DataClass_MongoDB

May 24, 2019

1 Data Classes and MongoDB

In this notebook, I will briefly describe how to set up some simple, astronomically-significant data classes and load them into a MongoDB NoSQL database for future use. More details are offered in my blog post at Strakul's Thoughts.

To run this notebook, you need to run `Python 3.7` or later and have `pymongo` installed.

1.0.1 Data Classes

Data classes were introduced in Python 3.7 and offer an easy way to quickly create python classes for storing rich, structured data. Let's create a simple class to store a type of astronomical object known as brown dwarf. Brown dwarfs are not massive enough to fuse hydrogen like stars but are more massive than planets, which make the particularly interesting to study.

```
In [23]: from dataclasses import dataclass, field

         @dataclass
         class BrownDwarf_v0:
             name: str
             ra: float
             dec: float
             name_list: list = field(default_factory=list)
```

```
In [24]: bd = BrownDwarf_v0(name='1207-3932', ra=181.889, dec=-39.548)
         bd.name_list = ['TWA 27', '2MASS J12073346-3932539']
```

```
In [25]: bd
```

```
Out[25]: BrownDwarf_v0(name='1207-3932', ra=181.889, dec=-39.548, name_list=['TWA 27', '2MASS J1
```

As an alternative, you can have a separate dataclass to store the ra/dec information:

```
In [26]: @dataclass
         class Coords:
             ra: float
             dec: float

         @dataclass
```

```
class BrownDwarf_v1:
    name: str
    coords: Coords
    name_list: list = field(default_factory=list)
```

```
In [27]: c = Coords(ra=181.889, dec=-39.548)
         bd = BrownDwarf_v1(name='1207-3932', coords=c)
         print(bd)
```

```
BrownDwarf_v1(name='1207-3932', coords=Coords(ra=181.889, dec=-39.548), name_list=[])
```

Or get even fancier and store an astropy SkyCoords object, if you wanted:

```
In [28]: from astropy.coordinates import SkyCoord
         import astropy.units as u

         @dataclass
         class BrownDwarf_v2:
             name: str
             coords: SkyCoord
             name_list: list = field(default_factory=list)
```

```
In [29]: s = SkyCoord(ra=181.889*u.deg, dec=-39.548*u.deg)
         bd = BrownDwarf_v2(name='1207-3932', coords=s)
         print(bd)
```

```
BrownDwarf_v2(name='1207-3932', coords=<SkyCoord (ICRS): (ra, dec) in deg
    (181.889, -39.548)>, name_list=[])
```

As a SkyCoord object, you get all the usual functionality you expect. For example, you can quickly check the constellation:

```
In [30]: print(bd.coords.get_constellation())
```

```
Centaurus
```

For the purposes of this demo, I'll stick to using custom-built objects to store my data. Let's define a more complete example to work with:

```
In [31]: import json

         @dataclass
         class Coords:
             ra: float
             dec: float

         @dataclass
```

```
class Photometry:
    value: float
    error: float
    unit: str = 'mag'

@dataclass
class BrownDwarf:
    source_id: int
    name: str
    coords: Coords
    J: Photometry = None
    H: Photometry = None
    Ks: Photometry = None
    spectral_type: str = None
    name_list: list = field(default_factory=list)

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

```
In [32]: c = Coords(ra=181.889, dec=-39.548)
         # 2MASS_J: 12.99 +/- 0.03
         # 2MASS_H: 12.39 +/- 0.03
         # 2MASS_Ks: 11.95 +/- 0.03
         j = Photometry(12.99, 0.03)
         bd = BrownDwarf(source_id=11, name='1207-3932', coords=c,
                         J = Photometry(12.99, 0.03),
                         H = Photometry(12.39, 0.03),
                         Ks = Photometry(11.95, 0.03))
         bd.name_list = ['TWA 27', '2MASS J12073346-3932539']
         bd.spectral_type = 'M8.0'
         print(bd)
```

