Getters: how to write your own

As discussed on the framework overview page, when a catalog class is asked to output a column, the first thing the catalog class does is search through its own methods to see if it has a method for calculating the column. These column-calculating methods are called 'getters'. This page will discuss how to write new getters so that users can add columns to catalog classes as needed. The first two sections of this page will be a purely operational discussion of how to write new getters. The third section will explain why getters work the way that they do.

Writing single-column getters

Getters are called by the method column_by_name which is defined in the InstanceCatalog class in /sims_catalogs_measures/python/lsst/sims/catalogs/measures/instance/InstanceCatalog.py. column_by_name accepts a string corresponding to the name of the column, searches for a method to calculate the column, and returns a numpy array containing the values of the column. column_by_name is meant to act on several rows of the database at once. To make a gross oversimplification,

```
self.column_by_name('lsst_u')
```

will calculate the value of the column 'lsst_u' for all rows in the database and return a numpy array containing the results.

If a getter is only meant to calculate a single column's value, then the only requirements are that it be named, literally, get_columnName (replacing columnName with the name of the column that is calculated) and that it return its results as a numpy array. get_EBV (defined in /sims_photUtils/python/lsst/sims/photUtils/EBV.py) is an example of a single_column getter

```
    #and finally, here is the getter
    @cached
    def get_EBV(self):
        """
        Getter for the InstanceCatalog framework
        """
        glon=self.column_by_name('glon')
        glat=self.column_by_name('glat')

        EBV_out=numpy.array(self.calculateEbv(gLon=glon,gLat=glat,interp=True))
        return EBV_out
```

Note that, to calculate the column EBV, get_EBV must know the galactic longitude and latitude associated with each row in the database. These are the columns 'glon' and 'glat,' which get_EBV calls using self.column_by_name (which will then call the getters associated with those columns).

Note that the getter is declared with the decorator @cached. @cached is defined in /sims_catalogs_measures/python/lsst/sims/catalogs/measures/instance/decorators.py. It modifies the getter so that, each time the getter is called for a given set of rows (a 'chunk' below), the result is stored in a dictionary. Subsequent calls to that getter for that set of rows (i.e. if another column relies on that column's result and tries to access it through column_by_name) will merely read the result from that dictionary, rather than re-calculating the same result for the same set of rows.

Writing compound getters

Sometimes it is more efficient to write a getter which calculates the values of many columns at once. 'glon' and 'glat' are, themselves, examples of this case. Since both the galactic longitude and the galactic latitude require knowledge of the RA and Dec of an object, it makes much more sense to calculate them together than separately. Here is an example of a compound getter taken from /sims_coordUtils/python/lsst/sims/coordUtils/Astrometry.py

```
    @compound('glon','glat')
    def get_galactic_coords(self):
        """
        Getter for galactic coordinates, in case the catalog class does not provide that
        Reads in the ra and dec from the data base and returns columns with galactic
        longitude and latitude.
        All angles are in radians
        """
        ra=self.column_by_name('raJ2000')
        dec=self.column_by_name('decJ2000')
        glon, glat = galacticFromEquatorial(ra,dec)
        return numpy.array([glon,glat])
```

In this case, the name of the method is only contained to be get_something ('something' should be descriptive of what the getter does, but should not correspond to the name of any of the returned columns). The getter is flagged for use by column_by_name with the decorator @compound. The arguments of @compound are the names the columns calculated by the getter (in this case, 'glon' and 'glat'). It is important that the compound getter return its results as a two dimensional numpy array with the rows set in the same order as the arguments of @compound. The decorator @compound is defined in /sims_catalogs_measures/python/lsst/sims/catalogs/measures/instance/decorators.py. It modified the getter method so that the rows of the output emerge in an Ordered Dict keyed to the arguments of @compound. This is how column_by_name knows how to process the results of a compound getter.

The @compound decorator includes @cached from above.

How does the framework know how to handle getters?

We said above that "Getters are called by the method column_by_name which is defined in the InstanceCatalog class in /sims_catalogs_measures/python /lsst/sims/catalogs/measures/instance/InstanceCatalog.py. column_by_name accepts a string corresponding to the name of the column, searches for a method to calculate the column, and returns the corresponding value of that column. column_by_name is meant to act on several rows of the database at once. To make a gross oversimplification,

```
self.column_by_name('lsst_u')
```

will calculate the value of the column 'lsst_u' for all rows in the database and return a numpy array containing the results." As we said, though, this is a gross oversimplification, first and foremost because at no time does InstanceCatalog query all of the columns of a database at once."

