FitsHeaders_MongoDB

June 6, 2019

1 FITS Headers in MongoDB

We continue our exploration of loading astronomical data into a MongoDB database. In this instance, we'll look at FITS Headers and how we can store them inside MongoDB. This is meant to accompany the blog post at Strakul's Thoughts.

1.1 Examining a FITS header

First off, we'll explore an ordinary fits file with `astropy`. In particular, we'll use some of the test files provided by the package itself.

```
In [1]: from astropy.io import fits

        fits_image_filename = fits.util.get_testdata_filepath('test0.fits')

        hdul = fits.open(fits_image_filename)
```

We can examine what header information is present for this file. For clarity, I only display a few of the cards and I'll only be considering the primary header.

```
In [2]: hdul.info()
```

```
Filename: /Users/drodriguez/anaconda3/lib/python3.7/site-packages/astropy/io/fits/tests/data/tes
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU     138   ()
  1  SCI           1 ImageHDU        61   (40, 40)   int16
  2  SCI           2 ImageHDU        61   (40, 40)   int16
  3  SCI           3 ImageHDU        61   (40, 40)   int16
  4  SCI           4 ImageHDU        61   (40, 40)   int16
```

```
In [3]: hdul[0].header[0:10]
```

```
Out[3]: SIMPLE  =                    T / file does conform to FITS standard
        BITPIX  =                   16 / number of bits per data pixel
        NAXIS   =                    0 / number of data axes
        EXTEND  =                    T / FITS dataset may contain extensions
        GROUPS  =                    F / data has groups
```

```
NEXTEND =                    4 / Number of standard extensions
BSCALE  =           1.000000E0 / REAL = TAPE*BSCALE + BZERO
BZERO   =           3.276800E4 /
ORIGIN  = 'NOAO-IRAF FITS Image Kernel Aug 1 1997' / FITS file originator
DATE    = '01/04/99  '         / Date FITS file was generated
```

```
In [4]: h = hdul[0].header
```

As we did in our prior exercise, we'll want to convert this output to JSON in some way. We'll make use of the built-in method `__dict__` to express the header as a dictionary and, in particular, store only the key `_cards` which stores the actual header information as a list of tuples.

```
In [5]: h.__dict__['_cards'][0:10]
```

```
Out[5]: [('SIMPLE', True, 'file does conform to FITS standard'),
         ('BITPIX', 16, 'number of bits per data pixel'),
         ('NAXIS', 0, 'number of data axes'),
         ('EXTEND', True, 'FITS dataset may contain extensions'),
         ('GROUPS', False, 'data has groups'),
         ('NEXTEND', 4, 'Number of standard extensions'),
         ('BSCALE', 1.0, 'REAL = TAPE*BSCALE + BZERO'),
         ('BZERO', 32768.0, ''),
         ('ORIGIN', 'NOAO-IRAF FITS Image Kernel Aug 1 1997', 'FITS file originator'),
         ('DATE', '01/04/99', 'Date FITS file was generated')]
```

