Programmer’s Guide to using DataVault

An instructional guide to using the python-coded DataVault module developed for use in the yet-to-be-coded Great Lakes Regulation and Routing Model (GLRRM).

**Background**

The Coordinated Great Lakes Regulation and Routing Model (CGLRRM) was developed in the mid-1990s by Matt McPherson of the US Army Corps of Engineers (Detroit District). He coded that model in Fortran, specifically the Fortran 77 dialect, for a number of good reasons. It was the programming language most universally used by the partners on the project, and most of the existing individual component models (Lake Superior regulation, MidLakes routing, etc) existed as Fortran code. As new developers tried to modify the CGLRRM ten to fifteen years later for use in the International Upper Great Lakes Study, it became clear that changes in technology were going to necessitate development of a new model framework. One of the significant issues with the CGLRRM is related to how Fortran 77 handles data storage and sharing between modules. The CGLRRM makes extensively use of COMMON blocks, the primary means for global information sharing in Fortran 77. COMMON blocks are a potentially powerful tool for the programmer, but they also have very high potential for causing issues when they are not coded correctly. Without detailing issues, after the IUGLS update issues were found that don’t seem to have any negative results, but underscore the pitfalls of the technology. Accordingly, when development of the new model framework was discussed, a major focus was internal data handling.

In the planning discussions, I described the way COMMON blocks work by using the analogy of a table. All of the various data items (represented by sheets of paper with data on them) are spread out on the table, and the various component models are told, “Add what you have. Grab what you need. Replace items if you want.” But there are no labels on the papers; they just have the data values. Every model has to “know” exactly where on the table is the page with the numbers it needs. The problem is that model A might accidentally put the net basin supplies for Lake Michigan in the spot where another model thinks is the spot for Niagara River flows. And there is no means to verify that, other than very careful coding by a programmer who understands the nuances of data typing, array alignment, etc. Many of the coders working on the development of these models are primarily physical scientists. They have a thorough understanding of the physical model they are building, but programming is a secondary skill for them, and they often do not have the detailed understanding of data representation and other relevant computing issues.

Thus, when developing a data repository for use with the GLRRM, I wanted to build something more robust. I chose the analogy of a bank vault. Rather than just spreading out the data on a table for every model to use and, more importantly, modify at will, the data will be stored in the vault. The models are not allowed to access the vault directly; rather they go through a teller. To get data, they simply request what they want and the teller retrieves the appropriate file from the vault, giving them a copy. When they want to store data, they give the data to the teller along with a description of exactly what they are storing, and the teller stores it in the vault, replacing any existing data with the same description.

Subsequent discussions by the team tasked with developing the GLRRM settled on Python as the development language, and development of the data repository was assigned to Tim Hunter at NOAA’s Great Lakes Environmental Research Laboratory (GLERL). Funding was received from the Inland Waters Initiate to also fund work by an additional (student?) programmer on this part of the project. Due to a number of issues, including scheduling, prompted GLERL leadership decided to assign this funding to James Kessler, an employee of the Cooperative Institute for Great Lakes Research (CIGLR) who is located at GLERL’s laboratory. Tim and James have been working on development of the repository along with associated file I/O procedures.

**Overview**

GLRRM’s DataVault class contains a python dictionary object as the underlying vault structure. That dictionary stores objects of type DataSeries. Each DataSeries object stores a single set of physical data, for example daily runoff to Lake Erie in millimeters for the period 01/01/1990 to 12/31/2015. Specification of each metadata item in a DataSeries object is accomplished via a short text string. When data is stored into the vault, it will be normalized to a standard value for units, and when that same data is requested again, unit conversion will be done to match the requested units.

**Details**

There are five types of metadata associated with a DataSeries object. They are:

kind (nbs, runoff, level, flow, etc.)

units (meters, cubic feet per second, etc.)

interval (daily, weekly, etc.)

location (Lake Superior, Welland Canal, etc)

dates (start and end dates)

The first four of those are stored as short text strings and the dates are stored as standard datetime.date objects. In order to facilitate ease of use when working with the text metadata, we have defined four metadata classes that function like translators. Each of these metadata classes has a pair of two-dimensional tuple structures, one for input names and one for output names. In each of those, there is a tuple defined for each valid element (e.g. nbs, runoff, flow), and each of those tuples is a list of the valid names for that element. For example, the DataKind class defines all of the different kinds of data. The input names tuple contains a list tuple for all of the valid nbs names, one for the valid runoff names, one for the valid flow names, etc. Those tuples can have any number (>1) of entries. The first entry in the list is the short “primary” name, and the remaining entries are all acceptable variants that will be translated into that primary name. This makes it easier to translate any name we might read from a file into the primary name that is used throughout the processing code. For example, nbs might be specified in a file as “nbs” or “net basin supply” or “net\_basin\_supply”. The translator will accept any of those variants and return “nbs”. Additionally, the output names tuple provides a couple of standard variants (short and long) for use in output files.

These metadata classes are implemented as derived classes from a class I have named BaseMeta, which provides the common functionality. Variables should never be declared as being of type BaseMeta(); it is only for use as the base class of the metadata classes, and has no strings defined. The derived subclasses are:

DataKind

The DataKind class defines an object for specifying metadata about the kind of a dataset. Kind is something that in other contexts might often be referred to as the data “type”. For example, precipitation, runoff, channel flow, or lake level. I have chosen to avoid the word “type”, however, because that is a python reserved word and it could easily get syntactically messy or illegal when using “type” in the code.