```
BrownDwarf(source_id=11, name='1207-3932', coords=Coords(ra=181.889, dec=-39.548), J=Photometry(
```

Now that we have a representation of the basic parameters we want to store for our object, we can loop over some table or input the values we need to store. At some point, though, we'll want to save our work in some more concrete fashion. You may have noticed I included a `to_json` method in the `BrownDwarf` object definition. This uses the `__dict__` method (or equivalently the `asdict` function of dataclasses) to represent the object as a dictionary and then convert it to JSON. Here's how the object looks like as a dictionary:

```
In [33]: from dataclasses import asdict

         asdict(bd)
         #bd.__dict__
```

```
Out[33]: {'source_id': 11,
          'name': '1207-3932',
```

```
        'coords': {'ra': 181.889, 'dec': -39.548},
        'J': {'value': 12.99, 'error': 0.03, 'unit': 'mag'},
        'H': {'value': 12.39, 'error': 0.03, 'unit': 'mag'},
        'Ks': {'value': 11.95, 'error': 0.03, 'unit': 'mag'},
        'spectral_type': 'M8.0',
        'name_list': ['TWA 27', '2MASS J12073346-3932539']}
```

And here's how it looks like when expressed as JSON:

```
In [34]: print(bd.to_json())
```

```
{
    "source_id": 11,
    "name": "1207-3932",
    "coords": {
        "ra": 181.889,
        "dec": -39.548
    },
    "J": {
        "value": 12.99,
        "error": 0.03,
        "unit": "mag"
    },
    "H": {
        "value": 12.39,
        "error": 0.03,
        "unit": "mag"
    },
    "Ks": {
        "value": 11.95,
        "error": 0.03,
        "unit": "mag"
    },
    "spectral_type": "M8.0",
    "name_list": [
        "TWA 27",
        "2MASS J12073346-3932539"
    ]
}
```

In the above example, I used `default= lambda x: x.__dict__` to tell json.dumps that by default it should attempt to use the dictionary representation of classes if it found cases it could not understand. This, or something similar, is needed to recursively convert any nested dataclasses you may have built (such as Photometry and Coords)

1.0.2 MongoDB

You may be wondering why use JSON at all? Why not just flatten it out and write a long table? The reason is that I want to use MongoDB to store my data. MongoDB is a NoSQL database that relies on JSON to store its documents. In fact, it's explicitly a type of database known as a document-store. By representing my data as JSON, I have a format that I can directly store into MongoDB without any major work.

You can download a free copy of MongoDB from https://www.mongodb.com/ and can run a server locally on your machine, which is what I've done. Alternatively you can connect to a Cloud instance, if you have access to one already or sign up for their Atlas service.

Let's connect to a local MongoDB server instance; you may need to start this instance separately, refer to the mongodb documentation. In my case, I had to run `mongod --dbpath PATH-TO-DB-DIR` in a Bash terminal, where `PATH-TO-DB-DIR` is the directory where I store my mongodb databases:

```
In [35]: import pymongo

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

Now, we can specify the database we'd like to use, as well as the *collection*. A MongoDB collection is the equivalent of a table in relational databases like SQL. Each collection is built up of multiple documents (equivalent to rows or entries). Unlike relational database, neither the database or collection is required to exist prior to loading documents into it. If one doesn't exist, it will be created when you load your first document. If you've been running this tutorial several times, you may already have a collection. If you want to clear it you can use the `.drop()` method on it.

```
In [36]: db_name = 'test'
         db = client[db_name]  # database
         dwarfs = db.dwarfs  # collection; can also call as db['dwarfs']
         dwarfs.drop()  # drop collection, if needed
```

Now, let's load up that JSON representation of the brown dwarf we saved (we need an actual JSON object, not the string representation we produced before):

```
In [37]: json_data = json.loads(bd.to_json())
         result = dwarfs.insert_one(json_data)

         # Quick check to confirm load
         cursor = dwarfs.find({'source_id': 11})
         for doc in cursor:
             print(doc)
```

```
{'_id': ObjectId('5cdff168244648147014faa5'), 'source_id': 11, 'name': '1207-3932', 'coords': {'
```