In order to prevent massive queries from overwhelming the memory of the computers running them, queries in the catalog simulations are divided into chunks – manageably-sized groups of rows that match the query. Methods that return catalogs do so one chunk at a time. Consider the following lines of code taken from the method InstanceCatalog.write_catalog:

```
    query_result = self.db_obj.query_columns(colnames=self._active_columns,
                                             obs_metadata=self.obs_metadata,
                                             constraint=self.constraint,
                                             chunk_size=chunk_size)
    for chunk in query_result:
        self._set_current_chunk(chunk)
        chunk_cols = [self.transformations[col](self.column_by_name(col))
                      if col in self.transformations.keys() else
                      self.column_by_name(col)
                      for col in self.iter_column_names()]
```

query_result returns an iterator over the chunks of the query. For each chunk, write_catalog assembles an iterator over the desired column names (the result of self.iter_column_names()) and passes the columns in that iterator to column_by_name. The source code from column_by_name is below (with comments added to draw attention to and clarify what the method is doing)

```
    def column_by_name(self, column_name, *args, **kwargs):
        """Given a column name, return the column data"""
        getfunc = "get_%s" % column_name

        ##########################
        #####do methods exist for calculating the desired column?
        ##########################
        if hasattr(self, getfunc):
            return getattr(self, getfunc)(*args, **kwargs)
        elif column_name in self._compound_column_names:
            getfunc = self._compound_column_names[column_name]
            compound_column = getattr(self, getfunc)(*args, **kwargs)
            return compound_column[column_name]
        ############################
        #####do the desired columns exist in the database?
        ############################
        elif isinstance(self._current_chunk, _MimicRecordArray) or column_name in self._current_chunk.dtype.
names:
            return self._current_chunk[column_name]
        ############################
        #####have default values been defined for the desired columns?
        ############################
        else:
            return getattr(self, "default_%s"%column_name)(*args, **kwargs)
```

As can be seen, column_by_name first determines if it knows how to calculate the requested column. If not, it determines whether or not the column exists natively in the database. If not, it determines whether or not it knows a default value for the column (if all of these fail, column_by_name will throw an exception). Note that, if the column is defined natively in the database, it is returned as self._current_chunk[column_name]. Before calling column_by_name, write_catalog defined the member variable self._current_chunk to be the chunk of the database it was currently iterating over. This means that, when column_by_name returns self._current_chunk[column_name], it will return a numpy array containing the values of the column for all of the rows in self._current_chunk. This is important, because it illustrates how getters know to return results on the entire chunk.

Let us return to our 'glon', 'glat' example from above.

```
    @compound('glon','glat')
    def get_galactic_coords(self):
        """
        Getter for galactic coordinates, in case the catalog class does not provide that
        Reads in the ra and dec from the data base and returns columns with galactic
        longitude and latitude.
        All angles are in radians
        """
        ra=self.column_by_name('raJ2000')
        dec=self.column_by_name('decJ2000')
        glon, glat = galacticFromEquatorial(ra,dec)
        return numpy.array([glon,glat])
```

If the user asked for the column 'glon' (galactic longitude), column_by_name would call the getter (more on how it would know to make that choice later). Nothing about the getter refers explicitly to the size of self._current_chunk, except that this getter requires the columns 'raJ2000' and 'decJ2000' (the RA and Dec in the J2000 coordinate system), which exist natively in the database. Thus, the calls to column_by_name made in get_galactic_coords will return numpy arrays of all of the raJ2000 and decJ2000 values associated with the rows in self._current_chunk. These arrays will be passed to galacticFromEquatorial, which knows to return arrays of the same length as the arrays it is passed. Finally, get_galactic_coords combines glon and glat into a single, two row numpy array containing the galactic coordinates associated with all of the rows in self._current_chunk.

Default columns are handled by the following method in the InstanceCatalogMeta class (the meta class for InstanceCatalog, also defined in InstanceCatalog.py)

```
        # add methods for default columns
        for default in cls.default_columns:
            setattr(cls, 'default_%s'%(default[0]),
                lambda self, value=default[1], type=default[2]:\
                        numpy.array([value for i in
                                 xrange(len(self._current_chunk))],
                                 dtype=type))
```

Recall from our example catalog class on the framework overview page that defaults columns are declared like

```
class myCatalogClass(InstanceCatalog):
    column_outputs=['id','raJ2000','decJ2000','nonsenseColumn']
    default_columns=[('nonsenseColumn','word',(str,4))]
```

the InstanceCatalogMeta class will iterate over all of the columns defined in default_columns and build a method that, for each default column, returns a numpy array the size of self._current_chunk containing the default values. In this way, default columns also know the size of self._current_chunk

Because any calculated column will depend either on default columns or columns that exist natively in the database, getters for calculated columns will also be able to return numpy arrays of the size of self._current_chunk

Return to the main catalog simulations documentation page