OBJECT,DISPERSED 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.72 [0.01]
SEARCH,OBJECT,DISPERSED 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.62 [0.01]
SEARCH,OBJECT,DISPERSED 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.61 [0.01]
SEARCH,OBJECT,DISPERSED 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.54 [0.01]
SEARCH,OBJECT,DISPERSED 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.53 [0.01]
 TRACK,OBJECT,DISPERSED 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.51 [0.01]
 TRACK,OBJECT,DISPERSED 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.51 [0.01]
 TRACK,OBJECT,DISPERSED 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.51 [0.01]
      PHOTOMETRY,OBJECT 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.54 [0.01]
      PHOTOMETRY,OBJECT 12:10:32.63 +39:24:20.7 155.076719 75.063247 ... STARINTF 0.54 [0.01]
Length = 38 rows
```

And indeed, 38 datasets are found, and the DATE OBS column is in the result table.

### 21.4.4 Querying all instruments

The ESO database can also be queried without a specific instrument in mind. This is what the method query_main() is for. The associated query form on the ESO archive website is http://archive.eso.org/wdb/wdb/eso/eso_archive_main/form. Except for the keyword specifying the instrument the behaviour of query_main() is identical to query_instrument().

ESO instruments without a specific query interface can be queried with query_main(), specifying the instrument constraint. This is the case of e.g. harps or feros.

## 21.5 Obtaining extended information on data products

Only a small subset of the keywords presents in the data products can be obtained with `query_instrument()` or `query_main()`. There is however a way to get the full primary header of the FITS data products, using `get_headers()`. This method is detailed in the example below, continuing with the previously obtained table.

```
>>> table_headers = eso.get_headers(table['DP.ID'])
>>> table_headers.pprint()
             ARCFILE              BITPIX ...   TELESCOP     UTC
--------------------------------- ------ ... ------------ -------
MIDI.2007-02-07T07:01:51.000.fits     16 ... ESO-VLTI-U23 25300.5
MIDI.2007-02-07T07:02:49.000.fits     16 ... ESO-VLTI-U23 25358.5
MIDI.2007-02-07T07:03:30.695.fits     16 ... ESO-VLTI-U23 25358.5
MIDI.2007-02-07T07:05:47.000.fits     16 ... ESO-VLTI-U23 25538.5
MIDI.2007-02-07T07:06:28.695.fits     16 ... ESO-VLTI-U23 25538.5
MIDI.2007-02-07T07:09:03.000.fits     16 ... ESO-VLTI-U23 25732.5
MIDI.2007-02-07T07:09:44.695.fits     16 ... ESO-VLTI-U23 25732.5
MIDI.2007-02-07T07:13:09.000.fits     16 ... ESO-VLTI-U23 25978.5
MIDI.2007-02-07T07:13:50.695.fits     16 ... ESO-VLTI-U23 25978.5
MIDI.2007-02-07T07:15:55.000.fits     16 ... ESO-VLTI-U23 26144.5
MIDI.2007-02-07T07:16:36.694.fits     16 ... ESO-VLTI-U23 26144.5
                              ...    ... ...          ...     ...
MIDI.2007-02-07T07:51:13.485.fits     16 ... ESO-VLTI-U23 28190.5
MIDI.2007-02-07T07:52:27.992.fits     16 ... ESO-VLTI-U23 28190.5
MIDI.2007-02-07T07:56:21.000.fits     16 ... ESO-VLTI-U23 28572.5
MIDI.2007-02-07T07:57:35.485.fits     16 ... ESO-VLTI-U23 28572.5
MIDI.2007-02-07T07:59:46.000.fits     16 ... ESO-VLTI-U23 28778.5
MIDI.2007-02-07T08:01:00.486.fits     16 ... ESO-VLTI-U23 28778.5
MIDI.2007-02-07T08:03:42.000.fits     16 ... ESO-VLTI-U23 29014.5
MIDI.2007-02-07T08:04:56.506.fits     16 ... ESO-VLTI-U23 29014.5
MIDI.2007-02-07T08:06:11.013.fits     16 ... ESO-VLTI-U23 29014.5
MIDI.2007-02-07T08:08:19.000.fits     16 ... ESO-VLTI-U23 29288.5
MIDI.2007-02-07T08:09:33.506.fits     16 ... ESO-VLTI-U23 29288.5
Length = 38 rows
>>> len(table_headers.columns)
340
```

As shown above, for each data product ID (`DP.ID`), the full header (570 columns in our case) of the archive FITS file is collected. In the above table `table_headers`, there are as many rows as in the column `table['DP.ID']`.

## 21.6 Downloading datasets from the archive

Continuing from the query with constraints example, the first two datasets are selected, using their data product IDs `DP.ID`, and retrieved from the ESO archive.

```
>>> data_files = eso.retrieve_data(table['DP.ID'][:2])
Staging request...
Downloading files...
Downloading MIDI.2007-02-07T07:01:51.000.fits.Z...
Downloading MIDI.2007-02-07T07:02:49.000.fits.Z...
Done!
```

The file names, returned in data_files, points to the decompressed datasets (without the .Z extension) that have been locally downloaded. They are ready to be used with `fits`.

The default location (in the astropy cache) of the decompressed datasets can be adjusted by providing a `location` keyword in the call to `retrieve_data()`.

In all cases, if a requested dataset is already found, it is not downloaded again from the archive.

## 21.7 Reference/API

### 21.7.1 astroquery.eso Package

ESO service.

#### Classes

| | |
|---|---|
| `EsoClass`() | |
| `Conf` | Configuration parameters for `astroquery.eso`. |

#### EsoClass

**class** astroquery.eso.**EsoClass**

Bases: `astroquery.query.QueryWithLogin`

##### Attributes Summary

| |
|---|
| `QUERY_INSTRUMENT_URL` |
| `ROW_LIMIT` |
| `USERNAME` |

##### Methods Summary

| | |
|---|---|
| `get_headers`(product_ids[, cache]) | Get the headers associated to a list of data product IDs |
| `list_instruments`([cache]) | List all the available instrument-specific queries offered by the ESO archive. |
| `list_surveys`([cache]) | List all the available surveys (phase 3) in the ESO archive. |
| `query_apex_quicklooks`([project_id, help, . . . ]) | APEX data are distributed with quicklook products identified with a different name than other ESO products. |
| `query_instrument`(instrument[, . . . ]) | Query instrument-specific raw data contained in the ESO archive. |
| `query_main`([column_filters, columns, . . . ]) | Query raw data contained in the ESO archive. |
| `query_surveys`([surveys, cache, help, open_form]) | Query survey Phase 3 data contained in the ESO archive. |
| `retrieve_data`(datasets[, continuation, . . . ]) | Retrieve a list of datasets form the ESO archive. |
| `verify_data_exists`(dataset) | Given a data set name, return 'True' if ESO has the file and 'False' otherwise |

### Attributes Documentation

**QUERY_INSTRUMENT_URL = 'http://archive.eso.org/wdb/wdb/eso'**

**ROW_LIMIT = 50**

**USERNAME = ''**

### Methods Documentation

**get_headers**(*product_ids*, *cache=True*)

Get the headers associated to a list of data product IDs

This method returns a `Table` where the rows correspond to the provided data product IDs, and the columns are from each of the Fits headers keywords.

Note: The additional column `'DP.ID'` found in the returned table corresponds to the provided data product IDs.

**Parameters**

**product_ids** : either a list of strings or a `Column`

List of data product IDs.

**Returns**

**result** : `Table`

A table where: columns are header keywords, rows are product_ids.

**list_instruments**(*cache=True*)

List all the available instrument-specific queries offered by the ESO archive.

**Returns**

**instrument_list** : list of strings

**cache** : bool

Cache the response for faster subsequent retrieval

**list_surveys**(*cache=True*)

List all the available surveys (phase 3) in the ESO archive.

**Returns**

**survey_list** : list of strings

**cache** : bool

Cache the response for faster subsequent retrieval

**query_apex_quicklooks**(*project_id=None*, *help=False*, *open_form=False*, *cache=True*, ***kwargs*)

APEX data are distributed with quicklook products identified with a different name than other ESO products. This query tool searches by project ID or any other supported keywords.

#### Examples

```
>>> tbl = Eso.query_apex_quicklooks('093.C-0144')
>>> files = Eso.retrieve_data(tbl['Product ID'])
```

**query_instrument**(*instrument*, *column_filters={}*, *columns=[]*, *open_form=False*, *help=False*, *cache=True*, ***kwargs*)
Query instrument-specific raw data contained in the ESO archive.

**Parameters**
**instrument** : string

Name of the instrument to query, one of the names returned by `list_instruments`.

**column_filters** : dict

Constraints applied to the query.

**columns** : list of strings

Columns returned by the query.

**open_form** : bool

If `True`, opens in your default browser the query form for the requested instrument.

**help** : bool

If `True`, prints all the parameters accepted in `column_filters` and `columns` for the requested `instrument`.

**cache** : bool

Cache the response for faster subsequent retrieval.

**Returns**
**table** : `Table`

A table representing the data available in the archive for the specified instrument, matching the constraints specified in `kwargs`. The number of rows returned is capped by the ROW_LIMIT configuration item.

**query_main**(*column_filters={}*, *columns=[]*, *open_form=False*, *help=False*, *cache=True*, ***kwargs*)
Query raw data contained in the ESO archive.

**Parameters**
**column_filters** : dict

Constraints applied to the query.

**columns** : list of strings

Columns returned by the query.

**open_form** : bool

If `True`, opens in your default browser the query form for the requested instrument.

**help** : bool

If `True`, prints all the parameters accepted in `column_filters` and `columns` for the requested `instrument`.

**cache** : bool

Cache the response for faster subsequent retrieval.

**Returns**
**table** : `Table`

A table representing the data available in the archive for the specified instrument, matching the constraints specified in `kwargs`. The number of rows returned is capped by the ROW_LIMIT configuration item.

**query_surveys**(*surveys=''*, *cache=True*, *help=False*, *open_form=False*, ***kwargs*)
Query survey Phase 3 data contained in the ESO archive.

**Parameters**

**survey** : string or list

Name of the survey(s) to query. Should beone or more of the names returned by `list_surveys`. If specified as a string, should be a comma-separated list of survey names.

**cache** : bool

Cache the response for faster subsequent retrieval

**Returns**

**table** : `Table` or `None`

A table representing the data available in the archive for the specified survey, matching the constraints specified in `kwargs`. The number of rows returned is capped by the ROW_LIMIT configuration item. `None` is returned when the query has no results.

**retrieve_data**(*datasets*, *continuation=False*, *destination=None*)
Retrieve a list of datasets form the ESO archive.

**Parameters**

**datasets** : list of strings or string

List of datasets strings to retrieve from the archive.

**destination: string**

Directory where the files are copied. Files already found in the destination directory are skipped. Default to astropy cache.

**Returns**

**files** : list of strings or string

List of files that have been locally downloaded from the archive.

### Examples

```
>>> dptbl = Eso.query_instrument('apex', pi_coi='ginsburg')
>>> dpids = [row['DP.ID'] for row in dptbl if 'Map' in row['Object']]
>>> files = Eso.retrieve_data(dpids)
```

**verify_data_exists**(*dataset*)
Given a data set name, return 'True' if ESO has the file and 'False' otherwise

## Conf

**class** astroquery.eso.**Conf**
Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.eso`.

### Attributes Summary

| | |
|---|---|
| query_instrument_url | Root query URL for main and instrument queries. |
| row_limit | Maximum number of rows returned (set to -1 for unlimited). |
| username | Optional default username for ESO archive. |

### Attributes Documentation

**query_instrument_url**
Root query URL for main and instrument queries.

**row_limit**
Maximum number of rows returned (set to -1 for unlimited).

**username**
Optional default username for ESO archive.

CHAPTER 22

# xMatch Queries (astroquery.xmatch)

## 22.1 Getting started

The xMatch service is a tool to cross-identify sources between very large data sets or between a user-uploaded list and a large catalogue. An example for the latter case can be found below.

First of all, a new CSV file is created which stores a list of coordinates. It has the following content:

```
ra,dec
267.22029,-20.35869
274.83971,-25.42714
275.92229,-30.36572
283.26621,-8.70756
306.01575,33.86756
322.493,12.16703
```

Next, the xMatch service will be used to find cross matches between the uploaded file and a VizieR catalogue. The parameters `cat1` and `cat2` define the catalogues where one of them may point to a local file (in this example, the CSV file is stored in `'/tmp/pos_list.csv'`). `max_distance` denotes the maximum distance in arcsec to look for counterparts; it is used here to limit the number of rows in the resulting table for demonstration purposes. Finally, `colRa1` and `colDec1` are used to denote the column names in the input file.

```
>>> from astropy import units as u
>>> from astroquery.xmatch import XMatch
>>> table = XMatch.query(cat1=open('/tmp/pos_list.csv'),
...                      cat2='vizier:II/246/out',
...                      max_distance=5 * u.arcsec, colRA1='ra',
...                      colDec1='dec')
>>> type(table)
<class 'astropy.table.table.Table'>
>>> print(table)
angDist      ra       dec          2MASS       ... Qfl Rfl  X   MeasureJD
-------- --------- --------- ---------------- ... --- --- --- ------------
1.352044 267.22029 -20.35869 17485281-2021323 ... EEU 226   2 2450950.8609
```

(continues on next page)

(continued from previous page)

```
1.578188 267.22029 -20.35869 17485288-2021328 ... UUB 662   2 2450950.8609
3.699368 267.22029 -20.35869 17485264-2021294 ... UUB 662   2 2450950.8609
3.822922 267.22029 -20.35869 17485299-2021279 ... EBA 222   2 2450950.8609
4.576677 267.22029 -20.35869 17485255-2021326 ... CEU 226   2 2450950.8609
0.219609 274.83971 -25.42714 18192154-2525377 ... AAA 211   0 2451407.5033
1.633225 275.92229 -30.36572 18234133-3021582 ... EEE 222   2 2451021.7212
0.536998 283.26621  -8.70756 18530390-0842276 ... AAA 222   0 2451301.7945
1.178542 306.01575  33.86756 20240382+3352021 ... AAA 222   0 2450948.9708
0.853178   322.493  12.16703 21295836+1210007 ... EEA 222   0 2451080.6935
 4.50395   322.493  12.16703 21295861+1210023 ... EEE 222   0 2451080.6935
```

## 22.2 Reference/API

### 22.2.1 astroquery.xmatch Package

#### Classes

| | |
|---|---|
| XMatchClass() | |
| Conf | Configuration parameters for astroquery.xmatch. |

#### XMatchClass

**class** astroquery.xmatch.**XMatchClass**

Bases: astroquery.query.BaseQuery

##### Attributes Summary

| |
|---|
| TIMEOUT |
| URL |

##### Methods Summary

| | |
|---|---|
| get_available_tables([cache]) | Get the list of the VizieR tables which are available in the xMatch service and return them as a list of strings. |
| is_table_available(table_id) | Return True if the passed CDS table identifier is one of the available VizieR tables, otherwise False. |
| query(cat1, cat2, max_distance[, colRA1, ...]) | Query the CDS cross-match service by finding matches between two (potentially big) catalogues. |
| query_async(cat1, cat2, max_distance[, ...]) | Query the CDS cross-match service by finding matches between two (potentially big) catalogues. |

##### Attributes Documentation

**TIMEOUT = 60**

`URL = 'http://cdsxmatch.u-strasbg.fr/xmatch/api/v1/sync'`

### Methods Documentation

**get_available_tables**(*cache=True*)

Get the list of the VizieR tables which are available in the xMatch service and return them as a list of strings.

**is_table_available**(*table_id*)

Return True if the passed CDS table identifier is one of the available VizieR tables, otherwise False.

**query**(*cat1*, *cat2*, *max_distance*, *colRA1=None*, *colDec1=None*, *colRA2=None*, *colDec2=None*, *cache=True*, *get_query_payload=False*)

Query the CDS cross-match service by finding matches between two (potentially big) catalogues.

**Parameters**

**cat1** : str, file or `Table`

Identifier of the first table. It can either be a URL, the payload of a local file being uploaded, a CDS table identifier (either *simbad* for a view of SIMBAD data / to point out a given VizieR table) or a an AstroPy table. If the table is uploaded or accessed through a URL, it must be in VOTable or CSV format with the positions in J2000 equatorial frame and as decimal degrees numbers.

**cat2** : str or file

Identifier of the second table. Follows the same rules as *cat1*.

**max_distance** : `Quantity`

Maximum distance to look for counterparts. Maximum allowed value is 180 arcsec.

**colRA1** : str

Name of the column holding the right ascension. Only required if `cat1` is an uploaded table or a pointer to a URL.

**colDec1** : str

Name of the column holding the declination. Only required if `cat1` is an uploaded table or a pointer to a URL.

**colRA2** : str

Name of the column holding the right ascension. Only required if `cat2` is an uploaded table or a pointer to a URL.

**colDec2** : str

Name of the column holding the declination. Only required if `cat2` is an uploaded table or a pointer to a URL.

**Returns**

**table** : `Table`

Query results table

**query_async**(*cat1*, *cat2*, *max_distance*, *colRA1=None*, *colDec1=None*, *colRA2=None*, *colDec2=None*, *cache=True*, *get_query_payload=False*)

Query the CDS cross-match service by finding matches between two (potentially big) catalogues.

**Parameters**

**cat1** : str, file or `Table`

Identifier of the first table. It can either be a URL, the payload of a local file being uploaded, a CDS table identifier (either *simbad* for a view of SIMBAD data / to point out a given VizieR table) or a an AstroPy table. If the table is uploaded or accessed through a URL, it must be in VOTable or CSV format with the positions in J2000 equatorial frame and as decimal degrees numbers.

**cat2** : str or file

Identifier of the second table. Follows the same rules as *cat1*.

**max_distance** : `Quantity`

Maximum distance to look for counterparts. Maximum allowed value is 180 arcsec.

**colRA1** : str

Name of the column holding the right ascension. Only required if `cat1` is an uploaded table or a pointer to a URL.

**colDec1** : str

Name of the column holding the declination. Only required if `cat1` is an uploaded table or a pointer to a URL.

**colRA2** : str

Name of the column holding the right ascension. Only required if `cat2` is an uploaded table or a pointer to a URL.

**colDec2** : str

Name of the column holding the declination. Only required if `cat2` is an uploaded table or a pointer to a URL.

**Returns**

**response** : `Response`

The HTTP response returned from the service.

## Conf

**class** `astroquery.xmatch.Conf`

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.xmatch`.

### Attributes Summary

| | |
|---|---|
| `timeout` | time limit for connecting to xMatch server |
| `url` | xMatch URL |

### Attributes Documentation

**`timeout`**

time limit for connecting to xMatch server

`url`
xMatch URL

CHAPTER 23

# Atomic Line List (astroquery.atomic)

## 23.1 Getting started

"Atomic Line List" is a collection of more than 900,000 atomic transitions in the range from 0.5 Å to 1000 µm (source). AtomicLineList has 13 parameters of which all are optional. In the example below, only a restricted set of the available parameters is used to keep it simple: wavelength_range, wavelength_type, wavelength_accuracy and element_spectrum. The respective web form for Atomic Line List can be found at http://www.pa.uky.edu/~peter/atomic/. As can be seen there, the first form fields are "Wavelength range" and "Unit". Because astroquery encourages the usage of AstroPy units, the expected type for the parameter wavelength_range is a tuple with two AstroPy quantities in it. This has the positive side-effect that even more units will be supported than by just using the web form directly.

In the following Python session you can see the atomic package in action. Note that Hz is actually not a supported unit by Atomic Line List, the atomic package takes care to support all spectral units.

```
>>> from astropy import units as u
>>> from astroquery.atomic import AtomicLineList
>>> wavelength_range = (15 * u.nm, 1.5e+16 * u.Hz)
>>> AtomicLineList.query_object(wavelength_range, wavelength_type='Air', wavelength_accuracy=20,␣
↪element_spectrum='C II-IV')
<Table rows=3 names=('LAMBDA VAC ANG','SPECTRUM','TT','TERM','J J','LEVEL ENERGY  CM 1')>
array([(196.8874, 'C IV', 'E1', '2S-2Po', '1/2-*', '0.00 -   507904.40'),
       (197.7992, 'C IV', 'E1', '2S-2Po', '1/2-*', '0.00 -   505563.30'),
       (199.0122, 'C IV', 'E1', '2S-2Po', '1/2-*', '0.00 -   502481.80')],
      dtype=[('LAMBDA VAC ANG', '<f8'), ('SPECTRUM', 'S4'), ('TT', 'S2'), ('TERM', 'S6'), ('J J', 'S5'),
↪ ('LEVEL ENERGY  CM 1', 'S18')])
```

## 23.2 Reference/API

### 23.2.1 astroquery.atomic Package

#### Classes

| | |
|---|---|
| AtomicLineListClass() | |
| Transition | |

#### AtomicLineListClass

**class** astroquery.atomic.**AtomicLineListClass**

Bases: astroquery.query.BaseQuery

##### Attributes Summary

| | |
|---|---|
| FORM_URL | |
| TIMEOUT | |

##### Methods Summary

| | |
|---|---|
| query_object([wavelength_range, ...]) | Queries Atomic Line List for the given parameters adnd returns the result as a Table. |
| query_object_async([wavelength_range, ...]) | Queries Atomic Line List for the given parameters adnd returns the result as a Table. |

##### Attributes Documentation

**FORM_URL = 'http://www.pa.uky.edu/~peter/atomic/'**

**TIMEOUT = 60**

##### Methods Documentation

**query_object**(*wavelength_range=None, wavelength_type=None, wavelength_accuracy=None, element_spectrum=None, minimal_abundance=None, depl_factor=None, lower_level_energy_range=None, upper_level_energy_range=None, nmax=None, multiplet=None, transitions=None, show_fine_structure=None, show_auto_ionizing_transitions=None, output_columns=('spec', 'type', 'conf', 'term', 'angm', 'prob', 'ener')*)

Queries Atomic Line List for the given parameters adnd returns the result as a Table. All parameters are optional.

**Parameters**

**wavelength_range** : pair of astropy.units.Unit values

Wavelength range. Can be done in two ways: supply a lower and upper limit for the region or, supply the central wavelength and the 1 sigma error (68 % confidence value) for that line. If the first number is smaller than the second number, this implies that the first option has been chosen, and otherwise the second option.

**wavelength_type** : str

Either `'Air'` or `'Vacuum'`.

**wavelength_accuracy** : str

All wavelengths in the line list have relative accuracies of 5% or better. The default is to list all lines, irrespective of their accuracy. When a relative accuracy in percent is given, only those lines with accuracies better than or equal to the passed value are included in the search. Values larger than 5% will be ignored.

**element_spectrum** : str

Restrict the search to a range of elements and/or ionization stages. The elements should be entered by their usual symbolic names (e.g. Fe) and the ionization stages by the usual spectroscopic notation (e.g. I for neutral, II for singly ionized etc.). To pass multiple values, separate them by `\n` (newline).

**minimal_abundance** : str

Impose a lower limit on the abundances of elements to be considered for possible identifications. Default is to consider arbitrary low abundances. The elements are assumed to have standard cosmic abundances.

**depl_factor** : str

For nebular conditions it is not a realistic assumption that the elements have standard cosmic abundances since most metals will be depleted on grains. To simulate this it is possible to supply a depletion factor df. This factor will be used to calculate the actual abundance A from the cosmic abundance Ac using the formula A(elm) = Ac(elm) - df*sd(elm) where sd is the standard depletion for each element.

**lower_level_energy_range** : `Quantity`

Default is to consider all values for the lower/upper level energy to find a possible identification. To restrict the search a range of energies can be supplied. The supported units are: Ry, eV, 1/cm, J, erg.

**upper_level_energy_range** : `Quantity`

See parameter `lower_level_energy_range`.

**nmax** : int

Maximum for principal quantum number n. Default is to consider all possible values for the principal quantum number n to find possible identifications. However, transitions involving electrons with a very high quantum number n tend to be weaker and can therefore be less likely identifications. These transitions can be suppressed using this parameter.

**multiplet** : str

This option (case sensitive) can be used to find all lines in a specific multiplet within a certain wavelength range. The lower and upper level term should be entered here exactly as they appear in the output of the query. The spectrum to which this multiplet belongs should of course also be supplied in the `element_spectrum` parameter.

**transitions** : str‘

**Possible values are:**

- **'all':**
  The default, consider all transition types.
- **'nebular':**
  Consider only allowed transitions of Hydrogen or Helium and only magnetic dipole or electric quadrupole transitions of other elements.
- A union of the values: One of the following: 'E1', 'IC', 'M1', 'E2' Refer to the documentation for the meaning of these values.

**show_fine_structure** : bool

If `True`, the fine structure components will be included in the output. Refer to the documentation for more information.

**show_auto_ionizing_transitions** : bool

If `True`, transitions originating from auto-ionizing levels will be included in the output. In this context, all levels with energies higher than the ionization potential going to the ground state of the next ion are considered auto-ionizing levels.

**output_columns** : tuple

A Tuple of strings indicating which output columns are retrieved. A subset of ('spec', 'type', 'conf', 'term', 'angm', 'prob', 'ener') should be used. Where each string corresponds to the column titled Spectrum, Transition type, Configuration, Term, Angular momentum, Transition probability and Level energies respectively.

**Returns**

**result** : `Table`

The result of the query as a `Table` object.

**query_object_async**(*wavelength_range=None, wavelength_type='', wavelength_accuracy=None, element_spectrum=None, minimal_abundance=None, depl_factor=None, lower_level_energy_range=None, upper_level_energy_range=None, nmax=None, multiplet=None, transitions=None, show_fine_structure=None, show_auto_ionizing_transitions=None, output_columns=('spec', 'type', 'conf', 'term', 'angm', 'prob', 'ener')*)

Queries Atomic Line List for the given parameters adnd returns the result as a `Table`. All parameters are optional.

**Parameters**

**wavelength_range** : pair of `astropy.units.Unit` values

Wavelength range. Can be done in two ways: supply a lower and upper limit for the region or, supply the central wavelength and the 1 sigma error (68 % confidence value) for that line. If the first number is smaller than the second number, this implies that the first option has been chosen, and otherwise the second option.

**wavelength_type** : str

Either 'Air' or 'Vacuum'.

**wavelength_accuracy** : str

All wavelengths in the line list have relative accuracies of 5% or better. The default is to list all lines, irrespective of their accuracy. When a relative accuracy in percent is given, only those lines with accuracies better than or equal to the passed value are included in the search. Values larger than 5% will be ignored.

**element_spectrum** : str

Restrict the search to a range of elements and/or ionization stages. The elements should be entered by their usual symbolic names (e.g. Fe) and the ionization stages by the usual spectroscopic notation (e.g. I for neutral, II for singly ionized etc.). To pass multiple values, separate them by `\n` (newline).

**minimal_abundance** : str

Impose a lower limit on the abundances of elements to be considered for possible identifications. Default is to consider arbitrary low abundances. The elements are assumed to have standard cosmic abundances.

**depl_factor** : str

For nebular conditions it is not a realistic assumption that the elements have standard cosmic abundances since most metals will be depleted on grains. To simulate this it is possible to supply a depletion factor df. This factor will be used to calculate the actual abundance A from the cosmic abundance Ac using the formula A(elm) = Ac(elm) - df*sd(elm) where sd is the standard depletion for each element.

**lower_level_energy_range** : `Quantity`

Default is to consider all values for the lower/upper level energy to find a possible identification. To restrict the search a range of energies can be supplied. The supported units are: Ry, eV, 1/cm, J, erg.

**upper_level_energy_range** : `Quantity`

See parameter `lower_level_energy_range`.

**nmax** : int

Maximum for principal quantum number n. Default is to consider all possible values for the principal quantum number n to find possible identifications. However, transitions involving electrons with a very high quantum number n tend to be weaker and can therefore be less likely identifications. These transitions can be suppressed using this parameter.

**multiplet** : str

This option (case sensitive) can be used to find all lines in a specific multiplet within a certain wavelength range. The lower and upper level term should be entered here exactly as they appear in the output of the query. The spectrum to which this multiplet belongs should of course also be supplied in the `element_spectrum` parameter.

**transitions** : str‘

**Possible values are:**

- **`'all'`:**
  The default, consider all transition types.
- **`'nebular'`:**
  Consider only allowed transitions of Hydrogen or Helium and only magnetic dipole or electric quadrupole transitions of other elements.
- A union of the values: One of the following: `'E1'`, `'IC'`, `'M1'`, `'E2'` Refer to the documentation for the meaning of these values.

**show_fine_structure** : bool

If `True`, the fine structure components will be included in the output. Refer to the documentation for more information.

**show_auto_ionizing_transitions** : bool

If `True`, transitions originating from auto-ionizing levels will be included in the output. In this context, all levels with energies higher than the ionization potential going to the ground state of the next ion are considered auto-ionizing levels.

**output_columns** : tuple

A Tuple of strings indicating which output columns are retrieved. A subset of ('spec', 'type', 'conf', 'term', 'angm', 'prob', 'ener') should be used. Where each string corresponds to the column titled Spectrum, Transition type, Configuration, Term, Angular momentum, Transition probability and Level energies respectively.

**.. _documentation: http://www.pa.uky.edu/~peter/atomic/instruction.html**

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service.

## Transition

**class** `astroquery.atomic.`**`Transition`**

Bases: `object`

### Attributes Summary

| |
|---|
| `E1` |
| `E2` |
| `IC` |
| `M1` |
| `all` |
| `nebular` |

### Attributes Documentation

**`E1 = MultiTransition<[AtomicTransition<'E1'>]>`**

**`E2 = MultiTransition<[AtomicTransition<'E2'>]>`**

**`IC = MultiTransition<[AtomicTransition<'IC'>]>`**

**`M1 = MultiTransition<[AtomicTransition<'M1'>]>`**

**`all = AtomicTransition<'All'>`**

**`nebular = AtomicTransition<'Neb'>`**

CHAPTER 24

# ALMA Queries (astroquery.alma)

## 24.1 Example Notebooks

A series of example notebooks can be found here:

- What has ALMA observed toward all Messier objects? (an example of querying many sources)
- ALMA finder chart of the Cartwheel galaxy and public Cycle 1 data quicklooks
- Finder charts toward many sources with different backgrounds
- Finder chart and downloaded data from Cycle 0 observations of Sombrero Galaxy

## 24.2 Getting started

astroquery.alma provides the astroquery interface to the ALMA archive. It supports object and region based querying and data staging and retrieval.

You can get interactive help to find out what keywords to query for:

```
>>> from astroquery.alma import Alma
>>> Alma.help()
Valid ALMA keywords:

Position
  Source name (Resolver)           : source_name_resolver
  Source name (ALMA)               : source_name_alma
  RA Dec                           : ra_dec

Energy
  Frequency                        : frequency
  Bandwidth                        : bandwidth
  Spectral resolution              : spectral_resolution
  Band                             : 3(84-116 GHz) = 3 , 4(125-163 GHz) = 4 , 6(211-275 GHz) = 6 ,
↪7(275-373 GHz) = 7 , 8(385-500 GHz) = 8 , 9(602-720 GHz) = 9 , 10(787-950 GHz) = 10
```

(continues on next page)

(continued from previous page)

```
Time
  Observation date                   : start_date
  Integration time                   : integration_time

Polarisation
  Polarisation type                  : Stokes I = 0 , Single = 1 , Dual = 2 , Full = =3|4

Observation
  Water vapour                       : water_vapour

Project
  Project code                       : project_code
  Project title                      : project_title
  PI name                            : pi_name

Options
  (I) View:                          : result_view          = raw
  ( ) View:                          : result_view          = project
  [x] public data only               : public_data          = public
  [x] science observations only      : science_observations = =%TARGET%
```

## 24.3 Authentication

Users can log in to acquire proprietary data products. Login is performed via the ALMA CAS (central authentication server).

```
>>> from astroquery.alma import Alma
>>> alma = Alma()
>>> # First example: TEST is not a valid username, it will fail
>>> alma.login("TEST")
TEST, enter your ALMA password:

Authenticating TEST on asa.alma.cl...
Authentication failed!
>>> # Second example: pretend ICONDOR is a valid username
>>> alma.login("ICONDOR", store_password=True)
ICONDOR, enter your ALMA password:

Authenticating ICONDOR on asa.alma.cl...
Authentication successful!
>>> # After the first login, your password has been stored
>>> alma.login("ICONDOR")
Authenticating ICONDOR on asa.alma.cl...
Authentication successful!
```

Your password will be stored by the keyring module. You can choose not to store your password by passing the argument `store_password=False` to `Alma.login`. You can delete your password later with the command `keyring.delete_password('astroquery:asa.alma.cl', 'username')`.

## 24.4 Querying Targets and Regions

You can query by object name or by circular region:

```
>>> from astroquery.alma import Alma
>>> m83_data = Alma.query_object('M83')
>>> print(len(m83_data))
830
>>> m83_data.colnames
['Project code', 'Source name', 'RA', 'Dec', 'Band',
'Frequency resolution', 'Integration', 'Release date', 'Frequency support',
'Velocity resolution', 'Pol products', 'Observation date', 'PI name',
'PWV', 'Member ous id', 'Asdm uid', 'Project title', 'Project type',
'Scan intent', 'Spatial resolution', 'QA0 Status', 'QA2 Status']
```

Region queries are just like any other in astroquery:

```
>>> from astropy import coordinates
>>> from astropy import units as u
>>> galactic_center = coordinates.SkyCoord(0*u.deg, 0*u.deg,
...                                        frame='galactic')
>>> gc_data = Alma.query_region(galactic_center, 1*u.deg)
>>> print(len(gc_data))
383
```

## 24.5 Querying by other parameters

As of version 0.3.4, you can also query other fields by keyword. For example, if you want to find all projects with a particular PI, you could do:

```
>>> rslt = Alma.query_object('W51', pi_name='Ginsburg', public=False)
```

## 24.6 Downloading Data

You can download ALMA data with astroquery, but be forewarned, cycle 0 and cycle 1 data sets tend to be >100 GB!

```
>>> import numpy as np
>>> uids = np.unique(m83_data['Member ous id'])
>>> print(uids)
     Member ous id
 -----------------------
  uid://A002/X3216af/X31
 uid://A002/X5a9a13/X689
```

You can then stage the data and see how big it is (you can ask for one or more UIDs):

```
>>> link_list = Alma.stage_data(uids)
INFO: Staging files... [astroquery.alma.core]
>>> link_list['size'].sum()
159.26999999999998
```

You can then go on to download that data. The download will be cached so that repeat queries of the same file will not re-download the data. The default cache directory is `~/.astropy/cache/astroquery/Alma/`, but this can be changed by changing the `cache_location` variable:

```
>>> myAlma = Alma()
>>> myAlma.cache_location = '/big/external/drive/'
>>> myAlma.download_files(link_list, cache=True)
```

You can also do the downloading all in one step:

```
>>> myAlma.retrieve_data_from_uid(uids[0])
```

## 24.7 Downloading FITS data

If you want just the QA2-produced FITS files, you can download the tarball, extract the FITS file, then delete the tarball:

```
>>> from astroquery.alma.core import Alma
>>> from astropy import coordinates
>>> from astropy import units as u
>>> orionkl = coordinates.SkyCoord('5:35:14.461 -5:21:54.41', frame='fk5',
...                                unit=(u.hour, u.deg))
>>> result = Alma.query_region(orionkl, radius=0.034*u.deg)
>>> uid_url_table = Alma.stage_data(result['Member ous id'])
>>> # Extract the data with tarball file size < 1GB
>>> small_uid_url_table = uid_url_table[uid_url_table['size'] < 1]
>>> # get the first 10 files...
>>> filelist = Alma.download_and_extract_files(small_uid_url_table[:10]['URL'])
```

You might want to look at the READMEs from a bunch of files so you know what kind of S/N to expect:

```
>>> filelist = Alma.download_and_extract_files(uid_url_table['URL'], regex='.*README$')
```

## 24.8 Further Examples

There are some nice examples of using the ALMA query tool in conjunction with other astroquery tools in *A Gallery of Queries*, especially *Example 7*.

## 24.9 Reference/API

### 24.9.1 astroquery.alma Package

ALMA Archive service.

#### Classes

| | |
|---|---|
| `AlmaClass`() | |
| `Conf` | Configuration parameters for `astroquery.alma`. |

## AlmaClass

**class** astroquery.alma.**AlmaClass**

Bases: astroquery.query.QueryWithLogin

### Attributes Summary

| | |
|---|---|
| TIMEOUT | |
| USERNAME | |
| archive_url | |
| cycle0_table | Return a table of Cycle 0 Project IDs and associated UIDs. |

### Methods Summary

| | |
|---|---|
| download_and_extract_files(urls[, delete, ...]) | Given a list of tarball URLs: |
| download_files(files[, savedir, cache, ...]) | Given a list of file URLs, download them |
| get_cycle0_uid_contents(uid) | List the file contents of a UID from Cycle 0. |
| get_files_from_tarballs(downloaded_files[, ...]) | Given a list of successfully downloaded tarballs, extract files with names matching a specified regular expression. |
| help([cache]) | Return the valid query parameters |
| query(*args, **kwargs) | Queries the service and returns a table object. |
| query_async(payload[, cache, public, ...]) | Perform a generic query with user-specified payload |
| query_object(*args, **kwargs) | Queries the service and returns a table object. |
| query_object_async(object_name[, cache, ...]) | Query the archive with a source name |
| query_region(*args, **kwargs) | Queries the service and returns a table object. |
| query_region_async(coordinate, radius[, ...]) | Query the ALMA archive with a source name and radius |
| retrieve_data_from_uid(uids[, cache]) | Stage & Download ALMA data. |
| stage_data(uids) | Stage ALMA data |
| validate_query(payload[, cache]) | Use the ALMA query validator service to check whether the keywords are valid |

### Attributes Documentation

**TIMEOUT = 60**

**USERNAME = ''**

**archive_url = 'http://almascience.org'**

**cycle0_table**

Return a table of Cycle 0 Project IDs and associated UIDs.

The table is distributed with astroquery and was provided by Felix Stoehr.

### Methods Documentation

**download_and_extract_files**(*urls*, *delete=True*, *regex='.*\\.fits$'*, *include_asdm=False*, *path='cache_path'*, *verbose=True*)

Given a list of tarball URLs:

1. Download the tarball
2. Extract all FITS files (or whatever matches the regex)
3. Delete the downloaded tarball

See `Alma.get_files_from_tarballs` for details

**Parameters**

**urls** : str or list

A single URL or a list of URLs

**include_asdm** : bool

Only affects cycle 1+ data. If set, the ASDM files will be downloaded in addition to the script and log files. By default, though, this file will be downloaded and deleted without extracting any information: you must change the regex if you want to extract data from an ASDM tarball

**download_files**(*files*, *savedir=None*, *cache=True*, *continuation=True*)

Given a list of file URLs, download them

Note: Given a list with repeated URLs, each will only be downloaded once, so the return may have a different length than the input list

**get_cycle0_uid_contents**(*uid*)

List the file contents of a UID from Cycle 0. Will raise an error if the UID is from cycle 1+, since those data have been released in a different and more consistent format. See http://almascience.org/documents-and-tools/cycle-2/ALMAQA2Productsv1.01.pdf for details.

**get_files_from_tarballs**(*downloaded_files*, *regex='.*\\.fits$'*, *path='cache_path'*, *verbose=True*)

Given a list of successfully downloaded tarballs, extract files with names matching a specified regular expression. The default is to extract all FITS files

**Parameters**

**downloaded_files** : list

A list of downloaded files. These should be paths on your local machine.

**regex** : str

A valid regular expression

**path** : 'cache_path' or str

If 'cache_path', will use the astroquery.Alma cache directory (`Alma.cache_location`), otherwise will use the specified path. Note that the subdirectory structure of the tarball will be maintained.

**Returns**

**filelist** : list

A list of the extracted file locations on disk

**help**(*cache=True*)

Return the valid query parameters

**query**(**args*, ***kwargs*)
Queries the service and returns a table object.

Perform a generic query with user-specified payload

**Parameters**
**payload** : dict

A dictionary of payload keywords that are accepted by the ALMA archive system. You can look these up by examining the forms at http://almascience.org/aq or using the `help` method

**cache** : bool

Cache the query? (note: HTML queries *cannot* be cached using the standard caching mechanism because the URLs are different each time

**public** : bool

Return only publicly available datasets?

**science** : bool

Return only data marked as "science" in the archive?

**Returns**
**table** : A `Table` object.

**query_async**(*payload*, *cache=True*, *public=True*, *science=True*, *max_retries=5*, *get_html_version=False*, *get_query_payload=False*, ***kwargs*)
Perform a generic query with user-specified payload

**Parameters**
**payload** : dict

A dictionary of payload keywords that are accepted by the ALMA archive system. You can look these up by examining the forms at http://almascience.org/aq or using the `help` method

**cache** : bool

Cache the query? (note: HTML queries *cannot* be cached using the standard caching mechanism because the URLs are different each time

**public** : bool

Return only publicly available datasets?

**science** : bool

Return only data marked as "science" in the archive?

**query_object**(**args*, ***kwargs*)
Queries the service and returns a table object.

Query the archive with a source name

**Parameters**
**object_name** : str

The object name. Will be parsed by SESAME on the ALMA servers.

**cache** : bool

Cache the query?

**public** : bool

Return only publicly available datasets?

**science** : bool

Return only data marked as "science" in the archive?

**payload** : dict

Dictionary of additional keywords. See `help`.

**kwargs** : dict

Passed to `query_async`

**Returns**

**table** : A `Table` object.

**query_object_async**(*object_name*, *cache=True*, *public=True*, *science=True*, *payload=None*, ***kwargs*)

Query the archive with a source name

**Parameters**

**object_name** : str

The object name. Will be parsed by SESAME on the ALMA servers.

**cache** : bool

Cache the query?

**public** : bool

Return only publicly available datasets?

**science** : bool

Return only data marked as "science" in the archive?

**payload** : dict

Dictionary of additional keywords. See `help`.

**kwargs** : dict

Passed to `query_async`

**query_region**(**args*, ***kwargs*)

Queries the service and returns a table object.

Query the ALMA archive with a source name and radius

**Parameters**

**coordinates** : str / `astropy.coordinates`

the identifier or coordinates around which to query.

**radius** : str / `Quantity`, optional

the radius of the region

**cache** : bool

Cache the query?

**public** : bool

Return only publicly available datasets?

**science** : bool

Return only data marked as "science" in the archive?

**payload** : dict

Dictionary of additional keywords. See `help`.

**kwargs** : dict

Passed to `query_async`

**Returns**

**table** : A `Table` object.

`query_region_async`(*coordinate*, *radius*, *cache=True*, *public=True*, *science=True*, *payload=None*, ***kwargs*)

Query the ALMA archive with a source name and radius

**Parameters**

**coordinates** : str / `astropy.coordinates`

the identifier or coordinates around which to query.

**radius** : str / `Quantity`, optional

the radius of the region

**cache** : bool

Cache the query?

**public** : bool

Return only publicly available datasets?

**science** : bool

Return only data marked as "science" in the archive?

**payload** : dict

Dictionary of additional keywords. See `help`.

**kwargs** : dict

Passed to `query_async`

`retrieve_data_from_uid`(*uids*, *cache=True*)

Stage & Download ALMA data. Will print out the expected file size before attempting the download.

**Parameters**

**uids** : list or str

A list of valid UIDs or a single UID. UIDs should have the form: 'uid://A002/X391d0b/X7b'

**cache** : bool

Whether to cache the downloads.

**Returns**

**downloaded_files** : list

A list of the downloaded file paths

`stage_data`(*uids*)

Stage ALMA data

**Parameters**

**uids** : list or str

A list of valid UIDs or a single UID. UIDs should have the form: 'uid://A002/X391d0b/X7b'

**Returns**

**data_file_table** : Table

A table containing 3 columns: the UID, the file URL (for future downloading), and the file size

**validate_query**(*payload*, *cache=True*)

Use the ALMA query validator service to check whether the keywords are valid

### Conf

**class** astroquery.alma.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.alma`.

#### Attributes Summary

| | |
|---|---|
| `archive_url` | The ALMA Archive mirror to use. |
| `timeout` | Timeout in seconds. |
| `username` | Optional default username for ALMA archive. |

#### Attributes Documentation

**archive_url**

The ALMA Archive mirror to use.

**timeout**

Timeout in seconds.

**username**

Optional default username for ALMA archive.

## 24.9.2 astroquery.alma.utils Module

Utilities for making finder charts and overlay images for ALMA proposing

### Functions

| | |
|---|---|
| `add_meta_to_reg`(reg, meta) | |
| `approximate_primary_beam_sizes`(…[, …]) | Using parse_frequency_support, determine the mean primary beam size in each observed band |

Continued on next page

Table 5 – continued from previous page

| | |
|---|---|
| footprint_to_reg(footprint) | ALMA footprints have the form: 'Polygon ICRS 266.519781 -28.724666 266.524678 -28.731930 266.536683 -28.737784 266.543860 -28.737586 266.549277 -28.733370 266.558133 -28.729545 266.560136 -28.724666 266.558845 -28.719605 266.560133 -28.694332 266.555234 -28.687069 266.543232 -28.681216 266.536058 -28.681414 266.530644 -28.685630 266.521788 -28.689453 266.519784 -28.694332 266.521332 -28.699778' Some of them have *additional* polygons |
| make_finder_chart(target, radius, save_prefix) | Create a "finder chart" showing where ALMA has pointed in various bands, including different color coding for public/private data and each band. |
| make_finder_chart_from_image(image, target, ...) | Create a "finder chart" showing where ALMA has pointed in various bands, including different color coding for public/private data and each band. |
| make_finder_chart_from_image_and_catalog(...) | Create a "finder chart" showing where ALMA has pointed in various bands, including different color coding for public/private data and each band. |
| parse_frequency_support(frequency_support_str) | ALMA "Frequency Support" strings have the form: |
| pyregion_subset(region, data, mywcs) | Return a subset of an image (data) given a region. |

### add_meta_to_reg

astroquery.alma.utils.**add_meta_to_reg**(*reg*, *meta*)

### approximate_primary_beam_sizes

astroquery.alma.utils.**approximate_primary_beam_sizes**(*frequency_support_str*, *dish_diameter=<Quantity 12. m>*, *first_null=1.22*)

Using parse_frequency_support, determine the mean primary beam size in each observed band

**Parameters**

**frequency_support_str** : str

The frequency support string, see parse_frequency_support

**dish_diameter** : Quantity

Meter-equivalent unit. The diameter of the dish.

**first_null** : float

The position of the first null of an Airy. Used to compute resolution as $R = 1.22\lambda/D$

### footprint_to_reg

astroquery.alma.utils.**footprint_to_reg**(*footprint*)

ALMA footprints have the form: 'Polygon ICRS 266.519781 -28.724666 266.524678 -28.731930 266.536683 -28.737784 266.543860 -28.737586 266.549277 -28.733370 266.558133 -28.729545 266.560136 -28.724666 266.558845 -28.719605 266.560133 -28.694332 266.555234 -28.687069 266.543232 -28.681216 266.536058

-28.681414 266.530644 -28.685630 266.521788 -28.689453 266.519784 -28.694332 266.521332 -28.699778' Some of them have *additional* polygons

### make_finder_chart

astroquery.alma.utils.**make_finder_chart**(*target*, *radius*, *save_prefix*, *service=<bound method SkyViewClass.get_images of <astroquery.skyview.core.SkyViewClass object>>*, *service_kwargs={'pixels': 500, 'survey': ['2MASS-K']}*, *alma_kwargs={'public': False, 'science': False}*, ***kwargs*)

Create a "finder chart" showing where ALMA has pointed in various bands, including different color coding for public/private data and each band.

Contours are set at various integration times.

**Parameters**

**target** : `astropy.coordinates` or str

A legitimate target name

**radius** : `Quantity`

A degree-equivalent radius.

**save_prefix** : str

The prefix for the output files. Both .reg and .png files will be written. The .reg files will have the band numbers and public/private appended, while the .png file will be named prefix_almafinderchart.png

**service** : function

The `get_images` function of an astroquery service, e.g. SkyView.

**service_kwargs** : dict

The keyword arguments to pass to the specified service. For example, for SkyView, you can give it the survey ID (e.g., 2MASS-K) and the number of pixels in the resulting image. See the documentation for the individual services for more details.

**alma_kwargs** : dict

Keywords to pass to the ALMA archive when querying.

**private_band_colors / public_band_colors** : tuple

A tuple or list of colors to be associated with private/public observations in the various bands

**integration_time_contour_levels** : list or np.array

The levels at which to draw contours in units of seconds. Default is log-spaced (2^n) seconds: [ 1., 2., 4., 8., 16., 32.])

### make_finder_chart_from_image

astroquery.alma.utils.**make_finder_chart_from_image**(*image*, *target*, *radius*, *save_prefix*, *alma_kwargs={'cache': False, 'public': False, 'science': False}*, ***kwargs*)

Create a "finder chart" showing where ALMA has pointed in various bands, including different color coding for public/private data and each band.

Contours are set at various integration times.

**Parameters**

**image** : fits.PrimaryHDU or fits.ImageHDU object

The image to overlay onto

**target** : `astropy.coordinates` or str

A legitimate target name

**radius** : `astropy.units.Quantity`

A degree-equivalent radius

**save_prefix** : str

The prefix for the output files. Both .reg and .png files will be written. The .reg files will have the band numbers and public/private appended, while the .png file will be named prefix_almafinderchart.png

**alma_kwargs** : dict

Keywords to pass to the ALMA archive when querying.

**private_band_colors / public_band_colors** : tuple

A tuple or list of colors to be associated with private/public observations in the various bands

**integration_time_contour_levels** : list or np.array

The levels at which to draw contours in units of seconds. Default is log-spaced (2^n) seconds: [ 1., 2., 4., 8., 16., 32.])

### make_finder_chart_from_image_and_catalog

`astroquery.alma.utils.`**`make_finder_chart_from_image_and_catalog`**(*image*, *catalog*, *save_prefix*, *alma_kwargs={'public': False, 'science': False}*, *bands=(3, 4, 5, 6, 7, 8, 9, 10)*, *private_band_colors=('maroon', 'red', 'orange', 'coral', 'brown', 'yellow', 'mediumorchid', 'palegoldenrod')*, *public_band_colors=('blue', 'cyan', 'green', 'turquoise', 'teal', 'darkslategrey', 'chartreuse', 'lime')*, *integration_time_contour_levels=array([ 1., 2., 4., 8., 16., 32.])*, *save_masks=False*, *use_saved_masks=False*, *linewidth=1*)

Create a "finder chart" showing where ALMA has pointed in various bands, including different color coding for public/private data and each band.

Contours are set at various integration times.

**Parameters**

**image** : fits.PrimaryHDU or fits.ImageHDU object

The image to overlay onto

**catalog** : astropy.Table object

The catalog of ALMA observations

**save_prefix** : str

The prefix for the output files. Both .reg and .png files will be written. The .reg files will have the band numbers and public/private appended, while the .png file will be named prefix_almafinderchart.png

**alma_kwargs** : dict

Keywords to pass to the ALMA archive when querying.

**private_band_colors / public_band_colors** : tuple

A tuple or list of colors to be associated with private/public observations in the various bands

**integration_time_contour_levels** : list or np.array

The levels at which to draw contours in units of seconds. Default is log-spaced (2^n) seconds: [ 1., 2., 4., 8., 16., 32.])

## parse_frequency_support

`astroquery.alma.utils.`**`parse_frequency_support`**(*frequency_support_str*)

ALMA "Frequency Support" strings have the form:

[100.63..101.57GHz,488.28kHz, XX YY] U [102.43..103.37GHz,488.28kHz, XX YY] U [112.74..113.68GHz,488.28kHz, XX YY] U [114.45..115.38GHz,488.28kHz, XX YY]

at least, as far as we have seen. The "U" is meant to be the Union symbol. This function will parse such a string into a list of pairs of astropy Quantities representing the frequency range. It will ignore the resolution and polarizations.

## pyregion_subset

`astroquery.alma.utils.`**`pyregion_subset`**(*region*, *data*, *mywcs*)

Return a subset of an image (`data`) given a region.

**Parameters**

**region** : `Shape`

A Shape from a pyregion-parsed region file

**data** : np.ndarray

An array with shape described by WCS

**mywcs** : `astropy.wcs.WCS`

A world coordinate system describing the data

# CHAPTER 25

## Skyview Queries (astroquery.skyview)

### 25.1 Getting started

The SkyView service offers a cutout service for a number of imaging surveys.

To see the list of surveys, use the list_surveys method:

```
>>> from astroquery.skyview import SkyView
>>> SkyView.list_surveys()
{'DiffuseX-ray': [u'RASS Background 1',
                  u'RASS Background 2',
                  u'RASS Background 3',
                  u'RASS Background 4',
                  u'RASS Background 5',
                  u'RASS Background 6',
                  u'RASS Background 7'],
 'GOODS/HDF/CDF(Allwavebands)': [u'GOODS: Chandra ACIS HB',
                                 u'GOODS: Chandra ACIS FB',
                                 u'GOODS: Chandra ACIS SB',
                                 u'GOODS: VLT VIMOS U',
                                 u'GOODS: VLT VIMOS R',
                                 u'GOODS: HST ACS B',
                                 u'GOODS: HST ACS V',
                                 u'GOODS: HST ACS I',
                                 u'GOODS: HST ACS Z',
                                 u'Hawaii HDF U',
                                 u'Hawaii HDF B',
                                 u'Hawaii HDF V0201',
                                 u'Hawaii HDF V0401',
                                 u'Hawaii HDF R',
                                 u'Hawaii HDF I',
                                 u'Hawaii HDF z',
                                 u'Hawaii HDF HK',
                                 u'GOODS: HST NICMOS',
```

(continues on next page)

(continued from previous page)

```
                                u'GOODS: VLT ISAAC J',
                                u'GOODS: VLT ISAAC H',
                                u'GOODS: VLT ISAAC Ks',
                                u'HUDF: VLT ISAAC Ks',
                                u'GOODS: Spitzer IRAC 3.6',
                                u'GOODS: Spitzer IRAC 4.5',
                                u'GOODS: Spitzer IRAC 5.8',
                                u'GOODS: Spitzer IRAC 8.0',
                                u'GOODS: Spitzer MIPS 24',
                                u'GOODS: Herschel 100',
                                u'GOODS: Herschel 160',
                                u'GOODS: Herschel 250',
                                u'GOODS: Herschel 350',
                                u'GOODS: Herschel 500',
                                u'CDFS: LESS',
                                u'GOODS: VLA North'],
 'GammaRay': [u'Fermi 5',
              u'Fermi 4',
              u'Fermi 3',
              u'Fermi 2',
              u'Fermi 1',
              u'EGRET (3D)',
              u'EGRET <100 MeV',
              u'EGRET >100 MeV',
              u'COMPTEL'],
 'HardX-ray': [u'INT GAL 17-35 Flux',
               u'INT GAL 17-60 Flux',
               u'INT GAL 35-80 Flux',
               u'INTEGRAL/SPI GC',
               u'GRANAT/SIGMA',
               u'RXTE Allsky 3-8keV Flux',
               u'RXTE Allsky 3-20keV Flux',
               u'RXTE Allsky 8-20keV Flux'],
 'IRAS': [u'IRIS  12',
          u'IRIS  25',
          u'IRIS  60',
          u'IRIS 100',
          u'SFD100m',
          u'SFD Dust Map',
          u'IRAS  12 micron',
          u'IRAS  25 micron',
          u'IRAS  60 micron',
          u'IRAS 100 micron'],
 'InfraredHighRes': [u'2MASS-J',
                     u'2MASS-H',
                     u'2MASS-K',
                     u'UKIDSS-Y',
                     u'UKIDSS-J',
                     u'UKIDSS-H',
                     u'UKIDSS-K',
                     u'WISE 3.4',
                     u'WISE 4.6',
                     u'WISE 12',
                     u'WISE 22'],
 'Optical:DSS': [u'DSS',
                 u'DSS1 Blue',
                 u'DSS1 Red',
```

(continues on next page)

(continued from previous page)

```
                u'DSS2 Red',
                u'DSS2 Blue',
                u'DSS2 IR'],
 'Optical:SDSS': [u'SDSSg',
                 u'SDSSi',
                 u'SDSSr',
                 u'SDSSu',
                 u'SDSSz',
                 u'SDSSdr7g',
                 u'SDSSdr7i',
                 u'SDSSdr7r',
                 u'SDSSdr7u',
                 u'SDSSdr7z'],
 'OtherOptical': [u'Mellinger Red',
                 u'Mellinger Green',
                 u'Mellinger Blue',
                 u'NEAT',
                 u'H-Alpha Comp',
                 u'SHASSA H',
                 u'SHASSA CC',
                 u'SHASSA C',
                 u'SHASSA Sm'],
 'Planck': [u'Planck 857',
           u'Planck 545',
           u'Planck 353',
           u'Planck 217',
           u'Planck 143',
           u'Planck 100',
           u'Planck 070',
           u'Planck 044',
           u'Planck 030'],
 'Radio': [u'GB6 (4850MHz)',
          u'VLA FIRST (1.4 GHz)',
          u'NVSS',
          u'Stripe82VLA',
          u'1420MHz (Bonn)',
          u'nH',
          u'SUMSS 843 MHz',
          u'0408MHz',
          u'WENSS',
          u'CO',
          u'VLSSr',
          u'0035MHz'],
 'SoftX-ray': [u'RASS-Cnt Soft',
              u'RASS-Cnt Hard',
              u'RASS-Cnt Broad',
              u'PSPC 2.0 Deg-Int',
              u'PSPC 1.0 Deg-Int',
              u'PSPC 0.6 Deg-Int',
              u'HRI',
              u'HEAO 1 A-2'],
 'SwiftBAT': [u'BAT SNR 14-195',
             u'BAT SNR 14-20',
             u'BAT SNR 20-24',
             u'BAT SNR 24-35',
             u'BAT SNR 35-50',
             u'BAT SNR 50-75',
```

(continues on next page)

(continued from previous page)

```
              u'BAT SNR 75-100',
              u'BAT SNR 100-150',
              u'BAT SNR 150-195'],
 'UV': [u'GALEX Near UV',
        u'GALEX Far UV',
        u'ROSAT WFC F1',
        u'ROSAT WFC F2',
        u'EUVE 83 A',
        u'EUVE 171 A',
        u'EUVE 405 A',
        u'EUVE 555 A'],
 'WMAP/COBE': [u'WMAP ILC',
               u'WMAP Ka',
               u'WMAP K',
               u'WMAP Q',
               u'WMAP V',
               u'WMAP W',
               u'COBE DIRBE/AAM',
               u'COBE DIRBE/ZSMA']}
```

There are two essential methods: `get_images` searches for and downloads files, while `get_image_list` just searches for the files.

```
>>> paths = SkyView.get_images(position='Eta Carinae',
...                            survey=['Fermi 5', 'HRI', 'DSS'])
Downloading http://skyview.gsfc.nasa.gov/tempspace/fits/skv668576311417_1.fits
|========================================================================================================================
↪371k/371k (100.00%)         0s
Downloading http://skyview.gsfc.nasa.gov/tempspace/fits/skv668576311417_2.fits
|========================================================================================================================
↪371k/371k (100.00%)         0s
Downloading http://skyview.gsfc.nasa.gov/tempspace/fits/skv668576311417_3.fits
|========================================================================================================================
↪374k/374k (100.00%)         0s
>>> print(paths)
[[<astropy.io.fits.hdu.image.PrimaryHDU object at 0x10ef3a250>], [<astropy.io.fits.hdu.image.PrimaryHDU␣
↪object at 0x10f096f10>], [<astropy.io.fits.hdu.image.PrimaryHDU object at 0x10f0aea50>]]
```

Without the download:

```
>>> SkyView.get_image_list(position='Eta Carinae',
                  survey=['Fermi 5', 'HRI', 'DSS'])
[u'http://skyview.gsfc.nasa.gov/tempspace/fits/skv669807193757_1.fits',
 u'http://skyview.gsfc.nasa.gov/tempspace/fits/skv669807193757_2.fits',
 u'http://skyview.gsfc.nasa.gov/tempspace/fits/skv669807193757_3.fits']
```

## 25.2 Reference/API

### 25.2.1 astroquery.skyview Package

#### Classes

| | |
|---|---|
| SkyViewClass() | |
| Conf | Configuration parameters for astroquery.skyview. |

## SkyViewClass

**class** astroquery.skyview.**SkyViewClass**

Bases: astroquery.query.BaseQuery

### Attributes Summary

| | |
|---|---|
| URL | |
| survey_dict | |

### Methods Summary

| | |
|---|---|
| get_image_list(position, survey[, ...]) | Query the SkyView service, download the FITS file that will be found and return a generator over the local paths to the downloaded FITS files. |
| get_images(position, survey[, coordinates, ...]) | Query the SkyView service, download the FITS file that will be found and return a generator over the local paths to the downloaded FITS files. |
| get_images_async(position, survey[, ...]) | Query the SkyView service, download the FITS file that will be found and return a generator over the local paths to the downloaded FITS files. |
| list_surveys() | Print out a formatted version of the survey dict |

### Attributes Documentation

**URL = 'http://skyview.gsfc.nasa.gov/current/cgi/basicform.pl'**

**survey_dict**

### Methods Documentation

**get_image_list**(*position*, *survey*, *coordinates=None*, *projection=None*, *pixels=None*, *scaling=None*, *sampler=None*, *resolver=None*, *deedger=None*, *lut=None*, *grid=None*, *gridlabels=None*, *radius=None*, *width=None*, *height=None*, *cache=True*)

Query the SkyView service, download the FITS file that will be found and return a generator over the local paths to the downloaded FITS files.

Note that the files will be downloaded when the generator will be exhausted, i.e. just calling this method alone without iterating over the result won't issue a connection to the SkyView server.

**Parameters**

**position** : str

Determines the center of the field to be retrieved. Both coordinates (also equatorial ones) and object names are supported. Object names are converted to coordinates via the SIMBAD or NED name resolver. See the reference for more info on the supported syntax for coordinates.

**survey** : str or list of str

Select data from one or more surveys. The number of surveys determines the number of resulting file downloads. Passing a list with just one string has the same effect as passing this string directly.

**coordinates** : str

Choose among common equatorial, galactic and ecliptic coordinate systems (`"J2000"`, `"B1950"`, `"Galactic"`, `"E2000"`, `"ICRS"`) or pass a custom string.

**projection** : str

Choose among the map projections (the value in parentheses denotes the string to be passed):

**Gnomonic (Tan), default value**
good for small regions

**Rectangular (Car)**
simplest projection

**Aitoff (Ait)**
Hammer-Aitoff, equal area projection good for all sky maps

**Orthographic (Sin)**
Projection often used in interferometry

**Zenith Equal Area (Zea)**
equal area, azimuthal projection

**COBE Spherical Cube (Csc)**
Used in COBE data

**Arc (Arc)**
Similar to Zea but not equal-area

**pixels** : str

Selects the pixel dimensions of the image to be produced. A scalar value or a pair of values separated by comma may be given. If the value is a scalar the number of width and height of the image will be the same. By default a 300x300 image is produced.

**scaling** : str

Selects the transformation between pixel intensity and intensity on the displayed image. The supported values are: `"Log"`, `"Sqrt"`, `"Linear"`, `"HistEq"`, `"LogLog"`.

**sampler** : str

The sampling algorithm determines how the data requested will be resampled so that it can be displayed.

**resolver** : str

The name resolver allows to choose a name resolver to use when looking up a name which was passed in the `position` parameter (as opposed to a numeric coordinate value). The default choice is to call the SIMBAD name resolver first and then the NED name resolver if the SIMBAD search fails.

**deedger** : str

When multiple input images with different backgrounds are resampled the edges between the images may be apparent because of the background shift. This parameter makes it possible to attempt to minimize these edges by applying a de-edging algorithm. The user can elect to choose the default given for that survey, to turn de-edging off, or to use the default de-edging algorithm. The supported values are: `"_skip_"` to use the survey default, `"skyview.process.Deedger"` (for enabling de-edging), and `"null"` to disable.

**lut** : str

Choose from the color table selections to display the data in false color.

**grid** : bool

overlay a coordinate grid on the image if True

**gridlabels** : bool

annotate the grid with coordinates positions if True

**radius** : `Quantity` or None

The radius of the specified field. Overrides width and height.

**width** : `Quantity` or None

The width of the specified field. Must be specified with `height`.

**height** : `Quantity` or None

The height of the specified field. Must be specified with `width`.

**Returns**

list of image urls

### References

*[R4]*

### Examples

```
>>> SkyView().get_image_list(position='Eta Carinae',
...                          survey=['Fermi 5', 'HRI', 'DSS'])
[u'http://skyview.gsfc.nasa.gov/tempspace/fits/skv6183161285798_1.fits',
 u'http://skyview.gsfc.nasa.gov/tempspace/fits/skv6183161285798_2.fits',
 u'http://skyview.gsfc.nasa.gov/tempspace/fits/skv6183161285798_3.fits']
```

**get_images**(*position*, *survey*, *coordinates=None*, *projection=None*, *pixels=None*, *scaling=None*, *sampler=None*, *resolver=None*, *deedger=None*, *lut=None*, *grid=None*, *gridlabels=None*, *radius=None*, *height=None*, *width=None*, *cache=True*, *show_progress=True*)

Query the SkyView service, download the FITS file that will be found and return a generator over the local paths to the downloaded FITS files.

Note that the files will be downloaded when the generator will be exhausted, i.e. just calling this method alone without iterating over the result won't issue a connection to the SkyView server.

**Parameters**

**position** : str

Determines the center of the field to be retrieved. Both coordinates (also equatorial ones) and object names are supported. Object names are converted to coordinates via the SIMBAD or NED name resolver. See the reference for more info on the supported syntax for coordinates.

**survey** : str or list of str

Select data from one or more surveys. The number of surveys determines the number of resulting file downloads. Passing a list with just one string has the same effect as passing this string directly.

**coordinates** : str

Choose among common equatorial, galactic and ecliptic coordinate systems (`"J2000"`, `"B1950"`, `"Galactic"`, `"E2000"`, `"ICRS"`) or pass a custom string.

**projection** : str

Choose among the map projections (the value in parentheses denotes the string to be passed):

**Gnomonic (Tan), default value**
good for small regions

**Rectangular (Car)**
simplest projection

**Aitoff (Ait)**
Hammer-Aitoff, equal area projection good for all sky maps

**Orthographic (Sin)**
Projection often used in interferometry

**Zenith Equal Area (Zea)**
equal area, azimuthal projection

**COBE Spherical Cube (Csc)**
Used in COBE data

**Arc (Arc)**
Similar to Zea but not equal-area

**pixels** : str

Selects the pixel dimensions of the image to be produced. A scalar value or a pair of values separated by comma may be given. If the value is a scalar the number of width and height of the image will be the same. By default a 300x300 image is produced.

**scaling** : str

Selects the transformation between pixel intensity and intensity on the displayed image. The supported values are: `"Log"`, `"Sqrt"`, `"Linear"`, `"HistEq"`, `"LogLog"`.

**sampler** : str

The sampling algorithm determines how the data requested will be resampled so that it can be displayed.

**resolver** : str

The name resolver allows to choose a name resolver to use when looking up a name which was passed in the `position` parameter (as opposed to a numeric coordinate value). The default choice is to call the SIMBAD name resolver first and then the NED name resolver if the SIMBAD search fails.

**deedger** : str

When multiple input images with different backgrounds are resampled the edges between the images may be apparent because of the background shift. This parameter makes it possible to attempt to minimize these edges by applying a de-edging algorithm. The user can elect to choose the default given for that survey, to turn de-edging off, or to use the default de-edging algorithm. The supported values are: `"_skip_"` to use the survey default, `"skyview.process.Deedger"` (for enabling de-edging), and `"null"` to disable.

**lut** : str

Choose from the color table selections to display the data in false color.

**grid** : bool

overlay a coordinate grid on the image if True

**gridlabels** : bool

annotate the grid with coordinates positions if True

**radius** : `Quantity` or None

The radius of the specified field. Overrides width and height.

**width** : `Quantity` or None

The width of the specified field. Must be specified with `height`.

**height** : `Quantity` or None

The height of the specified field. Must be specified with `width`.

**Returns**

A list of `HDUList` objects.

### References

*[R5]*

### Examples

```
>>> sv = SkyView()
>>> paths = sv.get_images(position='Eta Carinae',
...                       survey=['Fermi 5', 'HRI', 'DSS'])
>>> for path in paths:
...     print ' new file:', path
```

**get_images_async**(*position, survey, coordinates=None, projection=None, pixels=None, scaling=None, sampler=None, resolver=None, deedger=None, lut=None, grid=None, gridlabels=None, radius=None, height=None, width=None, cache=True, show_progress=True*)

Query the SkyView service, download the FITS file that will be found and return a generator over the local paths to the downloaded FITS files.

Note that the files will be downloaded when the generator will be exhausted, i.e. just calling this method alone without iterating over the result won't issue a connection to the SkyView server.

**Parameters**

**position** : str

Determines the center of the field to be retrieved. Both coordinates (also equatorial ones) and object names are supported. Object names are converted to coordinates via the SIMBAD or NED name resolver. See the reference for more info on the supported syntax for coordinates.

**survey** : str or list of str

Select data from one or more surveys. The number of surveys determines the number of resulting file downloads. Passing a list with just one string has the same effect as passing this string directly.

**coordinates** : str

Choose among common equatorial, galactic and ecliptic coordinate systems (`"J2000"`, `"B1950"`, `"Galactic"`, `"E2000"`, `"ICRS"`) or pass a custom string.

**projection** : str

Choose among the map projections (the value in parentheses denotes the string to be passed):

**Gnomonic (Tan), default value**
good for small regions

**Rectangular (Car)**
simplest projection

**Aitoff (Ait)**
Hammer-Aitoff, equal area projection good for all sky maps

**Orthographic (Sin)**
Projection often used in interferometry

**Zenith Equal Area (Zea)**
equal area, azimuthal projection

**COBE Spherical Cube (Csc)**
Used in COBE data

**Arc (Arc)**
Similar to Zea but not equal-area

**pixels** : str

Selects the pixel dimensions of the image to be produced. A scalar value or a pair of values separated by comma may be given. If the value is a scalar the number of width and height of the image will be the same. By default a 300x300 image is produced.

**scaling** : str

Selects the transformation between pixel intensity and intensity on the displayed image. The supported values are: `"Log"`, `"Sqrt"`, `"Linear"`, `"HistEq"`, `"LogLog"`.

**sampler** : str

The sampling algorithm determines how the data requested will be resampled so that it can be displayed.

**resolver** : str

The name resolver allows to choose a name resolver to use when looking up a name which was passed in the `position` parameter (as opposed to a numeric coordinate value). The default choice is to call the SIMBAD name resolver first and then the NED name resolver if the SIMBAD search fails.

**deedger** : str

When multiple input images with different backgrounds are resampled the edges between the images may be apparent because of the background shift. This parameter makes it possible to attempt to minimize these edges by applying a de-edging algorithm. The user can elect to choose the default given for that survey, to turn de-edging off, or to use the default de-edging algorithm. The supported values are: `"_skip_"` to use the survey default, `"skyview.process.Deedger"` (for enabling de-edging), and `"null"` to disable.

**lut** : str

Choose from the color table selections to display the data in false color.

**grid** : bool

overlay a coordinate grid on the image if True

**gridlabels** : bool

annotate the grid with coordinates positions if True

**radius** : `Quantity` or None

The radius of the specified field. Overrides width and height.

**width** : `Quantity` or None

The width of the specified field. Must be specified with `height`.

**height** : `Quantity` or None

The height of the specified field. Must be specified with `width`.

**Returns**

A list of context-managers that yield readable file-like objects

### References

*[R6]*

### Examples

```
>>> sv = SkyView()
>>> paths = sv.get_images(position='Eta Carinae',
...                       survey=['Fermi 5', 'HRI', 'DSS'])
>>> for path in paths:
...     print ' new file:', path
```

**list_surveys()**

Print out a formatted version of the survey dict

## Conf

**class** astroquery.skyview.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.skyview`.

### Attributes Summary

| | |
|---|---|
| `url` | SkyView URL |

### Attributes Documentation

**url**
SkyView URL

CHAPTER 26

# NASA ADS Queries (astroquery.nasa_ads)

## 26.1 Getting Started

This module provides an interface to the online SAO/NASA Astrophysics Data System. At the moment only the "simple search", i.e. omni-box search is available, and only a subset of the results are accessible.

## 26.2 Examples

### 26.2.1 Search works by specific identifier

```
from astroquery import nasa_ads as na
# the "^" makes ADS to return only papers where Persson
# is first author
results = na.ADS.query_simple('^Persson Origin of water\
     around deeply embedded low-mass protostars') results[0].title

# to sort after publication date
results.sort(['pubdate'])

#  get the title of the last hit
title = results[-1]['title'][0]

# printout the authors of the last hit
print(results[-1]['authors'])
```

### 26.2.2 Get links

Not yet implemented.

### 26.2.3 Download publisher/ArXiv PDF

Not yet implemented.

### 26.2.4 Get Bibtex

Not yet implemented.

## 26.3 Reference/API

### 26.3.1 astroquery.nasa_ads Package

#### SAO/NASA ADS Query Tool

**Author**
Magnus Vilhelm Persson (magnusp@vilhelm.nu)

#### Classes

| | |
|---|---|
| `ADSClass`(*args) | set some parameters |
| `Conf` | Configuration parameters for `astroquery.nasa_ads`. |

#### ADSClass

**class** astroquery.nasa_ads.**ADSClass**(**args*)
Bases: `astroquery.query.BaseQuery`

set some parameters

##### Attributes Summary

| |
|---|
| `QUERY_ADVANCED_PATH` |
| `QUERY_ADVANCED_URL` |
| `QUERY_SIMPLE_PATH` |
| `QUERY_SIMPLE_URL` |
| `SERVER` |
| `TIMEOUT` |

##### Methods Summary

| | |
|---|---|
| `query_simple`(query_string[, ...]) | Basic query. |

#### Attributes Documentation

`QUERY_ADVANCED_PATH = '/cgi-bin/nph-abs_connect'`

`QUERY_ADVANCED_URL = 'http://adswww.harvard.edu/cgi-bin/nph-abs_connect'`

`QUERY_SIMPLE_PATH = '/cgi-bin/basic_connect'`

`QUERY_SIMPLE_URL = 'http://adswww.harvard.edu/cgi-bin/basic_connect'`

`SERVER = 'http://adswww.harvard.edu'`

`TIMEOUT = 120`

#### Methods Documentation

**query_simple**(*query_string*, *get_query_payload=False*, *get_raw_response=False*, *cache=True*)
Basic query. Uses a string and the ADS generic query.

### Conf

**class** `astroquery.nasa_ads.Conf`
Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.nasa_ads`.

#### Attributes Summary

| | |
|---|---|
| `advanced_path` | Path for advanced query (unconfirmed) |
| `mirrors` | SAO/NASA ADS mirrors around the world |
| `server` | SAO/NASA ADS main server. |
| `simple_path` | Path for simple query (return XML) |
| `timeout` | Time limit for connecting to ADS server |

#### Attributes Documentation

**advanced_path**
Path for advanced query (unconfirmed)

**mirrors**
SAO/NASA ADS mirrors around the world

**server**
SAO/NASA ADS main server.

**simple_path**
Path for simple query (return XML)

**timeout**
Time limit for connecting to ADS server

CHAPTER 27

# HEASARC Queries (astroquery.heasarc)

## 27.1 Getting started

This is a python interface for querying the HEASARC archive web service.

The capabilities are currently very limited ... feature requests and contributions welcome!

### 27.1.1 Getting lists of available datasets

```
>>> from astroquery.heasarc import Heasarc
>>> heasarc = Heasarc()
>>> mission = 'rospublic'
>>> object_name = '3c273'
>>> table = heasarc.query_object(object_name, mission=mission)
>>> table[:3].pprint()
```

### 27.1.2 Downloading identified datasets

Not implemented yet.

## 27.2 Reference/API

### 27.2.1 astroquery.heasarc Package

#### HEASARC

The High Energy Astrophysics Science Archive Research Center (HEASARC) is the primary archive for NASA's (and other space agencies') missions.

The initial version of this was coded in a sprint at the "Python in astronomy" workshop in April 2015 by Jean-Christophe Leyder, Abigail Stevens, Antonio Martin-Carrillo and Christoph Deil.

## Classes

| | |
|---|---|
| `HeasarcClass`() | HEASARC query class. |
| `Conf` | Configuration parameters for `astroquery.heasarc`. |

## HeasarcClass

**class** astroquery.heasarc.**HeasarcClass**

Bases: `astroquery.query.BaseQuery`

HEASARC query class.

### Attributes Summary

| |
|---|
| `TIMEOUT` |
| `URL` |

### Methods Summary

| | |
|---|---|
| `query_object`(*args, **kwargs) | Queries the service and returns a table object. |
| `query_object_async`(object_name, mission[, ...]) | TODO: document this! |

### Attributes Documentation

**TIMEOUT = 30**

**URL = 'http://heasarc.gsfc.nasa.gov/cgi-bin/W3Browse/w3query_noredir.pl'**

### Methods Documentation

**query_object**(**args*, ***kwargs*)

Queries the service and returns a table object.

TODO: document this!

(maybe start by copying over from some other service.)

**Returns**

**table** : A `Table` object.

**query_object_async**(*object_name*, *mission*, *cache=True*, *get_query_payload=False*, *display_mode='FitsDisplay'*)

TODO: document this!

(maybe start by copying over from some other service.)

## Conf

`class astroquery.heasarc.Conf`

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.heasarc`.

### Attributes Summary

| | |
|---|---|
| `server` | Name of the HEASARC server to use. |
| `timeout` | Time limit for connecting to HEASARC server. |

### Attributes Documentation

**server**

Name of the HEASARC server to use.

**timeout**

Time limit for connecting to HEASARC server.

# CHAPTER 28

## Gaia TAP+ (`astroquery.gaia`)

Gaia is an ambitious mission to chart a three-dimensional map of our Galaxy, the Milky Way, in the process revealing the composition, formation and evolution of the Galaxy. Gaia will provide unprecedented positional and radial velocity measurements with the accuracies needed to produce a stereoscopic and kinematic census of about one billion stars in our Galaxy and throughout the Local Group. This amounts to about 1 per cent of the Galactic stellar population.

If you use public Gaia DR1 data in your paper, please take note of our guide on how to acknowledge and cite Gaia DR1.

This package allows the access to the European Space Agency Gaia Archive (http://archives.esac.esa.int/gaia)

Gaia Archive access is based on a TAP+ REST service. TAP+ is an extension of Table Access Protocol (TAP: http://www.ivoa.net/documents/TAP/) specified by the International Virtual Observatory Alliance (IVOA: http://www.ivoa.net).

The TAP query language is Astronomical Data Query Language (ADQL: http://www.ivoa.net/documents/ADQL/2.0), which is similar to Structured Query Language (SQL), widely used to query databases.

TAP provides two operation modes: Synchronous and Asynchronous:

- Synchronous: the response to the request will be generated as soon as the request received by the server. (Do not use this method for queries that generate a big amount of results.)
- Asynchronous: the server will start a job that will execute the request. The first response to the request is the required information (a link) to obtain the job status. Once the job is finished, the results can be retrieved.

Gaia TAP+ server provides two access mode: public and authenticated:

- Public: this is the standard TAP access. A user can execute ADQL queries and upload tables to be used in a query 'on-the-fly' (these tables will be removed once the query is executed). The results are available to any other user and they will remain in the server for a limited space of time.
- Authenticated: some functionalities are restricted to authenticated users only. The results are saved in a private user space and they will remain in the server for ever (they can be removed by the user).
  - ADQL queries and results are saved in a user private area.
  - Cross-match operations: a catalog cross-match operation can be executed. Cross-match operations results are saved in a user private area.

- Persistence of uploaded tables: a user can upload a table in a private space. These tables can be used in queries as well as in cross-matches operations.

This python module provides an Astroquery API access. Nevertheless, only `query_object` and `query_object_async` are implemented.

The Gaia Archive table used for the methods where no table is specified is `gaiadr1.gaia_source`

## 28.1 Examples

### 28.1.1 1. Non authenticated access

#### 1.1. Query object

```
>>> import astropy.units as u
>>> from astropy.coordinates.sky_coordinate import SkyCoord
>>> from astropy.units import Quantity
>>> from astroquery.gaia import Gaia
>>>
>>> coord = SkyCoord(ra=280, dec=-60, unit=(u.degree, u.degree), frame='icrs')
>>> width = Quantity(0.1, u.deg)
>>> height = Quantity(0.1, u.deg)
>>> r = Gaia.query_object_async(coordinate=coord, width=width, height=height)
>>> r.pprint()

         dist             solution_id     ...       ecl_lat
                                          ...      Angle[deg]
--------------------- ------------------- ... -------------------
0.0026029414438061079 1635378410781933568 ... -36.779151653783892
0.0038537557334594502 1635378410781933568 ... -36.773899692008634
0.0045451702670639632 1635378410781933568 ... -36.772645786277522
0.0056131312891700424 1635378410781933568 ... -36.781488832325074
0.0058494547209840585 1635378410781933568 ... -36.770812028764119
0.0062076788443168303 1635378410781933568 ... -36.780588167751368
0.00820184358662692 1 1635378410781933568 ... -36.784730288359086
0.0083377863521668077 1635378410781933568 ... -36.784848302904727
0.0084057202175603796 1635378410781933568 ... -36.784556953222634
0.0092437652172596384 1635378410781933568 ... -36.767784193150469
                  ...                 ... ...                 ...
0.049586988816560117 1635378410781933568 ... -36.824132319326232
0.049717306565450765 1635378410781933568 ... -36.823845008396503
0.049777020825344041 1635378410781933568 ...  -36.72857293240213
0.050385912463710505 1635378410781933568 ... -36.729880776402624
0.050826536195428054 1635378410781933568 ... -36.822968947436181
0.050859645206141363 1635378410781933568 ... -36.823021426398789
0.051040085912766479 1635378410781933568 ... -36.728589237516161
0.051211160779507325 1635378410781933568 ... -36.825120633172546
0.051958453766310551 1635378410781933568 ... -36.725819366872734
0.053207596589671176 1635378410781933568 ... -36.826600298826662
Length = 152 rows
```

### 1.2. Cone search

```
>>> import astropy.units as u
>>> from astropy.coordinates.sky_coordinate import SkyCoord
>>> from astropy.units import Quantity
>>> from astroquery.gaia import Gaia
>>>
>>> coord = SkyCoord(ra=280, dec=-60, unit=(u.degree, u.degree), frame='icrs')
>>> radius = Quantity(1.0, u.deg)
>>> j = Gaia.cone_search_async(coord, radius)
>>> r = j.get_results()
>>> r.pprint()

         dist             solution_id     ...       ecl_lat
                                        ...      Angle[deg]
--------------------- ------------------- ... -------------------
0.0026029414438061079 1635378410781933568 ... -36.779151653783892
0.0038537557334594502 1635378410781933568 ... -36.773899692008634
0.0045451702670639632 1635378410781933568 ... -36.772645786277522
0.0056131312891700424 1635378410781933568 ... -36.781488832325074
0.0058494547209840585 1635378410781933568 ... -36.770812028764119
0.0062076788443168303 1635378410781933568 ... -36.780588167751368
0.008201843586626921 1635378410781933568 ... -36.784730288359086
0.0083377863521668077 1635378410781933568 ... -36.784848302904727
0.0084057202175603796 1635378410781933568 ... -36.784556953222634
0.0092437652172596384 1635378410781933568 ... -36.767784193150469
                  ...                 ... ...                 ...
0.14654733241000259 1635378410781933568 ... -36.667789989774818
0.14657617264211745 1635378410781933568 ... -36.876849099093427
0.14674748663117593 1635378410781933568 ... -36.734323499168184
0.14678063354511475 1635378410781933568 ... -36.845214606267504
0.14679704339818228 1635378410781933568 ... -36.697986781654343
0.14684048305123779 1635378410781933568 ...   -36.6983554058179
0.14684061095346052 1635378410781933568 ... -36.854933118845658
0.14690380253776872 1635378410781933568 ... -36.700207569397797
0.1469069007730108 1635378410781933568 ...  -36.92092859296757
0.14690740362559238 1635378410781933568 ... -36.677757522466912
Length = 2000 rows
```

### 1.3 Getting public tables

To load only table names (TAP+ capability)

```
>>> from astroquery.gaia import Gaia
>>> tables = Gaia.load_tables(only_names=True)
>>> for table in (tables):
>>>   print(table.get_qualified_name())

public.dual
public.tycho2
public.igsl_source
public.hipparcos
public.hipparcos_newreduction
public.hubble_sc
public.igsl_source_catalog_ids
tap_schema.tables
```

(continues on next page)

(continued from previous page)

```
tap_schema.keys
tap_schema.columns
tap_schema.schemas
tap_schema.key_columns
gaiadr1.phot_variable_time_series_gfov
gaiadr1.ppmxl_neighbourhood
gaiadr1.gsc23_neighbourhood
gaiadr1.ppmxl_best_neighbour
gaiadr1.sdss_dr9_neighbourhood
...
gaiadr1.tgas_source
gaiadr1.urat1_original_valid
gaiadr1.allwise_original_valid
```

To load table names (TAP compatible)

```
>>> from astroquery.gaia import Gaia
>>> tables = Gaia.load_tables()
>>> for table in (tables):
>>>   print(table.get_qualified_name())

public.dual
public.tycho2
public.igsl_source
public.hipparcos
public.hipparcos_newreduction
public.hubble_sc
public.igsl_source_catalog_ids
tap_schema.tables
tap_schema.keys
tap_schema.columns
tap_schema.schemas
tap_schema.key_columns
gaiadr1.phot_variable_time_series_gfov
gaiadr1.ppmxl_neighbourhood
gaiadr1.gsc23_neighbourhood
gaiadr1.ppmxl_best_neighbour
gaiadr1.sdss_dr9_neighbourhood
...
gaiadr1.tgas_source
gaiadr1.urat1_original_valid
gaiadr1.allwise_original_valid
```

To load only a table (TAP+ capability)

```
>>> from astroquery.gaia import Gaia
>>> table = Gaia.load_table('gaiadr1.gaia_source')
>>> print(table)

Table name: gaiadr1.gaia_source
Description: This table has an entry for every Gaia observed source as listed in the
Main Database accumulating catalogue version from which the catalogue
release has been generated. It contains the basic source parameters,
that is only final data (no epoch data) and no spectra (neither final
nor epoch).
Num. columns: 57
```

Once a table is loaded, columns can be inspected

```
>>> from astroquery.gaia import Gaia
>>> gaiadr1_table = Gaia.load_table('gaiadr1.gaia_source')
>>> for column in (gaiadr1_table.get_columns()):
>>>   print(column.get_name())

solution_id
source_id
random_index
ref_epoch
ra
ra_error
dec
dec_error
...
ecl_lon
ecl_lat
```

### 1.4 Synchronous query

A synchronous query will not store the results at server side. These queries must be used when the amount of data to be retrieve is 'small'.

There is a limit of 2000 rows. If you need more than that, you must use asynchronous queries.

The results can be saved in memory (default) or in a file.

Query without saving results in a file:

```
>>> from astroquery.gaia import Gaia
>>>
>>> job = Gaia.launch_job("select top 100 \
>>> solution_id,ref_epoch,ra_dec_corr,astrometric_n_obs_al,matched_observations,duplicated_source,phot_
↪variable_flag \
>>> from gaiadr1.gaia_source order by source_id")
>>>
>>> print(job)

Jobid: None
Phase: COMPLETED
Owner: None
Output file: sync_20170223111452.xml.gz
Results: None

>>> r = job.get_results()
>>> print(r['solution_id'])

  solution_id
-------------------
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
```

(continues on next page)

(continued from previous page)

```
1635378410781933568
1635378410781933568
1635378410781933568
                ...
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
Length = 100 rows
```

Query saving results in a file:

```
>>> from astroquery.gaia import Gaia
>>> job = Gaia.launch_job("select top 100 \
>>> solution_id,ref_epoch,ra_dec_corr,astrometric_n_obs_al,matched_observations,duplicated_source,phot_
↪variable_flag \
>>> from gaiadr1.gaia_source order by source_id", dump_to_file=True)
>>>
>>> print(job)

Jobid: None
Phase: COMPLETED
Owner: None
Output file: sync_20170223111452.xml.gz
Results: None

>>> r = job.get_results()
>>> print(r['solution_id'])

  solution_id
-------------------
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
                ...
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
```

(continues on next page)

(continued from previous page)

```
1635378410781933568
1635378410781933568
1635378410781933568
Length = 100 rows
```

### 1.5 Synchronous query on an 'on-the-fly' uploaded table

A table can be uploaded to the server in order to be used in a query.