Now that we know how to extract the FITS header, we'll loop over the cards and store them in a dictionary which we'll convert to JSON. I decided to store only they key and value, but provide some commented out code to how one can also store the comment field.

```
In [6]: import json

        data = {}
        for i, c in enumerate(h.__dict__['_cards']):
            if c[0] == '': continue
            data[c[0]] = c[1]

            # An alternative to store comments in addition to values
            #data[c[0]] = {'value': c[1], 'comment': c[2]}

        print(json.dumps(data, default=lambda x: x.__dict__, indent=4))
```

```
{
    "SIMPLE": true,
    "BITPIX": 16,
    "NAXIS": 0,
    "EXTEND": true,
    "GROUPS": false,
    "NEXTEND": 4,
    "BSCALE": 1.0,
```

```
"BZERO": 32768.0,
"ORIGIN": "NOAO-IRAF FITS Image Kernel Aug 1 1997",
"DATE": "01/04/99",
"IRAF-TLM": "xxx",
"INSTRUME": "WFPC2",
"ROOTNAME": "U2EQ0201T",
"FILETYPE": "SCI",
"MODE": "AREA",
"SERIALS": "OFF",
"IMAGETYP": "EXT",
"CDBSFILE": "NO",
"PKTFMT": 96,
"FILTNAM1": "F673N",
"FILTNAM2": "",
"FILTER1": 33,
"FILTER2": 0,
"FILTROT": 0.0,
"LRFWAVE": 0.0,
"UCH1CJTM": -88.3486,
"UCH2CJTM": -88.8073,
"UCH3CJTM": -88.3945,
"UCH4CJTM": -88.9041,
"UBAY3TMP": 14.5969,
"KSPOTS": "OFF",
"SHUTTER": "B",
"ATODGAIN": 7.0,
"MASKCORR": "PERFORM",
"ATODCORR": "PERFORM",
"BLEVCORR": "PERFORM",
"BIASCORR": "PERFORM",
"DARKCORR": "OMIT",
"FLATCORR": "PERFORM",
"SHADCORR": "PERFORM",
"DOSATMAP": "OMIT",
"DOPHOTOM": "PERFORM",
"DOHISTOS": "OMIT",
"OUTDTYPE": "REAL",
"MASKFILE": "uref$fan15478u.r0h",
"ATODFILE": "uref$e1b09594u.r1h",
"BLEVFILE": "ucal$u2eq0201t.x0h",
"BLEVDFIL": "ucal$u2eq0201t.q1h",
"BIASFILE": "uref$e6o0937du.r2h",
"BIASDFIL": "uref$e6o0937du.b2h",
"DARKFILE": "uref$e6o09384u.r3h",
"DARKDFIL": "uref$e6o09384u.b3h",
"FLATFILE": "uref$e1c1404ju.r4h",
"FLATDFIL": "uref$e1c1404ju.b4h",
"SHADFILE": "uref$e6o09405u.r5h",
```

```
    "PHOTTAB": "",
    "GRAPHTAB": "mtab$f7d1401pm.tmg",
    "COMPTAB": "mtab$f7j1535pm.tmc",
    "SATURATE": 4095,
    "USCALE": 1.0,
    "UZERO": 0.0,
    "READTIME": 151,
    "PA_V3": 292.387786865,
    "RA_SUN": 56.2731208362,
    "DEC_SUN": 19.8289107164,
    "EQNX_SUN": 2000.0,
    "MTFLAG": true,
    "EQRADTRG": 0.0,
    "FLATNTRG": 0.0,
    "NPDECTRG": 0.0,
    "NPRATRG": 0.0,
    "ROTRTTRG": 0.0,
    "LONGPMER": 0.0,
    "EPLONGPM": 0.0,
    "SURFLATD": 0.0,
    "SURFLONG": 0.0,
    "SURFALTD": 0.0,
    "PODPSFF": false,
    "STDCFFF": false,
    "STDCFFP": "0x5569",
    "RSDPFILL": -100,
    "UEXPODUR": 0,
    "NSHUTA17": 0,
    "DARKTIME": 0.23,
    "UEXPOTIM": 20272,
    "PSTRTIME": "1994.139:15:41:39",
    "PSTPTIME": "1994.139:16:14:39",
    "ORIENTAT": 157.076,
    "SUNANGLE": 158.939133,
    "MOONANGL": 54.995346,
    "SUN_ALT": -11.552458,
    "FGSLOCK": "FINE",
    "DATE-OBS": "19/05/94",
    "TIME-OBS": "15:41:16",
    "EXPSTART": 49491.65366175,
    "EXPEND": 49491.65366441,
    "EXPTIME": 0.23,
    "EXPFLAG": "NORMAL",
    "FILENAME": "vtest3.fits"
}
```

1.2 A simple database load

This is a fairly simple document, so inserting into MongoDB should not be any issue. As before, we'll establish a connection to our locally running server and create a new collection, called 'fits', to store this.

```
In [7]: import pymongo

        client = pymongo.MongoClient()  # default connection (ie, local)

        db = client['test']  # database
        collection = db.fits  # collection; can also call as db['fits']
        collection.drop()  # drop collection, if needed
```

```
In [8]: json_data = json.loads(json.dumps(data, default=lambda x: x.__dict__, indent=4))
        result = collection.insert_one(json_data)

        # Quick check to confirm load
        cursor = collection.find({'FILENAME': 'vtest3.fits'})
        for doc in cursor:
            print(doc)
```

```
{'_id': ObjectId('5cf02ccc5abbd9a33e10c74e'), 'SIMPLE': True, 'BITPIX': 16, 'NAXIS': 0, 'EXTEND'
```

1.3 Creating a wrapper to handle inserts

Now, let's put all of those steps together into something we can run more easily. Below I've created a class called `FitsWrapper`, which takes a filename, gets the primary header, constructs the dictionary out of it, and has methods to both produce the JSON output and to store it in a collection the user defines. With this one class, I can do all of the steps above to quickly load FITS headers into my database. Note that I concatenate header keywords in the event multiple keywords are present, as is commonly the case with 'COMMENT'.

```
In [9]: import os
        import json
        from astropy.io import fits


        class FitsWrapper:
            def __init__(self, fits_filename):
                hdul = fits.open(fits_filename)
                self.h = hdul[0].header
                hdul.close()

                self.data = {}
                for i, c in enumerate(self.h.__dict__['_cards']):
                    if c[0] == '': continue
```

```
                    # Append to existing cards
                    if self.data.get(c[0], None) is None:
                        self.data[c[0]] = c[1]
                    else:
                        self.data[c[0]] = self.data[c[0]] + ' ' + c[1]

                if self.data.get('FILENAME', None) is None:
                    self.data['FILENAME'] = os.path.basename(fits_filename)

            def to_json(self):
                return json.dumps(self.data, default=lambda x: x.__dict__, indent=4)

            def load_mongodb(self, collection, verbose=False):
                json_data = json.loads(self.to_json())

                # This uses replace_one to replace any existing document that matches the filter
                # If none is matched, upsert=True creates a new document.
                result = collection.replace_one(filter={'FILENAME': self.data['FILENAME']}, repl

                if verbose:
                    print('Modified: ', result.modified_count)
                    print('Insert ID: ', result.upserted_id)

In [10]: fits_image_filename = fits.util.get_testdata_filepath('test0.fits')
         fw = FitsWrapper(fits_image_filename)
         fw.load_mongodb(collection, verbose=True)

Modified:  1
Insert ID:  None
```