1.0.3 Example Data Load

Let's load up a bunch of data first so we can better explore how to use mongodb. Here is a small sample of data from the BDNYC Brown Dwarf database. For simplicity, I've only included J and H 2MASS data and only a single spectral type estimate. For more details on the BDNYC database, I'll refer you to http://database.bdnyc.org

```
In [38]: bddata = """#id         sname         ra          dec         sptype        J         J_err
         2         1331-0116       202.95387       -1.280556       16         15.46        0.04
         4         1448+1031       222.106791       10.533056       13.5        14.556
         7         1439+1929       219.868167       19.487472       11         12.759        0.
         14         2249+0044       342.472709       0.734611       11         16.587        0.
         15         2208+2921       332.05679       29.355972       13         15.797        0.
         17         0027+0503       6.924875       5.061583       8         16.189        0.093
         19         2148+4003       327.068041       40.0665       16         14.147        0.0
         20         1102-3430       165.54097       -34.509869       8.5         13.034
         32         0415-0935       63.831417       -9.585167       28         15.695        0.
         34         0727+1710       111.826001       17.167       27         15.6        0.061
         36         0451-3402       72.753833       -34.0375       10.5         13.541        0
         53         1515+4847       228.753459       48.794889       16         14.111        0
         61         1245-4429       191.309       -44.485477       9         14.518        0.03
         63         0334-4953       53.537667       -49.893944       9         11.376        0.
         80         1552+2948       238.24591       29.81342       10         13.478        0.0
         82         1835+3259       278.90792       32.998497       8.5         10.27        0.
         83         1547-2423       236.94662       -24.397028       9         13.97        0.0
         86         0036+1821       9.067376       18.352889       13.5         12.466        0
         91         0518-2756       79.692333       -27.946028       10         15.262        0
         96         0248-1651       42.170846       -16.856022       8         12.551        0.
         98         0241-0326       40.297958       -3.449639       10         15.799        0.
```

```
In [39]: data = []
         for row in bddata.split('\n'):
             if row.startswith('#'): continue
             elems = row.split('\t')

             # Format the spectral type
             spnum = float(elems[4])
             if spnum >= 20:
                 sptype = 'T{:3.1f}'.format(spnum-20)
             elif spnum >= 10:
                 sptype = 'L{:3.1f}'.format(spnum-10)
             else:
                 sptype = 'M{:3.1f}'.format(spnum)

             temp = BrownDwarf(source_id=int(elems[0]),
                               name=elems[1],
                               coords=Coords(ra=float(elems[2]), dec=float(elems[3])),
                               spectral_type=sptype,
                               J=Photometry(value=float(elems[5]), error=float(elems[6])),
                               H=Photometry(value=float(elems[7]), error=float(elems[8])),
                               name_list=elems[9].split(',')
                              )
             data.append(temp)
```

Now, if we check the data variable we can see that it is *literally* a `list` of `BrownDwarfs`:

```
In [40]: for i, row in enumerate(data):
             if i>4: break  # only display the first 5
             print(row)
```

```
BrownDwarf(source_id=2, name='1331-0116', coords=Coords(ra=202.95387, dec=-1.280556), J=Photomet
BrownDwarf(source_id=4, name='1448+1031', coords=Coords(ra=222.106791, dec=10.533056), J=Photome
BrownDwarf(source_id=7, name='1439+1929', coords=Coords(ra=219.868167, dec=19.487472), J=Photome
BrownDwarf(source_id=14, name='2249+0044', coords=Coords(ra=342.472709, dec=0.734611), J=Photome
BrownDwarf(source_id=15, name='2208+2921', coords=Coords(ra=332.05679, dec=29.355972), J=Photome
```