```
from astroquery.gaia import Gaia

>>> upload_resource = 'my_table.xml'
>>> j = Gaia.launch_job(query="select * from tap_upload.table_test", upload_resource=upload_resource, \
>>> upload_table_name="table_test", verbose=True)
>>> r = j.get_results()
>>> r.pprint()

source_id alpha delta
--------- ----- -----
        a   1.0   2.0
        b   3.0   4.0
        c   5.0   6.0
```

### 1.6 Asynchronous query

Asynchronous queries save results at server side. These queries can be accessed at any time. For anonymous users, results are kept for three days.

The results can be saved in memory (default) or in a file.

Query without saving results in a file:

```
>>> from astroquery.gaia import Gaia
>>>
>>> job = Gaia.launch_job_async("select top 100 * from gaiadr1.gaia_source order by source_id")
>>>
>>> print(job)

Jobid: 1487845273526O
Phase: COMPLETED
Owner: None
Output file: async_20170223112113.vot
Results: None

>>> r = job.get_results()
>>> print(r['solution_id'])

  solution_id
-------------------
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
```

(continues on next page)

(continued from previous page)

```
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
                ...
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
Length = 100 rows
```

Query saving results in a file:

```
>>> from astroquery.gaia import Gaia
>>>
>>> job = Gaia.launch_job_async("select top 100 * from gaiadr1.gaia_source order by source_id", dump_to_
↪file=True)
>>>
>>> print(job)

Jobid: 1487845273526O
Phase: COMPLETED
Owner: None
Output file: async_20170223112113.vot
Results: None

>>> r = job.get_results()
>>> print(r['solution_id'])

  solution_id
-------------------
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
                ...
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
```

(continues on next page)

(continued from previous page)

```
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
Length = 100 rows
```

### 1.6 Asynchronous job removal

To remove asynchronous

```
>>> from astroquery.gaia import Gaia
>>> job = Gaia.remove_jobs(["job_id_1","job_id_2",...])
```

## 28.1.2 2. Authenticated access

Authenticated users are able to access to TAP+ capabilities (shared tables, persistent jobs, etc.) In order to authenticate a user, `login` or `login_gui` methods must be called. After a successful authentication, the user will be authenticated until `logout` method is called.

All previous methods (`query_object`, `cone_search`, `load_table`, `load_tables`, `launch_job`) explained for non authenticated users are applicable for authenticated ones.

The main differences are:

- Asynchronous results are kept at server side for ever (until the user decides to remove one of them).
- Users can access to shared tables.

### 2.1. Login/Logout

Graphic interface

*Note: Tkinter module is required to use login_gui method.*

```
>>> from astroquery.gaia import Gaia
>>> Gaia.login_gui()
```

Command line

```
>>> from astroquery.gaia import Gaia
>>> Gaia.login(user='userName', password='userPassword')
```

It is possible to use a file where the credentials are stored:

*The file must containing user and password in two different lines.*

```
>>> from astroquery.gaia import Gaia
>>> Gaia.login(credentials_file='my_credentials_file')
```

To perform a logout

```
>>> from astroquery.gaia import Gaia
>>> Gaia.logout()
```

## 2.2. Listing shared tables

```
>>> from astroquery.gaia import Gaia
>>> tables = Gaia.load_tables(only_names=True, include_shared_tables=True)
>>> for table in (tables):
>>>   print(table.get_qualified_name())

public.dual
public.tycho2
public.igsl_source
tap_schema.tables
tap_schema.keys
tap_schema.columns
tap_schema.schemas
tap_schema.key_columns
gaiadr1.phot_variable_time_series_gfov
gaiadr1.ppmxl_neighbourhood
gaiadr1.gsc23_neighbourhood
...
user_schema_1.table1
user_schema_2.table1
...
```

## Reference/API

### astroquery.gaia Package

@author: Juan Carlos Segovia @contact: juan.carlos.segovia@sciops.esa.int

European Space Astronomy Centre (ESAC) European Space Agency (ESA)

Created on 30 jun. 2016

#### Classes

| | |
|---|---|
| GaiaClass([tap_plus_handler]) | Proxy class to default TapPlus object (pointing to Gaia Archive) |
| Conf | Configuration parameters for astroquery.gaia. |

#### GaiaClass

**class** astroquery.gaia.**GaiaClass**(*tap_plus_handler=None*)

Bases: object

Proxy class to default TapPlus object (pointing to Gaia Archive)

##### Methods Summary

| | |
|---|---|
| cone_search(coordinate[, radius, ...]) | Cone search sorted by distance (sync.) TAP & TAP+ |
| cone_search_async(coordinate[, radius, ...]) | Cone search sorted by distance (async) TAP & TAP+ |

Continued on next page

Table 2 – continued from previous page

| | |
|---|---|
| launch_job(query[, name, output_file, ...]) | Launches a synchronous job TAP & TAP+ |
| launch_job_async(query[, name, output_file, ...]) | Launches an asynchronous job TAP & TAP+ |
| list_async_jobs([verbose]) | Returns all the asynchronous jobs TAP & TAP+ |
| load_async_job([jobid, name, verbose]) | Loads an asynchronous job TAP & TAP+ |
| load_table(table[, verbose]) | Loads the specified table TAP+ only |
| load_tables([only_names, ...]) | Loads all public tables TAP & TAP+ |
| login([user, password, credentials_file, ...]) | Performs a login. |
| login_gui([verbose]) | Performs a login using a GUI dialog TAP+ only |
| logout([verbose]) | Performs a logout TAP+ only |
| query_object(coordinate[, radius, width, ...]) | Launches a job TAP & TAP+ |
| query_object_async(coordinate[, radius, ...]) | Launches a job (async) TAP & TAP+ |
| remove_jobs(jobs_list[, verbose]) | Removes the specified jobs TAP+ |
| save_results(job[, verbose]) | Saves job results TAP & TAP+ |
| search_async_jobs([jobfilter, verbose]) | Searches for jobs applying the specified filter TAP+ only |

### Methods Documentation

**cone_search**(*coordinate*, *radius=None*, *output_file=None*, *output_format='votable'*, *verbose=False*, *dump_to_file=False*)

Cone search sorted by distance (sync.) TAP & TAP+

**Parameters**

**coordinate** : astropy.coordinate, mandatory

coordinates center point

**radius** : astropy.units, mandatory

radius

**output_file** : str, optional, default None

file name where the results are saved if dumpToFile is True. If this parameter is not provided, the jobid is used instead

**output_format** : str, optional, default 'votable'

results format

**verbose** : bool, optional, default 'False'

flag to display information about the process

**dump_to_file** : bool, optional, default 'False'

if True, the results are saved in a file instead of using memory

**Returns**

A Job object

**cone_search_async**(*coordinate*, *radius=None*, *background=False*, *output_file=None*, *output_format='votable'*, *verbose=False*, *dump_to_file=False*)

Cone search sorted by distance (async) TAP & TAP+

**Parameters**

**coordinate** : astropy.coordinate, mandatory

coordinates center point

**radius** : astropy.units, mandatory

radius

**background** : bool, optional, default 'False'

when the job is executed in asynchronous mode, this flag specifies whether the execution will wait until results are available

**output_file** : str, optional, default None

file name where the results are saved if dumpToFile is True. If this parameter is not provided, the jobid is used instead

**output_format** : str, optional, default 'votable'

results format

**verbose** : bool, optional, default 'False'

flag to display information about the process

**dump_to_file** : bool, optional, default 'False'

if True, the results are saved in a file instead of using memory

**Returns**

A Job object

**launch_job**(*query*, *name=None*, *output_file=None*, *output_format='votable'*, *verbose=False*, *dump_to_file=False*, *upload_resource=None*, *upload_table_name=None*)

Launches a synchronous job TAP & TAP+

**Parameters**

**query** : str, mandatory

query to be executed

**output_file** : str, optional, default None

file name where the results are saved if dumpToFile is True. If this parameter is not provided, the jobid is used instead

**output_format** : str, optional, default 'votable'

results format

**verbose** : bool, optional, default 'False'

flag to display information about the process

**dump_to_file** : bool, optional, default 'False'

if True, the results are saved in a file instead of using memory

**upload_resource: str, optional, default None**

resource to be uploaded to UPLOAD_SCHEMA

**upload_table_name: str, required if uploadResource is provided, default None**

resource temporary table name associated to the uploaded resource

**Returns**

A Job object

**launch_job_async**(*query*, *name=None*, *output_file=None*, *output_format='votable'*, *verbose=False*, *dump_to_file=False*, *background=False*, *upload_resource=None*, *upload_table_name=None*)

Launches an asynchronous job TAP & TAP+

**Parameters**

**query** : str, mandatory

query to be executed

**output_file** : str, optional, default None

file name where the results are saved if dumpToFile is True. If this parameter is not provided, the jobid is used instead

**output_format** : str, optional, default 'votable'

results format

**verbose** : bool, optional, default 'False'

flag to display information about the process

**dump_to_file** : bool, optional, default 'False'

if True, the results are saved in a file instead of using memory

**background** : bool, optional, default 'False'

when the job is executed in asynchronous mode, this flag specifies whether the execution will wait until results are available

**upload_resource: str, optional, default None**

resource to be uploaded to UPLOAD_SCHEMA

**upload_table_name: str, required if uploadResource is provided, default None**

resource temporary table name associated to the uploaded resource

**Returns**

A Job object

`list_async_jobs`(*verbose=False*)

Returns all the asynchronous jobs TAP & TAP+

**Parameters**

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**

A list of Job objects

`load_async_job`(*jobid=None*, *name=None*, *verbose=False*)

Loads an asynchronous job TAP & TAP+

**Parameters**

**jobid** : str, mandatory if no name is provided, default None

job identifier

**name** : str, mandatory if no jobid is provided, default None

job name

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**

A Job object

**load_table**(*table*, *verbose=False*)
Loads the specified table TAP+ only

**Parameters**
**table** : str, mandatory

full qualified table name (i.e. schema name + table name)

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**
A table object

**load_tables**(*only_names=False*, *include_shared_tables=False*, *verbose=False*)
Loads all public tables TAP & TAP+

**Parameters**
**only_names** : bool, TAP+ only, optional, default 'False'

True to load table names only

**include_shared_tables** : bool, TAP+, optional, default 'False'

True to include shared tables

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**
A list of table objects

**login**(*user=None*, *password=None*, *credentials_file=None*, *verbose=False*)
Performs a login. TAP+ only User and password can be used or a file that contains user name and password (2 lines: one for user name and the following one for the password)

**Parameters**
**user** : str, mandatory if 'file' is not provided, default None

login name

**password** : str, mandatory if 'file' is not provided, default None

user password

**credentials_file** : str, mandatory if no 'user' & 'password' are provided

file containing user and password in two lines

**verbose** : bool, optional, default 'False'

flag to display information about the process

**login_gui**(*verbose=False*)
Performs a login using a GUI dialog TAP+ only

**Parameters**
**verbose** : bool, optional, default 'False'

flag to display information about the process

**logout**(*verbose=False*)
Performs a logout TAP+ only

**Parameters**
**verbose** : bool, optional, default 'False'

flag to display information about the process

`query_object`(*coordinate*, *radius=None*, *width=None*, *height=None*, *verbose=False*)
Launches a job TAP & TAP+

**Parameters**
**coordinate** : astropy.coordinates, mandatory

coordinates center point

**radius** : astropy.units, required if no 'width' nor 'height' are provided

radius (deg)

**width** : astropy.units, required if no 'radius' is provided

box width

**height** : astropy.units, required if no 'radius' is provided

box height

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**
The job results (astropy.table).

`query_object_async`(*coordinate*, *radius=None*, *width=None*, *height=None*, *verbose=False*)
Launches a job (async) TAP & TAP+

**Parameters**
**coordinate** : astropy.coordinates, mandatory

coordinates center point

**radius** : astropy.units, required if no 'width' nor 'height' are provided

radius

**width** : astropy.units, required if no 'radius' is provided

box width

**height** : astropy.units, required if no 'radius' is provided

box height

**async_job** : bool, optional, default 'False'

executes the query (job) in asynchronous/synchronous mode (default synchronous)

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**
The job results (astropy.table).

`remove_jobs`(*jobs_list*, *verbose=False*)
Removes the specified jobs TAP+

**Parameters**
**jobs_list** : str, mandatory

jobs identifiers to be removed

**verbose** : bool, optional, default 'False'

flag to display information about the process

**save_results**(*job*, *verbose=False*)
Saves job results TAP & TAP+

**Parameters**
**job** : Job, mandatory

job

**verbose** : bool, optional, default 'False'

flag to display information about the process

**search_async_jobs**(*jobfilter=None*, *verbose=False*)
Searches for jobs applying the specified filter TAP+ only

**Parameters**
**jobfilter** : JobFilter, optional, default None

job filter

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**
A list of Job objects

## Conf

**class** astroquery.gaia.**Conf**
Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.gaia`.

# CHAPTER 29

## VO Simple Cone Search (`astroquery.vo_conesearch`)

Astroquery offers Simple Cone Search Version 1.03 as defined in IVOA Recommendation (February 22, 2008). Cone Search queries an area encompassed by a given radius centered on a given RA and DEC and returns all the objects found within the area in the given catalog.

This was ported from `astropy.vo`:

- `astropy.vo.client.conesearch` is now `astroquery.vo_conesearch.conesearch`
- `astropy.vo.validator` is now `astroquery.vo_conesearch.validator`

`astroquery.vo_conesearch.ConeSearch` is a Cone Search API that adheres to Astroquery standards but unlike Astropy's version, it only queries one given service URL, which defaults to HST Guide Star Catalog. This default is controlled by `astroquery.vo_conesearch.conf.fallback_url`.

### 29.1 Default Cone Search Services

For the "classic" API ported from Astropy, the default Cone Search services used are a subset of those found in the STScI VAO Registry. They were hand-picked to represent commonly used catalogs below:

- 2MASS All-Sky
- HST Guide Star Catalog (also default for "new" Astroquery-style API)
- SDSS Data Release 7
- SDSS-III Data Release 8
- USNO A1
- USNO A2
- USNO B1

This subset undergoes daily validations hosted by STScI using *Validation for Simple Cone Search*. Those that pass without critical warnings or exceptions are used by *Simple Cone Search* by default. They are controlled by `astroquery.vo_conesearch.conf.conesearch_dbname`:

1. `'conesearch_good'` Default. Passed validation without critical warnings and exceptions.
2. `'conesearch_warn'` Has critical warnings but no exceptions. Use at your own risk.
3. `'conesearch_exception'` Has some exceptions. *Never* use this.
4. `'conesearch_error'` Has network connection error. *Never* use this.

If you are a Cone Search service provider and would like to include your service in the list above, please open a GitHub issue on Astroquery.

## 29.2 Caching

Caching of downloaded contents is controlled by `astropy.utils.data`. To use cached data, some functions in this package have a `cache` keyword that can be set to `True`.

## 29.3 Getting Started

This section only contains minimal examples showing how to perform basic Cone Search.

Query STScI Guide Star Catalog using "new" Astroquery-style API around M31 with a 0.1-degree search radius:

```
>>> from astropy.coordinates import SkyCoord
>>> from astroquery.vo_conesearch import ConeSearch
>>> ConeSearch.URL
'http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&'
>>> c = SkyCoord.from_name('M31')
>>> c.ra, c.dec
(<Longitude 10.6847929 deg>, <Latitude 41.269065 deg>)
>>> result = ConeSearch.query_region(c, '0.1 deg')
>>> result
<Table masked=True length=4027>
    objID           gscID2      GSC1ID ... multipleFlag compassGSC2id    Mag
                                        ...                                 mag
    int64           object      object ...    object        object     float32
-------------- ---------------- ------ ... ------------ ------------- ---------
23323175812944 00424433+4116085        ...            0 6453800072293        --
23323175812933 00424455+4116103        ...            0 6453800072282        --
23323175812939 00424464+4116092        ...            0 6453800072288        --
23323175812931 00424464+4116106        ...            0 6453800072280        --
23323175812948 00424403+4116069        ...            0 6453800072297        --
23323175812930 00424403+4116108        ...            0 6453800072279        --
           ...              ...    ... ...          ...           ...       ...
 1330012210212     N330012210212        ...            0 6453800010212 20.276699
23323175812087 00425926+4121267        ...            0 6453800071571        --
 1330012231849     N330012231849        ...            0 6453800031849   20.2869
 1330012244053     N330012244053        ...            0 6453800044053        --
 1330012243728     N330012243728        ...            0 6453800043728        --
  133001228698      N33001228698        ...            0 6453800008698 20.563999
```

List the available Cone Search catalogs that passed daily validation:

```
>>> from astroquery.vo_conesearch import conesearch
>>> conesearch.list_catalogs()
Downloading http://stsdas.stsci.edu/astrolib/vo_databases/conesearch_good.json
```

(continues on next page)

(continued from previous page)

```
|==========================================|  37k/ 37k (100.00%)         0s
['Guide Star Catalog v2 1',
 'SDSS DR8 - Sloan Digital Sky Survey Data Release 8 1',
 'SDSS DR8 - Sloan Digital Sky Survey Data Release 8 2',
 'The HST Guide Star Catalog, Version 1.1 (Lasker+ 1992) 1',
 'The HST Guide Star Catalog, Version 1.2 (Lasker+ 1996) 1',
 'The HST Guide Star Catalog, Version GSC-ACT (Lasker+ 1996-99) 1',
 'The PMM USNO-A1.0 Catalogue (Monet 1997) 1',
 'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 'USNO-A2 Catalogue 1']
```

Query the HST Guide Star Catalog around M31 with a 0.1-degree search radius. This is the same query as above but using “classic” Astropy-style API:

```
>>> from astropy import units as u
>>> my_catname = 'Guide Star Catalog v2 1'
>>> result = conesearch.conesearch(c, 0.1 * u.degree, catalog_db=my_catname)
Trying http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&
>>> result
<Table masked=True length=4027>
    objID           gscID2      GSC1ID ... multipleFlag compassGSC2id    Mag
                                       ...                               mag
    int64           object      object ...    object        object     float32
-------------- ---------------- ------ ... ------------ ------------- ---------
23323175812944 00424433+4116085        ...            0 6453800072293        --
23323175812933 00424455+4116103        ...            0 6453800072282        --
23323175812939 00424464+4116092        ...            0 6453800072288        --
23323175812931 00424464+4116106        ...            0 6453800072280        --
23323175812948 00424403+4116069        ...            0 6453800072297        --
23323175812930 00424403+4116108        ...            0 6453800072279        --
           ...              ...    ... ...          ...           ...       ...
 1330012210212    N330012210212        ...            0 6453800010212 20.276699
23323175812087 00425926+4121267        ...            0 6453800071571        --
 1330012231849    N330012231849        ...            0 6453800031849   20.2869
 1330012244053    N330012244053        ...            0 6453800044053        --
 1330012243728    N330012243728        ...            0 6453800043728        --
  133001228698     N33001228698        ...            0 6453800008698 20.563999
>>> result.url
'http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&'
```

Get the number of matches and returned column names:

```
>>> result.array.size
4027
>>> result.array.dtype.names
(objID',
 'gscID2',
 'GSC1ID',
 'hstID',
 'ra',
 'dec', ...,
 'Mag')
```

Extract RA and DEC of the matches:

```
>>> result.array['ra']
```

(continues on next page)

(continued from previous page)

```
masked_array(data = [10.684737 10.685657 10.686026 ..., 10.8028268814087],
             mask = [False False False ..., False],
       fill_value = 1e+20)
>>> result.array['dec']
masked_array(data = [41.269035 41.26955 41.269226 ..., 41.2230453491211],
             mask = [False False False ..., False],
       fill_value = 1e+20)
```

## 29.4 Using `astroquery.vo_conesearch`

This package has four main components across two categories:

### 29.4.1 Using "Client" API

The "client" API contains modules supporting VO Cone Search's client-side operations.

#### Catalog Manipulation

You can manipulate a VO catalog using `VOSCatalog`, which is basically a dictionary with added functionalities.

#### Examples

You can create a VO catalog from scratch with your own VO service by providing its title and access URL, and optionally any other metadata as key-value pairs:

```
>>> from astroquery.vo_conesearch.vos_catalog import VOSCatalog
>>> my_cat = VOSCatalog.create(
...     'My Own', 'http://ex.org/cgi-bin/cs.pl?',
...     description='My first VO service.', creator='J. Doe', year=2013)
>>> print(my_cat)
title: My Own
url: http://ex.org/cgi-bin/cs.pl?
>>> print(my_cat.dumps())
{
    "creator": "J. Doe",
    "description": "My first VO service.",
    "title": "My Own",
    "url": "http://ex.org/cgi-bin/cs.pl?",
    "year": 2013
}
```

You can modify and add fields:

```
>>> my_cat['year'] = 2014
>>> my_cat['new_field'] = 'Hello world'
>>> print(my_cat.dumps())
{
    "creator": "J. Doe",
    "description": "My first VO service.",
    "new_field": "Hello world",
```

(continues on next page)

(continued from previous page)

```
    "title": "My Own",
    "url": "http://ex.org/cgi-bin/cs.pl?",
    "year": 2014
}
```

In addition, you can also delete an existing field, except the compulsory title and access URL:

```
>>> my_cat.delete_attribute('description')
>>> print(my_cat.dumps())
{
    "creator": "J. Doe",
    "new_field": "Hello world",
    "title": "My Own",
    "url": "http://ex.org/cgi-bin/cs.pl?",
    "year": 2014
}
```

### Database Manipulation

You can manipulate VO database using VOSDatabase, which is basically a nested dictionary with added functionalities.

### Examples

You can choose to start with an empty database:

```
>>> from astroquery.vo_conesearch.vos_catalog import VOSDatabase
>>> my_db = VOSDatabase.create_empty()
>>> print(my_db.dumps())
{
    "__version__": 1,
    "catalogs": {}
}
```

Add the custom catalog from *VO catalog examples* to database:

```
>>> my_db.add_catalog('My Catalog 1', my_cat)
>>> print(my_db)
My Catalog 1
>>> print(my_db.dumps())
{
    "__version__": 1,
    "catalogs": {
        "My Catalog 1": {
            "creator": "J. Doe",
            "new_field": "Hello world",
            "title": "My Own",
            "url": "http://ex.org/cgi-bin/cs.pl?",
            "year": 2014
        }
    }
}
```

You can write/read the new database to/from a JSON file:

```
>>> my_db.to_json('my_vo_database.json', overwrite=True)
>>> my_db = VOSDatabase.from_json('my_vo_database.json')
```

You can also load a database from a VO registry. The process is described in *Building the Database from Registry*, except that here, validation is not done, so `validate_xxx` keys are not added. This might generate a lot of warnings, especially if the registry has duplicate entries of similar services, so here, we silently ignore all the warnings:

```
>>> import warnings
>>> from astroquery.vo_conesearch.validator import conf as validator_conf
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore')
...     registry_db = VOSDatabase.from_registry(
...         validator_conf.conesearch_master_list, encoding='binary',
...         cache=False)
Downloading http://vao.stsci.edu/directory/NVORegInt.asmx/...
|==========================================|  73M/ 73M (100.00%)         0s
>>> len(registry_db)
17832
```

Find catalog names containing `'usno*a2'` in the registry database:

```
>>> usno_a2_list = registry_db.list_catalogs(pattern='usno*a2')
>>> usno_a2_list
['ROSAT All-Sky Survey Bright Source Catalog USNO A2 Cross-Associations 1',
 'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 'USNO-A2 Catalogue 1']
```

Find access URLs containing `'stsci'` in the registry database:

```
>>> stsci_urls = registry_db.list_catalogs_by_url(pattern='stsci')
>>> stsci_urls
[b'http://archive.stsci.edu/befs/search.php?',
 b'http://archive.stsci.edu/copernicus/search.php?',
 b'http://archive.stsci.edu/euve/search.php?', ...,
 b'http://galex.stsci.edu/gxWS/ConeSearch/gxConeSearch.aspx?',
 b'http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&']
```

Extract a catalog titled `'USNO-A2 Catalogue 1'` from the registry:

```
>>> usno_a2 = registry_db.get_catalog('USNO-A2 Catalogue 1')
>>> print(usno_a2)
title: USNO-A2 Catalogue
url: http://www.nofs.navy.mil/cgi-bin/vo_cone.cgi?CAT=USNO-A2&
```

Extract a catalog by known access URL from the registry (the iterator version of this functionality is `get_catalogs_by_url()`, which is useful in the case of multiple entries with same access URL):

```
>>> gsc_url = 'http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/305/out&'
>>> gsc = registry_db.get_catalog_by_url(gsc_url)
>>> print(gsc)
title: The Guide Star Catalog, Version 2.3.2 (GSC2.3) (STScI, 2006)
url: http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/305/out&
```

Add all `'usno*a2'` catalogs from registry to your database:

```
>>> for name, cat in registry_db.get_catalogs():
...     if name in usno_a2_list:
```

(continues on next page)

(continued from previous page)

```
...         my_db.add_catalog(name, cat)
>>> my_db.list_catalogs()
['My Catalog 1',
 'ROSAT All-Sky Survey Bright Source Catalog USNO A2 Cross-Associations 1',
 'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 'USNO-A2 Catalogue 1']
```

You can delete a catalog from the database either by name or access URL:

```
>>> my_db.delete_catalog('USNO-A2 Catalogue 1')
>>> my_db.delete_catalog_by_url('https://heasarc.gsfc.nasa.gov/cgi-bin/vo/'
...                             'cone/coneGet.pl?table=rassusnoid&')
>>> my_db.list_catalogs()
['My Catalog 1', 'The USNO-A2.0 Catalogue (Monet+ 1998) 1']
```

You can also merge two database together. In this example, the second database contains a simple catalog that only has given name and access URL:

```
>>> other_db = VOSDatabase.create_empty()
>>> other_db.add_catalog_by_url(
...     'My Guide Star Catalogue',
...     'http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/305/out&')
>>> print(other_db.dumps())
{
    "__version__": 1,
    "catalogs": {
        "My Guide Star Catalogue": {
            "title": "My Guide Star Catalogue",
            "url": "url": "http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/305/out&"
        }
    }
}
>>> merged_db = my_db.merge(other_db)
>>> merged_db.list_catalogs()
['My Catalog 1',
 'My Guide Star Catalogue',
 'The USNO-A2.0 Catalogue (Monet+ 1998) 1']
```

### General VO Services Access

`astroquery.vo_conesearch.vos_catalog` also contains common utilities for accessing simple VO services already validated by STScI (see *Validation for Simple Cone Search*).

### Configurable Items

These parameters are set via Configuration system (astropy.config):

- **`astroquery.vo_conesearch.conf.pedantic`**
  Set strictness of VO table parser (`False` is recommended).
- **`astroquery.vo_conesearch.conf.timeout`**
  Timeout for remote service access.
- **`astroquery.vo_conesearch.conf.vos_baseurl`**
  URL (or path) where VO Service database is stored.

### Examples

Get all catalogs from a database named `'conesearch_good'` (this contains cone search services that cleanly passed daily validations; also see *Cone Search Examples*):

```
>>> from astroquery.vo_conesearch import vos_catalog
>>> my_db = vos_catalog.get_remote_catalog_db('conesearch_good')
Downloading http://stsdas.stsci.edu/astrolib/vo_databases/conesearch_good.json
|=========================================|  37k/ 37k (100.00%)         0s
>>> print(my_db)
Guide Star Catalog v2 1
SDSS DR8 - Sloan Digital Sky Survey Data Release 8 1
# ...
USNO-A2 Catalogue 1
```

If you get timeout error, you need to use a custom timeout as follows:

```
>>> from astropy.utils import data
>>> with data.conf.set_temp('remote_timeout', 30):
...     my_db = vos_catalog.get_remote_catalog_db('conesearch_good')
```

To see validation warnings generated by *Validation for Simple Cone Search* for the one of the catalogs above:

```
>>> my_cat = my_db.get_catalog('Guide Star Catalog v2 1')
>>> for w in my_cat['validate_warnings']:
...     print(w)
/.../vo.xml:136:0: W50: Invalid unit string 'pixel'
/.../vo.xml:155:0: W48: Unknown attribute 'nrows' on TABLEDATA
```

By default, pedantic is `False`:

```
>>> from astroquery.vo_conesearch import conf
>>> conf.pedantic
False
```

To call a given VO service; In this case, a Cone Search (also see *Cone Search Examples*):

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> c = coord.SkyCoord.from_name('47 Tuc')
>>> c
<SkyCoord (ICRS): (ra, dec) in deg
    (6.0223292, -72.0814444)>
>>> sr = 0.5 * u.degree
>>> sr
<Quantity 0.5 deg>
>>> result = vos_catalog.call_vo_service(
...     'conesearch_good',
...     kwargs={'RA': c.ra.degree, 'DEC': c.dec.degree, 'SR': sr.value},
...     catalog_db='The PMM USNO-A1.0 Catalogue (Monet 1997) 1')
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/243/out&
Downloading ...
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
WARNING: W03: ... Implictly generating an ID from a name 'RA(ICRS)'...
WARNING: W03: ... Implictly generating an ID from a name 'DE(ICRS)'...
>>> result
<Table masked=True length=36184>
```

(continues on next page)

(continued from previous page)

```
   _r      USNO-A1.0     RA_ICRS_   DE_ICRS_   ...   Bmag    Rmag    Epoch
  deg                      deg        deg      ...   mag     mag      yr
float64     bytes13      float64    float64    ... float64 float64 float64
-------- ------------- ---------- ----------   ... ------- ------- --------
0.499298 0150-00088188   4.403473 -72.124045 ...    20.6    19.4 1977.781
0.499075 0150-00088198   4.403906 -72.122762 ...    21.2    18.0 1977.778
0.499528 0150-00088210   4.404531 -72.045198 ...    16.2    15.4 1977.781
     ...           ...        ...        ... ...     ...     ...      ...
0.499917 0150-00226223   7.647400 -72.087600 ...    23.4    21.7 1975.829
```

To repeat the above and suppress *all* the screen outputs (not recommended):

```
>>> import warnings
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore')
...     result = vos_catalog.call_vo_service(
...         'conesearch_good',
...         kwargs={'RA': c.ra.degree, 'DEC': c.dec.degree, 'SR': sr.value},
...         catalog_db='The PMM USNO-A1.0 Catalogue (Monet 1997) 1',
...         verbose=False)
```

You can also use custom VO database, say, `'my_vo_database.json'` from *VO database examples*:

```
>>> import os
>>> with conf.set_temp('vos_baseurl', os.curdir):
...     try:
...         result = vos_catalog.call_vo_service(
...             'my_vo_database',
...             kwargs={'RA': c.ra.degree, 'DEC': c.dec.degree,
...                     'SR': sr.value})
...     except Exception as e:
...         print(e)
Trying http://ex.org/cgi-bin/cs.pl?
WARNING: W25: ... failed with: <urlopen error timed out> [...]
None of the available catalogs returned valid results. (1 URL(s) timed out.)
```

### Simple Cone Search

`astroquery.vo_conesearch.conesearch` supports VO Simple Cone Search capabilities.

Available databases are generated on the server-side hosted by STScI using *Validation for Simple Cone Search*. The database used is controlled by `astroquery.vo_conesearch.conf.conesearch_dbname`, which can be changed in *Configurable Items* below. Here are the available options:

1. **`'conesearch_good'`**
   Default. Passed validation without critical warnings and exceptions.
2. **`'conesearch_warn'`**
   Has critical warnings but no exceptions. Use at your own risk.
3. **`'conesearch_exception'`**
   Has some exceptions. *Never* use this.
4. **`'conesearch_error'`**
   Has network connection error. *Never* use this.

In the default setting, it searches the good Cone Search services one by one, stops at the first one that gives non-zero match(es), and returns the result. Since the list of services are extracted from a Python dictionary, the search order might differ from call to call.

There are also functions, both synchronously and asynchronously, available to return *all* the Cone Search query results. However, this is not recommended unless one knows what one is getting into, as it could potentially take up significant run time and computing resources.

*Examples* below show how to use non-default search behaviors, where the user has more control of which catalog(s) to search, et cetera.

**Note:** Most services currently fail to parse when `pedantic=True`.

**Warning:** When Cone Search returns warnings, you should decide whether the results are reliable by inspecting the warning codes in `astropy.io.votable.exceptions`.

### Configurable Items

These parameters are set via Configuration system (astropy.config):

- **`astroquery.vo_conesearch.conf.conesearch_dbname`**
  Cone Search database name to query.

Also depends on *General VO Services Access Configurable Items*.

### Examples

```
>>> from astroquery.vo_conesearch import conesearch
```

Shows a sorted list of Cone Search services to be searched:

```
>>> conesearch.list_catalogs()
['Guide Star Catalog v2 1',
 'SDSS DR8 - Sloan Digital Sky Survey Data Release 8 1',
 'SDSS DR8 - Sloan Digital Sky Survey Data Release 8 2',
 'The HST Guide Star Catalog, Version 1.1 (Lasker+ 1992) 1',
 'The HST Guide Star Catalog, Version 1.2 (Lasker+ 1996) 1',
 'The HST Guide Star Catalog, Version GSC-ACT (Lasker+ 1996-99) 1',
 'The PMM USNO-A1.0 Catalogue (Monet 1997) 1',
 'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 'USNO-A2 Catalogue 1']
```

To inspect them in detail, do the following and then refer to the examples in *Database Manipulation*:

```
>>> from astroquery.vo_conesearch import vos_catalog
>>> good_db = vos_catalog.get_remote_catalog_db('conesearch_good')
```

Select a catalog to search:

```
>>> my_catname = 'The PMM USNO-A1.0 Catalogue (Monet 1997) 1'
```

By default, pedantic is `False`:

```
>>> from astroquery.vo_conesearch import conf
>>> conf.pedantic
False
```

Perform Cone Search in the selected catalog above for 0.5 degree radius around 47 Tucanae with minimum verbosity, if supported. The `catalog_db` keyword gives control over which catalog(s) to use. If running this for the first time, a copy of the catalogs database will be downloaded to local cache. To run this again without using cached data, set `cache=False`:

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> c = coord.SkyCoord.from_name('47 Tuc')
>>> c
<SkyCoord (ICRS): (ra, dec) in deg
    (6.0223292, -72.0814444)>
>>> sr = 0.5 * u.degree
>>> sr
<Quantity 0.5 deg>
>>> result = conesearch.conesearch(c, sr, catalog_db=my_catname)
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/243/out&
Downloading ...
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
WARNING: W03: ... Implictly generating an ID from a name 'RA(ICRS)'...
WARNING: W03: ... Implictly generating an ID from a name 'DE(ICRS)'...
```

To run the command above using custom timeout of 30 seconds for each Cone Search service query:

```
>>> result = conesearch.conesearch(c, sr, catalog_db=my_catname, timeout=30)
```

To suppress *all* the screen outputs (not recommended):

```
>>> import warnings
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore')
...     result = conesearch.conesearch(c, sr, catalog_db=my_catname,
...                                    verbose=False)
```

Extract Numpy array containing the matched objects. See `numpy` for available operations:

```
>>> cone_arr = result.array.data
>>> cone_arr
array([ (0.499298, '0150-00088188', 4.403473, -72.124045, ...),
        (0.499075, '0150-00088198', 4.403906, -72.122762, ...), ...],
      dtype=[('_r', '<f8'), ('USNO-A1.0', 'S13'), ...])
>>> cone_arr.dtype.names
('_r',
 'USNO-A1.0',
 'RA_ICRS_',
 'DE_ICRS_',
 'GSCflag',
 'Mflag',
 'Bmag',
 'Rmag',
 'Epoch')
>>> cone_arr.size
36184
>>> ra_list = cone_arr['RA_ICRS_']
```

(continues on next page)

(continued from previous page)

```
>>> ra_list
array([ 4.403473,  4.403906,  4.404531, ...,  7.641731,  7.645489,  7.6474  ])
>>> cone_arr[0]  # First row
(0.499298, '0150-00088188', 4.403473, -72.124045, '', '', 20.6, 19.4, 1977.781)
>>> cone_arr[-1]  # Last row
(0.499917, '0150-00226223', 7.6474, -72.0876, '', '', 23.4, 21.7, 1975.829)
>>> cone_arr[:10]  # First 10 rows
array([ (0.499298, '0150-00088188', 4.403473, -72.124045, ...),
        (0.499075, '0150-00088198', 4.403906, -72.122762, ...), ...],
      dtype=[('_r', '<f8'), ('USNO-A1.0', 'S13'), ...])
```

Sort the matched objects by angular separation in ascending order:

```
>>> import numpy as np
>>> sep = cone_arr['_r']
>>> i_sorted = np.argsort(sep)
>>> cone_arr[i_sorted]
array([ (0.081971, '0150-00145335', 5.917787, -72.006075, ...),
        (0.083181, '0150-00149799', 6.020339, -72.164623, ...), ...,
        (0.499989, '0150-00192872', 6.899589, -72.5043, ...)],
      dtype=[('_r', '<f8'), ('USNO-A1.0', 'S13'), ...])
```

Result can also be manipulated as VOTable XML handling (astropy.io.votable) and its unit can be manipulated as Units and Quantities (astropy.units). In this example, we convert RA values from degree to arcsec:

```
>>> from astropy import units as u
>>> ra_field = result.get_field_by_id('RA_ICRS_')
>>> ra_field.title
u'Right ascension (ICRS) mean of blue/red plates'
>>> ra_field.unit
Unit("deg")
>>> ra_field.unit.to(u.arcsec) * ra_list
array([ 15852.5028,  15854.0616,  15856.3116, ...,  27510.2316,
        27523.7604,  27530.64  ])
```

Perform the same Cone Search as above but asynchronously using `AsyncConeSearch`. Queries to individual Cone Search services are still governed by `astroquery.vo_conesearch.conf.timeout`. Cone Search is forced to run in silent mode asynchronously, but warnings are still controlled by `warnings`:

```
>>> async_search = conesearch.AsyncConeSearch(c, sr, catalog_db=my_catname)
```

Check asynchronous search status:

```
>>> async_search.running()
True
>>> async_search.done()
False
```

Get search results after a 30-second wait (not to be confused with `astroquery.vo_conesearch.conf.timeout` that governs individual Cone Search queries). If search is still not done after 30 seconds, `TimeoutError` is raised. Otherwise, Cone Search result is returned and can be manipulated as above. If no `timeout` keyword given, it waits until completion:

```
>>> async_result = async_search.get(timeout=30)
>>> cone_arr = async_result.array.data
```

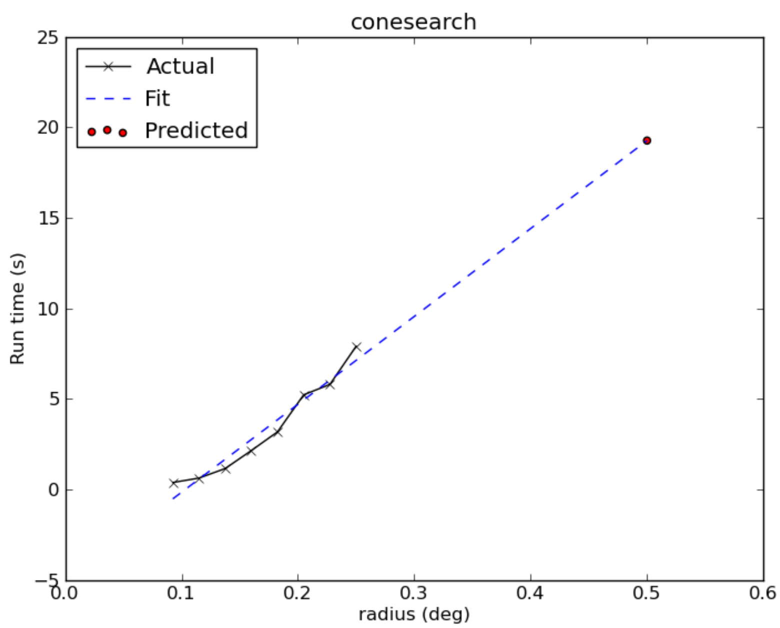
(continues on next page)

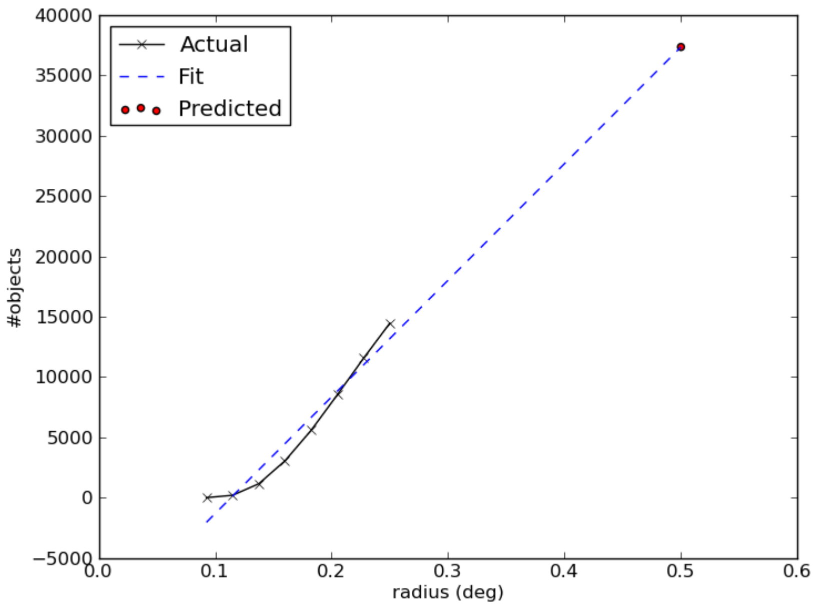
(continued from previous page)

```
>>> cone_arr.size
36184
```

Estimate the execution time and the number of objects for the Cone Search service URL from above. The prediction naively assumes a linear model, which might not be accurate for some cases. It also uses the normal Cone Search, not the asynchronous version. This example uses a custom timeout of 30 seconds and runs silently (except for warnings):

```
>>> result.url
'http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/243/out&'
>>> t_est, n_est = conesearch.predict_search(result.url, c, sr, verbose=False,
...                                          plot=True)
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
# ...
>>> t_est  # Predicted execution time
10.757875269998323
>>> n_est  # Predicted number of objects
37340
```






For debugging purpose, one can obtain the actual execution time and number of objects, and compare them with the predicted values above. The INFO message shown in controlled by `astropy.logger`. Keep in mind that running this for every prediction would defeat the purpose of the prediction itself:

```
>>> t_real, tab = conesearch.conesearch_timer(
...     c, sr, catalog_db=result.url, verbose=False)
INFO: conesearch_timer took 11.5103080273 s on AVERAGE for 1 call(s). [...]
>>> t_real  # Actual execution time
11.5103080273
>>> tab.array.size  # Actual number of objects
36184
```

One can also search in a list of catalogs instead of a single one. In this example, we look for all catalogs containing `'guide*star'` in their titles and only perform Cone Search using those services. The first catalog in the list to successfully return non-zero result is used. Therefore, the order of catalog names given in `catalog_db` is important:

```
>>> gsc_cats = conesearch.list_catalogs(pattern='guide*star')
>>> gsc_cats
[u'Guide Star Catalog v2 1',
 u'The HST Guide Star Catalog, Version 1.1 (Lasker+ 1992) 1',
 u'The HST Guide Star Catalog, Version 1.2 (Lasker+ 1996) 1',
 u'The HST Guide Star Catalog, Version GSC-ACT (Lasker+ 1996-99) 1']
>>> gsc_result = conesearch.conesearch(c, sr, catalog_db=gsc_cats)
Trying http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&
>>> gsc_result.array.size
74272
>>> gsc_result.url
'http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&'
```

To repeat the Cone Search above with the services listed in a different order:

```
>>> gsc_cats_reordered = [gsc_cats[i] for i in (3, 1, 2, 0)]
>>> gsc_cats_reordered
[u'The HST Guide Star Catalog, Version GSC-ACT (Lasker+ 1996-99) 1',
```

(continues on next page)

(continued from previous page)

```
 u'The HST Guide Star Catalog, Version 1.1 (Lasker+ 1992) 1',
 u'The HST Guide Star Catalog, Version 1.2 (Lasker+ 1996) 1',
 u'Guide Star Catalog v2 1']
>>> gsc_result = conesearch.conesearch(c, sr, catalog_db=gsc_cats_reordered)
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/255/out&
>>> gsc_result.array.size
2997
>>> gsc_result.url
'http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/255/out&'
```

To obtain results from *all* the services above:

```
>>> all_gsc_results = conesearch.search_all(c, sr, catalog_db=gsc_cats)
Trying http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/220/out&
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/254/out&
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/255/out&
>>> len(all_gsc_results)
4
>>> for url in sorted(all_gsc_results):
...     tab = all_gsc_results[url]
...     print('{} has {} results'.format(url, tab.array.size))
http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23& has 74272 results
http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/220/out& has 2997 results
http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/254/out& has 2998 results
http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=I/255/out& has 2997 results
```

To repeat the above asynchronously:

```
>>> async_search_all = conesearch.AsyncSearchAll(
...     c, sr, catalog_db=gsc_cats, timeout=60)
>>> async_search_all.running()
True
>>> async_search_all.done()
False
>>> all_gsc_results = async_search_all.get()
```

If one is unable to obtain any desired results using the default Cone Search database, `'conesearch_good'`, that only contains sites that cleanly passed validation, one can use Configuration system (astropy.config) to use another database, `'conesearch_warn'`, containing sites with validation warnings. One should use these sites with caution:

```
>>> from astroquery.vo_conesearch import conf
>>> conf.conesearch_dbname = 'conesearch_warn'
>>> conesearch.list_catalogs()
Downloading http://stsdas.stsci.edu/astrolib/vo_databases/conesearch_warn.json
|==========================================| 312k/312k (100.00%)         0s
['2MASS All-Sky Catalog of Point Sources (Cutri+ 2003) 1',
 'Data release 7 of Sloan Digital Sky Survey catalogs 1',
 'Data release 7 of Sloan Digital Sky Survey catalogs 2',
 'Data release 7 of Sloan Digital Sky Survey catalogs 3',
 'Data release 7 of Sloan Digital Sky Survey catalogs 4',
 'Data release 7 of Sloan Digital Sky Survey catalogs 5',
 'Data release 7 of Sloan Digital Sky Survey catalogs 6',
 'The 2MASS All-Sky Catalog 1',
 'The 2MASS All-Sky Catalog 2',
 'The USNO-B1.0 Catalog (Monet+ 2003) 1',
```

(continues on next page)

(continued from previous page)

```
 'The USNO-B1.0 Catalog 1',
 'USNO-A V2.0, A Catalog of Astrometric Standards 1',
 'USNO-B1 Catalogue 1']
>>> result = conesearch.conesearch(c, sr)
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-out.all&-source=II/284/out&
>>> result.array.data.size
50000
```

You can also use custom Cone Search database, say, `'my_vo_database.json'` from *VO database examples*:

```
>>> import os
>>> from astroquery.vo_conesearch import conf
>>> conf.vos_baseurl = os.curdir
>>> conf.conesearch_dbname = 'my_vo_database'
>>> conesearch.list_catalogs()
[u'My Catalog 1']
>>> result = conesearch.conesearch(c, sr)
Trying http://ex.org/cgi-bin/cs.pl?
# ...
VOSError: None of the available catalogs returned valid results. (1 URL(s) timed out.)
```

### 29.4.2 Using "Server" API

The "server" API contains modules supporting VO Cone Search's server-side operations, particularly to validate external Cone Search services for *Simple Cone Search*.

A typical user should not need the validator. However, this could be used by VO service providers to validate their services. Currently, any service to be validated has to be registered in STScI VAO Registry.

#### Validation for Simple Cone Search

`astroquery.vo_conesearch.validator.validate` validates VO services. Currently, only Cone Search validation is done using `check_conesearch_sites()`, which utilizes underlying `astropy.io.votable.validator` library.

A master list of all available Cone Search services is obtained from `astroquery.vo_conesearch.validator.conf.conesearch_master_list`, which is a URL query to STScI VAO Registry by default. However, by default, only the ones in `astroquery.vo_conesearch.validator.conf.conesearch_urls` are validated (also see *Default Cone Search Services*), while the rest are skipped. There are also options to validate a user-defined list of services or all of them.

All Cone Search queries are done using RA, DEC, and SR given by `<testQuery>` XML tag in the registry, and maximum verbosity. In an uncommon case where `<testQuery>` is not defined for a service, it uses a default search for `RA=0&DEC=0&SR=0.1`.

The results are separated into 4 groups below. Each group is stored as a JSON file of `VOSDatabase`:

1. **`conesearch_good.json`**
   Passed validation without critical warnings and exceptions. This database residing in `astroquery.vo_conesearch.conf.vos_baseurl` is the one used by *Simple Cone Search* by default.
2. **`conesearch_warn.json`**
   Has critical warnings but no exceptions. Users can manually set `astroquery.vo_conesearch.conf.conesearch_dbname` to use this at their own risk.
3. **`conesearch_exception.json`**
   Has some exceptions. *Never* use this. For informational purpose only.

4. **conesearch_error.json**
   Has network connection error. *Never* use this. For informational purpose only.

HTML pages summarizing the validation results are stored in `'results'` sub-directory, which also contains downloaded XML files from individual Cone Search queries.

### Warnings and Exceptions

A subset of `astropy.io.votable.exceptions` that is considered non-critical is defined by `astroquery.vo_conesearch.validator.conf.noncritical_warnings`, which will not be flagged as bad by the validator. However, this does not change the behavior of `astroquery.vo_conesearch.conf.pedantic`, which still needs to be set to `False` for them not to be thrown out by `conesearch()`. Despite being listed as non-critical, user is responsible to check whether the results are reliable; They should not be used blindly.

Some units recognized by VizieR are considered invalid by Cone Search standards. As a result, they will give the warning `'W50'`, which is non-critical by default.

User can also modify `astroquery.vo_conesearch.validator.conf.noncritical_warnings` to include or exclude any warnings or exceptions, as desired. However, this should be done with caution. Adding exceptions to non-critical list is not recommended.

### Building the Database from Registry

Each Cone Search service is a `VOSCatalog` in a `VOSDatabase` (see *Catalog Manipulation* and *Database Manipulation*).

In the master registry, there are duplicate catalog titles with different access URLs, duplicate access URLs with different titles, duplicate catalogs with slightly different descriptions, etc.

A Cone Search service is really defined by its access URL regardless of title, description, etc. By default, `from_registry()` ensures each access URL is unique across the database. However, for user-friendly catalog listing, its title will be the catalog key, not the access URL.

In the case of two different access URLs sharing the same title, each URL will have its own database entry, with a sequence number appended to their titles (e.g., 'Title 1' and 'Title 2'). For consistency, even if the title does not repeat, it will still be renamed to 'Title 1'.

In the case of the same access URL appearing multiple times in the registry, the validator will store the first catalog with that access URL and throw out the rest. However, it will keep count of the number of duplicates thrown out in the `'duplicatesIgnored'` dictionary key of the catalog kept in the database.

All the existing catalog tags will be copied over as dictionary keys, except `'accessURL'` that is renamed to `'url'` for simplicity. In addition, new keys from validation are added:

- **validate_expected**
  Expected validation result category, e.g., "good".
- **validate_network_error**
  Indication for connection error.
- **validate_nexceptions**
  Number of exceptions found.
- **validate_nwarnings**
  Number of warnings found.
- **validate_out_db_name**
  Cone Search database name this entry belongs to.

- **validate_version**
  Version of validation software.
- **validate_warning_types**
  List of warning codes.
- **validate_warnings**
  Descriptions of the warnings.
- **validate_xmllint**
  Indication of whether `xmllint` passed.
- **validate_xmllint_content**
  Output from `xmllint`.

### Configurable Items

These parameters are set via Configuration system (astropy.config):

- **astroquery.vo_conesearch.validator.conf.conesearch_master_list**
  VO registry query URL that should return a VO table with all the desired VO services.
- **astroquery.vo_conesearch.validator.conf.conesearch_urls**
  Subset of Cone Search access URLs to validate.
- **astroquery.vo_conesearch.validator.conf.noncritical_warnings**
  List of VO table parser warning codes that are considered non-critical.

Also depends on properties in *Simple Cone Search Configurable Items*.

### Examples

Validate default Cone Search sites with multiprocessing and write results in the current directory. Reading the master registry can be slow, so the default timeout is internally set to 60 seconds for it. In addition, all VO table warnings from the registry are suppressed because we are not trying to validate the registry itself but the services it contains:

```
>>> from astroquery.vo_conesearch.validator import validate
>>> validate.check_conesearch_sites()
Downloading http://vao.stsci.edu/directory/NVORegInt.asmx/...
|=========================================|  73M/ 73M (100.00%)         0s
INFO: Only 30/17832 site(s) are validated [...]
# ...
INFO: good: 9 catalog(s) [astroquery.vo_conesearch.validator.validate]
INFO: warn: 6 catalog(s) [astroquery.vo_conesearch.validator.validate]
INFO: excp: 4 catalog(s) [astroquery.vo_conesearch.validator.validate]
INFO: nerr: 9 catalog(s) [astroquery.vo_conesearch.validator.validate]
INFO: total: 28 out of 30 catalog(s) [...]
INFO: check_conesearch_sites took 26.626858234405518 s on AVERAGE...
```

Validate only Cone Search access URLs hosted by `'stsci.edu'` without verbose outputs (except warnings that are controlled by `warnings`) or multiprocessing, and write results in `'subset'` sub-directory instead of the current directory. For this example, we use `registry_db` from *VO database examples*:

```
>>> urls = registry_db.list_catalogs_by_url(pattern='stsci.edu')
>>> urls
['http://archive.stsci.edu/befs/search.php?',
 'http://archive.stsci.edu/copernicus/search.php?', ...,
```

(continues on next page)

(continued from previous page)

```
 'http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&']
>>> validate.check_conesearch_sites(
...     destdir='./subset', verbose=False, parallel=False, url_list=urls)
# ...
INFO: check_conesearch_sites took 22.44089651107788 s on AVERAGE...
```

Add 'W24' from `astropy.io.votable.exceptions` to the list of non-critical warnings to be ignored and re-run default validation. This is *not* recommended unless you know exactly what you are doing:

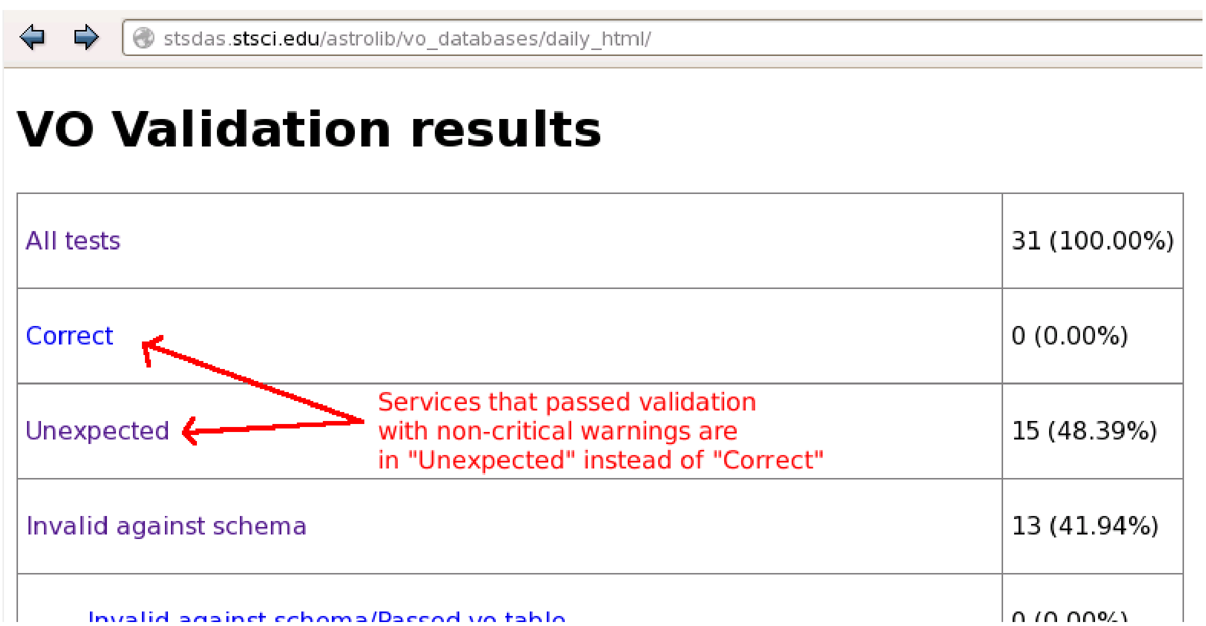
```
>>> from astroquery.vo_conesearch.validator import conf as validator_conf
>>> new_warns = validator_conf.noncritical_warnings + ['W24']
>>> with validator_conf.set_temp('noncritical_warnings', new_warns):
...     validate.check_conesearch_sites()
```

Validate *all* Cone Search services in the master registry (this will take a while) and write results in 'all' sub-directory:

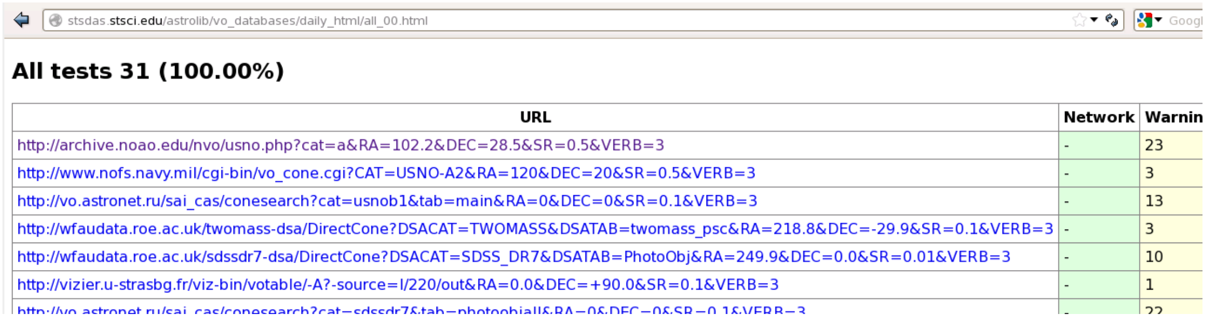
```
>>> validate.check_conesearch_sites(destdir='./all', url_list=None)
```

To look at the HTML pages of the validation results in the current directory using Firefox browser (images shown are from STScI server but your own results should look similar):

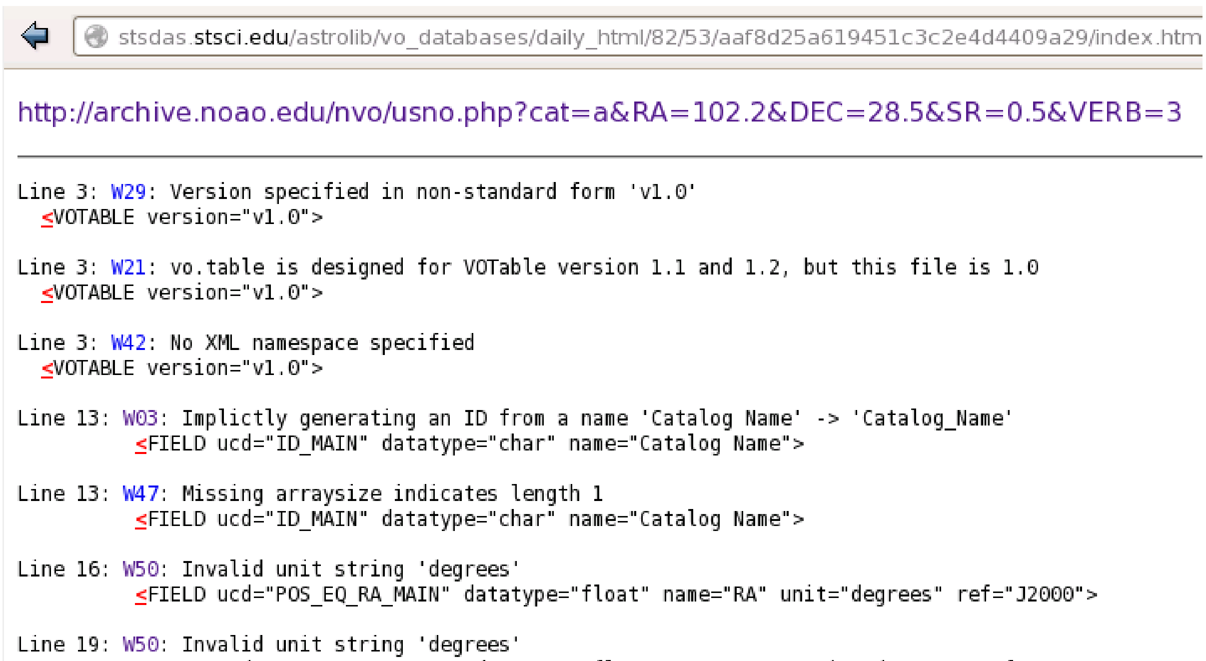
```
firefox results/index.html
```



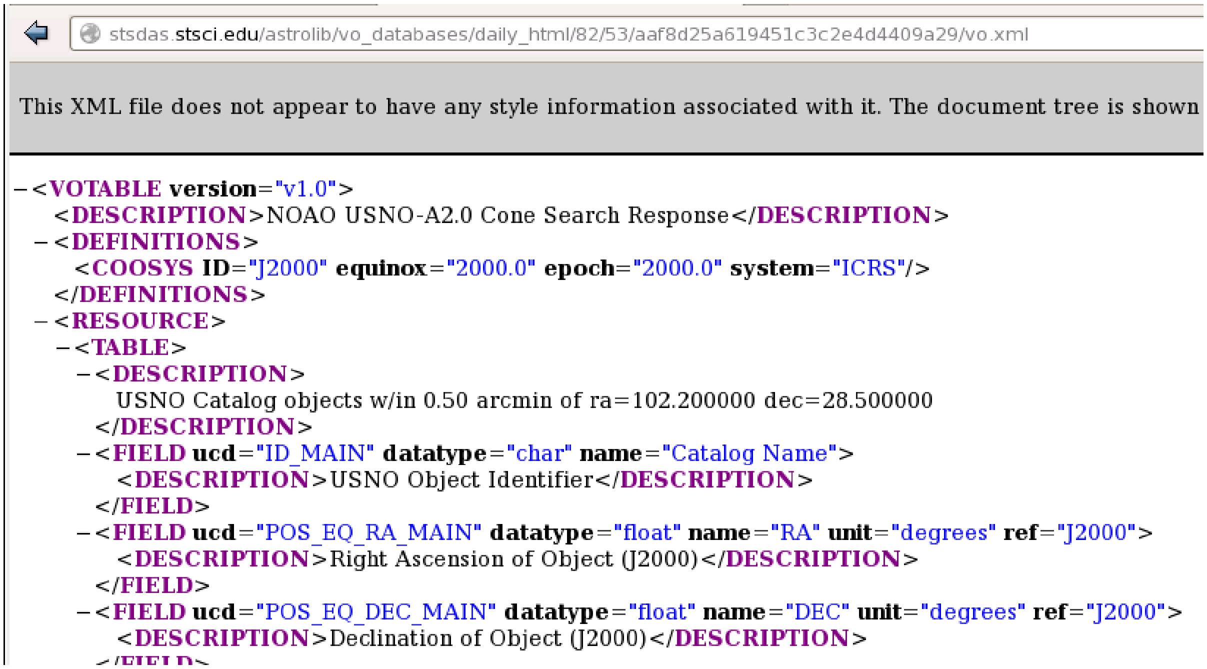

When you click on 'All tests' from the page above, you will see all the Cone Search services validated with a summary of validation results:




When you click on any of the listed URLs from above, you will see detailed validation warnings and exceptions for the selected URL:




When you click on the URL on top of the page above, you will see the actual VO Table returned by the Cone Search query:



stsdas.stsci.edu/astrolib/vo_databases/daily_html/82/53/aaf8d25a619451c3c2e4d4409a29/vo.xml

This XML file does not appear to have any style information associated with it. The document tree is shown

```
-<VOTABLE version="v1.0">
   <DESCRIPTION>NOAO USNO-A2.0 Cone Search Response</DESCRIPTION>
  -<DEFINITIONS>
     <COOSYS ID="J2000" equinox="2000.0" epoch="2000.0" system="ICRS"/>
   </DEFINITIONS>
  -<RESOURCE>
    -<TABLE>
      -<DESCRIPTION>
         USNO Catalog objects w/in 0.50 arcmin of ra=102.200000 dec=28.500000
       </DESCRIPTION>
      -<FIELD ucd="ID_MAIN" datatype="char" name="Catalog Name">
         <DESCRIPTION>USNO Object Identifier</DESCRIPTION>
       </FIELD>
      -<FIELD ucd="POS_EQ_RA_MAIN" datatype="float" name="RA" unit="degrees" ref="J2000">
         <DESCRIPTION>Right Ascension of Object (J2000)</DESCRIPTION>
       </FIELD>
      -<FIELD ucd="POS_EQ_DEC_MAIN" datatype="float" name="DEC" unit="degrees" ref="J2000">
         <DESCRIPTION>Declination of Object (J2000)</DESCRIPTION>
```

### Inspection of Validation Results

`astroquery.vo_conesearch.validator.inspect` inspects results from *Validation for Simple Cone Search*. It reads in JSON files of `VOSDatabase` residing in `astroquery.vo_conesearch.conf.vos_baseurl`, which can be changed to point to a different location.

### Configurable Items

This parameter is set via Configuration system (astropy.config):

- `astroquery.vo_conesearch.conf.vos_baseurl`

### Examples

```
>>> from astroquery.vo_conesearch.validator import inspect
```

Load Cone Search validation results from `astroquery.vo_conesearch.conf.vos_baseurl` (by default, the one used by *Simple Cone Search*):

```
>>> r = inspect.ConeSearchResults()
Downloading http://.../conesearch_good.json
|==========================================|  37k/ 37k (100.00%)         0s
Downloading http://.../conesearch_warn.json
|==========================================| 312k/312k (100.00%)         0s
Downloading http://.../conesearch_exception.json
|==========================================|  15k/ 15k (100.00%)         0s
Downloading http://.../conesearch_error.json
|==========================================|  44 / 44  (100.00%)         0s
```

Print tally. In this example, there are 9 Cone Search services that passed validation with non-critical warnings, 13 with critical warnings, 6 with exceptions, and 0 with network error:

```
>>> r.tally()
good: 9 catalog(s)
warn: 13 catalog(s)
exception: 6 catalog(s)
error: 0 catalog(s)
total: 28 catalog(s)
```

Print a list of good Cone Search catalogs, each with title, access URL, warning codes collected, and individual warnings:

```
>>> r.list_cats('good')
Guide Star Catalog v2 1
http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&
W48,W50
.../vo.xml:136:0: W50: Invalid unit string 'pixel'
.../vo.xml:155:0: W48: Unknown attribute 'nrows' on TABLEDATA
# ...
USNO-A2 Catalogue 1
http://www.nofs.navy.mil/cgi-bin/vo_cone.cgi?CAT=USNO-A2&
W17,W21,W42
.../vo.xml:4:0: W21: vo.table is designed for VOTable version 1.1 and 1.2...
.../vo.xml:4:0: W42: No XML namespace specified
.../vo.xml:15:15: W17: VOTABLE element contains more than one DESCRIPTION...
```

List Cone Search catalogs with warnings, excluding warnings that were ignored in `astroquery.vo_conesearch.validator.conf.noncritical_warnings`, and writes the output to a file named `'warn_cats.txt'` in the current directory. This is useful to see why the services failed validations:

```
>>> with open('warn_cats.txt', 'w') as fout:
...     r.list_cats('warn', fout=fout, ignore_noncrit=True)
```

List the titles of all good Cone Search catalogs:

```
>>> r.catkeys['good']
['Guide Star Catalog v2 1',
 'SDSS DR8 - Sloan Digital Sky Survey Data Release 8 1', ...,
 'USNO-A2 Catalogue 1']
```

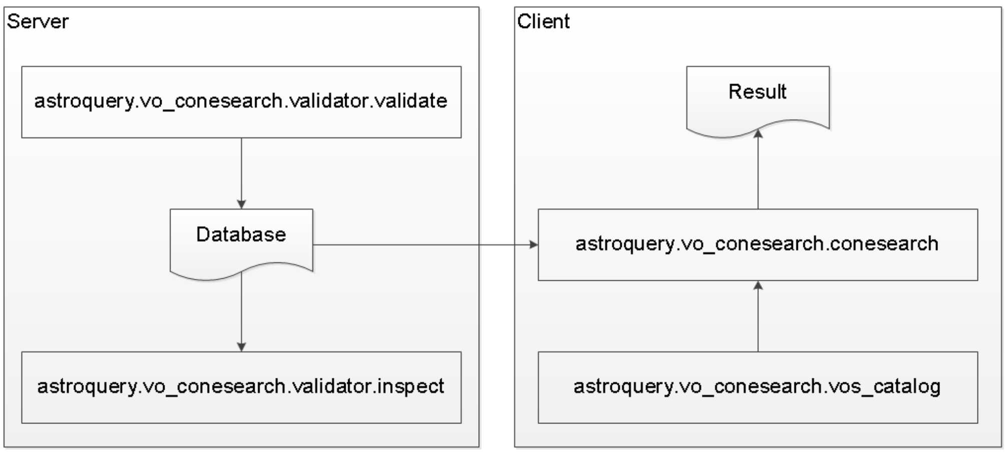
Print the details of catalog titled `'USNO-A2 Catalogue 1'`:

```
>>> r.print_cat('USNO-A2 Catalogue 1')
{
    "capabilityClass": "ConeSearch",
    "capabilityStandardID": "ivo://ivoa.net/std/ConeSearch",
    "capabilityValidationLevel": "2",
    "contentLevel": "#University#Research#Amateur#",
    # ...
    "version": "",
    "waveband": "#Optical#"
}
Found in good
```

Load Cone Search validation results from a local directory named `'subset'`. This is useful if you ran your own *Validation for Simple Cone Search* and wish to inspect the output databases. This example reads in validation of STScI Cone Search services done in *Validation for Simple Cone Search Examples*:

```
>>> from astroquery.vo_conesearch import conf
>>> with conf.set_temp('vos_baseurl', './subset/'):
>>>     r = inspect.ConeSearchResults()
>>> r.tally()
good: 11 catalog(s)
warn: 3 catalog(s)
exception: 15 catalog(s)
error: 0 catalog(s)
total: 29 catalog(s)
>>> r.catkeys['good']
[u'Berkeley Extreme and Far-UV Spectrometer 1',
 u'Copernicus Satellite 1', ...,
 u'Wisconsin Ultraviolet Photo-Polarimeter Experiment 1']
```

They are designed to be used in a work flow as illustrated below:




The one that a typical user needs is the *Simple Cone Search* component (see *Cone Search Examples*).

## 29.5 See Also

- Simple Cone Search Version 1.03, IVOA Recommendation (22 February 2008)
- STScI VAO Registry
- STScI VO Databases

## 29.6 Reference/API

### 29.6.1 astroquery.vo_conesearch.core Module

#### Classes

| | |
|---|---|
| `ConeSearchClass`() | The class for querying the Virtual Observatory (VO) Cone Search web service. |

### ConeSearchClass

**class** astroquery.vo_conesearch.core.**ConeSearchClass**

Bases: astroquery.query.BaseQuery

The class for querying the Virtual Observatory (VO) Cone Search web service.

#### Examples

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> from astroquery.vo_conesearch import ConeSearch
>>> ConeSearch.query_region(SkyCoord.from_name('M31'), 5 * u.arcsecond)
<Table masked=True length=6>
    objID           gscID2      ... compassGSC2id   Mag
                                ...                 mag
    int64           object      ...     object    float32
-------------- ---------------- ... ------------- -------
23323175812944 00424433+4116085 ... 6453800072293      --
23323175812933 00424455+4116103 ... 6453800072282      --
23323175812939 00424464+4116092 ... 6453800072288      --
23323175812931 00424464+4116106 ... 6453800072280      --
23323175812948 00424403+4116069 ... 6453800072297      --
23323175812930 00424403+4116108 ... 6453800072279      --
```

#### Attributes Summary

| |
|---|
| PEDANTIC |
| TIMEOUT |
| URL |

#### Methods Summary

| | |
|---|---|
| query_region(coordinates, radius[, verb, …]) | Perform Cone Search and returns the result of the first successful query. |
| query_region_async(*args, **kwargs) | This is not implemented. |

#### Attributes Documentation

**PEDANTIC = False**

**TIMEOUT = 30.0**

**URL = 'http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&'**

### Methods Documentation

**query_region**(*coordinates*, *radius*, *verb=1*, *get_query_payload=False*, *cache=True*, *verbose=False*)

Perform Cone Search and returns the result of the first successful query.

**Parameters**

**coordinates** : str, `astropy.coordinates` object, list, or tuple

Position of the center of the cone to search. It may be specified as an object from the Astronomical Coordinate Systems (astropy.coordinates) package, string as accepted by `parse_coordinates()`, or tuple/list. If given as tuple or list, it is assumed to be `(RA, DEC)` in the ICRS coordinate frame, given in decimal degrees.

**radius** : float or `Quantity`

Radius of the cone to search:

- If float is given, it is assumed to be in decimal degrees.
- If `Quantity` is given, it is internally converted to degrees.

**verb** : {1, 2, 3}, optional

Verbosity indicating how many columns are to be returned in the resulting table. Support for this parameter by a Cone Search service implementation is optional. If the service supports the parameter:

1. Return the bare minimum number of columns that the provider considers useful in describing the returned objects.
2. Return a medium number of columns between the minimum and maximum (inclusive) that are considered by the provider to most typically useful to the user.
3. Return all of the columns that are available for describing the objects.

If not supported, the service should ignore the parameter and always return the same columns for every request.

**get_query_payload** : bool, optional

Just return the dict of HTTP request parameters.

**cache** : bool, optional

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

**verbose** : bool, optional

Verbose output, including VO table warnings.

**Returns**

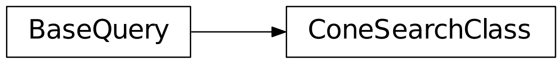
**result** : `astropy.io.votable.tree.Table`

Table from successful VO service request.

**query_region_async**(**args*, ***kwargs*)

This is not implemented. Use `AsyncConeSearch` instead.

#### Class Inheritance Diagram




### 29.6.2 astroquery.vo_conesearch.vos_catalog Module

Common utilities for accessing VO simple services.

**Note:** Some functions are not used by Astroquery but kept for backward-compatibility with `astropy.vo.client`.

#### Functions

| | |
|---|---|
| `get_remote_catalog_db`(dbname[, cache, verbose]) | Get a database of VO services (which is a JSON file) from a remote location. |
| `call_vo_service`(service_type[, catalog_db, . . . ]) | Makes a generic VO service call. |
| `list_catalogs`(service_type[, cache, verbose]) | List the catalogs available for the given service type. |

#### get_remote_catalog_db

`astroquery.vo_conesearch.vos_catalog.`**`get_remote_catalog_db`**(*dbname*, *cache=True*, *verbose=True*)

Get a database of VO services (which is a JSON file) from a remote location.

**Parameters**

**dbname** : str

Prefix of JSON file to download from `astroquery.vo_conesearch.conf.vos_baseurl`.

**cache** : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

**verbose** : bool

Show download progress bars.

**Returns**

**db** : `VOSDatabase`

A database of VO services.

### call_vo_service

`astroquery.vo_conesearch.vos_catalog.call_vo_service`(*service_type*, *catalog_db=None*, *pedantic=None*, *verbose=True*, *cache=True*, *kwargs={}*)

Makes a generic VO service call.

**Parameters**

**service_type** : str

Name of the type of service, e.g., 'conesearch_good'. Used in error messages and to select a catalog database if `catalog_db` is not provided.

**catalog_db**

May be one of the following, in order from easiest to use to most control:

- `None`: A database of `service_type` catalogs is downloaded from `astroquery.vo_conesearch.conf.vos_baseurl`. The first catalog in the database to successfully return a result is used.
- *catalog name*: A name in the database of `service_type` catalogs at `astroquery.vo_conesearch.conf.vos_baseurl` is used. For a list of acceptable names, use `list_catalogs()`.
- *url*: The prefix of a URL to a IVOA Service for `service_type`. Must end in either '?' or '&'.
- `VOSCatalog` object: A specific catalog manually downloaded and selected from the database (see *General VO Services Access*).
- Any of the above 3 options combined in a list, in which case they are tried in order.

**pedantic** : bool or `None`

When `True`, raise an error when the file violates the spec, otherwise issue a warning. Warnings may be controlled using `warnings` module. When not provided, uses the configuration setting `astroquery.vo_conesearch.conf.pedantic`, which defaults to `False`.

**verbose** : bool

Verbose output.

**cache** : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

**kwargs** : dictionary

Keyword arguments to pass to the catalog service. No checking is done that the arguments are accepted by the service, etc.

**Returns**

**obj** : `astropy.io.votable.tree.Table`

First table from first successful VO service request.

**Raises**

**VOSError**

If VO service request fails.

### list_catalogs

astroquery.vo_conesearch.vos_catalog.**list_catalogs**(*service_type*, *cache=True*, *verbose=True*, ***kwargs*)

List the catalogs available for the given service type.

**Parameters**

**service_type** : str

Name of the type of service, e.g., 'conesearch_good'.

**cache** : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

**verbose** : bool

Show download progress bars.

**pattern** : str or `None`

If given string is anywhere in a catalog name, it is considered a matching catalog. It accepts patterns as in `fnmatch` and is case-insensitive. By default, all catalogs are returned.

**sort** : bool

Sort output in alphabetical order. If not sorted, the order depends on dictionary hashing. Default is `True`.

**Returns**

**arr** : list of str

List of catalog names.

### Classes

| | |
|---|---|
| `VOSBase`(tree) | Base class for VO catalog and database. |
| `VOSCatalog`(tree) | A class to represent VO Service Catalog. |
| `VOSDatabase`(tree) | A class to represent a collection of `VOSCatalog`. |

### VOSBase

**class** astroquery.vo_conesearch.vos_catalog.**VOSBase**(*tree*)

Bases: `object`

Base class for VO catalog and database.

**Parameters**

**tree** : JSON tree

#### Methods Summary

| | |
|---|---|
| `dumps`() | Dump the contents into a string. |

### Methods Documentation

**dumps**()

Dump the contents into a string.

**Returns**

**s** : str

Contents as JSON string dump.

## VOSCatalog

**class** astroquery.vo_conesearch.vos_catalog.**VOSCatalog**(*tree*)

Bases: astroquery.vo_conesearch.vos_catalog.VOSBase

A class to represent VO Service Catalog.

**Parameters**

**tree** : JSON tree

**Raises**

**VOSError**

Missing necessary key(s).

### Methods Summary

| | |
|---|---|
| create(title, url, **kwargs) | Create a new VO Service Catalog with user parameters. |
| delete_attribute(key) | Delete given metadata key and its value from the catalog. |

### Methods Documentation

**classmethod create**(*title*, *url*, ***kwargs*)

Create a new VO Service Catalog with user parameters.

**Parameters**

**title** : str

Title of the catalog.

**url** : str

Access URL of the service. This is used to build queries.

**kwargs** : dict

Additional metadata as keyword-value pairs describing the catalog, except 'title' and 'url'.

**Returns**

**cat** : VOSCatalog

VO Service Catalog.

**Raises**

**TypeError**

Multiple values given for keyword argument.

**delete_attribute**(*key*)
Delete given metadata key and its value from the catalog.

**Parameters**
**key** : str

Metadata key to delete.

**Raises**
**KeyError**

Key not found.

**VOSError**

Key must exist in catalog, therefore cannot be deleted.

### VOSDatabase

**class** astroquery.vo_conesearch.vos_catalog.**VOSDatabase**(*tree*)
Bases: `astroquery.vo_conesearch.vos_catalog.VOSBase`

A class to represent a collection of `VOSCatalog`.

**Parameters**
**tree** : JSON tree

**Raises**
**VOSError**

If given `tree` does not have 'catalogs' key or catalog is invalid.

#### Attributes Summary

| | |
|---|---|
| `version` | Database version number. |

#### Methods Summary

| | |
|---|---|
| `add_catalog`(name, cat[, allow_duplicate_url]) | Add a catalog to database. |
| `add_catalog_by_url`(name, url, **kwargs) | Like `add_catalog()` but the catalog is created with only the given name and access URL. |
| `create_empty`() | Create an empty database of VO services. |
| `delete_catalog`(name) | Delete a catalog from database with given name. |
| `delete_catalog_by_url`(url) | Like `delete_catalog()` but using access URL. |
| `from_json`(filename, **kwargs) | Create a database of VO services from a JSON file. |
| `from_registry`(registry_url[, timeout]) | Create a database of VO services from VO registry URL. |
| `get_catalog`(name) | Get one catalog of given name. |
| `get_catalog_by_url`(url) | Like `get_catalog()` but using access URL look-up. |
| `get_catalogs`() | Iterator to get all catalogs. |
| `get_catalogs_by_url`(url) | Like `get_catalogs()` but using access URL look-up. |
| `list_catalogs`([pattern, sort]) | List catalog names. |

Continued on next page

Table 9 – continued from previous page

| | |
|---|---|
| `list_catalogs_by_url`([pattern, sort]) | Like `list_catalogs()` but using access URL. |
| `merge`(other, **kwargs) | Merge two database together. |
| `to_json`(filename[, overwrite]) | Write database content to a JSON file. |

### Attributes Documentation

**`version`**
Database version number.

### Methods Documentation

**`add_catalog`**(*name*, *cat*, *allow_duplicate_url=False*)
Add a catalog to database.

**Parameters**

**name** : str

Primary key for the catalog.

**cat** : `VOSCatalog`

Catalog to add.

**allow_duplicate_url** : bool

Allow catalog with duplicate access URL?

**Raises**

**VOSError**

Invalid catalog.

**DuplicateCatalogName**

Catalog with given name already exists.

**DuplicateCatalogURL**

Catalog with given access URL already exists.

**`add_catalog_by_url`**(*name*, *url*, ***kwargs*)
Like `add_catalog()` but the catalog is created with only the given name and access URL.

**Parameters**

**name** : str

Primary key for the catalog.

**url** : str

Access URL of the service. This is used to build queries.

**kwargs** : dict

Keywords accepted by `add_catalog()`.

**`classmethod create_empty`**()
Create an empty database of VO services.

Empty database format:

```
{
    "__version__": 1,
    "catalogs" : {
    }
}
```

**Returns**

**db** : VOSDatabase

Empty database.

**delete_catalog**(*name*)

Delete a catalog from database with given name.

**Parameters**

**name** : str

Primary key identifying the catalog.

**Raises**

**MissingCatalog**

If catalog is not found.

**delete_catalog_by_url**(*url*)

Like delete_catalog() but using access URL. On multiple matches, all matches are deleted.

**classmethod from_json**(*filename*, ***kwargs*)

Create a database of VO services from a JSON file.

Example JSON format for Cone Search:

```
{
    "__version__": 1,
    "catalogs" : {
        "My Cone Search": {
            "capabilityClass": "ConeSearch",
            "title": "My Cone Search",
            "url": "http://foo/cgi-bin/search?CAT=bar&",
            ...
        },
        "Another Cone Search": {
            ...
        }
    }
}
```

**Parameters**

**filename** : str

JSON file.

**kwargs** : dict

Keywords accepted by get_readable_fileobj().

**Returns**

**db** : VOSDatabase

Database from given file.

**classmethod from_registry**(*registry_url*, *timeout=60*, ***kwargs*)

Create a database of VO services from VO registry URL.

This is described in detail in *Building the Database from Registry*, except for the `validate_xxx` keys that are added by the validator itself.

**Parameters**

**registry_url** : str

URL of VO registry that returns a VO Table. For example, see `astroquery.vo_conesearch.validator.conf.cs_mstr_list`. Pedantic is automatically set to `False` for parsing.

**timeout** : number

Temporarily set `astropy.utils.data.conf.remote_timeout` to this value to avoid time out error while reading the entire registry.

**kwargs** : dict

Keywords accepted by `get_readable_fileobj()`.

**Returns**

**db** : `VOSDatabase`

Database from given registry.

**Raises**

**VOSError**

Invalid VO registry.

**get_catalog**(*name*)

Get one catalog of given name.

**Parameters**

**name** : str

Primary key identifying the catalog.

**Returns**

**obj** : `VOSCatalog`

**Raises**

**MissingCatalog**

If catalog is not found.

**get_catalog_by_url**(*url*)

Like `get_catalog()` but using access URL look-up. On multiple matches, only first match is returned.

**get_catalogs**()

Iterator to get all catalogs.

**get_catalogs_by_url**(*url*)

Like `get_catalogs()` but using access URL look-up.

**list_catalogs**(*pattern=None*, *sort=True*)

List catalog names.

**Parameters**

**pattern** : str or `None`

If given string is anywhere in a catalog name, it is considered a matching catalog. It accepts patterns as in `fnmatch` and is case-insensitive. By default, all catalogs are returned.

**sort** : bool

Sort output in alphabetical order. If not sorted, the order depends on dictionary hashing. Default is `True`.

**Returns**

**out_arr** : list of str

List of catalog names.

**list_catalogs_by_url**(*pattern=None*, *sort=True*)

Like `list_catalogs()` but using access URL.

**merge**(*other*, ***kwargs*)

Merge two database together.

**Parameters**

**other** : `VOSDatabase`

The other database to merge.

**kwargs** : dict

Keywords accepted by `add_catalog()`.

**Returns**

**db** : `VOSDatabase`

Merged database.

**Raises**

**VOSError**

Invalid database or incompatible version.

**to_json**(*filename*, *overwrite=False*)

Write database content to a JSON file.

**Parameters**

**filename** : str

JSON file.

**overwrite** : bool

If `True`, overwrite the output file if it exists.

**Raises**

**OSError**

If the file exists and `overwrite` is `False`.

### 29.6.3 astroquery.vo_conesearch.conesearch Module

Support VO Simple Cone Search capabilities.

**Note:** This maintains a similar API as `astropy.vo.client`.

### Functions

| | |
|---|---|
| conesearch(center, radius[, verb, . . . ]) | Perform Cone Search and returns the result of the first successful query. |
| search_all(*args, **kwargs) | Perform Cone Search and returns the results of all successful queries. |
| list_catalogs(**kwargs) | Return the available Cone Search catalogs as a list of strings. |
| predict_search(url, *args, **kwargs) | Predict the run time needed and the number of objects for a Cone Search for the given access URL, position, and radius. |
| conesearch_timer(*args, **kwargs) | Time a single Cone Search using `astropy.utils.timer.timefunc` with a single try and a verbose timer. |

### conesearch

`astroquery.vo_conesearch.conesearch.`**`conesearch`**(*center*, *radius*, *verb=1*, *catalog_db=None*, *pedantic=None*, *verbose=True*, *cache=True*, *timeout=None*, *query_all=False*)

Perform Cone Search and returns the result of the first successful query.

**Parameters**

**center** : str, `astropy.coordinates` object, list, or tuple

Position of the center of the cone to search. It may be specified as an object from the Astronomical Coordinate Systems (astropy.coordinates) package, string as accepted by `parse_coordinates()`, or tuple/list. If given as tuple or list, it is assumed to be `(RA, DEC)` in the ICRS coordinate frame, given in decimal degrees.

**radius** : float or `Quantity`

Radius of the cone to search:

- If float is given, it is assumed to be in decimal degrees.
- If astropy quantity is given, it is internally converted to degrees.

**verb** : {1, 2, 3}

Verbosity indicating how many columns are to be returned in the resulting table. Support for this parameter by a Cone Search service implementation is optional. If the service supports the parameter:

1. Return the bare minimum number of columns that the provider considers useful in describing the returned objects.
2. Return a medium number of columns between the minimum and maximum (inclusive) that are considered by the provider to most typically useful to the user.
3. Return all of the columns that are available for describing the objects.

If not supported, the service should ignore the parameter and always return the same columns for every request.

**catalog_db**

May be one of the following, in order from easiest to use to most control:

- `None`: A database of `astroquery.vo_conesearch.conf.conesearch_dbname` catalogs is downloaded from `astroquery.vo_conesearch.conf.vos_baseurl`. The first catalog in the database to successfully return a result is used.
- *catalog name*: A name in the database of `astroquery.vo_conesearch.conf.conesearch_dbname` catalogs at `astroquery.vo_conesearch.conf.vos_baseurl` is used. For a list of acceptable names, use `astroquery.vo_conesearch.vos_catalog.list_catalogs()`.
- *url*: The prefix of a URL to a IVOA Service for `astroquery.vo_conesearch.conf.conesearch_dbname`. Must end in either '?' or '&'.
- `VOSCatalog` object: A specific catalog manually downloaded and selected from the database (see *General VO Services Access*).
- Any of the above 3 options combined in a list, in which case they are tried in order.

**pedantic** : bool or `None`

When `True`, raise an error when the result violates the spec, otherwise issue a warning. Warnings may be controlled using `warnings` module. When not provided, uses the configuration setting `astroquery.vo_conesearch.conf.pedantic`, which defaults to `False`.

**verbose** : bool

Verbose output.

**cache** : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

**timeout** : number or `None`

Timeout limit in seconds for each service being queries. If `None`, use default.

**query_all** : bool

This is used by `search_all()`.

**Returns**

**obj** : `astropy.io.votable.tree.Table`

First table from first successful VO service request.

**Raises**

**ConeSearchError**

When invalid inputs are passed into Cone Search.

**VOSError**

If VO service request fails.

## search_all

`astroquery.vo_conesearch.conesearch.`**`search_all`**`(*args, **kwargs)`

Perform Cone Search and returns the results of all successful queries.

> **Warning:** Could potentially take up significant run time and computing resources.

**Parameters**

**args, kwargs**

Arguments and keywords accepted by `conesearch()`.

**Returns**

**result** : dict of `astropy.io.votable.tree.Table` objects

A dictionary of tables from successful VO service requests, with keys being the access URLs. If none is successful, an empty dictionary is returned.

**Raises**

**ConeSearchError**

When invalid inputs are passed into Cone Search.

### list_catalogs

`astroquery.vo_conesearch.conesearch.`**`list_catalogs`**`(**kwargs)`

Return the available Cone Search catalogs as a list of strings. These can be used for the `catalog_db` argument to `conesearch()`.

**Parameters**

**cache** : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

**verbose** : bool

Show download progress bars.

**pattern** : str or `None`

If given string is anywhere in a catalog name, it is considered a matching catalog. It accepts patterns as in `fnmatch` and is case-insensitive. By default, all catalogs are returned.

**sort** : bool

Sort output in alphabetical order. If not sorted, the order depends on dictionary hashing. Default is `True`.

**Returns**

**arr** : list of str

List of catalog names.

### predict_search

`astroquery.vo_conesearch.conesearch.`**`predict_search`**`(url, *args, **kwargs)`

Predict the run time needed and the number of objects for a Cone Search for the given access URL, position, and radius.

Run time prediction uses `astropy.utils.timer.RunTimePredictor`. Baseline searches are done with starting and ending radii at 0.05 and 0.5 of the given radius, respectively.

Extrapolation on good data uses least-square straight line fitting, assuming linear increase of search time and number of objects with radius, which might not be accurate for some cases. If there are less than 3 data points in the fit, it fails.

Warnings (controlled by warnings) are given when:

1. Fitted slope is negative.
2. Any of the estimated results is negative.
3. Estimated run time exceeds `astroquery.vo_conesearch.conf.timeout`.

---

**Note:** If `verbose=True`, extra log info will be provided. But unlike `conesearch_timer()`, timer info is suppressed.

The predicted results are just *rough* estimates.

Prediction is done using `astroquery.vo_conesearch.core.ConeSearchClass`. Prediction for `AsyncConeSearch` is not supported.

---

**Parameters**

**url** : str

Cone Search access URL to use.

**plot** : bool

If `True`, plot will be displayed. Plotting uses matplotlib.

**args, kwargs**

See `astroquery.vo_conesearch.core.ConeSearchClass.query_region()`.

**Returns**

**t_est** : float

Estimated time in seconds needed for the search.

**n_est** : int

Estimated number of objects the search will yield.

**Raises**

**AssertionError**

If prediction fails.

**ConeSearchError**

If input parameters are invalid.

**VOSError**

If VO service request fails.

### conesearch_timer

`astroquery.vo_conesearch.conesearch.`**`conesearch_timer`**`(*args, **kwargs)`

Time a single Cone Search using `astropy.utils.timer.timefunc` with a single try and a verbose timer.

**Parameters**

**args, kwargs**

See `conesearch()`.

**Returns**

**t** : float

Run time in seconds.

**obj** : astropy.io.votable.tree.Table

First table from first successful VO service request.

### Classes

| | |
|---|---|
| AsyncConeSearch(*args, **kwargs) | Perform a Cone Search asynchronously and returns the result of the first successful query. |
| AsyncSearchAll(*args, **kwargs) | Perform a Cone Search asynchronously, storing all results instead of just the result from first successful query. |

### AsyncConeSearch

**class** astroquery.vo_conesearch.conesearch.**AsyncConeSearch**(**args*, ***kwargs*)

Bases: astroquery.vo_conesearch.async.AsyncBase

Perform a Cone Search asynchronously and returns the result of the first successful query.

**Note:** See AsyncBase for more details.

**Parameters**

**args, kwargs**

See conesearch().