DataUnits

The DataUnits class defines an object for specifying metadata about the units of a dataset. Examples include inches, square meters, m3/sec.

DataInterval

The DataInterval class defines an object for specifying metadata about the interval of a dataset. Valid options are currently daily, weekly, qtr-monthly and monthly.

DataLocation

The DataLocation class defines an object for specifying metadata about the location or geographic extent of a dataset. Examples include Lake Superior, Detroit River, Welland Canal.

The available methods are:

DataKind(initvalue)

DataInterval(initvalue)

DataUnits(initvalue)

DataLocation(initvalue)

The constructor \_\_init\_\_() takes one optional argument, the initial value. This value is specified by one of the valid input strings. If not specified, the default “undefined” value will be assigned (index value of 0). Strings are converted to all lowercase prior to comparison. As an example, the following three lines have identical results:

a = DataInterval(“mon”)

a = DataInterval(“MON”)

a = DataInterval(“moNthLY”)

className()

Returns the class name string, e.g. “DataKind”.

primaryName()

Get the primary name associated with the metadata object. For example, if the object is a DataKind object created by a=DataKind(‘net basin supply’); then a.primaryName() will return ‘nbs’.

inputName(index)

Get the specified input name string for the value. Index is used to specify which of the defined strings from the tuple to get, and defaults to zero if not specified. For example, if the object is a DataKind object for net basin supply, the default return value is ‘nbs’, but if index is set to 2 it will return ‘net basin supply’. Note that index values outside the valid range will be collapsed to the nearest extent of the valid range, so negative values become zero, and values>max become the max valid value.

Proper use of this function will depend on the programmer examining the entries in the \_inputString tuple for the metadata class.

outputName(index)

Same idea as inputName, but this time using the values specified in \_outputStrings.

outputNameShort()

Get the shortest (2 or 3 character) output name string for the value. This is equivalent to getOutputName(0).

outputNameLong()

Get the longest output name string for the value, assuming that the strings are arranged in recommended order from short to long. This is equivalent to getOutputName(<max index>).

intValueFromString(item)

Get the index value that corresponds to the item string. Item string can be any of the valid input strings for this data kind/interval/location. For example, if the object is a DataInterval object representing quarter-month data, and item=’qtrmon’, this will return the value 3.

DataSeries

An object that stores a single timeseries of data (as a python list) along with all of the associated metadata items.

Methods:

DataSeries(kind, units, freq, loc, first, last, values)

The constructor method for a DataSeries object takes seven arguments; Four metadata specifications, two dates, and the list of data values (float values). All arguments MUST be specified and valid.

add\_data(newDS)

newDS is a DataSeries object that contains a new set of data to be added to the existing contents of the current object. Data in newDS will overwrite existing values if there is overlap. If there is a gap between the existing and new data, then missing\_data values will be used to fill the gap. If the new data is in a different units from the existing data (e.g. mm vs inches), the new data will be converted to match the existing data. If there is a mismatch in data interval or location an exception will be raised, and the data will not be merged.

getKind()

Returns the primary name for this DataSeries object’s kind. e.g. If it has precipitation data, then this function returns ‘prc’.

getUnits()

Returns the primary name for this DataSeries object’s units. e.g. If the data is stored in mm, then this function returns ‘mm’.

getFreq()

Returns the primary name for this DataSeries object’s interval. e.g. If it has daily data, then this function returns ‘dy’.

getLocation()

Returns the primary name for this DataSeries object’s location. e.g. If it has data for Lake Ontario, then this function returns ‘on’.

printSummary()

Prints a simple quick and dirty summary of the contents, including the data values list. Mainly useful for debugging.

printOneLineSummary()

Prints a single-line summary of the metadata contents (no values). Mainly useful for debugging.

getOneLineSummary()

Similar to printOneLineSummary(), but just returns a string without actually printing it. Mainly useful for debugging.

DataVault

Stores items of type DataSeries and provides methods to store and retrieve those objects.

Methods:

DataVault()

The constructor method for a DataVault object takes no arguments. My expectation is that the main GLRRM framework initialization will create a single object of this type, and then the various component models will all access that global object.

deposit(ds)

Deposit a complete DataSeries object into the vault. Compare to existing data objects in the vault. If there is a match (kind, interval and location) then merge this new data with the old, overwriting the old values anyplace where they overlap. If there is a gap between the two data items, then the new merged item will contain the missing data indicator value for that gap period. If no matching data exists, then this new item is just added. The method returns True or False, indicating the success of the operation.

deposit\_data(kind=k, units=u, freq=f, loc=l, first=sdate, last=edate, values=dv)

Deposit a set of data into the vault without explicitly creating a DataSeries object first. Note that this method will, itself, create a DataSeries object with the specified metadata and datavalues, then use the normal DataVault.deposit() method to add it to the vault. The method returns True or False, indicating the success of the operation.

**Usage Examples**

theVault = DataVault()

ds = DataSeries()

flag = theVault.deposit(ds)

flag = theVault.deposit(kind=’flow’, units=’cms’, freq=’monthly’, loc=’niariv’, first=’1950-01-01’, last=’2010-12-31’, values=myNiagaraFlows)