We can now loop over these and load them up into our database. One thing I do here is check the source_id value first to avoid re-inserting an existing brown dwarf (since the source_id's are unique in the core BDNYC database). It's an optional step I take to avoid duplicated documents.

```
In [41]: for row in data:
             source_id = row.source_id
             json_data = json.loads(row.to_json())

             count = dwarfs.count_documents({'source_id': source_id})

             if count > 0:
                 # Replace existing
                 cursor = dwarfs.find({'source_id': source_id})
                 for doc in cursor:
                     result = dwarfs.replace_one({'_id': doc['_id']}, json_data)
             else:
                 # Insert new
                 result = dwarfs.insert_one(json_data)
```

1.0.4 Database Queries

Now, we can examine the data in the database with standard MongoDB queries. Below are a few examples, but I encourage you to read through the MongoDB and pymongo documentation for more details. Note that the `_id` field is automatically generated by MongoDB when storing the document.

```
In [42]: count = dwarfs.count_documents({})
         print('Total documents: ', count)
         cursor = dwarfs.find({'spectral_type': 'M8.0'})
         for doc in cursor:
             print(doc)
```

```
Total documents:  22
{'_id': ObjectId('5cdff168244648147014faa5'), 'source_id': 11, 'name': '1207-3932', 'coords': {'
{'_id': ObjectId('5cdff168244648147014faab'), 'source_id': 17, 'name': '0027+0503', 'coords': {'
{'_id': ObjectId('5cdff168244648147014fab9'), 'source_id': 96, 'name': '0248-1651', 'coords': {'
```

We can 'project', or return only the fields we are interested in by suppling a second parameter to our queries:

```
In [43]: cursor = dwarfs.find({'spectral_type': 'M8.0'}, {'_id':0, 'source_id': 1, 'name': 1, 's
         for doc in cursor:
             print(doc)
```

```
{'source_id': 11, 'name': '1207-3932', 'spectral_type': 'M8.0'}
{'source_id': 17, 'name': '0027+0503', 'spectral_type': 'M8.0'}
{'source_id': 96, 'name': '0248-1651', 'spectral_type': 'M8.0'}
```

With a large dataset, you can create indices to better search for your data. The index will update as more data is added to it. Or you can always drop and re-create it. I'll touch more on indexes in a future post, but for now here's an example of creating a text index on the name and name_list fields:

```
In [44]: #dwarfs.drop_index('text_fields')
         dwarfs.create_index([('name', pymongo.TEXT),
                              ('name_list', pymongo.TEXT)],
                               name='text_fields',
                               background=True)

Out[44]: 'text_fields'
```

```
In [45]: cursor = dwarfs.find({'$text': {'$search': 'TWA'}}, {'_id': 0, 'name': 1, 'name_list':
         for doc in cursor:
             print(doc)
```

```
{'name': '1245-4429', 'name_list': ['TWA 29', ' 2MASS J12451416-4429077', ' DENIS J124514.1-4429
{'name': '1102-3430', 'name_list': ['TWA 28', ' 2MASS J11020983-3430355', ' SSSPM 1102-3431']}
{'name': '1207-3932', 'name_list': ['TWA 27', '2MASS J12073346-3932539']}
```

For partial name matching, you can also use regular expressions:

```
In [46]: cursor = dwarfs.find({'name_list': {'$regex': '12'}}, {'_id': 0, 'name': 1, 'name_list'
         for doc in cursor:
             print(doc)
```

```
{'name': '1207-3932', 'name_list': ['TWA 27', '2MASS J12073346-3932539']}
{'name': '2208+2921', 'name_list': ['2MASSW J2208136+292121', ' 2MASS J22081363+2921215', ' 2MAS
{'name': '0727+1710', 'name_list': ['2MASSI J0727182+171001', ' 2MASS J07271824+1710012', ' WISE
{'name': '1245-4429', 'name_list': ['TWA 29', ' 2MASS J12451416-4429077', ' DENIS J124514.1-4429
{'name': '0334-4953', 'name_list': ['2MASS J03341218-4953322', ' DENIS-P J033411.3-495333', ' 2M
{'name': '0248-1651', 'name_list': ['LP 771-21', ' 2MASS J02484096-1651249', ' BR 0246-1703', '
```