#### Examples

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> from astroquery.vo_conesearch import conesearch
>>> c = coord.ICRS(6.0223 * u.degree, -72.0814 * u.degree)
>>> async_search = conesearch.AsyncConeSearch(
...     c, 0.5 * u.degree,
...     catalog_db='The PMM USNO-A1.0 Catalogue (Monet 1997) 1')
```

Check search status:

```
>>> async_search.running()
True
>>> async_search.done()
False
```

Get search results after a 30-second wait (not to be confused with astroquery.vo_conesearch.conf.timeout that governs individual Cone Search queries). If search is still not done after 30 seconds, TimeoutError is raised. Otherwise, Cone Search result is returned and can be manipulated as in *Simple Cone Search Examples*. If no timeout keyword given, it waits until completion:

```
>>> async_result = async_search.get(timeout=30)
>>> cone_arr = async_result.array.data
```

(continues on next page)

(continued from previous page)

```
>>> cone_arr.size
36184
```

### AsyncSearchAll

**class** astroquery.vo_conesearch.conesearch.**AsyncSearchAll**(**args*, ***kwargs*)

Bases: astroquery.vo_conesearch.async.AsyncBase

Perform a Cone Search asynchronously, storing all results instead of just the result from first successful query.

**Note:** See AsyncBase for more details.

**Parameters**

**args, kwargs**

See search_all().

#### Examples

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> from astroquery.vo_conesearch import conesearch
>>> c = coord.ICRS(6.0223 * u.degree, -72.0814 * u.degree)
>>> async_search = conesearch.AsyncSearchAll(c, 0.5 * u.degree)
```

Check search status:

```
>>> async_search.running()
True
>>> async_search.done()
False
```

Get a dictionary of all search results after a 30-second wait (not to be confused with astroquery.vo_conesearch.conf.timeout that governs individual Cone Search queries). If search is still not done after 30 seconds, TimeoutError is raised. Otherwise, a dictionary is returned and can be manipulated as in *Simple Cone Search Examples*. If no timeout keyword given, it waits until completion:

```
>>> async_allresults = async_search.get(timeout=30)
>>> all_catalogs = list(async_allresults)
>>> first_cone_arr = async_allresults[all_catalogs[0]].array.data
>>> first_cone_arr.size
36184
```

## 29.6.4 astroquery.vo_conesearch.async Module

Asynchronous VO service requests.

### Classes

| | |
|---|---|
| AsyncBase(func, *args, **kwargs) | Base class for asynchronous VO service requests using concurrent.futures.ThreadPoolExecutor. |

### AsyncBase

**class** astroquery.vo_conesearch.async.**AsyncBase**(*func*, **args*, ***kwargs*)

Bases: object

Base class for asynchronous VO service requests using concurrent.futures.ThreadPoolExecutor.

Service request will be forced to run in silent mode by setting verbose=False. Warnings are controlled by warnings module.

**Note:** Methods of the attributes can be accessed directly, with priority given to executor.

**Parameters**

**func** : function

The function to run.

**args, kwargs**

Arguments and keywords accepted by the service request function to be called asynchronously.

#### Attributes

| | |
|---|---|
| **executor** | (concurrent.futures.ThreadPoolExecutor) Executor running the function on single thread. |
| **future** | (concurrent.futures.Future) Asynchronous execution created by executor. |

#### Methods Summary

| | |
|---|---|
| get([timeout]) | Get result, if available, then shut down thread. |

#### Methods Documentation

**get**(*timeout=None*)

Get result, if available, then shut down thread.

**Parameters**

**timeout** : int or float

Wait the given amount of time in seconds before obtaining result. If not given, wait indefinitely until function is done.

**Returns**

result

Result returned by the function.

**Raises**

**Exception**

Errors raised by `concurrent.futures.Future`.

## 29.6.5 astroquery.vo_conesearch.exceptions Module

Exceptions related to Virtual Observatory (VO).

### Classes

| | |
|---|---|
| `BaseVOError` | Base class for VO exceptions. |
| `VOSError` | General VO service exception. |
| `MissingCatalog` | VO catalog is missing. |
| `DuplicateCatalogName` | VO catalog of the same title already exists. |
| `DuplicateCatalogURL` | VO catalog of the same access URL already exists. |
| `InvalidAccessURL` | Invalid access URL. |
| `ConeSearchError` | General Cone Search exception. |

### BaseVOError

**exception** `astroquery.vo_conesearch.exceptions.`**`BaseVOError`**

Base class for VO exceptions.

### VOSError

**exception** `astroquery.vo_conesearch.exceptions.`**`VOSError`**

General VO service exception.

### MissingCatalog

**exception** `astroquery.vo_conesearch.exceptions.`**`MissingCatalog`**

VO catalog is missing.

### DuplicateCatalogName

**exception** `astroquery.vo_conesearch.exceptions.`**`DuplicateCatalogName`**

VO catalog of the same title already exists.

### DuplicateCatalogURL

**exception** `astroquery.vo_conesearch.exceptions.`**`DuplicateCatalogURL`**

VO catalog of the same access URL already exists.

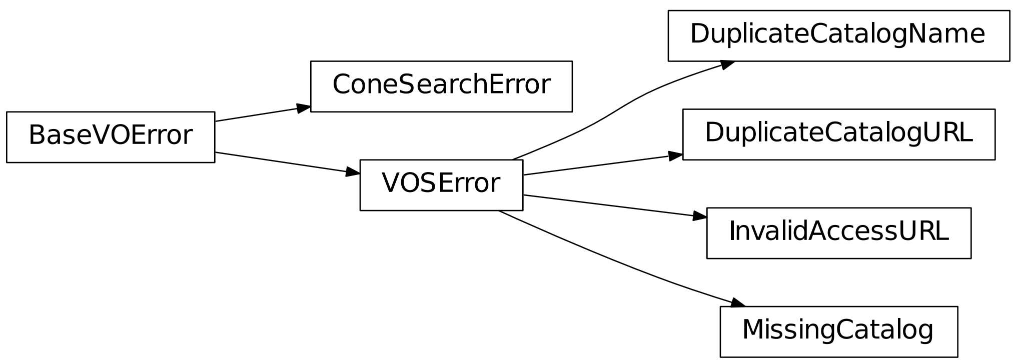
### InvalidAccessURL

**exception** astroquery.vo_conesearch.exceptions.**InvalidAccessURL**
Invalid access URL.

### ConeSearchError

**exception** astroquery.vo_conesearch.exceptions.**ConeSearchError**
General Cone Search exception.

### Class Inheritance Diagram




## 29.6.6 astroquery.vo_conesearch.validator.validate Module

Validate VO Services.

### Functions

| | |
|---|---|
| check_conesearch_sites([destdir, verbose, ...]) | Validate Cone Search Services. |

### check_conesearch_sites

astroquery.vo_conesearch.validator.validate.**check_conesearch_sites**(*destdir='.'*, *verbose=True*, *parallel=True*, *url_list='default'*)
Validate Cone Search Services.

**Note:** URLs are unescaped prior to validation.

Only check queries with <testQuery> parameters. Does not perform meta-data and erroneous queries.

**Parameters**

**destdir** : str, optional

Directory to store output files. Will be created if does not exist. Existing files with these names will be deleted or replaced:

- conesearch_good.json
- conesearch_warn.json
- conesearch_exception.json
- conesearch_error.json

**verbose** : bool, optional

Print extra info to log.

**parallel** : bool, optional

Enable multiprocessing.

**url_list** : list of string, optional

Only check these access URLs against `astroquery.vo_conesearch.validator.conf.conesearch_master_list` and ignore the others, which will not appear in output files. By default, check those in `astroquery.vo_conesearch.validator.conf.conesearch_urls`. If `None`, check everything.

**Raises**

**IOError**

Invalid destination directory.

**timeout**

URL request timed out.

**ValidationMultiprocessingError**

Multiprocessing failed.

### 29.6.7 astroquery.vo_conesearch.validator.inspect Module

Inspect results from `astroquery.vo_conesearch.validator.validate()`.

#### Classes

| | |
|---|---|
| `ConeSearchResults`([cache, verbose]) | A class to store Cone Search validation results. |

#### ConeSearchResults

**class** astroquery.vo_conesearch.validator.inspect.**ConeSearchResults**(*cache=False*, *verbose=True*)

Bases: `object`

A class to store Cone Search validation results.

**Parameters**

**cache** : bool

Read from cache, if available. Default is False to ensure the latest data are read.

**verbose** : bool

Show download progress bars.

### Attributes

| | |
|---|---|
| **dbtypes** | (list) Cone Search database identifiers. |
| **dbs** | (dict) Stores VOSDatabase for each dbtypes. |
| **catkeys** | (dict) Stores sorted catalog keys for each dbtypes. |

### Methods Summary

| | |
|---|---|
| list_cats(typ[, fout, ignore_noncrit]) | List catalogs in given database. |
| print_cat(key[, fout]) | Display a single catalog of given key. |
| tally([fout]) | Tally databases. |

### Methods Documentation

**list_cats**(*typ*, *fout=None*, *ignore_noncrit=False*)

List catalogs in given database.

Listing contains:

1. Catalog key
2. Cone search access URL
3. Warning codes
4. Warning descriptions

**Parameters**

**typ** : str

Any value in self.dbtypes.

**fout** : output stream

Default is screen output.

**ignore_noncrit** : bool

Exclude warnings in astroquery.vo_conesearch.validator.conf.noncritical_warnings. This is useful to see why a catalog failed validation.

**print_cat**(*key*, *fout=None*)

Display a single catalog of given key.

If not found, nothing is written out.

**Parameters**

**key** : str

Catalog key.

**fout** : output stream

Default is screen output.

**tally**(*fout=None*)

Tally databases.

**Parameters**

**fout** : output stream

Default is screen output.

### 29.6.8 astroquery.vo_conesearch.validator.exceptions Module

Exceptions related to Virtual Observatory (VO) validation.

#### Classes

| | |
|---|---|
| BaseVOValidationError | Base class for VO validation exceptions. |
| ValidationMultiprocessingError | Validation using multiprocessing failed. |

#### BaseVOValidationError

**exception** astroquery.vo_conesearch.validator.exceptions.**BaseVOValidationError**

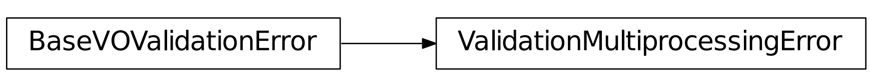
Base class for VO validation exceptions.

#### ValidationMultiprocessingError

**exception** astroquery.vo_conesearch.validator.exceptions.**ValidationMultiprocessingError**

Validation using multiprocessing failed.

#### Class Inheritance Diagram



# CHAPTER 30

## MAST Queries (`astroquery.mast`)

### 30.1 Getting Started

This module can be used to query the Barbara A. Mikulski Archive for Space Telescopes (MAST). Below are examples of the types of queries that can be used, and how to access data products.

#### 30.1.1 Positional Queries

Positional queries can be based on a sky position or a target name. The observation fields are documented here.

```
>>> from astroquery.mast import Observations
>>> obsTable = Observations.query_region("322.49324 12.16683")
>>> print(obsTable[:10])

dataproduct_type obs_collection instrument_name ... distance
---------------- -------------- --------------- ... --------
            cube          SWIFT            UVOT ...      0.0
            cube          SWIFT            UVOT ...      0.0
            cube          SWIFT            UVOT ...      0.0
            cube          SWIFT            UVOT ...      0.0
            cube          SWIFT            UVOT ...      0.0
            cube          SWIFT            UVOT ...      0.0
            cube          SWIFT            UVOT ...      0.0
            cube          SWIFT            UVOT ...      0.0
            cube          SWIFT            UVOT ...      0.0
            cube          SWIFT            UVOT ...      0.0
```

Radius is an optional parameter and the default is 0.2 degrees.

```
>>> from astroquery.mast import Observations
>>> observations = Observations.query_object("M8",radius=".02 deg")
>>> print(observations[:10])
```

(continues on next page)

(continued from previous page)

```
dataproduct_type obs_collection instrument_name ...    distance
---------------- -------------- --------------- ... -------------
            cube             K2          Kepler ... 39.4914065162
        spectrum            IUE             LWP ...           0.0
        spectrum            IUE             LWP ...           0.0
        spectrum            IUE             LWP ...           0.0
        spectrum            IUE             LWR ...           0.0
        spectrum            IUE             LWR ...           0.0
        spectrum            IUE             LWR ...           0.0
        spectrum            IUE             LWR ...           0.0
        spectrum            IUE             LWR ...           0.0
        spectrum            IUE             LWR ...           0.0
```

### 30.1.2 Observation Criteria Queries

To search for observations based on parameters other than position or target name, use `query_criteria`. Criteria are supplied as keyword arguments, where valid criteria are “coordinates”, “objectname”, “radius” (as in `query_region` and `query_object`), and all observation fields listed here.

Argument values are one or more acceptable values for the criterion, except for fields with a float datatype where the argument should be in the form [minVal, maxVal]. For non-float type criteria, wildcards (both * and %) may be used. However, only one wildcarded value can be processed per criterion.

RA and Dec must be given in decimal degrees, and datetimes in MJD.

```
>>> from astroquery.mast import Observations
>>> obsTable = Observations.query_criteria(dataproduct_type=["image"],
                                           proposal_pi="Osten",
                                           s_dec=[43.5,45.5])
>>> print(obsTable)

dataproduct_type calib_level obs_collection ... dataURL   obsid      objID
---------------- ----------- -------------- ... ------- ---------- ----------
           image           1            HST ...    None 2003520266 2011133418
           image           1            HST ...    None 2003520267 2011133419
           image           1            HST ...    None 2003520268 2011133420

>>> obsTable = Observations.query_criteria(filters=["*UV","Kepler"],objectname="M101")
>>> print(obsTable)

dataproduct_type calib_level obs_collection ...   objID1      distance
---------------- ----------- -------------- ... ---------- -------------
           image           2          GALEX ... 1000055044           0.0
           image           2          GALEX ... 1000004937 3.83290685323
           image           2          GALEX ... 1000045953 371.718371962
           image           2          GALEX ... 1000055047 229.810616011
           image           2          GALEX ... 1000016644 229.810616011
           image           2          GALEX ... 1000045952           0.0
           image           2          GALEX ... 1000048357           0.0
           image           2          GALEX ... 1000001326           0.0
           image           2          GALEX ... 1000001327 371.718371962
           image           2          GALEX ... 1000004203           0.0
           image           2          GALEX ... 1000016641           0.0
           image           2          GALEX ... 1000048943 3.83290685323
```

### 30.1.3 Getting Observation Counts

To get the number of observations and not the observations themselves, query_counts functions are available. This can be useful if trying to decide whether the available memory is sufficient for the number of observations.

```
>>> from astroquery.mast import Observations
>>> print(Observations.query_region_count("322.49324 12.16683"))
1804

>>> print(Observations.query_object_count("M8",radius=".02 deg"))
196

>>> print(Observations.query_criteria_count(dataproduct_type="image",
                                            filters=["NUV","FUV"],
                                            t_max=[52264.4586,54452.8914]))
59033
```

### 30.1.4 Listing Available Missions

To list data missions archived by MAST and avaiable through `astroquery.mast`, use the `list_missions` function.

```
>>> from astroquery.mast import Observations
>>> print(Observations.list_missions())
['IUE', 'Kepler', 'K2FFI', 'EUVE', 'HLA', 'KeplerFFI','FUSE',
'K2', 'HST', 'WUPPE', 'BEFS', 'GALEX', 'TUES','HUT', 'SWIFT']
```

## 30.2 Downloading Data

### 30.2.1 Getting Product Lists

Each observation returned from a MAST query can have one or more associated data products. Given one or more observations or observation ids ("obsid") `get_product_list` will return a `Table` containing the associated data products. The product fields are documented here.

```
>>> from astroquery.mast import Observations
>>> obsTable = Observations.query_object("M8",radius=".02 deg")
>>> dataProductsByObservation = Observations.get_product_list(obsTable[0:2])
>>> print(dataProductsByObservation)

  obsID    obs_collection ...          productFilename           size
---------- -------------- ... --------------------------------- --------
3000007760            IUE ...                lwp13058.elbll.gz   185727
3000007760            IUE ...                lwp13058.elbls.gz   183350
3000007760            IUE ...                 lwp13058.lilo.gz   612715
3000007760            IUE ...                lwp13058.melol.gz    12416
3000007760            IUE ...                lwp13058.melos.gz    12064
3000007760            IUE ...                  lwp13058.raw.gz   410846
3000007760            IUE ...                 lwp13058.rilo.gz   416435
3000007760            IUE ...                 lwp13058.silo.gz   100682
3000007760            IUE ...                     lwp13058.gif     8971
3000007760            IUE ...                 lwp13058.mxlo.gz    18206
3000007760            IUE ...              lwp13058mxlo_vo.fits    48960
3000007760            IUE ...                     lwp13058.gif     3967
```

(continues on next page)

(continued from previous page)

```
9500243833             K2 ...     k2-tpf-only-target_bw_large.png     9009
9500243833             K2 ... ktwo200071160-c91_lpd-targ.fits.gz 39930404
9500243833             K2 ... ktwo200071160-c92_lpd-targ.fits.gz 62213068
9500243833             K2 ...     k2-tpf-only-target_bw_thumb.png     1301

>>> obsids = obsTable[0:2]['obsid']
>>> dataProductsByID = Observations.get_product_list(obsids)
>>> print(dataProductsByID)

  obsID    obs_collection ...          productFilename            size
---------- -------------- ... ---------------------------------- --------
3000007760            IUE ...                  lwp13058.elbll.gz   185727
3000007760            IUE ...                  lwp13058.elbls.gz   183350
3000007760            IUE ...                   lwp13058.lilo.gz   612715
3000007760            IUE ...                  lwp13058.melol.gz    12416
3000007760            IUE ...                  lwp13058.melos.gz    12064
3000007760            IUE ...                    lwp13058.raw.gz   410846
3000007760            IUE ...                   lwp13058.rilo.gz   416435
3000007760            IUE ...                   lwp13058.silo.gz   100682
3000007760            IUE ...                       lwp13058.gif     8971
3000007760            IUE ...                   lwp13058.mxlo.gz    18206
3000007760            IUE ...               lwp13058mxlo_vo.fits    48960
3000007760            IUE ...                       lwp13058.gif     3967
9500243833             K2 ...     k2-tpf-only-target_bw_large.png     9009
9500243833             K2 ... ktwo200071160-c91_lpd-targ.fits.gz 39930404
9500243833             K2 ... ktwo200071160-c92_lpd-targ.fits.gz 62213068
9500243833             K2 ...     k2-tpf-only-target_bw_thumb.png     1301

>>> print((dataProductsByObservation == dataProductsByID).all())
True
```

### 30.2.2 Downloading Data Products

Products can be downloaded by using `download_products`, with a `Table` of data products, or a list (or single) obsid as the argument. **By default only Minimum Recommended Products will be downloaded.**

```
>>> from astroquery.mast import Observations
>>> obsid = '3000007760'
>>> dataProductsByID = Observations.get_product_list(obsid)
>>> manifest = Observations.download_products(dataProductsByID)
Downloading URL http://archive.stsci.edu/pub/iue/data/lwp/13000/lwp13058.mxlo.gz to ./mastDownload/IUE/
↪lwp13058/lwp13058.mxlo.gz ... [Done]
Downloading URL http://archive.stsci.edu/pub/vospectra/iue2/lwp13058mxlo_vo.fits to ./mastDownload/IUE/
↪lwp13058/lwp13058mxlo_vo.fits ... [Done]
>>> print(manifest)

                   Local Path                     Status  Message URL
------------------------------------------------ -------- ------- ----
    ./mastDownload/IUE/lwp13058/lwp13058.mxlo.gz COMPLETE    None None
./mastDownload/IUE/lwp13058/lwp13058mxlo_vo.fits COMPLETE    None None
```

As an alternative to downloading the data files now, the curl_flag can be used instead to instead get a curl script that can be used to download the files at a later time.

```
>>> from astroquery.mast import Observations
>>> Observations.download_products('2003839997',
                                   mrp_only=False,
                                   productType="SCIENCE",
                                   curl_flag=True)

Downloading URL https://mast.stsci.edu/portal/Download/stage/anonymous/public/514cfaa9-fdc1-4799-b043-
↪4488b811db4f/mastDownload_20170629162916.sh to ./mastDownload_20170629162916.sh ... [Done]
```

### Filtering

Filter keyword arguments can be applied to download only data products that meet the given criteria. Available filters are "mrp_only" (Minium Recommended Products), "extension" (file extension), and all products fields listed here.

**Important: mrp_only defaults to True.**

The 'AND' operation is performed for a list of filters, and the 'OR' operation is performed within a filter set. The below example illustrates downloading all product files with the extension "fits" that are either "RAW" or "UNCAL."

```
>>> from astroquery.mast import Observations
>>> Observations.download_products('2003839997',
                                   mrp_only=False,
                                   productSubGroupDescription=["RAW", "UNCAL"],
                                   extension="fits")
Downloading URL https://mast.stsci.edu/api/v0/download/file/HST/product/ib3p11p7q_raw.fits to ./
↪mastDownload/HST/IB3P11P7Q/ib3p11p7q_raw.fits ... [Done]
Downloading URL https://mast.stsci.edu/api/v0/download/file/HST/product/ib3p11p8q_raw.fits to ./
↪mastDownload/HST/IB3P11P8Q/ib3p11p8q_raw.fits ... [Done]
Downloading URL https://mast.stsci.edu/api/v0/download/file/HST/product/ib3p11phq_raw.fits to ./
↪mastDownload/HST/IB3P11PHQ/ib3p11phq_raw.fits ... [Done]
Downloading URL https://mast.stsci.edu/api/v0/download/file/HST/product/ib3p11q9q_raw.fits to ./
↪mastDownload/HST/IB3P11Q9Q/ib3p11q9q_raw.fits ... [Done]
```

Product filtering can also be appllied directly to a table of products without proceeding to the download step.

```
>>> from astroquery.mast import Observations
>>> dataProductsByID = Observations.get_product_list('2003839997')
>>> print(len(dataProductsByID))
31

>>> dataProductsByID = Observations.filter_products(dataProductsByID,
                                                    mrp_only=False,
                                                    productSubGroupDescription=["RAW", "UNCAL"],
                                                    extenstion="fits")
>>> print(len(dataProductsByID))
4
```

## 30.3 Accessing Proprietary Data

To access data that is not publicly available users may log into their MyST Account. This can be done by using the `login` function, or by initializing a class instance with a username/password. If a password is not supplied, the user will be prompted to enter one.

```
>>> from astroquery.mast import Observations
>>> Observations.login(username="testUser@stsci.edu",password="testPwd")

Authentication successful!
Session Expiration: 600 minute(s)

>>> sessionInfo = Observations.session_info()

Session Expiration: 559.0 min
Username: testUser@stsci.edu
First Name: Test
Last Name: User
```

```
>>> from astroquery.mast import Observations
>>> mySession = Observations(username="testUser@stsci.edu",password="testPwd")

Authentication successful!
Session Expiration: 600 minute(s)

>>> sessionInfo = mySession.session_info()

Session Expiration: 559.0 min
Username: testUser@stsci.edu
First Name: Test
Last Name: User
```

* For security passwords should not be typed into a terminal or Jupyter notebook but instead input using a more secure method such as `getpass`.

MAST login can also be achieved with a "session token," from an existing valid MAST session.

```
>>> from astroquery.mast import Observations
>>> from astroquery.mast import Mast
>>> myObsSession = Observations(username="testUser@stsci.edu",password="testPwd")

Authentication successful!
Session Expiration: 600 minute(s)

>>> myToken = myObsSession.get_token()
>>> Mast.login(session_token=myToken)

Authentication successful!
Session Expiration: 599 minute(s)
```

MAST sessions expire after 600 minutes, at which point the user must login again. The `store_password` argument can be used to store the username and password securely in the user's keyring. If the username/password are thus stored, only the username need be entered to login. This password can be overwritten using the `reenter_password` argument. To logout before a session expires, the `logout` method may be used.

## 30.4 Direct Mast Queries

The Mast class provides more direct access to the MAST interface. It requires more knowledge of the inner workings of the MAST API, and should be rarely needed. However in the case of new functionality not yet implemented in astroquery, this class does allow access. See the MAST api documentation for more information.

The basic MAST query function returns query results as an `Table`.

```
>>> from astroquery.mast import Mast
>>> service = 'Mast.Caom.Cone'
>>> params = {'ra':184.3,
              'dec':54.5,
              'radius':0.2}

>>> observations = Mast.mashup_request(service, params)
>>> print(observations)

dataproduct_type obs_collection instrument_name ...    distance   _selected_
---------------- -------------- --------------- ... ------------- ----------
           image          GALEX           GALEX ...           0.0      False
           image          GALEX           GALEX ...           0.0      False
           image          GALEX           GALEX ...           0.0      False
           image          GALEX           GALEX ...           0.0      False
           image          GALEX           GALEX ...           0.0      False
           image          GALEX           GALEX ... 302.405835798      False
           image          GALEX           GALEX ... 302.405835798      False
```

If the output is not the MAST json result type it cannot be properly parsed into a `Table`. In this case, the async method should be used to get the raw http response, which can then be manually parsed.

```
>>> from astroquery.mast import Mast
>>> service = 'Mast.Name.Lookup'
>>> params ={'input':"M8",
            'format':'json'}

>>> response = Mast.mashup_request_async(service,params)
>>> result = response[0].json()
>>> print(result)

{'resolvedCoordinate': [{'cacheDate': 'Apr 12, 2017 9:28:24 PM',
                         'cached': True,
                         'canonicalName': 'MESSIER 008',
                         'decl': -24.38017,
                         'objectType': 'Neb',
                         'ra': 270.92194,
                         'resolver': 'NED',
                         'resolverTime': 113,
                         'searchRadius': -1.0,
                         'searchString': 'm8'}],
 'status': ''}
```

## 30.5 Reference/API

### 30.5.1 astroquery.mast Package

#### MAST Query Tool

This module contains various methods for querying the MAST Portal.

#### Classes

| | |
|---|---|
| ObservationsClass([username, password, . . . ]) | MAST Observations query class. |
| MastClass([username, password, session_token]) | MAST query class. |
| Conf | Configuration parameters for astroquery.mast. |

### ObservationsClass

**class** astroquery.mast.**ObservationsClass**(*username=None*, *password=None*, *session_token=None*)

Bases: astroquery.mast.MastClass

MAST Observations query class.

Class for querying MAST observational data.

#### Methods Summary

| | |
|---|---|
| download_products(products[, download_dir, . . . ]) | Download data products. |
| filter_products(products[, mrp_only]) | Takes an astropy.table.Table of MAST observation data products and filters it based on given filters. |
| get_product_list(*args, **kwargs) | Queries the service and returns a table object. |
| get_product_list_async(observations) | Given a "Product Group Id" (column name obsid) returns a list of associated data products. |
| list_missions() | Lists data missions archived by MAST and avaiable through astroquery.mast. |
| query_criteria(*args, **kwargs) | Queries the service and returns a table object. |
| query_criteria_async([pagesize, page]) | Given an set of filters, returns a list of MAST observations. |
| query_criteria_count([pagesize, page]) | Given an set of filters, returns the number of MAST observations meeting those criteria. |
| query_object(*args, **kwargs) | Queries the service and returns a table object. |
| query_object_async(objectname[, radius, . . . ]) | Given an object name, returns a list of MAST observations. |
| query_object_count(objectname[, radius, . . . ]) | Given an object name, returns the number of MAST observations. |
| query_region(*args, **kwargs) | Queries the service and returns a table object. |
| query_region_async(coordinates[, radius, . . . ]) | Given a sky position and radius, returns a list of MAST observations. |
| query_region_count(coordinates[, radius, . . . ]) | Given a sky position and radius, returns the number of MAST observations in that region. |

#### Methods Documentation

**download_products**(*products*, *download_dir=None*, *cache=True*, *curl_flag=False*, *mrp_only=True*, ***filters*)

Download data products.

**Parameters**

**products** : str, list, astropy.table.Table

Either a single or list of obsids (as can be given to get_product_list), or a Table of products (as is returned by get_product_list)

**download_dir** : str, optional

Optional. Directory to download files to. Defaults to current directory.

**cache** : bool, optional

Default is True. If file is found on disc it will not be downloaded again. Note: has no affect when downloading curl script.

**curl_flag** : bool, optional

Default is False. If true instead of downloading files directly, a curl script will be downloaded that can be used to download the data files at a later time.

**mrp_only** : bool, optional

Default True. When set to true only "Minimum Recommended Products" will be returned.

****filters :**

Filters to be applied. Valid filters are all products fields listed here and 'extension' which is the desired file extension. The Column Name (or 'extension') is the keyword, with the argument being one or more acceptable values for that parameter. Filter behavior is AND between the filters and OR within a filter set. For example: productType="SCIENCE",extension=["fits","jpg"]

**Returns**

**response** : `Table`

The manifest of files downloaded, or status of files on disk if curl option chosen.

**filter_products**(*products*, *mrp_only=True*, ***filters*)

Takes an `astropy.table.Table` of MAST observation data products and filters it based on given filters.

**Parameters**

**products** : `astropy.table.Table`

Table containing data products to be filtered.

**mrp_only** : bool, optional

Default True. When set to true only "Minimum Recommended Products" will be returned.

****filters :**

Filters to be applied. Valid filters are all products fields listed here and 'extension' which is the desired file extension. The Column Name (or 'extension') is the keyword, with the argument being one or more acceptable values for that parameter. Filter behavior is AND between the filters and OR within a filter set. For example: productType="SCIENCE",extension=["fits","jpg"]

**Returns**

**response** : `Table`

**get_product_list**(**args*, ***kwargs*)

Queries the service and returns a table object.

Given a "Product Group Id" (column name obsid) returns a list of associated data products. See column documentation here.

**Parameters**

**observations** : str or `astropy.table.Row` or list/Table of same

Row/Table of MAST query results (e.g. output from `query_object`) or single/list of MAST Product Group Id(s) (obsid). See description here.

**Returns**

**table** : A `Table` object.

`get_product_list_async`(*observations*)

Given a "Product Group Id" (column name obsid) returns a list of associated data products. See column documentation here.

**Parameters**

**observations** : str or `astropy.table.Row` or list/Table of same

Row/Table of MAST query results (e.g. output from `query_object`) or single/list of MAST Product Group Id(s) (obsid). See description here.

**Returns**

**response** : list(`requests.Response`)

`list_missions`()

Lists data missions archived by MAST and avaiable through `astroquery.mast`.

**Returns**

**response** : list

List of available missions.

`query_criteria`(**args*, ***kwargs*)

Queries the service and returns a table object.

Given an set of filters, returns a list of MAST observations. See column documentation here.

**Parameters**

**pagesize** : int, optional

Can be used to override the default pagesize. E.g. when using a slow internet connection.

**page** : int, optional

Can be used to override the default behavior of all results being returned to obtain one sepcific page of results.

****criteria**

Criteria to apply. At least one non-positional criteria must be supplied. Valid criteria are coordinates, objectname, radius (as in `query_region` and `query_object`), and all observation fields listed here. The Column Name is the keyword, with the argument being one or more acceptable values for that parameter, except for fields with a float datatype where the argument should be in the form [minVal, maxVal]. For non-float type criteria wildcards maybe used (both * and % are considered wildcards), however only one wildcarded value can be processed per criterion. RA and Dec must be given in decimal degrees, and datetimes in MJD. For example: filters=["FUV","NUV"],proposal_pi="Ost*",t_max=[52264.4586,54452.8914]

**Returns**

**table** : A `Table` object.

`query_criteria_async`(*pagesize=None*, *page=None*, ***criteria*)

Given an set of filters, returns a list of MAST observations. See column documentation here.

**Parameters**

**pagesize** : int, optional

Can be used to override the default pagesize. E.g. when using a slow internet connection.

**page** : int, optional

Can be used to override the default behavior of all results being returned to obtain one sepcific page of results.

****criteria**

Criteria to apply. At least one non-positional criteria must be supplied. Valid criteria are coordinates, objectname, radius (as in `query_region` and `query_object`), and all observation fields listed here. The Column Name is the keyword, with the argument being one or more acceptable values for that parameter, except for fields with a float datatype where the argument should be in the form [minVal, maxVal]. For non-float type criteria wildcards maybe used (both * and % are considered wildcards), however only one wildcarded value can be processed per criterion. RA and Dec must be given in decimal degrees, and datetimes in MJD. For example: filters=["FUV","NUV"],proposal_pi="Ost*",t_max=[52264.4586,54452.8914]

**Returns**

**response** : list(`requests.Response`)

**query_criteria_count**(*pagesize=None*, *page=None*, ***criteria*)

Given an set of filters, returns the number of MAST observations meeting those criteria.

**Parameters**

**pagesize** : int, optional

Can be used to override the default pagesize. E.g. when using a slow internet connection.

**page** : int, optional

Can be used to override the default behavior of all results being returned to obtain one sepcific page of results.

****criteria**

Criteria to apply. At least one non-positional criterion must be supplied. Valid criteria are coordinates, objectname, radius (as in `query_region` and `query_object`), and all observation fields listed here. The Column Name is the keyword, with the argument being one or more acceptable values for that parameter, except for fields with a float datatype where the argument should be in the form [minVal, maxVal]. For non-float type criteria wildcards maybe used (both * and % are considered wildcards), however only one wildcarded value can be processed per criterion. RA and Dec must be given in decimal degrees, and datetimes in MJD. For example: filters=["FUV","NUV"],proposal_pi="Ost*",t_max=[52264.4586,54452.8914]

**Returns**

**response** : int

**query_object**(**args*, ***kwargs*)

Queries the service and returns a table object.

Given an object name, returns a list of MAST observations. See column documentation here.

**Parameters**

**objectname** : str

The name of the target around which to search.

**radius** : str or `Quantity` object, optional

Default 0.2 degrees. The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 0.2 deg.

**pagesize** : int, optional

Default None. Can be used to override the default pagesize for (set in configs) this query only. E.g. when using a slow internet connection.

**page** : int, optional

Defaulte None. Can be used to override the default behavior of all results being returned to obtain a specific page of results.

**Returns**

**table** : A `Table` object.

**query_object_async**(*objectname*, *radius=<Quantity 0.2 deg>*, *pagesize=None*, *page=None*)

Given an object name, returns a list of MAST observations. See column documentation here.

**Parameters**

**objectname** : str

The name of the target around which to search.

**radius** : str or `Quantity` object, optional

Default 0.2 degrees. The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 0.2 deg.

**pagesize** : int, optional

Default None. Can be used to override the default pagesize for (set in configs) this query only. E.g. when using a slow internet connection.

**page** : int, optional

Defaulte None. Can be used to override the default behavior of all results being returned to obtain a specific page of results.

**Returns**

**response** : list of `requests.Response`

**query_object_count**(*objectname*, *radius=<Quantity 0.2 deg>*, *pagesize=None*, *page=None*)

Given an object name, returns the number of MAST observations.

**Parameters**

**objectname** : str

The name of the target around which to search.

**radius** : str or `Quantity` object, optional

The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 0.2 deg.

**pagesize** : int, optional

Can be used to override the default pagesize. E.g. when using a slow internet connection.

**page** : int, optional

Can be used to override the default behavior of all results being returned to obtain one sepcific page of results.

**Returns**

**response** : int

**query_region**(**args*, ***kwargs*)

Queries the service and returns a table object.

Given a sky position and radius, returns a list of MAST observations. See column documentation here.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string or as the appropriate `astropy.coordinates` object.

**radius** : str or `Quantity` object, optional

Default 0.2 degrees. The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 0.2 deg.

**pagesize** : int, optional

Default None. Can be used to override the default pagesize for (set in configs) this query only. E.g. when using a slow internet connection.

**page** : int, optional

Default None. Can be used to override the default behavior of all results being returned to obtain a specific page of results.

**Returns**

**table** : A `Table` object.

**query_region_async**(*coordinates*, *radius=<Quantity 0.2 deg>*, *pagesize=None*, *page=None*)

Given a sky position and radius, returns a list of MAST observations. See column documentation here.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string or as the appropriate `astropy.coordinates` object.

**radius** : str or `Quantity` object, optional

Default 0.2 degrees. The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 0.2 deg.

**pagesize** : int, optional

Default None. Can be used to override the default pagesize for (set in configs) this query only. E.g. when using a slow internet connection.

**page** : int, optional

Default None. Can be used to override the default behavior of all results being returned to obtain a specific page of results.

**Returns**

**response** : list of `requests.Response`

**query_region_count**(*coordinates*, *radius=<Quantity 0.2 deg>*, *pagesize=None*, *page=None*)

Given a sky position and radius, returns the number of MAST observations in that region.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string or as the appropriate `astropy.coordinates` object.

**radius** : str or `Quantity` object, optional

The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 0.2 deg.

**pagesize** : int, optional

Can be used to override the default pagesize for. E.g. when using a slow internet connection.

**page** : int, optional

Can be used to override the default behavior of all results being returned to obtain a specific page of results.

**Returns**

**response** : int

## MastClass

**class** astroquery.mast.**MastClass**(*username=None*, *password=None*, *session_token=None*)

Bases: `astroquery.query.QueryWithLogin`

MAST query class.

Class that allows direct programatic access to the MAST Portal, more flexible but less user friendly than `ObservationsClass`.

### Methods Summary

| | |
|---|---|
| `get_token`() | Returns MAST session cookie. |
| `login`([username, password, session_token, . . . ]) | Log into the MAST portal. |
| `logout`() | Log out of current MAST session. |
| `service_request`(*args, **kwargs) | Queries the service and returns a table object. |
| `service_request_async`(service, params[, . . . ]) | Given a Mashup service and parameters, builds and excecutes a Mashup query. |
| `session_info`([silent]) | Displays information about current MAST session, and returns session info dictionary. |

### Methods Documentation

**get_token**()

Returns MAST session cookie.

**Returns**

**response** : `Cookie`

**login**(*username=None*, *password=None*, *session_token=None*, *store_password=False*, *reenter_password=False*)

Log into the MAST portal.

**Parameters**

**username** : string, optional

Default is None. The username for the user logging in. Usually this will be the user's email address. If a username is necessary but not supplied it will be prompted for.

**password** : string, optional

Default is None. The password associated with the given username. For security passwords should not be typed into the terminal or jupyter notebook, but input using a more secure method such as `getpass`. If a password is necessary but not supplied it will be prompted for.

**session_token** : dict or `Cookie`, optional

A valid MAST session cookie that will be attached to the current session in lieu of logging in with a username/password. If username and/or password is supplied, this argument will be ignored.

**store_password** : bool, optional

Default False. If true, username and password will be stored securely in your keyring.

**reenter_password** : bool, optional

Default False. Asks for the password even if it is already stored in the keyring. This is the way to overwrite an already stored password on the keyring.

**logout**()

Log out of current MAST session.

**service_request**(**args*, ***kwargs*)

Queries the service and returns a table object.

Given a Mashup service and parameters, builds and excecutes a Mashup query. See documentation here for information about how to build a Mashup request.

**Parameters**

**service** : str

The Mashup service to query.

**params** : dict

JSON object containing service parameters.

**pagesize** : int, optional

Default None. Can be used to override the default pagesize (set in configs) for this query only. E.g. when using a slow internet connection.

**page** : int, optional

Default None. Can be used to override the default behavior of all results being returned to obtain a specific page of results.

****kwargs :**

See MashupRequest properties here for additional keyword arguments.

**Returns**

**table** : A `Table` object.

**service_request_async**(*service*, *params*, *pagesize=None*, *page=None*, ***kwargs*)

Given a Mashup service and parameters, builds and excecutes a Mashup query. See documentation here for information about how to build a Mashup request.

**Parameters**

**service** : str

The Mashup service to query.

**params** : dict

JSON object containing service parameters.

**pagesize** : int, optional

Default None. Can be used to override the default pagesize (set in configs) for this query only. E.g. when using a slow internet connection.

**page** : int, optional

Default None. Can be used to override the default behavior of all results being returned to obtain a specific page of results.

****kwargs :**

See MashupRequest properties here for additional keyword arguments.

**Returns**

**response** : list of `requests.Response`

`session_info`(*silent=False*)

Displays information about current MAST session, and returns session info dictionary.

**Parameters**

**silent** : bool, optional

Default False. Suppresses output to stdout.

**Returns**

**response** : dict

## Conf

**class** `astroquery.mast.Conf`

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.mast`.

### Attributes Summary

| | |
|---|---|
| `pagesize` | Number of results to request at once from the STScI server. |
| `server` | Name of the MAST server. |
| `ssoserver` | MAST SSO Portal server. |
| `timeout` | Time limit for requests from the STScI server. |

### Attributes Documentation

`pagesize`

Number of results to request at once from the STScI server.

`server`

Name of the MAST server.

**ssoserver**
MAST SSO Portal server.

**timeout**
Time limit for requests from the STScI server.

These others are functional, but do not follow a common & consistent API:

# CHAPTER 31

## Fermi Queries (astroquery.fermi)

## 31.1 Getting started

The following example illustrates a Fermi LAT query, centered on M 31 for the energy range 1 to 100 GeV for the first day in 2013.

```
>>> from astroquery import fermi
>>> result = fermi.FermiLAT.query_object('M31', energyrange_MeV='1000, 100000',
...                                       obsdates='2013-01-01 00:00:00, 2013-01-02 00:00:00')
>>> print(result)
['http://fermi.gsfc.nasa.gov/FTP/fermi/data/lat/queries/L1309041729388EB8D2B447_SC00.fits',
 'http://fermi.gsfc.nasa.gov/FTP/fermi/data/lat/queries/L1309041729388EB8D2B447_PH00.fits']
>>> from astropy.io import fits
>>> sc = fits.open('http://fermi.gsfc.nasa.gov/FTP/fermi/data/lat/queries/L1309041729388EB8D2B447_SC00.
↪fits')
```

## 31.2 Reference/API

### 31.2.1 astroquery.fermi Package

Access to Fermi Gamma-ray Space Telescope data.

http://fermi.gsfc.nasa.gov http://fermi.gsfc.nasa.gov/ssc/data/

#### Classes

| | |
|---|---|
| FermiLATClass() | TODO: document |

Continued on next page

Table 1 – continued from previous page

| | |
|---|---|
| GetFermilatDatafile | TODO: document TODO: Fail with useful failure messages on genuine failures (this doesn't need to be implemented as a class) |
| Conf | Configuration parameters for astroquery.fermi. |

## FermiLATClass

**class** astroquery.fermi.**FermiLATClass**

Bases: astroquery.query.BaseQuery

TODO: document

### Attributes Summary

| |
|---|
| TIMEOUT |
| request_url |
| result_url_re |

### Methods Summary

| | |
|---|---|
| query_object(*args, **kwargs) | Queries the service and returns a table object. |
| query_object_async(*args, **kwargs) | Query the FermiLAT database |

### Attributes Documentation

**TIMEOUT = 60**

**request_url = 'http://fermi.gsfc.nasa.gov/cgi-bin/ssc/LAT/LATDataQuery.cgi'**

**result_url_re = re.compile('The results of your query may be found at <a href="(http://fermi.gsfc.nasa.g**

### Methods Documentation

**query_object**(**args*, ***kwargs*)

Queries the service and returns a table object.

Query the FermiLAT database

**Returns**

**table** : A Table object.

**query_object_async**(**args*, ***kwargs*)

Query the FermiLAT database

**Returns**

**url** : str

The URL of the page with the results (still need to scrape this page to download the data: easy for wget)

### GetFermilatDatafile

**class** astroquery.fermi.**GetFermilatDatafile**

Bases: object

TODO: document TODO: Fail with useful failure messages on genuine failures (this doesn't need to be implemented as a class)

#### Attributes Summary

| |
|---|
| TIMEOUT |
| check_frequency |
| fitsfile_re |

#### Methods Summary

| | |
|---|---|
| `__call__`(result_url[, check_frequency, verbose]) | Call self as a function. |

#### Attributes Documentation

**TIMEOUT = 120**

**check_frequency = 1**

**fitsfile_re = re.compile('<a href="(.*?)">Available</a>')**

#### Methods Documentation

**`__call__`**(*result_url*, *check_frequency=1*, *verbose=False*)

Call self as a function.

### Conf

**class** astroquery.fermi.**Conf**

Bases: astropy.config.ConfigNamespace

Configuration parameters for astroquery.fermi.

#### Attributes Summary

| | |
|---|---|
| retrieval_timeout | Time limit for retrieving a data file once it has been located. |
| timeout | Time limit for connecting to Fermi server. |
| url | Fermi query URL. |

### Attributes Documentation

**retrieval_timeout**
Time limit for retrieving a data file once it has been located.

**timeout**
Time limit for connecting to Fermi server.

**url**
Fermi query URL.

CHAPTER 32

# SDSS Queries (astroquery.sdss)

## 32.1 Getting started

This example shows how to perform an object cross-ID with SDSS. We'll start with the position of a source found in another survey, and search within a 5 arcsecond radius for optical counterparts in SDSS. Note use of the keyword argument spectro, which requires matches to have spectroscopy, not just photometry:

```
>>> from astroquery.sdss import SDSS
>>> from astropy import coordinates as coords
>>> pos = coords.SkyCoord('0h8m05.63s +14d50m23.3s', frame='icrs')
>>> xid = SDSS.query_region(pos, spectro=True)
>>> print(xid)
    ra          dec            objid         run  rerun camcol field    z     plate  mjd  fiberID     ␣
↪specobjid      specClass
---------- ----------- ------------------ ---- ----- ------ ----- ------- ----- ----- ------- ----------
↪--------- ---------
2.02344483 14.83982059 587727221951234166 1739    40      3   315 0.04541   751 52251     160␣
↪211612124516974592         3
```

The result is an astropy.Table.

## 32.2 Downloading data

If we'd like to download spectra and/or images for our match, we have all the information we need in the elements of "xid" from the above example.

```
>>> sp = SDSS.get_spectra(matches=xid)
>>> im = SDSS.get_images(matches=xid, band='g')
```

The variables "sp" and "im" are lists of HDUList objects, one entry for each corresponding object in xid.

Note that in SDSS, image downloads retrieve the entire plate, so further processing will be required to excise an image centered around the point of interest (i.e. the object(s) returned by query_region).

