

Catalog Class Interface

InstanceCatalog Redesign

Jake Vanderplas, June 20, 2013

In this document I will outline the elements of the redesign of the instance catalog generation and measures API. The overarching goal of this redesign was to provide an efficient as well as easily configurable and extensible interface to the catalogs generated from LSST simulation runs. The redesign covers two main pieces of the stack, within the generation and measures codebases

Goals of Review

The goals of this review are to evaluate the effectiveness of the design in terms of **extensibility** and **maintainability**. We want users to be able to define custom catalog outputs in a very seamless way: no config files, just Python code. So please address the following questions in this review:

- Does the method of introspection for defining the catalog class seem appropriate?
- Are there use cases where introspection may cause problems?
- Does the custom class interface make sense from a user's perspective?
- Is the code written clearly enough?
- Checking out the code

The code is in the "rewrite" directory under catalogs and measures; use the git branch `origin/u/vanderplas/queryDB_rewrite` for measures, the git branch is `origin/u/vanderplas/instance_rewrite`

Core Principles

Both pieces have a core design principle, which is the idea of subclasses as a primary interface. That is, users who wish to modify the default behavior can do so through the creation of subclasses of the core functionality. In order to make these subclasses as terse as possible, metaclasses are used to process specified class attributes and build the appropriate methods.

Database Interface

So, for example, the Star database object is defined (in `StarModels.py`) as follows:

```
from dbConnection import DBObject
class StarObj(DBObject):
    objid = 'msstars'
    tableid = 'starsMSRGB_forcesseek'
    # ... several more string attributes
    columns = [('id', 'simobjid', int),
               ('umag', None),
               ('gmag', None),
               #... several more column attributes
              ]
```

The class declaration contains no methods: all applicable methods are defined within the `DBObject` base class. Additionally, upon creation the new class is registered within the `DBObject` instance, so that it can be referred to by its specified object id:

```
star_obj = DBObject.from_objid('msstars', address=...)
```

This is true not only of built-in types, but also of user-defined types. What this results in is a very simple user-end API for extending the behavior of the built-in objects, with no separate config files needed.

Instance Catalog Interface

Where this approach really becomes interesting is within the instance catalog definitions. The `InstanceCatalog` object has been rewritten with the same principles, but with a very flexible way to define the source of data for columns. As above, the column names are specified as class attributes. So, a new type of test catalog can be specified like this:

```
class TestCatalog(InstanceCatalog):
    catalog_type = 'test_catalog'
    column_outputs = ['galID', 'raJ2000', 'decJ2000']
    refIdCol = 'galid'
    transformations = {'raJ2000':np.degrees,
                      'decJ2000':np.degrees}
```

When the catalog is instantiated from a database and the ``write_catalog`` method is called, the column_outputs are searched and the appropriate data is written. This data can be sourced from the database, from ``get`` methods in base classes, or from ``get`` methods in the class itself. The core of the logic required for this is in the ``column_by_name()`` method. It takes a column name as an argument, and returns the column values.

For example, imagine you have a database and are searching for the column called 'gmag'. The search path is as follows:

- If the class has a method called ``get_gmag``, then return the result of this method. Note that this will search the entire method resolution order of the class, so that parent classes and class mixins can be used in an intuitive way to extend built-in functionality.
- If there is no getter for the column, the next place to be searched is within compound column names (see below). Compound column names are groups of attributes which, for efficiency and convenience, may be computed together. If 'gmag' exists in a compound column, then its value will be computed and returned.
- Finally, if 'gmag' does not have an associated getter or compound column, its value is taken from the database.

Compound Columns

For a cleaner syntax and efficiency, a single getter function can return multiple columns. For example, when correcting magnitude systems, it may be more efficient to apply photometric corrections to all magnitudes at once. So your 'gmag' column may be defined as follows:

```
class TestCatalog(object):
    # ...
    @compound('gmag', 'rmag', 'imag')
    def get_magnitudes(self):
        g_raw, r_raw, i_raw = map(self.column_by_name,
                                ('g_raw', 'r_raw', 'i_raw'))
        # convert the raw magnitudes using some function
        gmag, rmag, imag = convert(g_raw, r_raw, i_raw)
        return gmag, rmag, imag
```

Because the results of compound column calculations will in general be accessed one at a time, the compound decorator also automatically caches the results the first time the function is computed.

Validation: Introspecting the Database

One potential problem with this approach is the validation of user-designed classes. For example, what if the catalog asks for an attribute that has neither a getter, a compound column entry, or a database entry? We'd like this failure to be caught as early as possible, before (for example) the necessary data is pulled from the database. Also, if we simply query the database each time a column is requested, it will lead to a lot of overhead. It would be better to query the database only once to pull down all the required columns, after which the remaining derived attributes can be computed.

This behavior is implemented within the InstanceCatalog metaclass through a clever bit of introspection. When the class is instantiated, the metaclass does a dry-run of outputting the table. To do this dry run, it temporarily replaces the real database with a duck-typed standin (_MimicRecordArray) which logs the columns that are requested. At the end of the dry run, we are left with a list of the columns that are required to be available in the database, and a quick check is performed to make sure the database has all the required columns. If not, an error is raised before any computation or data movement takes place.

Toy Examples

The result is a very intuitive Python API for defining various catalog types. Because the API takes advantage of Python's built-in method resolution order, it allows for very clean and powerful catalogs. Here is an example of a series of basic catalog types defined with this API:

```

class BasicCatalog(InstanceCatalog):
    """Simple catalog with columns directly from the database"""
    catalog_type = 'basic_catalog'
    refIdCol = 'id'
    column_outputs = ['id', 'ra_J2000', 'dec_J2000', 'g_raw', 'r_raw', 'i_raw']
    # transformations specify conversions when moving from the database
    # to the catalog. In this case, we take RA/DEC in radians and convert
    # to degrees.
    transformations = {"ra_J2000":np.degrees,
                      "dec_J2000":np.degrees}

class AstrometryMixin(object):
    @compound('ra_corrected', 'dec_corrected')
    def get_points_corrected(self):
        ra_J2000 = self.column_by_name('ra_J2000')
        dec_J2000 = self.column_by_name('dec_J2000')
        # ... do the conversions: these are just standins
        ra_corrected = ra_J2000 + 0.001
        dec_corrected = dec_J2000 - 0.001
        return ra_corrected, dec_corrected

class PhotometryMixin(object):
    @compound('g', 'r', 'i')
    def get_mags(self):
        g_mag, r_mag, i_mag = map(self.column_by_name,
                                   ['g_mag', 'r_mag', 'i_mag'])
        # ... do conversions here: these are just standins
        g = g_mag + 0.01
        r = r_mag - 0.01
        i = i_mag + 0.02
        return g, r, i

class CustomCatalog(BasicCatalog, AstrometryMixin, PhotometryMixin):
    column_outputs = ['id', 'redshift', 'points_corrected', 'mags']
    transformations = {"ra_corrected":np.degrees,
                      "dec_corrected":np.degrees}

```

Now to create a catalog, we connect to a database and call write_catalog

```

db = GalaxyObj()
catalog = CustomCatalog(db)
catalog.write_catalog("out.txt")
# out.txt has the following columns:
# id redshift ra_corrected dec_corrected g r i

```