It's that simple.

Now, if you look closely at the `load_mongodb` method, you'll notice I did not use `insert_one` as before, but instead `replace_one`. This is because I wanted to do something different. This method takes a filter and performs a search on the database for all documents that satisfy it, then it replaces them with the supplied JSON information. If no match is found and upsert is set to True, it creates that new document. This allows me to insert and update documents very easily and avoid duplication. By setting `verbose=True` in the call to `load_mongodb`, I output some debug statements showing that I modified 1 document and inserted no new ones.

In this particular example, I'm querying by the FITS FILENAME, which I assume to be unique, but if it isn't, or you want to query for something else, it's straightforward to update the filter setting in that method.

1.4 Loading more data

Now, let's go ahead and load up some more data. The text below contains links of FITS files drawn from the BDNYC database of brown dwarfs. These are all online and easily accessible though our interface at http://database.bdnyc.org. I also added a few from the Astropy Tutorials.

```
In [11]: fits_links = """https://s3.amazonaws.com/bdnyc/nir_spectra/spex_prism_1254-0122_030522.
         https://s3.amazonaws.com/bdnyc/U11122_1305-2541_kelu1_lris.fits
         https://s3.amazonaws.com/bdnyc/L2_Kelu-1AB.fits
         https://s3.amazonaws.com/bdnyc/SpeX/IRTF%20Library%20%28Prism%2BLXD%29/T4.5_2MASSJ0559-
         http://data.astropy.org/tutorials/FITS-images/HorseHead.fits
         https://astropy.stsci.edu/data/tutorials/FITS-images/M13_blue_0001.fits"""

In [12]: for f in fits_links.split('\n'):
             try:
                 fw = FitsWrapper(f)
                 fw.load_mongodb(collection)
             except Exception as e:
                 print(f'Error when loading {f}: {e}')
```

Let's check how many documents we've loaded thus far:

```
In [13]: count = collection.count_documents({})
         print('Total documents: ', count)

Total documents:  7
```

It's not a lot, but it's a start. Now, something to bear in mind is that the metadata in this FITS files is very varied. They're coming from different telescopes, different instruments, and different processing pipelines. Searching through these documents can be tricky if you don't know the exact header keywords have been used. One thing we can do to alleviate this is to create a text index accross all fields. You can only have one text index in a MongoDB collection, I believe, but it can include any number of fields. If we include all of them, then no matter what the header keyword is, we can still end up finding the values of interest.

```
In [14]: #collection.drop_index('text_fields')
         collection.create_index([('$**', pymongo.TEXT)], name='text_fields', background=True)

Out[14]: 'text_fields'

In [15]: cursor = collection.find({'$text': {'$search': 'OG590 F673N'}},
                                   {'_id': 0, 'FILENAME': 1, 'FILTER': 1, 'FILTNAM1': 1, 'FILTNAM
         for doc in cursor:
             print(doc)

{'FILTNAM1': 'F673N', 'FILTNAM2': '', 'FILENAME': 'vtest3.fits'}
{'FILTER': 'OG590', 'FILENAME': 'HorseHead.fits'}
```

In the above example, we've searched for either OG590 or F673N, two filters names (`$text` uses a logical OR search for the terms unless you enclose them in quotes, see the documentation). There are two documents that match this search, one for each filter. In one case, the filter keyword is FILTNAM1, but in the other its FILTER yet because the text index was created, any keyword with text content is included when doing a text search.

1.5 Final Thoughts

To wrap up, this has been a simple example of how you can populate a database with metadata from FITS headers. For simplicity, I only saved the primary header, but the example code here can be modified to run on any extension, or perhaps loop over all extensions. At that point you can decide if each extension is it's own single document or if they are embedded together to keep all the metadata together.

Due to the varied nature of headers, a text index was created in the database to facilitate searches. If you have hundreds of FITS files as part of your work (I certainly did in my graduate school and postdoc days), you could use something like this to organize your data and make searching through it somewhat simpler.