## 32.3 Spectral templates

It is also possible to download spectral templates from SDSS. To see what is available, do

```
>>> from astroquery.sdss import SDSS
>>> print(SDSS.AVAILABLE_TEMPLATES)
```

Then, to download your favorite template, do something like

```
>>> template = SDSS.get_spectral_template('qso')
```

The variable "template" is a list of PyFITS HDUList objects (same object as "sp" in the above example). In this case there is only one result, but in a few cases there are multiple templates available to choose from (e.g. the "galaxy" spectral template will actually return 3 templates).

## 32.4 Reference/API

### 32.4.1 astroquery.sdss Package

#### SDSS Spectra/Image/SpectralTemplate Archive Query Tool

#### Classes

| | |
|---|---|
| `Conf` | Configuration parameters for `astroquery.sdss`. |
| `SDSSClass`() | |

#### Conf

**class** astroquery.sdss.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.sdss`.

##### Attributes Summary

| | |
|---|---|
| `sas_baseurl` | Base URL for downloading data products like spectra and images. |
| `skyserver_baseurl` | Base URL for catalog-related queries like SQL and Cross-ID. |
| `timeout` | Time limit for connecting to SDSS server. |

##### Attributes Documentation

**sas_baseurl**

Base URL for downloading data products like spectra and images.

**skyserver_baseurl**

Base URL for catalog-related queries like SQL and Cross-ID.

**timeout**
Time limit for connecting to SDSS server.

## SDSSClass

**class** astroquery.sdss.**SDSSClass**
Bases: `astroquery.query.BaseQuery`

### Attributes Summary

| | |
|---|---|
| AVAILABLE_TEMPLATES | |
| IMAGING_URL_SUFFIX | |
| QUERY_URL_SUFFIX_DR_10 | |
| QUERY_URL_SUFFIX_DR_NEW | |
| QUERY_URL_SUFFIX_DR_OLD | |
| SPECTRA_URL_SUFFIX | |
| TEMPLATES_URL | |
| TIMEOUT | |
| XID_URL_SUFFIX_NEW | |
| XID_URL_SUFFIX_OLD | |

### Methods Summary

| | |
|---|---|
| get_images([coordinates, radius, matches, ...]) | Download an image from SDSS. |
| get_images_async([coordinates, radius, ...]) | Download an image from SDSS. |
| get_spectra([coordinates, radius, matches, ...]) | Download spectrum from SDSS. |
| get_spectra_async([coordinates, radius, ...]) | Download spectrum from SDSS. |
| get_spectral_template([kind, timeout, ...]) | Download spectral templates from SDSS DR-2. |
| get_spectral_template_async([kind, timeout, ...]) | Download spectral templates from SDSS DR-2. |
| query_crossid(*args, **kwargs) | Queries the service and returns a table object. |
| query_crossid_async(coordinates[, ...]) | Query using the cross-identification web interface. |
| query_photoobj(*args, **kwargs) | Queries the service and returns a table object. |
| query_photoobj_async([run, rerun, camcol, ...]) | Used to query the PhotoObjAll table with run, rerun, camcol and field values. |
| query_region(*args, **kwargs) | Queries the service and returns a table object. |
| query_region_async(coordinates[, radius, ...]) | Used to query a region around given coordinates. |
| query_specobj(*args, **kwargs) | Queries the service and returns a table object. |
| query_specobj_async([plate, mjd, fiberID, ...]) | Used to query the SpecObjAll table with plate, mjd and fiberID values. |
| query_sql(*args, **kwargs) | Queries the service and returns a table object. |
| query_sql_async(sql_query[, timeout, ...]) | Query the SDSS database. |

### Attributes Documentation

**AVAILABLE_TEMPLATES = {'galaxy': [24, 25, 26], 'galaxy_early': 23, 'galaxy_late': 27, 'galaxy_lrg': 28,**

**IMAGING_URL_SUFFIX = '{base}/dr{dr}/boss/photoObj/frames/{rerun}/{run}/{camcol}/frame-{band}-{run:06d}-{**

**QUERY_URL_SUFFIX_DR_10 = '/dr{dr}/en/tools/search/x_sql.aspx'**

**QUERY_URL_SUFFIX_DR_NEW = '/dr{dr}/en/tools/search/x_results.aspx'**

**QUERY_URL_SUFFIX_DR_OLD = '/dr{dr}/en/tools/search/x_sql.asp'**

**SPECTRA_URL_SUFFIX = '{base}/dr{dr}/{instrument}/spectro/redux/{run2d}/spectra/{plate:04d}/spec-{plate:0**

**TEMPLATES_URL = 'http://classic.sdss.org/dr7/algorithms/spectemplates/spDR2'**

**TIMEOUT = 60**

**XID_URL_SUFFIX_NEW = '/dr{dr}/en/tools/search/X_Results.aspx'**

**XID_URL_SUFFIX_OLD = '/dr{dr}/en/tools/crossid/x_crossid.aspx'**

### Methods Documentation

**get_images**(*coordinates=None, radius=<Quantity 2. arcsec>, matches=None, run=None, rerun=301, camcol=None, field=None, band='g', timeout=60, cache=True, get_query_payload=False, data_release=12, show_progress=True*)

Download an image from SDSS.

Querying SDSS for images will return the entire plate. For subsequent analyses of individual objects

The query can be made with one the following groups of parameters (whichever comes first is used):

- `matches` (result of a call to `query_region`);
- `coordinates`, `radius`;
- `run`, `rerun`, `camcol`, `field`.

See below for examples.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 2 arcsec.

**matches** : `Table`

Result of `query_region`.

**run** : integer, optional

Length of a strip observed in a single continuous image observing scan.

**rerun** : integer, optional

Reprocessing of an imaging run. Defaults to 301 which is the most recent rerun.

**camcol** : integer, optional

Output of one camera column of CCDs.

**field** : integer, optional

Part of a camcol of size 2048 by 1489 pixels.

**band** : str, list

Could be individual band, or list of bands. Options: `'u'`, `'g'`, `'r'`, `'i'`, or `'z'`.

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**cache** : bool

Cache the images using astropy's caching system

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**list** : List of `HDUList` objects.

### Examples

Using results from a call to `query_region`:

```
>>> from astropy import coordinates as coords
>>> from astroquery.sdss import SDSS
>>> co = coords.SkyCoord('0h8m05.63s +14d50m23.3s')
>>> result = SDSS.query_region(co)
>>> imgs = SDSS.get_images(matches=result)
```

Using coordinates directly:

```
>>> imgs = SDSS.get_images(co)
```

Fetch the images from all runs with camcol 3 and field 164:

```
>>> imgs = SDSS.get_images(camcol=3, field=164)
```

Fetch only images from run 1904, camcol 3 and field 164:

```
>>> imgs = SDSS.get_images(run=1904, camcol=3, field=164)
```

**get_images_async**(*coordinates=None, radius=<Quantity 2. arcsec>, matches=None, run=None, rerun=301, camcol=None, field=None, band='g', timeout=60, get_query_payload=False, cache=True, data_release=12, show_progress=True*)

Download an image from SDSS.

Querying SDSS for images will return the entire plate. For subsequent analyses of individual objects

The query can be made with one the following groups of parameters (whichever comes first is used):

- `matches` (result of a call to `query_region`);
- `coordinates`, `radius`;
- `run`, `rerun`, `camcol`, `field`.

See below for examples.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 2 arcsec.

**matches** : `Table`

Result of `query_region`.

**run** : integer, optional

Length of a strip observed in a single continuous image observing scan.

**rerun** : integer, optional

Reprocessing of an imaging run. Defaults to 301 which is the most recent rerun.

**camcol** : integer, optional

Output of one camera column of CCDs.

**field** : integer, optional

Part of a camcol of size 2048 by 1489 pixels.

**band** : str, list

Could be individual band, or list of bands. Options: `'u'`, `'g'`, `'r'`, `'i'`, or `'z'`.

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**cache** : bool

Cache the images using astropy's caching system

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**list** : List of `HDUList` objects.

### Examples

Using results from a call to `query_region`:

```
>>> from astropy import coordinates as coords
>>> from astroquery.sdss import SDSS
>>> co = coords.SkyCoord('0h8m05.63s +14d50m23.3s')
>>> result = SDSS.query_region(co)
>>> imgs = SDSS.get_images(matches=result)
```

Using coordinates directly:

```
>>> imgs = SDSS.get_images(co)
```

Fetch the images from all runs with camcol 3 and field 164:

```
>>> imgs = SDSS.get_images(camcol=3, field=164)
```

Fetch only images from run 1904, camcol 3 and field 164:

```
>>> imgs = SDSS.get_images(run=1904, camcol=3, field=164)
```

**get_spectra**(*coordinates=None*, *radius=<Quantity 2. arcsec>*, *matches=None*, *plate=None*, *fiberID=None*, *mjd=None*, *timeout=60*, *cache=True*, *data_release=12*, *show_progress=True*)

Download spectrum from SDSS.

The query can be made with one the following groups of parameters (whichever comes first is used):

- `matches` (result of a call to `query_region`);
- `coordinates`, `radius`;
- `plate`, `mjd`, `fiberID`.

See below for examples.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 2 arcsec.

**matches** : `Table`

Result of `query_region`.

**plate** : integer, optional

Plate number.

**mjd** : integer, optional

Modified Julian Date indicating the date a given piece of SDSS data was taken.

**fiberID** : integer, optional

Fiber number.

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use. With the default server, this only supports DR8 or later.

**Returns**

**list** : List of `HDUList` objects.

### Examples

Using results from a call to `query_region`:

```
>>> from astropy import coordinates as coords
>>> from astroquery.sdss import SDSS
>>> co = coords.SkyCoord('0h8m05.63s +14d50m23.3s')
>>> result = SDSS.query_region(co, spectro=True)
>>> spec = SDSS.get_spectra(matches=result)
```

Using coordinates directly:

```
>>> spec = SDSS.get_spectra(co)
```

Fetch the spectra from all fibers on plate 751 with mjd 52251:

```
>>> specs = SDSS.get_spectra(plate=751, mjd=52251)
```

**get_spectra_async**(*coordinates=None*, *radius=<Quantity 2. arcsec>*, *matches=None*, *plate=None*, *fiberID=None*, *mjd=None*, *timeout=60*, *get_query_payload=False*, *data_release=12*, *cache=True*, *show_progress=True*)

Download spectrum from SDSS.

The query can be made with one the following groups of parameters (whichever comes first is used):

- `matches` (result of a call to `query_region`);
- `coordinates`, `radius`;
- `plate`, `mjd`, `fiberID`.

See below for examples.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 2 arcsec.

**matches** : `Table`

Result of `query_region`.

**plate** : integer, optional

Plate number.

**mjd** : integer, optional

Modified Julian Date indicating the date a given piece of SDSS data was taken.

**fiberID** : integer, optional

Fiber number.

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use. With the default server, this only supports DR8 or later.

**Returns**

**list** : list

A list of context-managers that yield readable file-like objects. The function returns the spectra for only one of `matches`, or `coordinates` and `radius`, or `plate`, `mjd` and `fiberID`.

### Examples

Using results from a call to `query_region`:

```
>>> from astropy import coordinates as coords
>>> from astroquery.sdss import SDSS
>>> co = coords.SkyCoord('0h8m05.63s +14d50m23.3s')
>>> result = SDSS.query_region(co, spectro=True)
>>> spec = SDSS.get_spectra(matches=result)
```

Using coordinates directly:

```
>>> spec = SDSS.get_spectra(co)
```

Fetch the spectra from all fibers on plate 751 with mjd 52251:

```
>>> specs = SDSS.get_spectra(plate=751, mjd=52251)
```

**get_spectral_template**(*kind='qso', timeout=60, show_progress=True*)

Download spectral templates from SDSS DR-2.

Location: http://www.sdss.org/dr7/algorithms/spectemplates/

There 32 spectral templates available from DR-2, from stellar spectra, to galaxies, to quasars. To see the available templates, do:

from astroquery.sdss import SDSS print SDSS.AVAILABLE_TEMPLATES

**Parameters**

**kind** : str, list

Which spectral template to download? Options are stored in the dictionary astroquery.sdss.SDSS.AVAILABLE_TEMPLATES

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**Returns**

**list** : List of `HDUList` objects.

### Examples

```
>>> qso = SDSS.get_spectral_template(kind='qso')
>>> Astar = SDSS.get_spectral_template(kind='star_A')
>>> Fstar = SDSS.get_spectral_template(kind='star_F')
```

**get_spectral_template_async**(*kind='qso'*, *timeout=60*, *show_progress=True*)

Download spectral templates from SDSS DR-2.

Location: http://www.sdss.org/dr7/algorithms/spectemplates/

There 32 spectral templates available from DR-2, from stellar spectra, to galaxies, to quasars. To see the available templates, do:

from astroquery.sdss import SDSS print SDSS.AVAILABLE_TEMPLATES

**Parameters**

**kind** : str, list

Which spectral template to download? Options are stored in the dictionary astroquery.sdss.SDSS.AVAILABLE_TEMPLATES

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**Returns**

**list** : List of `HDUList` objects.

### Examples

```
>>> qso = SDSS.get_spectral_template(kind='qso')
>>> Astar = SDSS.get_spectral_template(kind='star_A')
>>> Fstar = SDSS.get_spectral_template(kind='star_F')
```

**query_crossid**(**args*, ***kwargs*)

Queries the service and returns a table object.

Query using the cross-identification web interface.

**Parameters**

**coordinates** : str or `astropy.coordinates` object or list of

coordinates or `Column` of coordinates The target(s) around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

Example: ra = np.array([220.064728084,220.064728467,220.06473483]) dec = np.array([0.870131920218,0.87013210119,0.870138329659]) coordinates = SkyCoord(ra, dec, frame='icrs', unit='deg')

**radius** : str or `Quantity` object, optional The

string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 2 arcsec.

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**photoobj_fields** : list, optional

PhotoObj quantities to return. If photoobj_fields is None and specobj_fields is None then the value of fields is used

**specobj_fields** : list, optional

SpecObj quantities to return. If photoobj_fields is None and specobj_fields is None then the value of fields is used

**obj_names** : str, or list or `Column`, optional

Target names. If given, every coordinate should have a corresponding name, and it gets repeated in the query result. It generates unique object names by default.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**table** : A `Table` object.

**query_crossid_async**(*coordinates*, *obj_names=None*, *photoobj_fields=None*, *specobj_fields=None*, *get_query_payload=False*, *timeout=60*, *radius=<Quantity 5. arcsec>*, *data_release=12*, *cache=True*)

Query using the cross-identification web interface.

**Parameters**

**coordinates** : str or `astropy.coordinates` object or list of

coordinates or `Column` of coordinates The target(s) around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

Example: ra = np.array([220.064728084,220.064728467,220.06473483]) dec = np.array([0.870131920218,0.87013210119,0.870138329659]) coordinates = SkyCoord(ra, dec, frame='icrs', unit='deg')

**radius** : str or `Quantity` object, optional The

string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 2 arcsec.

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**photoobj_fields** : list, optional

PhotoObj quantities to return. If photoobj_fields is None and specobj_fields is None then the value of fields is used

**specobj_fields** : list, optional

SpecObj quantities to return. If photoobj_fields is None and specobj_fields is None then the value of fields is used

**obj_names** : str, or list or `Column`, optional

Target names. If given, every coordinate should have a corresponding name, and it gets repeated in the query result. It generates unique object names by default.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**query_photoobj**(**args*, ***kwargs*)

Queries the service and returns a table object.

Used to query the PhotoObjAll table with run, rerun, camcol and field values.

At least one of `run`, `camcol` or `field` parameters must be specified.

**Parameters**

**run** : integer, optional

Length of a strip observed in a single continuous image observing scan.

**rerun** : integer, optional

Reprocessing of an imaging run. Defaults to 301 which is the most recent rerun.

**camcol** : integer, optional

Output of one camera column of CCDs.

**field** : integer, optional

Part of a camcol of size 2048 by 1489 pixels.

**fields** : list, optional

SDSS PhotoObj or SpecObj quantities to return. If None, defaults to quantities required to find corresponding spectra and images of matched objects (e.g. plate, fiberID, mjd, etc.).

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**field_help: str or bool, optional**

Field name to check whether a valid PhotoObjAll or SpecObjAll field name. If `True` or it is an invalid field name all the valid field names are returned as a dict.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**table** : A `Table` object.

### Examples

```
>>> from astroquery.sdss import SDSS
>>> result = SDSS.query_photoobj(run=5714, camcol=6)
>>> print(result[:5])
      ra           dec             objid        run  rerun camcol field
------------- ------------- ------------------- ---- ----- ------ -----
30.4644529079 7.86460794626 1237670017266024498 5714   301      6    75
38.7635496073 7.47083098197 1237670017269628978 5714   301      6   130
22.2574304026 8.43175488904 1237670017262485671 5714   301      6    21
23.3724928784 8.32576993103 1237670017262944491 5714   301      6    28
25.4801226435 8.27642390025 1237670017263927330 5714   301      6    43
```

**query_photoobj_async**(*run=None*, *rerun=301*, *camcol=None*, *field=None*, *fields=None*, *timeout=60*, *get_query_payload=False*, *field_help=False*, *data_release=12*, *cache=True*)

Used to query the PhotoObjAll table with run, rerun, camcol and field values.

At least one of `run`, `camcol` or `field` parameters must be specified.

**Parameters**

**run** : integer, optional

Length of a strip observed in a single continuous image observing scan.

**rerun** : integer, optional

Reprocessing of an imaging run. Defaults to 301 which is the most recent rerun.

**camcol** : integer, optional

Output of one camera column of CCDs.

**field** : integer, optional

Part of a camcol of size 2048 by 1489 pixels.

**fields** : list, optional

SDSS PhotoObj or SpecObj quantities to return. If None, defaults to quantities required to find corresponding spectra and images of matched objects (e.g. plate, fiberID, mjd, etc.).

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**field_help: str or bool, optional**

Field name to check whether a valid PhotoObjAll or SpecObjAll field name. If `True` or it is an invalid field name all the valid field names are returned as a dict.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**result** : `Table`

The result of the query as a `Table` object.

### Examples

```
>>> from astroquery.sdss import SDSS
>>> result = SDSS.query_photoobj(run=5714, camcol=6)
>>> print(result[:5])
      ra           dec             objid        run  rerun camcol field
------------- ------------- ------------------- ---- ----- ------ -----
30.4644529079 7.86460794626 1237670017266024498 5714   301      6    75
38.7635496073 7.47083098197 1237670017269628978 5714   301      6   130
22.2574304026 8.43175488904 1237670017262485671 5714   301      6    21
23.3724928784 8.32576993103 1237670017262944491 5714   301      6    28
25.4801226435 8.27642390025 1237670017263927330 5714   301      6    43
```

**`query_region`**(**args*, ***kwargs*)

Queries the service and returns a table object.

Used to query a region around given coordinates. Equivalent to the object cross-ID from the web interface.

**Parameters**

**coordinates** : str or `astropy.coordinates` object or list of

coordinates or `Column` of coordinates The target(s) around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

Example: ra = np.array([220.064728084,220.064728467,220.06473483]) dec = np.array([0.870131920218,0.87013210119,0.870138329659]) coordinates = SkyCoord(ra, dec, frame='icrs', unit='deg')

**radius** : str or `Quantity` object, optional The

string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 2 arcsec.

**fields** : list, optional

SDSS PhotoObj or SpecObj quantities to return. If None, defaults to quantities required to find corresponding spectra and images of matched objects (e.g. plate, fiberID, mjd, etc.).

**spectro** : bool, optional

Look for spectroscopic match in addition to photometric match? If True, objects will only count as a match if photometry *and* spectroscopy exist. If False, will look for photometric matches only.

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**photoobj_fields** : list, optional

PhotoObj quantities to return. If photoobj_fields is None and specobj_fields is None then the value of fields is used

**specobj_fields** : list, optional

SpecObj quantities to return. If photoobj_fields is None and specobj_fields is None then the value of fields is used

**field_help: str or bool, optional**

Field name to check whether a valid PhotoObjAll or SpecObjAll field name. If `True` or it is an invalid field name all the valid field names are returned as a dict.

**obj_names** : str, or list or `Column`, optional

Target names. If given, every coordinate should have a corresponding name, and it gets repeated in the query result.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**table** : A `Table` object.

### Examples

```
>>> from astroquery.sdss import SDSS
>>> from astropy import coordinates as coords
>>> co = coords.SkyCoord('0h8m05.63s +14d50m23.3s')
>>> result = SDSS.query_region(co)
>>> print(result[:5])
      ra           dec             objid        run  rerun camcol field
------------- ------------- ------------------- ---- ----- ------ -----
2.02344282607 14.8398204075 1237653651835781245 1904   301      3   163
2.02344283666 14.8398204143 1237653651835781244 1904   301      3   163
2.02344596595 14.8398237229 1237652943176138867 1739   301      3   315
2.02344596303 14.8398237521 1237652943176138868 1739   301      3   315
2.02344772021 14.8398201105 1237653651835781243 1904   301      3   163
```

**query_region_async**(*coordinates*, *radius=<Quantity 2. arcsec>*, *fields=None*, *spectro=False*, *timeout=60*, *get_query_payload=False*, *photoobj_fields=None*, *specobj_fields=None*, *field_help=False*, *obj_names=None*, *data_release=12*, *cache=True*)

Used to query a region around given coordinates. Equivalent to the object cross-ID from the web interface.

**Parameters**

**coordinates** : str or `astropy.coordinates` object or list of

coordinates or `Column` of coordinates The target(s) around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

Example: ra = np.array([220.064728084,220.064728467,220.06473483]) dec = np.array([0.870131920218,0.87013210119,0.870138329659]) coordinates = SkyCoord(ra, dec, frame='icrs', unit='deg')

**radius** : str or `Quantity` object, optional The

string must be parsable by `Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 2 arcsec.

**fields** : list, optional

SDSS PhotoObj or SpecObj quantities to return. If None, defaults to quantities required to find corresponding spectra and images of matched objects (e.g. plate, fiberID, mjd, etc.).

**spectro** : bool, optional

Look for spectroscopic match in addition to photometric match? If True, objects will only count as a match if photometry *and* spectroscopy exist. If False, will look for photometric matches only.

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**photoobj_fields** : list, optional

PhotoObj quantities to return. If photoobj_fields is None and specobj_fields is None then the value of fields is used

**specobj_fields** : list, optional

SpecObj quantities to return. If photoobj_fields is None and specobj_fields is None then the value of fields is used

**field_help: str or bool, optional**

Field name to check whether a valid PhotoObjAll or SpecObjAll field name. If `True` or it is an invalid field name all the valid field names are returned as a dict.

**obj_names** : str, or list or `Column`, optional

Target names. If given, every coordinate should have a corresponding name, and it gets repeated in the query result.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**result** : Table

The result of the query as a Table object.

### Examples

```
>>> from astroquery.sdss import SDSS
>>> from astropy import coordinates as coords
>>> co = coords.SkyCoord('0h8m05.63s +14d50m23.3s')
>>> result = SDSS.query_region(co)
>>> print(result[:5])
      ra           dec             objid        run  rerun camcol field
------------- ------------- ------------------- ---- ----- ------ -----
2.02344282607 14.8398204075 1237653651835781245 1904   301      3   163
2.02344283666 14.8398204143 1237653651835781244 1904   301      3   163
2.02344596595 14.8398237229 1237652943176138867 1739   301      3   315
2.02344596303 14.8398237521 1237652943176138868 1739   301      3   315
2.02344772021 14.8398201105 1237653651835781243 1904   301      3   163
```

**query_specobj**(**args*, ***kwargs*)

Queries the service and returns a table object.

Used to query the SpecObjAll table with plate, mjd and fiberID values.

At least one of `plate`, `mjd` or `fiberID` parameters must be specified.

**Parameters**

**plate** : integer, optional

Plate number.

**mjd** : integer, optional

Modified Julian Date indicating the date a given piece of SDSS data was taken.

**fiberID** : integer, optional

Fiber number.

**fields** : list, optional

SDSS PhotoObj or SpecObj quantities to return. If None, defaults to quantities required to find corresponding spectra and images of matched objects (e.g. plate, fiberID, mjd, etc.).

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**field_help: str or bool, optional**

Field name to check whether a valid PhotoObjAll or SpecObjAll field name. If True or it is an invalid field name all the valid field names are returned as a dict.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**table** : A `Table` object.

### Examples

```
>>> from astroquery.sdss import SDSS
>>> result = SDSS.query_specobj(plate=2340,
...     fields=['ra', 'dec','plate', 'mjd', 'fiberID', 'specobjid'])
>>> print(result[:5])
      ra           dec       plate  mjd  fiberID      specobjid
------------- ------------- ----- ----- ------- -------------------
49.2020613611 5.20883041368  2340 53733      60 2634622337315530752
48.3745360119 5.26557511598  2340 53733     154 2634648175838783488
47.1604269095 5.48241410994  2340 53733     332 2634697104106219520
48.6634992214 6.69459110287  2340 53733     553 2634757852123654144
48.0759195428 6.18757403485  2340 53733     506 2634744932862027776
```

**query_specobj_async**(*plate=None*, *mjd=None*, *fiberID=None*, *fields=None*, *timeout=60*, *get_query_payload=False*, *field_help=False*, *data_release=12*, *cache=True*)

Used to query the SpecObjAll table with plate, mjd and fiberID values.

At least one of `plate`, `mjd` or `fiberID` parameters must be specified.

**Parameters**

**plate** : integer, optional

Plate number.

**mjd** : integer, optional

Modified Julian Date indicating the date a given piece of SDSS data was taken.

**fiberID** : integer, optional

Fiber number.

**fields** : list, optional

SDSS PhotoObj or SpecObj quantities to return. If None, defaults to quantities required to find corresponding spectra and images of matched objects (e.g. plate, fiberID, mjd, etc.).

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**field_help: str or bool, optional**

Field name to check whether a valid PhotoObjAll or SpecObjAll field name. If `True` or it is an invalid field name all the valid field names are returned as a dict.

**get_query_payload** : bool

If True, this will return the data the query would have sent out, but does not actually do the query.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**result** : Table

The result of the query as an Table object.

### Examples

```
>>> from astroquery.sdss import SDSS
>>> result = SDSS.query_specobj(plate=2340,
...     fields=['ra', 'dec','plate', 'mjd', 'fiberID', 'specobjid'])
>>> print(result[:5])
      ra           dec       plate  mjd  fiberID      specobjid
------------- ------------- ----- ----- ------- -------------------
49.2020613611 5.20883041368  2340 53733      60 2634622337315530752
48.3745360119 5.26557511598  2340 53733     154 2634648175838783488
47.1604269095 5.48241410994  2340 53733     332 2634697104106219520
48.6634992214 6.69459110287  2340 53733     553 2634757852123654144
48.0759195428 6.18757403485  2340 53733     506 2634744932862027776
```

**query_sql**(**args*, ***kwargs*)

Queries the service and returns a table object.

Query the SDSS database.

**Parameters**

**sql_query** : str

An SQL query

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to SDSSClass.TIMEOUT.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**table** : A Table object.

### Examples

```
>>> from astroquery.sdss import SDSS
>>> query = "select top 10                       z, ra, dec, bestObjID                  ␣
↪from                       specObj                     where                       ␣
↪class = 'galaxy'                       and z > 0.3                        and zWarning = 0
↪"
>>> res = SDSS.query_sql(query)
>>> print(res[:5])
    z          ra       dec         bestObjID
--------- --------- --------- -------------------
0.3000011 16.411075 4.1197892 1237678660894327022
0.3000012 49.459411  0.847754 1237660241924063461
0.3000027 156.25024 7.6586271 1237658425162858683
```

(continues on next page)

(continued from previous page)

```
0.3000027 256.99461 25.566255 1237661387086693265
 0.300003 175.65125  34.37548 1237665128003731630
```

**query_sql_async**(*sql_query*, *timeout=60*, *data_release=12*, *cache=True*, ***kwargs*)

Query the SDSS database.

**Parameters**

**sql_query** : str

An SQL query

**timeout** : float, optional

Time limit (in seconds) for establishing successful connection with remote server. Defaults to `SDSSClass.TIMEOUT`.

**data_release** : int

The data release of the SDSS to use.

**Returns**

**result** : `Table`

The result of the query as a `Table` object.

## Examples

```
>>> from astroquery.sdss import SDSS
>>> query = "select top 10                        z, ra, dec, bestObjID                    ␣
↪from                          specObj                      where                    ␣
↪class = 'galaxy'                        and z > 0.3                         and zWarning = 0
↪"
>>> res = SDSS.query_sql(query)
>>> print(res[:5])
    z         ra       dec         bestObjID
--------- --------- --------- -------------------
0.3000011 16.411075 4.1197892 1237678660894327022
0.3000012 49.459411  0.847754 1237660241924063461
0.3000027 156.25024 7.6586271 1237658425162858683
0.3000027 256.99461 25.566255 1237661387086693265
 0.300003 175.65125  34.37548 1237665128003731630
```

CHAPTER 33

# ALFALFA Queries (astroquery.alfalfa)

## 33.1 Getting started

This example shows how to perform an object cross-ID with ALFALFA. We'll start with the position of a source that exists in another survey (same object we used in the SDSS example).

```
>>> from astroquery.alfalfa import Alfalfa
>>> from astropy import coordinates as coords
>>> pos = coords.SkyCoord('0h8m05.63s +14d50m23.3s')
>>> agc = Alfalfa.query_region(pos, optical_counterpart=True)
```

This retrieves the AGC number of the object closest to the supplied ra and dec (within search radius dr=3 arcminutes by default). The "optical_counterpart" keyword argument above tells the crossID function to look for matches using the positions of the optical counterparts of HI detected sources (painstakingly determined by members of the ALFALFA team), rather than their radio centroids. The AGC number is an identification number for objects in the ALFALFA survey, and once we know it, we can download spectra (if they are available) easily,

```
>>> sp = Alfalfa.get_spectrum(agc)
```

This returns a PyFITS HDUList object. If we want to have a look at the entire ALFALFA catalog, we can do that too:

```
>>> cat = Alfalfa.get_catalog()
```

which returns a dictionary containing HI measurements for nearly 16,000 objects.

## 33.2 Reference/API

## 33.3 astroquery.alfalfa Package

### 33.3.1 ALFALFA Spectra Archive Query Tool

**Author**
Jordan Mirocha (mirochaj@gmail.com)

This package is for querying the ALFALFA data repository hosted at http://arecibo.tc.cornell.edu/hiarchive/alfalfa/

### 33.3.2 Classes

| |
|---|
| `AlfalfaClass`() |

#### AlfalfaClass

**class** astroquery.alfalfa.**AlfalfaClass**
Bases: `astroquery.query.BaseQuery`

##### Attributes Summary

| |
|---|
| `CATALOG_PREFIX` |
| `FITS_PREFIX` |
| `PLACEHOLDER` |

##### Methods Summary

| | |
|---|---|
| `get_catalog`() | Download catalog of ALFALFA source properties. |
| `get_spectrum`(agc[, show_progress]) | Download spectrum from ALFALFA catalogue. |
| `get_spectrum_async`(agc[, show_progress]) | Download spectrum from ALFALFA catalogue. |
| `query_region`(coordinates[, radius, ...]) | Perform object cross-ID in ALFALFA. |

##### Attributes Documentation

**CATALOG_PREFIX = 'http://egg.astro.cornell.edu/alfalfa/data/a40files/a40.datafile1.csv'**

**FITS_PREFIX = 'http://arecibo.tc.cornell.edu/hiarchive/alfalfa/spectraFITS'**

**PLACEHOLDER = -999999**

### Methods Documentation

**get_catalog**()

Download catalog of ALFALFA source properties.

**Returns**

**result** : Dictionary of results, each element is a masked array.

### Notes

This catalog has ~15,000 entries, so after it's downloaded, it is made global to save some time later.

**get_spectrum**(*agc*, *show_progress=True*)

Download spectrum from ALFALFA catalogue.

**Parameters**

**agc** : int

Identification number for object in ALFALFA catalog.

**ascii** : bool

Download spectrum from remote server in ASCII or FITS format?

**Returns**

**spectrum** : `HDUList`

Spectrum is in `hdulist[0].data[0][2]`

**See also:**

`get_catalog`
method that downloads ALFALFA catalog

`query_region`
find object in catalog closest to supplied position (use this to determine AGC number first)

**get_spectrum_async**(*agc*, *show_progress=True*)

Download spectrum from ALFALFA catalogue.

**Parameters**

**agc** : int

Identification number for object in ALFALFA catalog.

**ascii** : bool

Download spectrum from remote server in ASCII or FITS format?

**Returns**

**result** : A file context manager

**See also:**

`get_catalog`
method that downloads ALFALFA catalog

`query_region`
find object in catalog closest to supplied position (use this to determine AGC number first)

**query_region**(*coordinates*, *radius=<Quantity 3. arcmin>*, *optical_counterpart=False*)
Perform object cross-ID in ALFALFA.

Search for objects near position (ra, dec) within some radius.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

The target around which to search. It may be specified as a string in which case it is resolved using online services or as the appropriate `astropy.coordinates` object. ICRS coordinates may also be entered as strings as specified in the `astropy.coordinates` module.

**radius** : str or `Quantity` object, optional

The string must be parsable by `astropy.coordinates.Angle`. The appropriate `Quantity` object from `astropy.units` may also be used. Defaults to 3 arcmin.

**optical_counterpart** : bool

Search for position match using radio positions or position of any optical counterpart identified by ALFALFA team? Keep in mind that the ALFA beam size is about 3x3 arcminutes.

**See documentation for astropy.coordinates.angles for more information**

**about ('ra', 'dec', 'unit') parameters.**

**Returns**

**result** : AGC number for object nearest supplied position.

## Examples

```
>>> from astroquery.alfalfa import Alfalfa
>>> from astropy import coordinates as coords
>>> C = coords.SkyCoord('0h8m05.63s +14d50m23.3s')
>>> agc = Alfalfa.query_region(C,'3 arcmin')
```

CHAPTER 34

# Spitzer Heritage Archive (astroquery.sha)

## 34.1 Querying catalogs

There are four types of supported queries for the Spitzer Heritage Archive (SHA) module, searching by: position, NAIFID, PID, and ReqKey. Examples for each are shown below.

Using the standard imports:

```
>>> from astroquery import sha
>>> from astropy import coordinates as coord
>>> from astropy import units as u
```

Query with an astropy coordinate instance (preferred):

```
>>> pos_t1 = sha.query(coord=coord.SkyCoord(ra=163.6136, dec=-11.784,
... unit=(u.degree, u.degree)), size=0.5)
```

Query with the alternate `ra` and `dec` parameters:

```
>>> pos_t2 = sha.query(ra=163.6136, dec=-11.784, size=0.5)
```

Query by NAIFID:

```
>>> nid_t = sha.query(naifid=2003226)
```

Query by PID:

```
>>> pid_t = sha.query(pid=30080)
```

Query by ReqKey:

```
>>> # by ReqKey
>>> rqk_t = sha.query(reqkey=21641216)
```

## 34.2 Additional Documentation

For column descriptions, metadata, and other information visit the SHA query API help page.

## 34.3 Saving files to disk

Using the access URLs found in the SHA queries, the functions `astroquery.sha.save_file` writes the file to disk. To save a file:

```
>>> pid_t = sha.query(pid=30080)
>>> url = pid_t['accessUrl'][0].strip()
>>> sha.save_file(url)
```

or alternatively with a name and path specified:

```
>>> sha.save_file(url, out_dir='proj_files/', out_name='sha_file1')
```

The extension will automatically be added depending on the filetype.

## 34.4 Reading files into python

Given an access URL, `astroquery.sha.get_file` returns an appropriate astropy object, either a `Table` instance for tabular data, or `PrimaryHDU` instance for FITS files.

```
>>> pid_t = sha.query(pid=30080)
>>> url = pid_t['accessUrl'][0].strip()
>>> img = sha.get_file(url)
```

## 34.5 Reference/API

### 34.5.1 astroquery.sha Package

#### SHA Query Tool

**Author**
Brian Svoboda (svobodb@email.arizona.edu)

This package is for querying the Spitzer Heritage Archive (SHA) found at: http://sha.ipac.caltech.edu/applications/Spitzer/SHA.

#### Functions

| | |
|---|---|
| `get_file`(url) | Return object from SHA query URL. |
| `query`([coord, ra, dec, size, naifid, pid, ...]) | Query the Spitzer Heritage Archive (SHA). |
| `save_file`(url[, out_dir, out_name]) | Download image to output directory given a URL from a SHA query. |

### get_file

astroquery.sha.**get_file**(*url*)

Return object from SHA query URL.

Currently only supports FITS files.

**Parameters**

**url** : str

Access URL from SHA query. Requires complete URL, valid URLs from the SHA query include columns:

```
accessUrl -> The URL to be used to retrieve an image or table
withAnc1  -> The URL to be used to retrieve the image or spectra
             with important ancillary products (mask, uncertainty,
             etc.) as a zip archive
```

**Returns**

**obj** : `Table`, `astropy.io.fits`, list

Return object depending if link points to a table, fits image, or zip file of products.

#### Examples

```
>>> from astroquery import sha
>>> url = sha.query(pid=30080)['accessUrl'][0]
>>> img = sha.get_file(url)
```

### query

astroquery.sha.**query**(*coord=None*, *ra=None*, *dec=None*, *size=None*, *naifid=None*, *pid=None*, *reqkey=None*, *dataset=2*, *verbosity=3*, *return_response=False*, *return_payload=False*)

Query the Spitzer Heritage Archive (SHA).

Four query types are valid to search by position, NAIFID, PID, and ReqKey:

```
position -> search a region
naifid   -> NAIF ID, which is a unique number allocated to solar
            system objects (e.g. planets, asteroids, comets,
            spacecraft) by the NAIF at JPL.
pid      -> program ID
reqkey   -> AOR ID: Astronomical Observation Request ID
```

For a valid query, enter only parameters related to a single query type:

```
position -> ra, dec, size
naifid   -> naifid
pid      -> pid
reqkey   -> reqkey
```

**Parameters**

**coord** : astropy.coordinates.builtin_systems

Astropy coordinate object. (query_type = 'position')

**ra** : number

Right ascension in degrees, alternative to using `coord`. (query_type = 'position')

**dec** : number

Declination in degrees, alternative to using `coord`. (query_type = 'position')

**size** : number

Region size in degrees. (query_type = 'position')

**naifid** : number

NAIF ID. (query_type = 'naifid')

**pid** : number

Program ID. (query_type = 'pid')

**reqkey** : number

Astronomical Observation Request ID. (query_type = 'reqkey')

**dataset** : number, default 2

Data set. Valid options:

```
1 -> BCD data
2 -> PBCD data
```

**verbosity** : number, default 3

Verbosity level, controls the number of columns to output.

**Returns**

**table** : `Table`

### Notes

For column descriptions, metadata, and other information visit the SHA query API help page

### Examples

Position query using an astropy coordinate object

```
>>> import astropy.coordinates as coord
>>> import astropy.units as u
>>> from astroquery import sha
>>> pos_t = sha.query(coord=coord.SkyCoord(ra=163.6136, dec=-11.784,
... unit=(u.degree, u.degree)), size=0.5)
```

Position query with optional `ra` and `dec` parameters

```
>>> pos_t = sha.query(ra=163.6136, dec=-11.784, size=0.5)
```

NAIFID query

```
>>> nid_t = sha.query(naifid=2003226)
```

PID query

```
>>> pid_t = sha.query(pid=30080)
```

ReqKey query

```
>>> rqk_t = sha.query(reqkey=21641216)
```

### save_file

`astroquery.sha.save_file`(*url*, *out_dir='sha_tmp/'*, *out_name=None*)

Download image to output directory given a URL from a SHA query.

**Parameters**

**url** : str

Access URL from SHA query. Requires complete URL, valid URLs from the SHA query include columns:

```
accessUrl -> The URL to be used to retrieve an image or table
withAnc1  -> The URL to be used to retrieve the image or spectra
             with important ancillary products (mask, uncertainty,
             etc.) as a zip archive
```

**out_dir** : str

Path for output table or image

**out_name** : str

Name for output table or image, if None use the file ID as name.

#### Examples

```
>>> from astroquery import sha
>>> url = sha.query(pid=30080)['accessUrl'][0]
>>> sha.save_file(url)
```

CHAPTER 35

# LAMDA Queries (astroquery.lamda)

## 35.1 Getting started

The Leiden Atomic and Molecular Database (LAMDA) stores information for energy levels, radiative transitions, and collisional rates for many astrophysically relevant atoms and molecules. To print the list of available molecules for query, use:

```
>>> from astroquery.lamda import Lamda
>>> Lamda.molecule_dict
```

The dictionary is created dynamically from the LAMDA website the first time it is called, then cached for future use. If there has been an update and you want to reload the cache, you can find the cache file 'molecules.json' and remove it:

```
>>> Lamda.cache_location
u'/Users/your_username/.astropy/cache/astroquery/Lamda'
>>> Lamda.moldict_path
u'/Users/your_username/.astropy/cache/astroquery/Lamda/molecules.json'
>>> os.remove(Lamda.moldict_path)
```

You can query for any molecule in that dictionary.

```
>>> collrates, radtransitions, enlevels = Lamda.query(mol='co')
```

Catalogs are returned as Table instances, except for collrates, which is a dictionary of tables, with one table for each collisional partner.

## 35.2 Reference/API

### 35.2.1 astroquery.lamda Package

#### LAMDA Query Tool

**Author**
Brian Svoboda (svobodb@email.arizona.edu)

This package is for querying the Leiden Atomic and Molecular Database (LAMDA) hosted at: http://home.strw.leidenuniv.nl/~moldata/.

**Note:**
If you use the data files from LAMDA in your research work please refer to the publication by Schoier, F.L., van der Tak, F.F.S., van Dishoeck E.F., Black, J.H. 2005, A&A 432, 369-379. When individual molecules are considered, references to the original papers providing the spectroscopic and collisional data are encouraged.

#### Functions

| | |
|---|---|
| `parse_lamda_datafile`(filename) | Read a datafile that follows the format adopted for the atomic and molecular data in the LAMDA database. |
| `write_lamda_datafile`(filename, tables) | Write tuple of tables with LAMDA data into a datafile that follows the format adopted for the LAMDA database. |

#### parse_lamda_datafile

`astroquery.lamda.parse_lamda_datafile`(*filename*)
Read a datafile that follows the format adopted for the atomic and molecular data in the LAMDA database.

**Parameters**
**filename** : str

Fully qualified path of the file to read.

**Returns**
Tuple of tables: ({rateid: Table, },

Table, Table)

#### write_lamda_datafile

`astroquery.lamda.write_lamda_datafile`(*filename*, *tables*)
Write tuple of tables with LAMDA data into a datafile that follows the format adopted for the LAMDA database.

**Parameters**
**filename** : str

Fully qualified path of the file to write.

**tables: tuple**

Tuple of Tables ({rateid: coll_table}, rad_table, mol_table)

# CHAPTER 36

## OGLE Queries (astroquery.ogle)

### 36.1 Getting started

The Optical Gravitational Lensing Experiment III (OGLE-III) stores information on the interstellar extinction towards the Galactic Bulge. The astroquery.ogle module queries the online extinction calculator and returns an Table instance with the same data. To run a single query using an astropy.coordinates instance use:

```
>>> from astropy import coordinates
>>> from astropy import units as u
>>> from astroquery.ogle import Ogle
>>> co = coordinates.SkyCoord(0*u.deg, 3*u.deg, frame='galactic')
>>> t = Ogle.query_region(coord=co)
```

Arguments can be passed to choose the interpolation algorithm, quality factor, and coordinate system. Multiple coordinates may be queried simultaneously by passing a list-like object of string/float values or a list-like object of astropy.coordinates instances. All of coordinates will be internally converted to FK5.

```
>>> # list of coordinate instances
>>> co_list = [co, co, co]
>>> t1 = Ogle.query_region(coord=co_list)
>>> # (2 x N) list of values
>>> co_list_values = [[0, 0, 0], [3, 3, 3]]
>>> t2 = Ogle.query_region(coord=co_list_values, coord_sys='LB')
```

Note that non-Astropy coordinates may not be supported in a future version.

## 36.2 Reference/API

### 36.2.1 astroquery.ogle Package

#### OGLE Query Tool

**Author**
Brian Svoboda (svobodb@email.arizona.edu)

This package is for querying interstellar extinction toward the Galactic bulge from OGLE-III data hosted at.

**Note:**
If you use the data from OGLE please refer to the publication by Nataf et al. (2012).

#### Classes

| | |
|---|---|
| OgleClass() | |
| Conf | Configuration parameters for astroquery.ogle. |

#### OgleClass

**class** astroquery.ogle.**OgleClass**
Bases: astroquery.query.BaseQuery

##### Attributes Summary

| |
|---|
| DATA_URL |
| TIMEOUT |
| algorithms |
| coord_systems |
| quality_codes |
| result_dtypes |

##### Methods Summary

| | |
|---|---|
| query_region(*args, **kwargs) | Queries the service and returns a table object. |
| query_region_async(*args, **kwargs) | Query the OGLE-III interstellar extinction calculator. |

##### Attributes Documentation

**DATA_URL = 'http://ogle.astrouw.edu.pl/cgi-ogle/getext.py'**

**TIMEOUT = 60**

**algorithms = ['NG', 'NN']**

**coord_systems = ['RD', 'LB']**

**quality_codes = ['GOOD', 'ALL']**

**result_dtypes = ['f8', 'f8', 'f8', 'f8', 'f8', 'f8', 'f8', 'f8', 'i8', 'a2', 'f8']**

### Methods Documentation

**query_region**(**args*, ***kwargs*)

Queries the service and returns a table object.

Query the OGLE-III interstellar extinction calculator.