If you wanted to, you could now go back and re-create the Python `BrownDwarf` class objects with the data from the database. In practice, this is a little tricky since the JSON return doesn't explictly tell you what dataclass it came from. Below is a rough example of how you could manually re-build the dataclass, though I've seen some StackOverflow examples that can set this up a bit more automatically.

```
In [47]: cursor = dwarfs.find({'source_id': 11})
         doc = list(cursor)[0]
         del doc['_id']
         print(doc)
```

```
{'source_id': 11, 'name': '1207-3932', 'coords': {'ra': 181.889, 'dec': -39.548}, 'J': {'value':
```

```
In [48]: j = Photometry(**doc['J'])
         h = Photometry(**doc['H'])
         ks = Photometry(**doc['Ks'])
         c = Coords(**doc['coords'])
         print(j)
         print(c)
```

```
Photometry(value=12.99, error=0.03, unit='mag')
Coords(ra=181.889, dec=-39.548)
```

```
In [49]: del doc['coords']
         del doc['J']
         del doc['H']
         del doc['Ks']
         new_bd = BrownDwarf(**doc, coords=c, J=j, H=h, Ks=ks)
         print(new_bd)
```

```
BrownDwarf(source_id=11, name='1207-3932', coords=Coords(ra=181.889, dec=-39.548), J=Photometry(
```

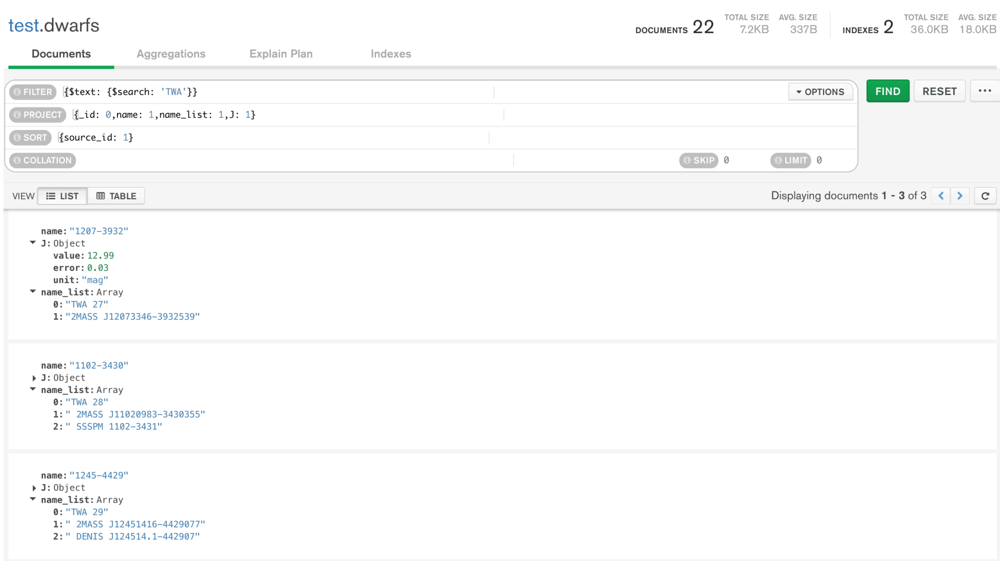
```
In [50]: # Original, for comparison
         print(bd)
```

```
BrownDwarf(source_id=11, name='1207-3932', coords=Coords(ra=181.889, dec=-39.548), J=Photometry(
```

```
In [51]: bd == new_bd
```

```
Out[51]: True
```

There are a lot more ways you can query this database and, as you can image, lots more ways to create and work with dataclasses. I'll leave it up to the reader to examine the documentation and play around with the code. I can recommend downloading the MongoDB Compass application from `https://www.mongodb.com/products/compass` which provides a nice GUI to directly access your database. Some advanced queries aren't possible in it, but it can serve as useful introduction to how to explore the data.



compass screenshot