**Parameters**

**coord** : list-like

Pointings to evaluate interstellar extinction. Three forms of coordinates may be passed:

```
* single astropy coordinate instance
* list-like object (1 x N) of astropy coordinate instances
* list-like object (2 x N) of RA/Decs or Glon/Glat as strings
or  floats. (May not be supported in future versions.)
```

**algorithm** : string

Algorithm to interpolate data for desired coordinate. Valid options:

```
* 'NG': nearest grid point
* 'NN': natural neighbor interpolation
```

**quality** : string

Quality factor for data. Valid options:

```
* 'All': all points
* 'GOOD': QF=0 as described in Nataf et al. (2012).
```

**coord_sys** : string

Coordinate system if using lists of RA/Decs in `coord`. Valid options:

```
* 'RD': equatorial coordinates
* 'LB': Galactic coordinates.
```

**Returns**

**table** : A `Table` object.

**Raises**

**CoordParseError**

Exception raised for malformed coordinate input

### Examples

Using astropy coordinates:

```
>>> from astropy.coordinates import SkyCoord
>>> from astropy import units as u
>>> co = SkyCoord(0.0, 3.0, unit=(u.degree, u.degree),
...               frame='galactic')
>>> from astroquery.ogle import Ogle
>>> t = Ogle.query_region(coord=co)
>>> t.pprint()
RA/LON   Dec/Lat    A_I  E(V-I) S_E(V-I) R_JKVI   mu    S_mu
--------- ---------- ----- ------ -------- ------ ------ ----- ...
17.568157 -27.342475 3.126  2.597    0.126 0.3337 14.581 0.212
```

**query_region_async**(**args*, ***kwargs*)

Query the OGLE-III interstellar extinction calculator.

**Parameters**

**coord** : list-like

Pointings to evaluate interstellar extinction. Three forms of coordinates may be passed:

```
* single astropy coordinate instance
* list-like object (1 x N) of astropy coordinate instances
* list-like object (2 x N) of RA/Decs or Glon/Glat as strings
or  floats. (May not be supported in future versions.)
```

**algorithm** : string

Algorithm to interpolate data for desired coordinate. Valid options:

```
* 'NG': nearest grid point
* 'NN': natural neighbor interpolation
```

**quality** : string

Quality factor for data. Valid options:

```
* 'All': all points
* 'GOOD': QF=0 as described in Nataf et al. (2012).
```

**coord_sys** : string

Coordinate system if using lists of RA/Decs in `coord`. Valid options:

```
* 'RD': equatorial coordinates
* 'LB': Galactic coordinates.
```

**Returns**

**response** : `requests.Response`

The HTTP response returned from the service.

**Raises**

**CoordParseError**

Exception raised for malformed coordinate input

### Examples

Using astropy coordinates:

```
>>> from astropy.coordinates import SkyCoord
>>> from astropy import units as u
>>> co = SkyCoord(0.0, 3.0, unit=(u.degree, u.degree),
...               frame='galactic')
>>> from astroquery.ogle import Ogle
>>> t = Ogle.query_region(coord=co)
>>> t.pprint()
RA/LON    Dec/Lat    A_I  E(V-I) S_E(V-I) R_JKVI   mu    S_mu
--------- ---------- ----- ------ -------- ------ ------ ----- ...
17.568157 -27.342475 3.126  2.597    0.126 0.3337 14.581 0.212
```

## Conf

**class** astroquery.ogle.**Conf**

Bases: astropy.config.ConfigNamespace

Configuration parameters for astroquery.ogle.

### Attributes Summary

| | |
|---|---|
| server | Name of the OGLE mirror to use. |
| timeout | Time limit for connecting to OGLE server. |

### Attributes Documentation

**server**

Name of the OGLE mirror to use.

**timeout**

Time limit for connecting to OGLE server.

# CHAPTER 37

# Open Exoplanet Catalogue(astroquery.open_exoplanet_catalogue)

## 37.1 Getting started

This module gives easy access to the open exoplanet catalogue in the form of an XML element tree.

To start import the catalog and generate the catalogue.

```
from astroquery import open_exoplanet_catalogue as oec
from astroquery.open_exoplanet_catalogue import findvalue

# getting the catalogue from the default remote source
cata = oec.get_catalogue()

# getting the catalogue from a local path
cata = oec.get_catalogue("path/to/file/systems.xml.gz")
```

## 37.2 Examples

First import the module and generate the catalogue. The findvalue function provides a simple method of getting values from Elements.

```
from astroquery import open_exoplanet_catalogue as oec
from astroquery.open_exoplanet_catalogue import findvalue

cata = oec.get_catalogue()
```

Prints all planets and their masses.

```
for planet in oec.findall(".//planet"):
    print(findvalue(planet, 'name'), findvalue(planet, 'mass'))
```

Prints all of the planets with known mass around stars of known mass in a machine readable format.

```
for star in oec.findall(".//star[mass]"):
    for planet in star.findall(".//planet[mass]"):
        print(findvalue(planet, 'mass').machine_readable(), findvalue(star, 'mass').machine_readable())
```

Print all the names of stars in binaries.

```
for star in oec.findall(".//binary/star"):
    print(findvalue(star, 'name'))
```

Prints all the planet names and period of planets around binaries

```
for planet in oec.findall(".//binary/planet"):
    print(findvalue( planet, 'name'), findvalue( planet, 'period'))
```

Prints the name, radius and mass of the planet Kepler-68 b.

```
planet = oec.find(".//planet[name='Kepler-68 b']")
print(findvalue( planet, 'name'), findvalue(planet, 'radius'), findvalue(planet, 'mass'))
```

Prints the name and radius of planets with a radius greater than 1 jupiter radius.

```
for planet in oec.findall(".//planet[radius]"):
    if findvalue(planet, 'radius') > 1:
        print(findvalue( planet, 'name'), findvalue( planet, 'radius'))
```

Prints the names of the planets around a single star in a binary.

```
for binary in oec.findall(".//binary/star/planet"):
    print(findvalue( binary, 'name'))
```

Prints a ratio of star and planet mass.

```
for star in oec.findall(".//star[mass]/planet[mass].."):
    if findvalue(star, 'mass') != None:
        for planet in star.findall(".//planet"):
            if findvalue(planet, 'mass') != None:
                print(findvalue( star, 'name'), findvalue( planet, 'name'), "Ratio:", findvalue( star,
↪'mass')/findvalue( planet, 'mass'))
```

Prints planets whose mass has an upper limit

```
for planet in oec.findall(".//planet/mass[@upperlimit].."):
    print(findvalue( planet, 'name'), findvalue(planet, 'mass'))
```

Prints all stars with the number of planets orbiting them

```
for star in oec.findall(".//star[planet]"):
    print(findvalue( star, 'name'), len(star.findall(".//planet")))
```

Prints all the properties of Kepler-20 b.

```
for properties in oec.findall(".//planet[name='Kepler-20 b']/*"):
    print("\t" + properties.tag + ":", properties.text)
```

Prints the right ascension and declination of systems with planets of known mass.

```
for systems in oec.findall(".//system[declination][rightascension]"):
    for planet in systems.findall(".//planet[mass]"):
        print(findvalue( systems, 'name'), findvalue( systems, 'rightascension'), findvalue( systems,
↪'declination'), findvalue( planet, 'mass'))
```

Prints the names of rogue planets.

```
for planets in oec.findall(".//system/planet"):
    print(findvalue( planets, 'name'))
```

## 37.3 Reference/API

### 37.3.1 astroquery.open_exoplanet_catalogue Package

Access to the Open Exoplanet Catalogue. Hanno Rein 2013

https://github.com/hannorein/open_exoplanet_catalogue https://github.com/hannorein/oec_meta http://openexoplanetcatalogue.com

#### Functions

| | |
|---|---|
| `findvalue`(element, searchstring) | Searches given string in element. |
| `get_catalogue`([filepath]) | Parses the Open Exoplanet Catalogue file. |
| `xml_element_to_dict`(e) | Creates a dictionary of the given xml tree. |

#### findvalue

`astroquery.open_exoplanet_catalogue.`**`findvalue`**(*element*, *searchstring*)

Searches given string in element.

**Parameters**

**element** : Element

Element from the ElementTree module.

**searchstring** : str

name of the tag to look for in element

**Returns**

None if tag does not exist.

str if the tag cannot be expressed as a float.

Number if the tag is a numerical value

#### get_catalogue

`astroquery.open_exoplanet_catalogue.`**`get_catalogue`**(*filepath=None*)

Parses the Open Exoplanet Catalogue file.

**Parameters**

**filepath** : str or None

if no filepath is given, remote source is used.

**Returns**

An Element Tree containing the open exoplanet catalogue

### xml_element_to_dict

`astroquery.open_exoplanet_catalogue.`**`xml_element_to_dict`**`(`*e*`)`

Creates a dictionary of the given xml tree.

**Parameters**

**e** : str

str of an xml tree

**Returns**

A dictionary of the given xml tree

To contribute to the open exoplanet catalogue, fork the project on github! https://github.com/OpenExoplanetCatalogue/open_exoplanet_catalogue

CHAPTER 38

# CosmoSim Queries (astroquery.cosmosim)

This module allows the user to query and download from one of three cosmological simulation projects: the MultiDark project, the BolshoiP project, and the CLUES project. For accessing these databases a CosmoSim object must first be instantiated with valid credentials (no public username/password are implemented). Below are a couple of examples of usage.

## 38.1 Requirements

The following packages are required for the use of this module:

- requests
- keyring
- getpass
- bs4

## 38.2 Getting started

```
>>> from astroquery.cosmosim import CosmoSim
>>> CS = CosmoSim()
```

Next, enter your credentials; caching is enabled, so after the initial successful login no further password is required if desired.

```
>>> CS.login(username="uname")
uname, enter your CosmoSim password:
Authenticating uname on www.cosmosim.org...
Authentication successful!
>>> # If running from a script (rather than an interactive python session):
>>> # CS.login(username="uname",password="password")
```

To store the password associated with your username in the keychain:

```
>>> CS.login(username="uname",store_password=True)
WARNING: No password was found in the keychain for the provided username. [astroquery.cosmosim.core]
uname, enter your CosmoSim password:
Authenticating uname on www.cosmosim.org...
Authentication successful!
```

Logging out is as simple as:

```
>>> CS.logout(deletepw=True)
Removed password for uname in the keychain.
```

The deletepw option will undo the storage of any password in the keychain. Checking whether you are successfully logged in (or who is currently logged in):

```
>>> CS.check_login_status()
Status: You are logged in as uname.
```

Below is an example of running an SQL query (BDMV mass function of the MDR1 cosmological simulation at a redshift of z=0):

```
>>> sql_query = "SELECT 0.25*(0.5+FLOOR(LOG10(mass)/0.25)) AS log_mass, COUNT(*) AS num FROM MDR1.FOF␣
↪WHERE snapnum=85 GROUP BY FLOOR(LOG10(mass)/0.25) ORDER BY log_mass"
>>> CS.run_sql_query(query_string=sql_query)
Job created: 359748449665484 #jobid; note: is unique to each and
every query
```

## 38.3 Managing CosmoSim Queries

The cosmosim module provides functionality for checking the completion status of queries, in addition to deleting them from the server. Below are a few examples of functions available to the user for these purposes.

```
>>> CS.check_all_jobs()
     JobID        Phase
--------------- ---------
359748449665484 COMPLETED
>>> CS.delete_job(jobid='359748449665484')
Deleted job: 359748449665484
>>> CS.check_all_jobs()
     JobID        Phase
--------------- ---------
```

The above function 'check_all_jobs' also supports the usage of a job's phase status in order to filter through all available CosmoSim jobs.

```
>>> CS.check_all_jobs()
     JobID        Phase
--------------- ---------
359748449665484 COMPLETED
359748449682647 ABORTED
359748449628375 ERROR
>>> CS.check_all_jobs(phase=['Completed','Aborted'])
     JobID        Phase
--------------- ---------
```

(continues on next page)

(continued from previous page)

```
359748449665484 COMPLETED
359748449682647 ABORTED
```

Additionally, 'check_all_jobs' (and 'delete_all_jobs') accepts both phase and/or tablename (via a regular expression) as criteria for deletion of all available CosmoSim jobs. But be careful: Leaving both arguments blank will delete ALL jobs!

```
>>> CS.check_all_jobs()
     JobID        Phase
--------------- ---------
359748449665484 COMPLETED
359748449682647 ABORTED
359748449628375 ERROR
>>> CS.table_dict()
{'359748449665484': '2014-09-07T05:01:40:0458'}
{'359748449682647': 'table2'}
{'359748449628375': 'table3'}
>>> CS.delete_all_jobs(phase=['Aborted','error'],regex='[a-z]*[0-9]*')
Deleted job: 359748449682647 (Table: table2)
Deleted job: 359748449628375 (Table: table3)
```

Note: Arguments for phase are case insensitive. Now, check to see if the jobs have been deleted:

```
>>> CS.check_all_jobs()
     JobID        Phase
--------------- ---------
359748449665484 COMPLETED
```

Getting rid of this last job can be done by deleting all jobs with phase COMPLETED, or it can be done simply by providing the 'delete_job' function with its unique jobid. Lastly, this could be accomplished by matching its tablename to the following regular expression: '[0-9]*-[0-9]*-[0-9]*[A-Z]*[0-9]*:[0-9]*:[0-9]*:[0-9]*'. All jobs created without specifying the tablename argument in 'run_sql_query' are automatically assigned one based upon the creation date and time of the job, and is therefore the default tablename format.

Deleting all jobs, regardless of tablename, and job phase:

```
>>> CS.check_all_jobs()
     JobID        Phase
--------------- ---------
359748449665484 ABORTED
359748586913123 COMPLETED

>>> CS.delete_all_jobs()
Deleted job: 359748449665484
Deleted job: 359748586913123
>>> CS.check_all_jobs()
     JobID        Phase
--------------- ---------
```

In addition to the phase and regex arguments for 'check_all_jobs', selected jobs can be sorted using two properties:

```
>>> CS.check_all_jobs(phase=['completed'],regex='[a-z]*[0-9]*',sortby='tablename')
     JobID        Phase    Tablename          Starttime
--------------- --------- ---------- -------------------------
361298054830707 COMPLETED    table1 2014-09-21T19:28:48+02:00
361298050841687 COMPLETED    table2 2014-09-21T19:20:23+02:00
```

```
>>> CS.check_all_jobs(phase=['completed'],regex='[a-z]*[0-9]*',sortby='starttime')
     JobID        Phase    Tablename          Starttime
--------------- --------- --------- -------------------------
361298050841687 COMPLETED    table2 2014-09-21T19:20:23+02:00
361298054830707 COMPLETED    table1 2014-09-21T19:28:48+02:00
```

## 38.4 Exploring Database Schema

A database exploration tool is available to help the user navigate the structure of any simulation database in the CosmoSim database.

Note: '@' precedes entries which are dictionaries

```
>>> CS.explore_db()
Must first specify a database.
        Projects          Project Items                                       Information
------------------------ ------------- -------------------------------------------------------------------
↪---------------------
              @ Bolshoi       @ tables
                                   id:                                                                    ␣
↪                     2
                          description:                                                                    ␣
↪The Bolshoi Database.
------------------------ ------------- -------------------------------------------------------------------
↪---------------------
             @ BolshoiP       @ tables
                                   id:                                                                    ␣
↪                   119
                          description:                                                                ␣
↪Bolshoi Planck simulation
------------------------ ------------- -------------------------------------------------------------------
↪---------------------
          @ Clues3_LGDM       @ tables
                                   id:                                                                    ␣
↪                   134
                          description: CLUES simulation, B64, 186592, WMAP3, Local Group resimulation,␣
↪4096, Dark Matter only
------------------------ ------------- -------------------------------------------------------------------
↪---------------------
         @ Clues3_LGGas       @ tables
                                   id:                                                                    ␣
↪                   124
                          description:          CLUES simulation, B64, 186592, WMAP3, Local Group␣
↪resimulation, 4096, Gas+SFR
------------------------ ------------- -------------------------------------------------------------------
↪---------------------
                 @ MDPL       @ tables
                                   id:                                                                    ␣
↪                   114
                          description:                                                          The␣
↪MDR1-Planck simulation.
------------------------ ------------- -------------------------------------------------------------------
↪---------------------
                 @ MDR1       @ tables
                                   id:                                                                    ␣
↪                     7
```

(continues on next page)

(continued from previous page)

```
                          description:                                                       The␣
↪MultiDark Run 1 Simulation.
------------------------ ------------- -----------------------------------------------------------------
↪---------------------
@ cosmosim_user_username      @ tables
                               id:                                                                    ␣
↪               userdb
                          description:                                                              ␣
↪Your personal database
------------------------ ------------- -----------------------------------------------------------------
↪---------------------
```

```
>>> CS.explore_db(db='MDPL')
  Projects  Project Items     Tables
----------- ------------- -------------
--> @ MDPL: --> @ tables:         @ FOF
                       id        @ FOF5
              description        @ FOF4
                                 @ FOF3
                                 @ FOF2
                                 @ FOF1
                                 @ BDMW
                            @ Redshifts
                           @ LinkLength
                           @ AvailHalos
                          @ Particles88
```

```
>>> CS.explore_db(db='MDPL',table='FOF')
  Projects  Project Items     Tables     Table Items  Table Info Columns
----------- ------------- ------------- ------------ ---------- --------
--> @ MDPL: --> @ tables:     --> @ FOF:          id:        934        y
                       id        @ FOF5    @ columns                    x
              description        @ FOF4 description:                    z
                                 @ FOF3                                ix
                                 @ FOF2                                iz
                                 @ FOF1                                vx
                                 @ BDMW                                vy
                            @ Redshifts                                vz
                           @ LinkLength                                iy
                           @ AvailHalos                                np
                          @ Particles88                              disp
                                                                     size
                                                                     spin
                                                                     mass
                                                                    axis1
                                                                    axis2
                                                                    axis3
                                                                    fofId
                                                                    phkey
                                                                    delta
                                                                    level
                                                                   angMom
                                                                   disp_v
                                                                  axis1_z
                                                                  axis1_x
                                                                  axis1_y
```

(continues on next page)

(continued from previous page)

```
                                                                axis3_x
                                                                axis3_y
                                                                axis3_z
                                                                axis2_y
                                                                axis2_x
                                                                NInFile
                                                                axis2_z
                                                                snapnum
                                                               angMom_x
                                                               angMom_y
                                                               angMom_z
```

```
>>> CS.explore_db(db='MDPL',table='FOF',col='fofId')
  Projects  Project Items     Tables     Table Items     Columns
----------- ------------- ------------- -------------- ------------
--> @ MDPL: --> @ tables:    --> @ FOF: --> @ columns: --> @ fofId:
                         id        @ FOF5             id       @ disp
              description          @ FOF4    description    @ axis1_z
                                   @ FOF3                   @ axis1_x
                                   @ FOF2                   @ axis1_y
                                   @ FOF1                        @ ix
                                   @ BDMW                        @ iz
                              @ Redshifts                   @ axis3_x
                             @ LinkLength                   @ axis3_y
                             @ AvailHalos                   @ axis3_z
                            @ Particles88                        @ vx
                                                                 @ vy
                                                                 @ vz
                                                            @ axis2_y
                                                            @ axis2_x
                                                               @ size
                                                              @ axis1
                                                              @ axis2
                                                              @ axis3
                                                                 @ iy
                                                             @ angMom
                                                            @ NInFile
                                                                 @ np
                                                            @ axis2_z
                                                             @ disp_v
                                                              @ phkey
                                                              @ delta
                                                            @ snapnum
                                                               @ spin
                                                              @ level
                                                           @ angMom_x
                                                           @ angMom_y
                                                           @ angMom_z
                                                               @ mass
                                                                  @ y
                                                                  @ x
                                                                  @ z
```

## 38.5 Downloading data

Query results can be downloaded and used in real-time from the command line, or alternatively they can be stored on your local machine.

```
>>> CS.check_all_jobs()
     JobID        Phase
--------------- ---------
359750704009965 COMPLETED

>>> data = CS.download(jobid='359750704009965',format='csv')
>>> print(data)
(['row_id', 'log_mass', 'num'],
 [[1, 10.88, 3683],
  [2, 11.12, 452606],
  [3, 11.38, 3024674],
  [4, 11.62, 3828931],
  [5, 11.88, 2638644],
  [6, 12.12, 1572685],
  [7, 12.38, 926764],
  [8, 12.62, 544650],
  [9, 12.88, 312360],
  [10, 13.12, 174164],
  [11, 13.38, 95263],
  [12, 13.62, 50473],
  [13, 13.88, 25157],
  [14, 14.12, 11623],
  [15, 14.38, 4769],
  [16, 14.62, 1672],
  [17, 14.88, 458],
  [18, 15.12, 68],
  [19, 15.38, 4]])
```

Unless the filename attribute is specified, data is not saved out to file.

```
>>> data = CS.download(jobid='359750704009965',filename='/Users/uname/Desktop/test.csv',format='csv')
|=========================================================================================================================
↪1.5k/1.5k (100.00%)         0s
```

Other formats include votable, votableb1, and votableb2 (the latter two are binary files, for easier handling of large data sets; these formats can not be used in an interactive python session).

Data can be stored and/or written out as a VOTable.

```
>>> data = CS.download(jobid='359750704009965',format='votable')
>>> data
<astropy.io.votable.tree.VOTableFile at 0x10b440150>
>>> data = CS.download(jobid='359750704009965',filename='/Users/uname/Desktop/test.xml',format='votable
↪')
>>>␣
↪|=======================================================================================================================
↪4.9k/4.9k (100.00%)         0s
```

## 38.6 Reference/API

### 38.6.1 astroquery.cosmosim Package

#### CosmoSim Database Query Tool

**Revision History**

Access to all cosmological simulations stored in the CosmoSim database, via the uws service.

http://www.cosmosim.org/uws/query

**Author**
Austen M. Groener <Austen.M.Groener@drexel.edu>

#### Classes

| | |
|---|---|
| `CosmoSimClass`() | |
| `Conf` | Configuration parameters for `astroquery.cosmosim`. |

#### CosmoSimClass

**class** astroquery.cosmosim.**CosmoSimClass**
Bases: `astroquery.query.QueryWithLogin`

##### Attributes Summary

| |
|---|
| `QUERY_URL` |
| `SCHEMA_URL` |
| `TIMEOUT` |
| `USERNAME` |

##### Methods Summary

| | |
|---|---|
| `abort_job`([jobid]) | |
| `check_all_jobs`([phase, regex, sortby]) | Public function which builds a dictionary whose keys are each jobid for a given set of user credentials and whose values are the phase status (e.g. |
| `check_job_status`([jobid]) | A public function which sends an http GET request for a given jobid, and checks the server status. |
| `check_login_status`() | Public function which checks the status of a user login attempt. |
| `completed_job_info`([jobid, output]) | A public function which sends an http GET request for a given jobid with phase COMPLETED. |

Continued on next page

Table 3 – continued from previous page

| | |
|---|---|
| delete_all_jobs([phase, regex]) | A public function which deletes any/all jobs from the server in any phase and/or with its tablename matching any desired regular expression. |
| delete_job([jobid, squash]) | A public function which deletes a stored job from the server in any phase. |
| download([jobid, filename, format, cache]) | A public function to download data from a job with COMPLETED phase. |
| explore_db([db, table, col]) | A public function which allows for the exploration of any simulation and its tables within the database. |
| general_job_info([jobid, output]) | A public function which sends an http GET request for a given jobid in any phase. |
| logout([deletepw]) | Public function which allows the user to logout of their cosmosim credentials. |
| run_sql_query(query_string[, tablename, ...]) | Public function which sends a POST request containing the sql query string. |

### Attributes Documentation

**QUERY_URL = 'http://www.cosmosim.org/uws/query'**

**SCHEMA_URL = 'http://www.cosmosim.org/query/account/databases/json'**

**TIMEOUT = 60.0**

**USERNAME = ''**

### Methods Documentation

**abort_job**(*jobid=None*)

**check_all_jobs**(*phase=None*, *regex=None*, *sortby=None*)

Public function which builds a dictionary whose keys are each jobid for a given set of user credentials and whose values are the phase status (e.g. - EXECUTING,COMPLETED,PENDING,ERROR).

**Parameters**

**phase** : list

A list of phase(s) of jobs to be checked on. If nothing provided, all are checked.

**regex** : string

A regular expression to match all tablenames to. Matching table names will be included. Note - Only tables/starttimes are associated with jobs which have phase COMPLETED.

**sortby** : string

An option to sort jobs (after phase and regex criteria have been taken into account) by either the execution start time (`starttime`), or by the table name (`'tablename'`).

**Returns**

**checkalljobs** : `Response` object

The requests response for the GET request for finding all existing jobs.

**check_job_status**(*jobid=None*)

A public function which sends an http GET request for a given jobid, and checks the server status. If no jobid is provided, it uses the most recent query (if one exists).

**Parameters**

**jobid** : string

The jobid of the sql query. If no jobid is given, it attempts to use the most recent job (if it exists in this session).

**Returns**

**result** : content of `Response` object

The requests response phase

**check_login_status**()

Public function which checks the status of a user login attempt.

**completed_job_info**(*jobid=None*, *output=False*)

A public function which sends an http GET request for a given jobid with phase COMPLETED. If output is True, the function prints a dictionary to the screen, while always generating a global dictionary `response_dict_current`. If no jobid is provided, a visual of all responses with phase COMPLETED is generated.

**Parameters**

**jobid** : string

The jobid of the sql query.

**output** : bool

Print output of response(s) to the terminal

**delete_all_jobs**(*phase=None*, *regex=None*)

A public function which deletes any/all jobs from the server in any phase and/or with its tablename matching any desired regular expression.

**Parameters**

**phase** : list

A list of job phases to be deleted. If nothing provided, all are deleted.

**regex** : string

A regular expression to match all tablenames to. Matching table names will be deleted.

**delete_job**(*jobid=None*, *squash=None*)

A public function which deletes a stored job from the server in any phase. If no jobid is given, it attempts to use the most recent job (if it exists in this session). If jobid is specified, then it deletes the corresponding job, and if it happens to match the existing current job, that variable gets deleted.

**Parameters**

**jobid** : string

The jobid of the sql query. If no jobid is given, it attempts to use the most recent job (if it exists in this session).

**output** : bool

Print output of response(s) to the terminal

**Returns**

**result** : list

A list of response object(s)

**download**(*jobid=None*, *filename=None*, *format=None*, *cache=True*)

A public function to download data from a job with COMPLETED phase.

**Parameters**

**jobid :**

Completed jobid to be downloaded

**filename** : str

If left blank, downloaded to the terminal. If specified, data is written out to file (directory can be included here).

**format** : str

The format of the data to be downloaded. Options are `'csv'`, `'votable'`, `'votableB1'`, and `'votableB2'`.

**cache** : bool

Whether to cache the data. By default, this is set to True.

**Returns**

**headers, data** : list, list

**explore_db**(*db=None*, *table=None*, *col=None*)

A public function which allows for the exploration of any simulation and its tables within the database. This function is meant to aid the user in constructing sql queries.

**Parameters**

**db** : string

The database to explore.

**table** : string

The table to explore.

**col** : string

The column to explore.

**general_job_info**(*jobid=None*, *output=False*)

A public function which sends an http GET request for a given jobid in any phase. If no jobid is provided, a summary of all jobs is generated.

**Parameters**

**jobid** : string

The jobid of the sql query.

**output** : bool

Print output of response(s) to the terminal

**logout**(*deletepw=False*)

Public function which allows the user to logout of their cosmosim credentials.

**Parameters**

**deletepw** : bool

A hard logout - delete the password to the associated username from the keychain. The default is True.

**run_sql_query**(*query_string*, *tablename=None*, *queue=None*, *mail=None*, *text=None*, *cache=True*)
Public function which sends a POST request containing the sql query string.

**Parameters**

**query_string** : string

The sql query to be sent to the CosmoSim.org server.

**tablename** : string

The name of the table for which the query data will be stored under. If left blank or if it already exists, one will be generated automatically.

**queue** : string

The short/long queue option. Default is short.

**mail** : string

The user's email address for receiving job completion alerts.

**text** : string

The user's cell phone number for receiving job completion alerts.

**cache** : bool

Whether to cache the query locally

**Returns**

**result** : jobid

The jobid of the query

## Conf

**class** astroquery.cosmosim.**Conf**
Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.cosmosim`.

### Attributes Summary

| | |
|---|---|
| `query_url` | CosmoSim UWS query URL. |
| `schema_url` | CosmoSim json query URL for generating database schema. |
| `timeout` | Timeout for CosmoSim query. |
| `username` | Optional default username for CosmoSim database. |

### Attributes Documentation

**query_url**
CosmoSim UWS query URL.

**schema_url**
CosmoSim json query URL for generating database schema.

**timeout**
Timeout for CosmoSim query.

`username`

Optional default username for CosmoSim database.

CHAPTER 39

# HITRAN Queries (astroquery.hitran)

## 39.1 Getting started

This module provides an interface to the HITRAN database API. It can download a data file including transitions for a particular molecule in a given wavenumber range. The file is downloaded in the default cache directory `~/.astropy/cache/astroquery/hitran` and can be opened with a reader function that returns a table of spectral lines including all accessible parameters.

## 39.2 Examples

This will download all transitions of the main isotopologue of water between the wavenumbers of 3400 and 4100 $cm^{-1}$.

```
>>> import os
>>> from astroquery.hitran import read_hitran_file, cache_location, download_hitran
>>> download_hitran(1, 1, 3400, 4100)
>>> tbl = read_hitran_file(os.path.join(cache_location, 'H2O.data'))
```

Transitions are returned as an `Table` instance.

## 39.3 Reference/API

### 39.3.1 astroquery.hitran Package

#### HITRAN Catalog Query Tool

**Author**

Adam Ginsburg (adam.g.ginsburg@gmail.com)

### Functions

| | |
|---|---|
| download_hitran(m, i, numin, numax) | Download HITRAN data for a particular molecule. |
| read_hitran_file(filename[, formats, formatfile]) | |

#### download_hitran

astroquery.hitran.**download_hitran**(*m*, *i*, *numin*, *numax*)

Download HITRAN data for a particular molecule. Based on fetch function from hapi.py.

**Parameters**

**m** : int

HITRAN molecule number

**i** : int

HITRAN isotopologue number

**numin** : real

lower wavenumber bound

**numax** : real

upper wavenumber bound

#### read_hitran_file

astroquery.hitran.**read_hitran_file**(*filename*, *formats=None*, *formatfile='/home/docs/checkouts/readthedocs.org/user_builds/ast packages/astroquery-0.3.7-py3.6.egg/astroquery/hitran/data/readme.txt'*)

# CHAPTER 40

## NASA Exoplanet Archive (astroquery.nasa_exoplanet_archive)

### 40.1 Accessing the planet table

You can access the complete tables from each table source, with units assigned to columns wherever possible.

```
>>> from astroquery.nasa_exoplanet_archive import NasaExoplanetArchive
>>> exoplanet_archive_table = NasaExoplanetArchive.get_confirmed_planets_table()

>>> exoplanet_archive_table[:2]
<Table masked=True length=2>
pl_hostname pl_letter pl_discmethod ... pl_nnotes rowupdate  NAME_LOWERCASE
                                    ...
   str27       str1       str29     ...   int64      str10        str29
----------- --------- ------------- ... --------- ---------- --------------
 Kepler-151         b       Transit ...         1 2014-05-14   kepler-151 b
 Kepler-152         b       Transit ...         1 2014-05-14   kepler-152 b
```

You can query for the row from each table corresponding to one exoplanet:

```
>>> from astroquery.nasa_exoplanet_archive import NasaExoplanetArchive
>>> hatp11b = NasaExoplanetArchive.query_planet('HAT-P-11 b')
```

### 40.2 Properties of a particular planet

The properties of each planet are stored in a table, with columns defined by the NASA Explanet Archive. There is also a special column of sky coordinates for each target, named sky_coord.

```
>>> from astroquery.nasa_exoplanet_archive import NasaExoplanetArchive
>>> hatp11b = NasaExoplanetArchive.query_planet('HAT-P-11 b')

>>> hatp11b['pl_orbper']  # Planet period
```

(continues on next page)

(continued from previous page)

```
<Quantity 4.8878162 d>

>>> hatp11b['pl_radj']  # Planet radius
<Quantity 0.422 jupiterRad>

>>> hatp11b['sky_coord'] # Position of host star
<SkyCoord (ICRS): (ra, dec) in deg
    ( 297.709351,  48.080856)>
```

## 40.3 Reference/API

### 40.3.1 astroquery.nasa_exoplanet_archive Package

#### NASA Exoplanet Archive Query Tool

**Author**

Brett M. Morris (brettmorris21@gmail.com)

#### Classes

| | |
|---|---|
| NasaExoplanetArchiveClass() | Exoplanet Archive querying object. |
| Conf | Configuration parameters for astroquery.nasa_exoplanet_archive. |

#### NasaExoplanetArchiveClass

**class** astroquery.nasa_exoplanet_archive.**NasaExoplanetArchiveClass**

Bases: object

Exoplanet Archive querying object. Use the get_confirmed_planets_table or query_planet methods to get information about exoplanets via the NASA Exoplanet Archive.

##### Attributes Summary

| |
|---|
| param_units |

##### Methods Summary

| | |
|---|---|
| get_confirmed_planets_table([cache, . . . ]) | Download (and optionally cache) the NExScI Exoplanet Archive Confirmed Planets table. |
| query_planet(planet_name[, table_path]) | Get table of exoplanet properties. |

#### Attributes Documentation

**param_units**

#### Methods Documentation

**get_confirmed_planets_table**(*cache=True*, *show_progress=True*, *table_path=None*)

Download (and optionally cache) the NExScI Exoplanet Archive Confirmed Planets table.

The Exoplanet Archive table returns lots of columns of data. A full description of the columns can be found here

**Parameters**

**cache** : bool (optional)

Cache exoplanet table to local astropy cache? Default is `True`.

**show_progress** : bool (optional)

Show progress of exoplanet table download (if no cached copy is available). Default is `True`.

**table_path** : str (optional)

Path to a local table file. Default `None` will trigger a download of the table from the internet.

**Returns**

**——-**

**table** : `QTable`

Table of exoplanet properties.

**query_planet**(*planet_name*, *table_path=None*)

Get table of exoplanet properties.

**Parameters**

**planet_name** : str

Name of planet

**table_path** : str (optional)

Path to a local table file. Default `None` will trigger a download of the table from the internet.

**Return**

**——**

**table** : `QTable`

Table of one exoplanet's properties.

### Conf

**class** astroquery.nasa_exoplanet_archive.**Conf**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for astroquery.nasa_exoplanet_archive.

CHAPTER 41

---

# Exoplanet Orbit Database (astroquery.exoplanet_orbit_database)

---

## 41.1 Accessing the planet table

You can access the complete tables from each table source, with units assigned to columns wherever possible.

```
>>> from astroquery.exoplanet_orbit_database import ExoplanetOrbitDatabase
>>> eod_table = ExoplanetOrbitDatabase.get_confirmed_planets_table()

>>> eod_table[:2]
<Table masked=True length=2>
pl_hostname pl_letter pl_discmethod ... pl_nnotes rowupdate  NAME_LOWERCASE
                                    ...
   str27       str1       str29     ...   int64     str10        str29
----------- --------- ------------- ... --------- ---------- --------------
 Kepler-151         b       Transit ...         1 2014-05-14   kepler-151 b
 Kepler-152         b       Transit ...         1 2014-05-14   kepler-152 b
```

You can query for the row from each table corresponding to one exoplanet:

```
>>> from astroquery.exoplanet_orbit_database import ExoplanetOrbitDatabase
>>> hatp11b = ExoplanetOrbitDatabase.query_planet('HAT-P-11 b')
```

## 41.2 Properties of a particular planet

The properties of each planet are stored in a table, with columns defined by the Exoplanet Orbit Database. There is also a special column of sky coordinates for each target, named sky_coord.

```
>>> from astroquery.exoplanet_orbit_database import ExoplanetOrbitDatabase
>>> hatp11b = ExoplanetOrbitDatabase.query_planet('HAT-P-11 b')

>>> hatp11b['PER']  # Planet period
```

(continues on next page)

(continued from previous page)

```
<Quantity 4.8878162 d>

>>> hatp11b['R']  # Planet radius
<Quantity 0.422 jupiterRad>

>>> hatp11b['sky_coord'] # Position of host star
<SkyCoord (ICRS): (ra, dec) in deg
    ( 297.70891666,  48.08029444)>
```

## 41.3 Reference/API

### 41.3.1 astroquery.exoplanet_orbit_database Package

#### Exoplanet Orbit Database Query Tool

**Author**
Brett M. Morris (brettmorris21@gmail.com)

#### Classes

| | |
|---|---|
| ExoplanetOrbitDatabaseClass() | Exoplanet Orbit Database querying object. |
| Conf | Configuration parameters for astroquery.exoplanet_orbit_database. |

#### ExoplanetOrbitDatabaseClass

**class** astroquery.exoplanet_orbit_database.**ExoplanetOrbitDatabaseClass**
Bases: object

Exoplanet Orbit Database querying object. Use the get_table or query_planet methods to get information about exoplanets via the Exoplanet Orbit Database.

##### Attributes Summary

| |
|---|
| param_units |

##### Methods Summary

| | |
|---|---|
| get_table([cache, show_progress, table_path]) | Download (and optionally cache) the Exoplanet Orbit Database planets table. |
| query_planet(planet_name[, table_path]) | Get table of exoplanet properties. |

#### Attributes Documentation

**param_units**

#### Methods Documentation

**get_table**(*cache=True*, *show_progress=True*, *table_path=None*)

Download (and optionally cache) the Exoplanet Orbit Database planets table.

**Parameters**

**cache** : bool (optional)

Cache exoplanet table to local astropy cache? Default is `True`.

**show_progress** : bool (optional)

Show progress of exoplanet table download (if no cached copy is available). Default is `True`.

**table_path** : str (optional)

Path to a local table file. Default `None` will trigger a download of the table from the internet.

**Returns**

**table** : `QTable`

Table of exoplanet properties.

**query_planet**(*planet_name*, *table_path=None*)

Get table of exoplanet properties.

**Parameters**

**planet_name** : str

Name of planet

**table_path** : str (optional)

Path to a local table file. Default `None` will trigger a download of the table from the internet.

**Returns**

**table** : `QTable`

Table of one exoplanet's properties.

### Conf

**class** `astroquery.exoplanet_orbit_database.`**`Conf`**

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astroquery.exoplanet_orbit_database`.

# Part VI

# Catalog, Archive, and Other

A second index of the services by the type of data they serve. Some services perform many tasks and are listed more than once.

# CHAPTER 42

## Catalogs

The first serve catalogs, which generally return one row of information for each source (though they may return many catalogs that *each* have one row for each source)

# CHAPTER 43

# Archives

Archive services provide data, usually in FITS images or spectra. They will generally return a table listing the available data first.

# CHAPTER 44

# Simulations

Simulation services query databases of simulated or synthetic data

CHAPTER 45

# Other

There are other astronomically significant services, e.g. line list and atomic/molecular cross section and collision rate services, that don't fit the above categories.

## 45.1 TAP/TAP+ (`astroquery.utils.tap`)

Table Access Protocol (TAP: http://www.ivoa.net/documents/TAP/) specified by the International Virtual Observatory Alliance (IVOA: http://www.ivoa.net) defines a service protocol for accessing general table data.

TAP+ is the ESAC Space Data Centre (ESDC: http://www.cosmos.esa.int/web/esdc/) extension of the Table Access Protocol.

The TAP query language is Astronomical Data Query Language (ADQL: http://www.ivoa.net/documents/ADQL/2.0), which is similar to Structured Query Language (SQL), widely used to query databases.

TAP provides two operation modes: Synchronous and Asynchronous:

- Synchronous: the response to the request will be generated as soon as the request received by the server. (Do not use this method for queries that generate a big amount of results.)
- Asynchronous: the server will start a job that will execute the request. The first response to the request is the required information (a link) to obtain the job status. Once the job is finished, the results can be retrieved.

TAP+ is fully compatible with TAP specification. TAP+ adds more capabilities like authenticated access and persistent user storage area.

Please, check methods documentation to determine whether a method is TAP compatible.

### 45.1.1 Examples

#### 1. Non authenticated access

#### 1.1 Getting public tables

To load only table names (TAP+ capability)

```
>>> from astroquery.utils.tap.core import TapPlus
>>>
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> tables = gaia.load_tables(only_names=True)
>>> for table in (tables):
>>>   print(table.get_qualified_name())

public.dual
public.tycho2
public.igsl_source
public.hipparcos
public.hipparcos_newreduction
public.hubble_sc
public.igsl_source_catalog_ids
tap_schema.tables
tap_schema.keys
tap_schema.columns
tap_schema.schemas
tap_schema.key_columns
gaiadr1.phot_variable_time_series_gfov
gaiadr1.ppmxl_neighbourhood
gaiadr1.gsc23_neighbourhood
gaiadr1.ppmxl_best_neighbour
gaiadr1.sdss_dr9_neighbourhood
...
gaiadr1.tgas_source
gaiadr1.urat1_original_valid
gaiadr1.allwise_original_valid
```

To load table names (TAP compatible)

```
>>> from astroquery.utils.tap.core import TapPlus
>>>
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> tables = gaia.load_tables()
>>> for table in (tables):
>>>   print(table.get_qualified_name())

public.dual
public.tycho2
public.igsl_source
public.hipparcos
public.hipparcos_newreduction
public.hubble_sc
public.igsl_source_catalog_ids
tap_schema.tables
tap_schema.keys
tap_schema.columns
tap_schema.schemas
```

(continues on next page)

(continued from previous page)

```
tap_schema.key_columns
gaiadr1.phot_variable_time_series_gfov
gaiadr1.ppmxl_neighbourhood
gaiadr1.gsc23_neighbourhood
gaiadr1.ppmxl_best_neighbour
gaiadr1.sdss_dr9_neighbourhood
...
gaiadr1.tgas_source
gaiadr1.urat1_original_valid
gaiadr1.allwise_original_valid
```

To load only a table (TAP+ capability)

```
>>> from astroquery.utils.tap.core import TapPlus
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> table = gaia.load_table('gaiadr1.gaia_source')
>>> print(table)

Table name: gaiadr1.gaia_source
Description: This table has an entry for every Gaia observed source as listed in the
Main Database accumulating catalogue version from which the catalogue
release has been generated. It contains the basic source parameters,
that is only final data (no epoch data) and no spectra (neither final
nor epoch).
Num. columns: 57
```

Once a table is loaded, columns can be inspected

```
>>> from astroquery.utils.tap.core import TapPlus
>>>
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> table = gaia.load_table('gaiadr1.gaia_source')
>>> for column in (gaiadr1_table.get_columns()):
>>>   print(column.get_name())

solution_id
source_id
random_index
ref_epoch
ra
ra_error
dec
dec_error
...
ecl_lon
ecl_lat
```

## 1.2 Synchronous query

A synchronous query will not store the results at server side. These queries must be used when the amount of data to be retrieve is 'small'.

There is a limit of 2000 rows. If you need more than that, you must use asynchronous queries.

The results can be saved in memory (default) or in a file.

Query without saving results in a file:

```
>>> from astroquery.utils.tap.core import TapPlus
>>>
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>>
>>> job = gaia.launch_job("select top 100 \
>>> solution_id,ref_epoch,ra_dec_corr,astrometric_n_obs_al,matched_observations,duplicated_source,phot_
↪variable_flag \
>>> from gaiadr1.gaia_source order by source_id")
>>>
>>> print(job)

Jobid: None
Phase: COMPLETED
Owner: None
Output file: sync_20170223111452.xml.gz
Results: None

>>> r = job.get_results()
>>> print(r['solution_id'])

  solution_id
-------------------
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
                ...
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
Length = 100 rows
```

Query saving results in a file:

```
>>> from astroquery.utils.tap.core import TapPlus
>>>
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> job = gaia.launch_job("select top 100 \
>>> solution_id,ref_epoch,ra_dec_corr,astrometric_n_obs_al,matched_observations,duplicated_source,phot_
↪variable_flag \
>>> from gaiadr1.gaia_source order by source_id", dump_to_file=True)
>>>
>>> print(job)
```

(continues on next page)

(continued from previous page)

```
Jobid: None
Phase: COMPLETED
Owner: None
Output file: sync_20170223111452.xml.gz
Results: None

>>> r = job.get_results()
>>> print(r['solution_id'])

  solution_id
-------------------
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
                ...
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
Length = 100 rows
```

## 1.3 Synchronous query on an 'on-the-fly' uploaded table

A table can be uploaded to the server in order to be used in a query.

```
>>> from astroquery.utils.tap.core import TapPlus
>>>
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>>
>>> upload_resource = 'my_table.xml'
>>> j = gaia.launch_job(query="select * from tap_upload.table_test", upload_resource=upload_resource, \
>>> upload_table_name="table_test", verbose=True)
>>> r = j.get_results()
>>> r.pprint()

source_id alpha delta
--------- ----- -----
        a   1.0   2.0
        b   3.0   4.0
        c   5.0   6.0
```

### 1.4 Asynchronous query

Asynchronous queries save results at server side. These queries can be accessed at any time. For anonymous users, results are kept for three days.

The results can be saved in memory (default) or in a file.

Query without saving results in a file:

```
>>> from astroquery.utils.tap.core import TapPlus
>>>
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> job = gaia.launch_job_async("select top 100 * from gaiadr1.gaia_source order by source_id")
>>>
>>> print(job)

Jobid: 1487845273526O
Phase: COMPLETED
Owner: None
Output file: async_20170223112113.vot
Results: None

>>> r = job.get_results()
>>> print(r['solution_id'])

  solution_id
-------------------
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
                ...
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
Length = 100 rows
```

Query saving results in a file:

```
>>> from astroquery.utils.tap.core import TapPlus
>>>
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> job = gaia.launch_job_async("select top 100 * from gaiadr1.gaia_source order by source_id", dump_to_
↪file=True)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> print(job)

Jobid: 1487845273526O
Phase: COMPLETED
Owner: None
Output file: async_20170223112113.vot
Results: None

>>> r = job.get_results()
>>> print(r['solution_id'])

  solution_id
-------------------
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
                ...
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
1635378410781933568
Length = 100 rows
```

### 1.5 Asynchronous job removal

To remove asynchronous

```
>>> from astroquery.utils.tap.core import TapPlus
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> job = gaia.remove_jobs(["job_id_1","job_id_2",...])
```

## 2. Authenticated access (TAP+ only)

Authenticated users are able to access to TAP+ capabilities (shared tables, persistent jobs, etc.) In order to authenticate a user, `login` or `login_gui` methods must be called. After a successful authentication, the user will be authenticated until `logout` method is called.

All previous methods (`query_object`, `cone_search`, `load_table`, `load_tables`, `launch_job`) explained for non authenticated users are applicable for authenticated ones.

The main differences are:

- Asynchronous results are kept at server side for ever (until the user decides to remove one of them).
- Users can access to shared tables.

## 2.1. Login/Logout

Graphic interface

*Note: Tkinter module is required to use login_gui method.*

```
>>> from astroquery.utils.tap.core import TapPlus
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> gaia.login_gui()
```

Command line

```
>>> from astroquery.utils.tap.core import TapPlus
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> gaia.login(user='userName', password='userPassword')
```

It is possible to use a file where the credentials are stored:

*The file must containing user and password in two different lines.*

```
>>> from astroquery.utils.tap.core import TapPlus
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> gaia.login(credentials_file='my_credentials_file')
```

To perform a logout

```
>>> from astroquery.utils.tap.core import TapPlus
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> gaia.login(credentials_file='my_credentials_file')
>>> ...
>>>
>>> gaia.logout()
```

## 2.2. Listing shared tables

```
>>> from astroquery.utils.tap.core import TapPlus
>>> gaia = TapPlus(url="http://gea.esac.esa.int/tap-server/tap")
>>> gaia.login(credentials_file='my_credentials_file')
>>> tables = gaia.load_tables(only_names=True, include_shared_tables=True)
>>> for table in (tables):
>>>   print(table.get_qualified_name())

public.dual
public.tycho2
public.igsl_source
tap_schema.tables
tap_schema.keys
tap_schema.columns
tap_schema.schemas
tap_schema.key_columns
```

(continues on next page)

(continued from previous page)

```
gaiadr1.phot_variable_time_series_gfov
gaiadr1.ppmxl_neighbourhood
gaiadr1.gsc23_neighbourhood
...
user_schema_1.table1
user_schema_2.table1
...
```

## 3. Using TAP+ to connect other TAP services

TAP+ can be used to connect other TAP services.

Example 1: TAPVizieR.u-strasbg.fr

```
>>> from astroquery.utils.tap.core import TapPlus
>>> tap = TapPlus(url="http://TAPVizieR.u-strasbg.fr/TAPVizieR/tap")
>>> #Inspect tables
>>> tables = tap.load_tables()
>>> for table in (tables):
>>>   print(table.get_name())

...
J/ApJS/173/104/memb
J/A+A/376/441/table1
J/A+AS/110/81/table2
J/ApJS/73/781/snr_indx
V/15/notes
J/A+AS/115/285/refs
J/ApJS/165/338/table1
IX/24/obsnames
J/A+AS/122/463/tab2-14
J/ApJS/107/521/table1
J/MNRAS/275/1102/table1a
J/ApJ/647/328/table4
J/A+A/402/1/table1a
J/AJ/115/1856/v12
...

>>> #Launch sync job
>>> job = tap.launch_job("SELECT top 10 * from " + tables[0].get_name())
>>> r = job.get_results()
>>> r.pprint()

                       title                           class [1] ... comment
----------------------------------------------------- --------- ... -------
The 2MASS Point Source and 2MASS6x catalogues (2003)         2 ...
        The 2MASS Extended Source Catalogue (2003)           2 ...
     Astrographic catalog (mean epoch around 1900)           2 ...
AKARI IRC (9/18um) and FIS (60-160um)all-sky Surveys         2 ...
         All-Sky Compiled Catalog of 2.5M*  (2003)           2 ...
     The DENIS database (3rd Release 2005 version)           2 ...
   The Carlsberg Meridian Catalog 14 (-30<Dec<+50)           2 ...
         GALEX-DR5 sources from AIS and MIS (2011)           2 ...
       Spitzer's GLIMPSE catalogs (Galactic Plane)           2 ...
 The HST Guide Star Catalog reduced on Tycho (ACT)           2 ...
Example 2: irsa.ipac.caltech.edu
```

```
>>> from astroquery.utils.tap.core import TapPlus
>>> tap = TapPlus(url="http://irsa.ipac.caltech.edu/TAP")
>>> job = tap.launch_job_async("SELECT TOP 10 * FROM fp_psc")
>>> r = job.get_results()
>>> r.pprint()

   name      dtype   unit format n_bad
------------- ------- ----- ------ -----
       cntr   int32                   0
      hemis  object                   0
      xdate  object                   0
       scan   int32                   0
         id   int32                   0
         ra float64   deg     %r      0
        dec float64   deg     %r      0
       glon float64   deg     %r      0
       glat float64   deg     %r      0
          x float64           %r      0
          y float64           %r      0
          z float64           %r      0
    err_maj float64  arcs     %r      0
    err_min float64  arcs     %r      0
    err_ang   int32   deg             0
     x_scan float64  arcs     %r      0
     y_scan float64  arcs     %r      0
...
```

Please, check methods documentation to determine whether a method is TAP compatible.

### 45.1.2 Reference/API

#### astroquery.utils.tap Package

@author: Juan Carlos Segovia @contact: juan.carlos.segovia@sciops.esa.int

European Space Astronomy Centre (ESAC) European Space Agency (ESA)

Created on 30 jun. 2016

#### Classes

| | |
|---|---|
| Tap([url, host, server_context, . . . ]) | TAP class Provides TAP capabilities |
| TapPlus([url, host, server_context, . . . ]) | TAP plus class Provides TAP and TAP+ capabilities |
| TapTableMeta() | TAP table metadata object |
| TapColumn() | TAP column object |

#### Tap

**class** astroquery.utils.tap.**Tap**(*url=None*, *host=None*, *server_context=None*, *tap_context=None*, *port=80*, *sslport=443*, *default_protocol_is_https=False*, *connhandler=None*, *verbose=False*)

Bases: object

TAP class Provides TAP capabilities

Constructor

**Parameters**

**url** : str, mandatory if no host is specified, default None

TAP URL

**host** : str, optional, default None

host name

**server_context** : str, optional, default None

server context

**tap_context** : str, optional, default None

tap context

**port** : int, optional, default '80'

HTTP port

**sslport** : int, optional, default '443'

HTTPS port

**default_protocol_is_https** : bool, optional, default False

Specifies whether the default protocol to be used is HTTPS

**connhandler connection handler object, optional, default None**

HTTP(s) connection hander (creator). If no handler is provided, a new one is created.

**verbose** : bool, optional, default 'False'

flag to display information about the process

### Methods Summary

| | |
|---|---|
| `launch_job`(query[, name, output_file, ...]) | Launches a synchronous job |
| `launch_job_async`(query[, name, output_file, ...]) | Launches an asynchronous job |
| `list_async_jobs`([verbose]) | Returns all the asynchronous jobs |
| `load_async_job`([jobid, name, verbose]) | Loads an asynchronous job |
| `load_tables`([verbose]) | Loads all public tables |
| `save_results`(job[, verbose]) | Saves job results |

### Methods Documentation

**launch_job**(*query*, *name=None*, *output_file=None*, *output_format='votable'*, *verbose=False*, *dump_to_file=False*, *upload_resource=None*, *upload_table_name=None*)

Launches a synchronous job

**Parameters**

**query** : str, mandatory

query to be executed

**output_file** : str, optional, default None

file name where the results are saved if dumpToFile is True. If this parameter is not provided, the jobid is used instead

**output_format** : str, optional, default 'votable'

results format

**verbose** : bool, optional, default 'False'

flag to display information about the process

**dump_to_file** : bool, optional, default 'False'

if True, the results are saved in a file instead of using memory

**upload_resource: str, optional, default None**

resource to be uploaded to UPLOAD_SCHEMA

**upload_table_name: str, required if uploadResource is provided, default None**

resource temporary table name associated to the uploaded resource

**Returns**

A Job object

`launch_job_async`(*query*, *name=None*, *output_file=None*, *output_format='votable'*, *verbose=False*, *dump_to_file=False*, *background=False*, *upload_resource=None*, *upload_table_name=None*)

Launches an asynchronous job

**Parameters**

**query** : str, mandatory

query to be executed

**output_file** : str, optional, default None

file name where the results are saved if dumpToFile is True. If this parameter is not provided, the jobid is used instead

**output_format** : str, optional, default 'votable'

results format

**verbose** : bool, optional, default 'False'

flag to display information about the process

**dump_to_file** : bool, optional, default 'False'

if True, the results are saved in a file instead of using memory

**background** : bool, optional, default 'False'

when the job is executed in asynchronous mode, this flag specifies whether the execution will wait until results are available

**upload_resource: str, optional, default None**

resource to be uploaded to UPLOAD_SCHEMA

**upload_table_name: str, required if uploadResource is provided, default None**

resource temporary table name associated to the uploaded resource

**Returns**

A Job object

**list_async_jobs**(*verbose=False*)
Returns all the asynchronous jobs

**Parameters**
**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**
A list of Job objects

**load_async_job**(*jobid=None*, *name=None*, *verbose=False*)
Loads an asynchronous job

**Parameters**
**jobid** : str, mandatory if no name is provided, default None

job identifier

**name** : str, mandatory if no jobid is provided, default None

job name

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**
A Job object

**load_tables**(*verbose=False*)
Loads all public tables

**Parameters**
**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**
A list of table objects

**save_results**(*job*, *verbose=False*)
Saves job results

**Parameters**
**job** : Job, mandatory

job

**verbose** : bool, optional, default 'False'

flag to display information about the process

### TapPlus

**class** astroquery.utils.tap.**TapPlus**(*url=None*, *host=None*, *server_context=None*, *tap_context=None*, *port=80*, *sslport=443*, *default_protocol_is_https=False*, *connhandler=None*, *verbose=True*)
Bases: astroquery.utils.tap.Tap

TAP plus class Provides TAP and TAP+ capabilities

Constructor

**Parameters**

**url** : str, mandatory if no host is specified, default None

TAP URL

**host** : str, optional, default None

host name

**server_context** : str, optional, default None

server context

**tap_context** : str, optional, default None

tap context

**port** : int, optional, default '80'

HTTP port

**sslport** : int, optional, default '443'

HTTPS port

**default_protocol_is_https** : bool, optional, default False

Specifies whether the default protocol to be used is HTTPS

**connhandler connection handler object, optional, default None**

HTTP(s) connection hander (creator). If no handler is provided, a new one is created.

**verbose** : bool, optional, default 'True'

flag to display information about the process

### Methods Summary

| | |
|---|---|
| `load_table`(table[, verbose]) | Loads the specified table |
| `load_tables`([only_names, . . . ]) | Loads all public tables |
| `login`([user, password, credentials_file, . . . ]) | Performs a login. |
| `login_gui`([verbose]) | Performs a login using a GUI dialog |
| `logout`([verbose]) | Performs a logout |
| `remove_jobs`(jobs_list[, verbose]) | Removes the specified jobs |
| `search_async_jobs`([jobfilter, verbose]) | Searches for jobs applying the specified filter |

### Methods Documentation

**load_table**(*table*, *verbose=False*)

Loads the specified table

**Parameters**

**table** : str, mandatory

full qualified table name (i.e. schema name + table name)

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**

A table object

**load_tables**(*only_names=False*, *include_shared_tables=False*, *verbose=False*)
Loads all public tables

**Parameters**
**only_names** : bool, TAP+ only, optional, default 'False'

True to load table names only

**include_shared_tables** : bool, TAP+, optional, default 'False'

True to include shared tables

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**
A list of table objects

**login**(*user=None*, *password=None*, *credentials_file=None*, *verbose=False*)
Performs a login. User and password can be used or a file that contains user name and password (2 lines: one for user name and the following one for the password)

**Parameters**
**user** : str, mandatory if 'file' is not provided, default None

login name

**password** : str, mandatory if 'file' is not provided, default None

user password

**credentials_file** : str, mandatory if no 'user' & 'password' are provided

file containing user and password in two lines

**verbose** : bool, optional, default 'False'

flag to display information about the process

**login_gui**(*verbose=False*)
Performs a login using a GUI dialog

**Parameters**
**verbose** : bool, optional, default 'False'

flag to display information about the process

**logout**(*verbose=False*)
Performs a logout

**Parameters**
**verbose** : bool, optional, default 'False'

flag to display information about the process

**remove_jobs**(*jobs_list*, *verbose=False*)
Removes the specified jobs

**Parameters**
**jobs_list** : str, mandatory

jobs identifiers to be removed

**verbose** : bool, optional, default 'False' flag to display information about the process

**search_async_jobs**(*jobfilter=None*, *verbose=False*)

Searches for jobs applying the specified filter

**Parameters**

**jobfilter** : JobFilter, optional, default None

job filter

**verbose** : bool, optional, default 'False'

flag to display information about the process

**Returns**

A list of Job objects

### TapTableMeta

**class** astroquery.utils.tap.**TapTableMeta**

Bases: `object`

TAP table metadata object

Constructor

#### Methods Summary

| | |
|---|---|
| `add_column`(tap_column) | Adds a table TAP column |
| `get_columns`() | Returns the TAP table columns |
| `get_description`() | Returns the TAP table description |
| `get_name`() | Returns the TAP table name |
| `get_qualified_name`() | Returns the qualified TAP table name. |
| `get_schema`() | Returns the TAP table schema name |
| `set_description`(description) | Sets the TAP table description |
| `set_name`(name) | Sets the TAP table name |
| `set_schema`(schema) | Sets the TAP table schema name |

#### Methods Documentation

**add_column**(*tap_column*)

Adds a table TAP column

**Parameters**

**tap_column** : TAP Column object, mandatory

table TAP column

**get_columns**()

Returns the TAP table columns

**Returns**

The TAP table columns (a list)

**get_description**()

Returns the TAP table description

**Returns**
The TAP table description

**get_name**()
Returns the TAP table name

**Returns**
The TAP table name

**get_qualified_name**()
Returns the qualified TAP table name. I.e. schema+table

**Returns**
The the qualified TAP table name (schema+table)

**get_schema**()
Returns the TAP table schema name

**Returns**
The TAP table schema name

**set_description**(*description*)
Sets the TAP table description

**Parameters**
**description** : str, mandatory

TAP table description

**set_name**(*name*)
Sets the TAP table name

**Parameters**
**name** : str, mandatory

TAP table name

**set_schema**(*schema*)
Sets the TAP table schema name

**Parameters**
**schema** : str, mandatory

TAP table schema name

## TapColumn

**class** astroquery.utils.tap.**TapColumn**
Bases: `object`

TAP column object

Constructor

### Methods Summary

| | |
|---|---|
| `get_array_size`() | Returns the TAP column data array size |
| `get_data_type`() | Returns the TAP column data type |
| `get_description`() | Returns the TAP column description |

Continued on next page

Table 5 – continued from previous page

| | |
|---|---|
| get_flag() | Returns the TAP column flag (TAP+) |
| get_name() | Returns the TAP column name |
| get_ucd() | Returns the TAP column ucd |
| get_unit() | Returns the TAP column unit |
| get_utype() | Returns the TAP column utype |
| set_array_size(arraySize) | Sets the TAP column data array size |
| set_data_type(dataType) | Sets the TAP column data type |
| set_description(description) | Sets the TAP column description |
| set_flag(flag) | Sets the TAP column flag (TAP+) |
| set_name(name) | Sets the TAP column name |
| set_ucd(ucd) | Sets the TAP column ucd |
| set_unit(unit) | Sets the TAP column unit |
| set_utype(utype) | Sets the TAP column utype |

### Methods Documentation

**get_array_size()**
Returns the TAP column data array size

**Returns**
The TAP column data array size

**get_data_type()**
Returns the TAP column data type

**Returns**
The TAP column data type

**get_description()**
Returns the TAP column description

**Returns**
The TAP column description

**get_flag()**
Returns the TAP column flag (TAP+)

**Returns**
The TAP column flag

**get_name()**
Returns the TAP column name

**Returns**
The TAP column name

**get_ucd()**
Returns the TAP column ucd

**Returns**
The TAP column ucd

**get_unit()**
Returns the TAP column unit

**Returns**
The TAP column unit

**get_utype**()
Returns the TAP column utype

**Returns**
The TAP column utype

**set_array_size**(*arraySize*)
Sets the TAP column data array size

**Parameters**
**description** : str, mandatory

TAP column data array size

**set_data_type**(*dataType*)
Sets the TAP column data type

**Parameters**
**description** : str, mandatory

TAP column data type

**set_description**(*description*)
Sets the TAP column description

**Parameters**
**description** : str, mandatory

TAP column description

**set_flag**(*flag*)
Sets the TAP column flag (TAP+)

**Parameters**
**description** : str, mandatory

TAP column flag

**set_name**(*name*)
Sets the TAP column name

**Parameters**
**name** : str, mandatory

TAP column name

**set_ucd**(*ucd*)
Sets the TAP column ucd

**Parameters**
**description** : str, mandatory

TAP column ucd

**set_unit**(*unit*)
Sets the TAP column unit

**Parameters**
**description** : str, mandatory

TAP column unit

**set_utype**(*utype*)
Sets the TAP column utype

**Parameters**

**description** : str, mandatory

TAP column utype

# CHAPTER 46

# Developer documentation

The modules and their maintainers are listed on the Maintainers wiki page.

The *Astroquery API Specification* is intended to be kept as consistent as possible, such that any web service can be used with a minimal learning curve imposed on the user.

## 46.1 Astroquery API Specification

### 46.1.1 Service Class

The query tools will be implemented as class methods, so that the standard approach for a given web service (e.g., IRSA, UKIDSS, SIMBAD) will be

```
from astroquery.service import Service

result = Service.query_object('M 31')
```

for services that do not require login, and

```
from astroquery.service import Service

S = Service(user='username',password='password')
result = S.query_object('M 31')
```

for services that do.

#### Query Methods

The classes will have the following methods where appropriate:

```
query_object(objectname, ...)
query_region(coordinate, radius=, width=)
get_images(coordinate)
```

They may also have other methods for querying non-standard data types (e.g., ADS queries that may return a `bibtex` text block).

### query_object

`query_object` is only needed for services that are capable of parsing an object name (e.g., SIMBAD, Vizier, NED), otherwise `query_region` is an adequate approach, as any name can be converted to a coordinate via the SIMBAD name parser.

### query_region

Query a region around a coordinate.

One of these keywords *must* be specified (no default is assumed):

```
radius - an astropy Quantity object, or a string that can be parsed into one.
        e.g., '1 degree' or 1*u.degree.
        If radius is specified, the shape is assumed to be a circle
width - a Quantity.  Specifies the edge length of a square box
height - a Quantity.  Specifies the height of a rectangular box.  Must be passed with width.
```

Returns a `Table`.

### get_images

Perform a coordinate-based query to acquire images.

Returns a list of `HDUList` objects.

Shape keywords are optional - some query services allow searches for images that overlap with a specified coordinate.

### (query)_async

Includes `get_images_async`, `query_region_async`, `query_object_async`

Same as the above query tools, but returns a list of readable file objects instead of a parsed object so that the data is not downloaded until `result.get_data()` is run.

### Common Keywords

These keywords are common to all query methods:

```
return_query_payload - Return the POST data that will be submitted as a dictionary
savename - [optional - see discussion below] File path to save the downloaded query to
timeout - timeout in seconds
```

### 46.1.2 Asynchronous Queries

Some services require asynchronous query submission & download, e.g. Besancon, the NRAO Archive, the Fermi archive, etc. The data needs to be "staged" on the remote server before it can be downloaded. For these queries, the approach is

```
result = Service.query_region_async(coordinate)

data = result.get_data()
# this will periodically check whether the data is available at the specified URL
```

Additionally, any service can be queried asynchronously - `get_images_async` will return readable objects that can be downloaded at a later time.

### 46.1.3 Outline of an Example Module

Directory Structure:

```
module/
module/__init__.py
module/core.py
module/tests/test_module.py
```

`__init__.py` contains:

```
from astropy import config as _config

class Conf(_config.ConfigNamespace):
    """
    Configuration parameters for `astroquery.template_module`.
    """
    server = _config.ConfigItem(
        ['http://dummy_server_mirror_1',
         'http://dummy_server_mirror_2',
         'http://dummy_server_mirror_n'],
        'Name of the template_module server to use.'
        )
    timeout = _config.ConfigItem(
        30,
        'Time limit for connecting to template_module server.'
        )

from .core import QueryClass

__all__ = ['QueryClass']
```

`core.py` contains:

```
from ..utils.class_or_instance import class_or_instance
from ..utils import commons, async_to_sync

__all__ = ['QueryClass']  # specifies what to import

@async_to_sync
class QueryClass(astroquery.BaseQuery):
```

(continues on next page)

(continued from previous page)

```
    server = SERVER()

    def __init__(self, *args):
        """ set some parameters """
        # do login here
        pass

    @class_or_instance
    def query_region_async(self, *args, get_query_payload=False):

        request_payload = self._args_to_payload(*args)

        response = commons.send_request(self.server, request_payload, TIMEOUT)

        # primarily for debug purposes, but also useful if you want to send
        # someone a URL linking directly to the data
        if get_query_payload:
            return request_payload

        return response

    @class_or_instance
    def get_images_async(self, *args):
        image_urls = self.get_image_list(*args)
        return [get_readable_fileobj(U) for U in image_urls]
        # get_readable_fileobj returns need a "get_data()" method?

    @class_or_instance
    def get_image_list(self, *args):

        request_payload = self.args_to_payload(*args)

        result = requests.post(url, data=request_payload)

        return self.extract_image_urls(result)

    def _parse_result(self, result):
        # do something, probably with regexp's
        return astropy.table.Table(tabular_data)

    def _args_to_payload(self, *args):
        # convert arguments to a valid requests payload

        return dict
```

### 46.1.4 Parallel Queries

For multiple parallel queries logged in to the same object, you could do:

```
from astroquery.module import QueryClass

QC = QueryClass(login_information)

results = parallel_map(QC.query_object, ['m31', 'm51', 'm17'],
                       radius=['1"', '1"', '1"'])
```

(continues on next page)

(continued from previous page)

```
results = [QC.query_object_async(obj, radius=r)
           for obj,r in zip(['m31', 'm51', 'm17'], ['1"', '1"', '1"'])]
```

Here `parallel_map()` is a parallel implementation of some map function.

### 46.1.5 Exceptions

- What errors should be thrown if queries fail? Failed queries should raise a custom Exception that will include the full html (or xml) of the failure, but where possible should parse the web page's error message into something useful.
- How should timeouts be handled? Timeouts should raise a `TimeoutError`.

### 46.1.6 Examples

Standard usage should be along these lines:

```
from astroquery.simbad import Simbad

result = Simbad.query_object("M 31")
# returns astropy.Table object

from astroquery.irsa import Irsa

images = Irsa.get_images("M 31","5 arcmin")
# searches for images in a 5-arcminute circle around M 31
# returns list of HDU objects

images = Irsa.get_images("M 31")
# searches for images overlapping with the SIMBAD position of M 31, if supported by the service?
# returns list of HDU objects

from astroquery.ukidss import Ukidss

Ukidss.login(username, password)

result = Ukidss.query_region("5.0 0.0 gal", catalog='GPS')
# FAILS: no radius specified!
result = Ukidss.query_region("5.0 0.0 gal", catalog='GPS', radius=1)
# FAILS: no assumed units!
result = Ukidss.query_region("5.0 0.0 gal", catalog='GPS', radius='1 arcmin')
# SUCCEEDS!  returns an astropy.Table

from astropy.coordinates import SkyCoord
import astropy.units as u
result = Ukidss.query_region(
    SkyCoord(5,0,unit=('deg','deg'), frame='galactic'),
    catalog='GPS', region='circle', radius=5*u.arcmin)
# returns an astropy.Table

from astroquery.nist import Nist

hydrogen = Nist.query(4000*u.AA, 7000*u.AA, linename='H I', energy_level_unit='eV')
# returns an astropy.Table
```

For tools in which multiple catalogs can be queried, e.g. as in the UKIDSS examples, they must be specified. There should also be a `list_catalogs` function that returns a `list` of catalog name strings:

```
print(Ukidss.list_catalogs())
```

### Unparseable Data

If data cannot be parsed into its expected form (`Table`, `astropy.io.fits.PrimaryHDU`), the raw unparsed data will be returned and a `Warning` issued.

## 46.2 Template Module

The template module, written by Madhura Parikh, shows how to build your own module for a new online service.

```
# Licensed under a 3-clause BSD style license - see LICENSE.rst
from __future__ import print_function

# put all imports organized as shown below
# 1. standard library imports

# 2. third party imports
import astropy.units as u
import astropy.coordinates as coord
import astropy.io.votable as votable
from astropy.table import Table
from astropy.io import fits

# 3. local imports - use relative imports
# commonly required local imports shown below as example
# all Query classes should inherit from BaseQuery.
from ..query import BaseQuery
# has common functions required by most modules
from ..utils import commons
# prepend_docstr is a way to copy docstrings between methods
from ..utils import prepend_docstr_nosections
# async_to_sync generates the relevant query tools from _async methods
from ..utils import async_to_sync
# import configurable items declared in __init__.py
from . import conf


# export all the public classes and methods
__all__ = ['Template', 'TemplateClass']

# declare global variables and constants if any


# Now begin your main class
# should be decorated with the async_to_sync imported previously
@async_to_sync
class TemplateClass(BaseQuery):

    """
    Not all the methods below are necessary but these cover most of the common
```

(continues on next page)

(continued from previous page)

```
    cases, new methods may be added if necessary, follow the guidelines at
    <http://astroquery.readthedocs.io/en/latest/api.html>
    """
    # use the Configuration Items imported from __init__.py to set the URL,
    # TIMEOUT, etc.
    URL = conf.server
    TIMEOUT = conf.timeout

    # all query methods are implemented with an "async" method that handles
    # making the actual HTTP request and returns the raw HTTP response, which
    # should be parsed by a separate _parse_result method.   The query_object
    # method is created by async_to_sync automatically.  It would look like
    # this:
    """
    def query_object(object_name, get_query_payload=False)
        response = self.query_object_async(object_name,
                                           get_query_payload=get_query_payload)
        if get_query_payload:
            return response
        result = self._parse_result(response, verbose=verbose)
        return result
    """

    def query_object_async(self, object_name, get_query_payload=False,
                           cache=True):
        """
        This method is for services that can parse object names. Otherwise
        use :meth:`astroquery.template_module.TemplateClass.query_region`.
        Put a brief description of what the class does here.

        Parameters
        ----------
        object_name : str
            name of the identifier to query.
        get_query_payload : bool, optional
            This should default to False. When set to `True` the method
            should return the HTTP request parameters as a dict.
        verbose : bool, optional
           This should default to `False`, when set to `True` it displays
           VOTable warnings.
        any_other_param : <param_type>
            similarly list other parameters the method takes

        Returns
        -------
        response : `requests.Response`
            The HTTP response returned from the service.
            All async methods should return the raw HTTP response.

        Examples
        --------
        While this section is optional you may put in some examples that
        show how to use the method. The examples are written similar to
        standard doctests in python.

        """
        # the async method should typically have the following steps:
```

(continues on next page)

(continued from previous page)

```
        # 1. First construct the dictionary of the HTTP request params.
        # 2. If get_query_payload is `True` then simply return this dict.
        # 3. Else make the actual HTTP request and return the corresponding
        #    HTTP response
        # All HTTP requests are made via the `BaseQuery._request` method. This
        # use a generic HTTP request method internally, similar to
        # `requests.Session.request` of the Python Requests library, but
        # with added caching-related tools.

        # See below for an example:

        # first initialize the dictionary of HTTP request parameters
        request_payload = dict()

        # Now fill up the dictionary. Here the dictionary key should match
        # the exact parameter name as expected by the remote server. The
        # corresponding dict value should also be in the same format as
        # expected by the server. Additional parsing of the user passed
        # value may be required to get it in the right units or format.
        # All this parsing may be done in a separate private `_args_to_payload`
        # method for cleaner code.

        request_payload['object_name'] = object_name
        # similarly fill up the rest of the dict ...

        if get_query_payload:
            return request_payload
        # BaseQuery classes come with a _request method that includes a
        # built-in caching system
        response = self._request('GET', self.URL, params=request_payload,
                                 timeout=self.TIMEOUT, cache=cache)
        return response

    # For services that can query coordinates, use the query_region method.
    # The pattern is similar to the query_object method. The query_region
    # method also has a 'radius' keyword for specifying the radius around
    # the coordinates in which to search. If the region is a box, then
    # the keywords 'width' and 'height' should be used instead. The coordinates
    # may be accepted as an `astropy.coordinates` object or as a string, which
    # may be further parsed.

    # similarly we write a query_region_async method that makes the
    # actual HTTP request and returns the HTTP response

    def query_region_async(self, coordinates, radius, height, width,
                           get_query_payload=False, cache=True):
        """
        Queries a region around the specified coordinates.

        Parameters
        ----------
        coordinates : str or `astropy.coordinates`.
            coordinates around which to query
        radius : str or `astropy.units.Quantity`.
            the radius of the cone search
        width : str or `astropy.units.Quantity`
            the width for a box region
```

(continues on next page)

(continued from previous page)

```
        height : str or `astropy.units.Quantity`
            the height for a box region
        get_query_payload : bool, optional
            Just return the dict of HTTP request parameters.
        verbose : bool, optional
            Display VOTable warnings or not.

        Returns
        -------
        response : `requests.Response`
            The HTTP response returned from the service.
            All async methods should return the raw HTTP response.
        """
        request_payload = self._args_to_payload(coordinates, radius, height,
                                                width)
        if get_query_payload:
            return request_payload
        response = self._request('GET', self.URL, params=request_payload,
                                 timeout=self.TIMEOUT, cache=cache)
        return response

    # as we mentioned earlier use various python regular expressions, etc
    # to create the dict of HTTP request parameters by parsing the user
    # entered values. For cleaner code keep this as a separate private method:

    def _args_to_payload(self, *args, **kwargs):
        request_payload = dict()
        # code to parse input and construct the dict
        # goes here. Then return the dict to the caller
        return request_payload

    # the methods above call the private _parse_result method.
    # This should parse the raw HTTP response and return it as
    # an `astropy.table.Table`. Below is the skeleton:

    def _parse_result(self, response, verbose=False):
        # if verbose is False then suppress any VOTable related warnings
        if not verbose:
            commons.suppress_vo_warnings()
        # try to parse the result into an astropy.Table, else
        # return the raw result with an informative error message.
        try:
            # do something with regex to get the result into
            # astropy.Table form. return the Table.
            pass
        except ValueError:
            # catch common errors here, but never use bare excepts
            # return raw result/ handle in some way
            pass

        return Table()

    # Image queries do not use the async_to_sync approach: the "synchronous"
    # version must be defined explicitly.  The example below therefore presents
    # a complete example of how to write your own synchronous query tools if
    # you prefer to avoid the automatic approach.
    #
```

(continues on next page)

(continued from previous page)

```
    # For image queries, the results should be returned as a
    # list of `astropy.fits.HDUList` objects. Typically image queries
    # have the following method family:
    # 1. get_images - this is the high level method that interacts with
    #        the user. It reads in the user input and returns the final
    #        list of fits images to the user.
    # 2. get_images_async - This is a lazier form of the get_images function,
    #        in that it returns just the list of handles to the image files
    #        instead of actually downloading them.
    # 3. extract_image_urls - This takes in the raw HTTP response and scrapes
    #        it to get the downloadable list of image URLs.
    # 4. get_image_list - this is similar to the get_images, but it simply
    #        takes in the list of URLs scrapped by extract_image_urls and
    #        returns this list rather than the actual FITS images
    # NOTE : in future support may be added to allow the user to save
    # the downloaded images to a preferred location. Here we look at the
    # skeleton code for image services

    def get_images(self, coordinates, radius, get_query_payload):
        """
        A query function that searches for image cut-outs around coordinates

        Parameters
        ----------
        coordinates : str or `astropy.coordinates`.
            coordinates around which to query
        radius : str or `astropy.units.Quantity`.
            the radius of the cone search
        get_query_payload : bool, optional
            If true than returns the dictionary of query parameters, posted to
            remote server. Defaults to `False`.

        Returns
        -------
        A list of `astropy.fits.HDUList` objects
        """
        readable_objs = self.get_images_async(coordinates, radius,
                                              get_query_payload=get_query_payload)
        if get_query_payload:
            return readable_objs  # simply return the dict of HTTP request params
        # otherwise return the images as a list of astropy.fits.HDUList
        return [obj.get_fits() for obj in readable_objs]

    @prepend_docstr_nosections(get_images.__doc__)
    def get_images_async(self, coordinates, radius, get_query_payload=False):
        """
        Returns
        -------
        A list of context-managers that yield readable file-like objects
        """
        # As described earlier, this function should return just
        # the handles to the remote image files. Use the utilities
        # in commons.py for doing this:

        # first get the links to the remote image files
        image_urls = self.get_image_list(coordinates, radius,
                                         get_query_payload=get_query_payload)
```

(continues on next page)

(continued from previous page)

```
        if get_query_payload:  # if true then return the HTTP request params dict
            return image_urls
        # otherwise return just the handles to the image files.
        return [commons.FileContainer(U) for U in image_urls]

    # the get_image_list method, simply returns the download
    # links for the images as a list

    @prepend_docstr_nosections(get_images.__doc__)
    def get_image_list(self, coordinates, radius, get_query_payload=False,
                       cache=True):
        """
        Returns
        -------
        list of image urls
        """
        # This method should implement steps as outlined below:
        # 1. Construct the actual dict of HTTP request params.
        # 2. Check if the get_query_payload is True, in which
        #    case it should just return this dict.
        # 3. Otherwise make the HTTP request and receive the
        #    HTTP response.
        # 4. Pass this response to the extract_image_urls
        #    which scrapes it to extract the image download links.
        # 5. Return the download links as a list.
        request_payload = self._args_to_payload(coordinates, radius)
        if get_query_payload:
            return request_payload
        response = self._request(method="GET", url=self.URL,
                                 data=request_payload,
                                 timeout=self.TIMEOUT, cache=cache)

        return self.extract_image_urls(response.text)

    # the extract_image_urls method takes in the HTML page as a string
    # and uses regexps, etc to scrape the image urls:

    def extract_image_urls(self, html_str):
        """
        Helper function that uses regex to extract the image urls from the
        given HTML.

        Parameters
        ----------
        html_str : str
            source from which the urls are to be extracted

        Returns
        -------
        list of image URLs
        """
        # do something with regex on the HTML
        # return the list of image URLs
        pass


# the default tool for users to interact with is an instance of the Class
```

(continues on next page)

(continued from previous page)

```
Template = TemplateClass()

# once your class is done, tests should be written
# See ./tests for examples on this

# Next you should write the docs in astroquery/docs/module_name
# using Sphinx.
```

## 46.3 Astroquery Testing

Testing in astroquery is a bit more complicated than in other modules since we depend on remote servers to supply data. In order to keep the tests green and fast, we use monkeypatching to test most functions on local copies of the data.

In order to set up testing for any given module, you therefore need to have local copies of the data.

The testing directory structure should look like:

```
module/tests/__init__.py
module/tests/test_module.py
module/tests/test_module_remote.py
module/tests/setup_package.py
module/tests/data/
module/tests/data/test_data.xml
```

### 46.3.1 `test_module.py`

This file should contain only tests that do not require an internet connection. It also contains the tricky monkeypatching components. At a minimum, monkeypatching requires a few methods that are defined locally in the test file for each module.

#### Monkeypatching

At a minimum, monkeypatching will require these changes:

```
class MockResponse(object):

    def __init__(self, content):
        self.content = content
```

`MockResponse` is an object intended to have any of the attributes that a normal `requests.Response` object would have. However, it only needs to implement the methods that are actually used within the tests.

The tricky bits are in the `pytest.fixture`.

The first little magical function is the `patch_x` function, where `x` should either be `post` or `get`.

```
@pytest.fixture
def patch_get(request):
    try:
        mp = request.getfixturevalue("monkeypatch")
    except AttributeError:  # pytest < 3
```

(continues on next page)

(continued from previous page)

```
        mp = request.getfuncargvalue("monkeypatch")
    mp.setattr(requests, 'get', get_mockreturn)
    return mp
```

This function, when called, changes the `requests.get` method (i.e., the `get` method of the `requests` module) to call the `get_mockreturn` function, defined below. `@pytest.fixture` means that, if any function in this `test_module.py` file accepts `patch_get` as an argument, `patch_get` will be called prior to running that function.

`get_mockreturn` is simple but important: this is where you define a function to return the appropriate data stored in the `data/` directory as a readable object within the `MockResponse` class:

```
def get_mockreturn(url, params=None, timeout=10):
    filename = data_path(DATA_FILES['votable'])
    content = open(filename, 'r').read()
    return MockResponse(content)
```

`data_path` is a simple function that looks for the `data` directory local to the `test_module.py` file.

```
def data_path(filename):
    data_dir = os.path.join(os.path.dirname(__file__), 'data')
    return os.path.join(data_dir, filename)
```

### 46.3.2 `test_module_remote.py`

The remote tests are much easier. Just decorate the test class or test functions with `astropy.tests.helper.remote_data`.

### 46.3.3 `setup_package.py`

This file only needs the `get_package_data()` function, which will tell `setup.py` to include the relevant files when installing.

```
import os

def get_package_data():
    paths_test = [os.path.join('data', '*.xml')]

    return {'astroquery.module.tests': paths_test}
```

The following Astroquery modules are mostly meant for internal use of services in Astroquery, you can use them for your scripts, but we don't guarantee API stability.

## 46.4 Astroquery utils (`astroquery.utils`)

### 46.4.1 Reference/API

#### astroquery.utils Package

Common non-package specific utility functions that will ultimately be merged into `astropy.utils`.

### Functions

| | |
|---|---|
| `async_to_sync`(cls) | Convert all query_x_async methods to query_x methods |
| `chunk_read`(response[, chunk_size, report_hook]) | |
| `chunk_report`(bytes_so_far, chunk_size, . . . ) | |
| `download_list_of_fitsfiles`(linklist[, . . . ]) | Given a list of file URLs, download them and (optionally) rename them. |
| `parse_coordinates`(coordinates) | Takes a string or astropy.coordinates object. |
| `prepend_docstr_nosections`(doc[, sections]) | Decorator to prepend to the function's docstr after stripping out the list of sections provided (by default "Returns" only). |
| `send_request`(url, data, timeout[, . . . ]) | A utility function that post HTTP requests to remote server and returns the HTTP response. |
| `suppress_vo_warnings`() | Suppresses all warnings of the class `astropy.io.votable.exceptions.VOWarning`. |
| `validate_email`(email) | E-mail address validation. |

#### async_to_sync

`astroquery.utils.`**`async_to_sync`**(*cls*)

Convert all query_x_async methods to query_x methods

(see http://stackoverflow.com/questions/18048341/add-methods-to-a-class-generated-from-other-methods for help understanding)

#### chunk_read

`astroquery.utils.`**`chunk_read`**(*response*, *chunk_size=1024*, *report_hook=None*)

#### chunk_report

`astroquery.utils.`**`chunk_report`**(*bytes_so_far*, *chunk_size*, *total_size*)

#### download_list_of_fitsfiles

`astroquery.utils.`**`download_list_of_fitsfiles`**(*linklist*, *output_directory=None*, *output_prefix=None*, *save=False*, *overwrite=False*, *verbose=False*, *output_coord_format=None*, *filename_header_keywords=None*, *include_input_filename=True*)

Given a list of file URLs, download them and (optionally) rename them.

##### Examples

```
>>> linklist = ['http://fermi.gsfc.nasa.gov/FTP/fermi/data/lat/queries/L130413170713F15B52BC06_
↪PH00.fits',
...             'http://fermi.gsfc.nasa.gov/FTP/fermi/data/lat/queries/L130413170713F15B52BC06_
↪PH01.fits',
...             'http://fermi.gsfc.nasa.gov/FTP/fermi/data/lat/queries/L130413170713F15B52BC06_
↪SC00.fits']
>>> download_list_of_fitsfiles(linklist,
...     output_directory='fermi_m31',
...     output_prefix='FermiLAT',
...     save=True,
...     overwrite=False,
...     verbose=True,
...     output_coord_format=None, # FITS tables don't have crval/crpix, good one is: "%08.3g%+08.
↪3g",
...     filename_header_keywords=None, # couldn't find any useful ones
...     include_input_filename=True)
```

### parse_coordinates

`astroquery.utils.`**`parse_coordinates`**(*coordinates*)

Takes a string or astropy.coordinates object. Checks if the string is parsable as an `astropy.coordinates` object or is a name that is resolvable. Otherwise asserts that the argument is an astropy.coordinates object.

**Parameters**

**coordinates** : str or `astropy.coordinates` object

Astronomical coordinate

**Returns**

**coordinates** : a subclass of `astropy.coordinates.BaseCoordinateFrame`

**Raises**

**astropy.units.UnitsError**

**TypeError**

### prepend_docstr_nosections

`astroquery.utils.`**`prepend_docstr_nosections`**(*doc, sections=['Returns']*)

Decorator to prepend to the function's docstr after stripping out the list of sections provided (by default "Returns" only).

### send_request

`astroquery.utils.`**`send_request`**(*url*, *data*, *timeout*, *request_type='POST'*, *headers={}*, ***kwargs*)

A utility function that post HTTP requests to remote server and returns the HTTP response.

**Parameters**

**url** : str

The URL of the remote server

**data** : dict

A dictionary representing the payload to be posted via the HTTP request

**timeout** : int, quantity_like

Time limit for establishing successful connection with remote server

**request_type** : str

options are 'POST' (default) and 'GET'. Determines whether to perform an HTTP POST or an HTTP GET request

**headers** : dict

POST or GET headers. user-agent will be set to astropy:astroquery.version

**Returns**

**response** : `requests.Response`

Response object returned by the remote server

### suppress_vo_warnings

`astroquery.utils.`**`suppress_vo_warnings`**`()`

Suppresses all warnings of the class `astropy.io.votable.exceptions.VOWarning`.

### validate_email

`astroquery.utils.`**`validate_email`**`(`*email*`)`

E-mail address validation. Uses validate_email if available, else a simple regex that will let through some invalid e-mails but will catch the most common violators.

### Classes

| | |
|---|---|
| `TableList`(inp) | A class that inherits from `list` but included some pretty printing methods for an OrderedDict of `astropy.table.Table` objects. |
| `class_or_instance`(fn) | |

### TableList

**`class`** `astroquery.utils.`**`TableList`**`(`*inp*`)`

Bases: `list`

A class that inherits from `list` but included some pretty printing methods for an OrderedDict of `astropy.table.Table` objects.

HINT: To access the tables by # instead of by table ID: >>> t = TableList([('a',1),('b',2)]) >>> t[1] 2 >>> t['b'] 2

#### Methods Summary

| | |
|---|---|
| format_table_list() | Prints the names of all astropy.table.Table objects, with their respective number of row and columns, contained in the TableList instance. |
| keys() | |
| pprint(**kwargs) | Helper function to make API more similar to astropy.Tables |
| print_table_list() | |
| values() | |

#### Methods Documentation

**format_table_list()**
Prints the names of all astropy.table.Table objects, with their respective number of row and columns, contained in the TableList instance.

**keys()**

**pprint**(***kwargs*)
Helper function to make API more similar to astropy.Tables

**print_table_list()**

**values()**

### class_or_instance

**class** astroquery.utils.**class_or_instance**(*fn*)
Bases: object

### TAP/TAP+

Table Access Protocol implementation. See *TAP/TAP+ (astroquery.utils.tap)*

## 46.5 Astroquery query (astroquery.query)

### 46.5.1 Reference/API

#### astroquery.query Module

#### Classes

| | |
|---|---|
| BaseQuery() | This is the base class for all the query classes in astroquery. |
| QueryWithLogin() | This is the base class for all the query classes which are required to have a login to access the data. |

## BaseQuery

**class** astroquery.query.**BaseQuery**

Bases: object

This is the base class for all the query classes in astroquery. It is implemented as an abstract class and must not be directly instantiated.

### Methods Summary

| | |
|---|---|
| __call__(*args, **kwargs) | init a fresh copy of self |

### Methods Documentation

**__call__**(**args*, ***kwargs*)

init a fresh copy of self

## QueryWithLogin

**class** astroquery.query.**QueryWithLogin**

Bases: astroquery.query.BaseQuery

This is the base class for all the query classes which are required to have a login to access the data.

The abstract method _login() must be implemented. It is wrapped by the login() method, which turns off the cache. This way, login credentials are not stored in the cache.

### Methods Summary

| |
|---|
| authenticated() |
| login(*args, **kwargs) |

### Methods Documentation

**authenticated**()

**login**(**args*, ***kwargs*)

# CHAPTER 47

# License

Astroquery is licensed under a 3-clause BSD style license - see license.

# Bibliography


[R4] http://skyview.gsfc.nasa.gov/current/help/fields.html

[R5] http://skyview.gsfc.nasa.gov/current/help/fields.html

[R6] http://skyview.gsfc.nasa.gov/current/help/fields.html

# Python Module Index

## a

# Index

## Symbols



## A

## B



## C

## H



## I



## K



## L

## M



## N



## O



## P



## Q

## R

# S



# T

## U

## V



## W



## X