Programmer's Guide to using Databank

Foreword:
This is an instructional guide to using the python-coded Databank package developed for use in the yet-to-be-coded Great Lakes Regulation and Routing Model (GLRRM). Databank provides functionality that will serve as a backbone for handling data used by the new GLRRM. Like any python package, its modules can be imported and used interactively or called by other python scripts. Throughout this documentation, the term "user" will refer to whom/whatever is calling databank's functions and methods whether it be an actual person (model developer) or a routing model.

Table of Contents:

1. 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. (preceding sentence is hard to understand) Accordingly, when development of the new model framework was discussed, a major focus was internal data handling.

The way COMMON blocks work can be described with 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, the goal was to build something more robust. The analogy of a bank vault is appropriate here. Rather than just spreading out the data on a table for every component model to use and, more importantly, modify at will, the data will be stored in the vault. The models

(users) 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 or appending any existing data with the same description. The role of the teller in this scenario is acted out by the Databank module which is essentially a comprehensive data handler.

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, 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 developed the data repository functionality along with associated file I/O procedures.

2. Overview

The databank module uses objects of class DataSeries to store all the physical data that will be used and created by the (to be developed) routing and regulation models (for example, daily runoff to Lake Erie in millimeters for the period 01/01/1990 to 12/31/2015.) These DataSeries objects have attributes containing both the metadata (described below) and the actual data values. Note that the attributes of any object in python can be can be accessed by appending a period to the object name followed by the attribute name (i.e. `someDataSeries.attributeName`)

DataSeries Attributes:

dataKind	(nbs, runoff, level, flow, etc.)
dataUnits	(meters, cubic feet per second, etc.)
dataInterval	(daily, weekly, quarter-monthly, monthly)
dataLocation	(Lake Superior, Welland Canal, etc)
startDate or "first"?	(python datetime object, 1900-01-01)
endDate or "last"?	(python datetime object, 1900-12-31)
dataVals	(list of floats containing the actual data values)

These DataSeries objects are stored in the "Vault" which, as mentioned above, can only be accessed through the methods of a vault object (acting as the bank teller). When the user (or model) deposits data into the Vault, databank first converts its values to a standard unit. If the vault already contains data of the same location, interval and kind, then the data being deposited is used to append or overwrite the existing data as necessary to prevent duplicate records of the same data from being stored in the vault. In order to withdraw data from the vault, the user must specify the metadata and Databank will return a DataSeries object containing the requested data for the given kind, interval, units, location and date range.

3. Getting Started

There are three modules in the Databank package:

- **databank.py** generates and modifies DataSeries objects and interacts with the Vault
- **databank_io.py** contains the functions to read from and write to text files.
- **databank_util.py** contains utilities useful for both databank and databank_io

The primary ways the user will interact with Databank are by creating DataSeries, depositing/withdrawing DataSeries, and reading/writing text files.

There are three different ways to create a DataSeries object:

I. Instantiate it directly with databank.DataSeries() method by specifying metadata and a list of data values
II. Read in data and meta data from a text file (return value is a DataSeries)
III. Withdraw data already stored in the vault (return value is a DataSeries)

There are two ways to deposit DataSeries:

I. Deposit a pre-existing DataSeries into the vault using the vault's deposit() method
II. Deposit a list of data values into the vault using the vault's deposit_data() method which requires that metadata is specified (because unlike a DataSeries, the list has no attributes)

There's only one way to: withdraw DataSeries objects, read from files and write to files. All the above common uses are demonstrated below:

```
# import the modules
import databank as db
import databank_io as io
import databank_util as util

# instantiate the Vault
the_vault = db.DataVault()


#
# PART 1: Three ways to create DataSeries objects:
#

# I) generate data as a list (in practice, this would be output from a model)
myNiagraFlows = [1.0, 2.0, 1.0, 1.5, 1.5, 1.0, 1.0, 1.5, 1.0, 1.5, 1.0, 1.5]

# use this list to generate a DataSeries by specifying metadata
dataseries1 = db.DataSeries(kind='flow', units='cms', intvl='monthly', loc='niariv', first='1900-01-01',
last='1900-12-31', values=myNiagraFlows)

# II) generate DataSeries by reading in a properly formatted text file
# CGLRRM formatted files are still accepted but they don't contain kind or location, so this metadata must
be specified calling read_file)
dataseries2 = io.read_file('somedatafile.txt')
classic_ds = io.read_file('someCLGRRMfile.txt', kind='prec', loc='erie')


# III) pretend vault already contains data and we want to withdraw some of it
dataseries3 = the_vault.withdraw(kind='nbs', units='m3', intvl='daily', loc='erie', first='2000-01-01',
last='2000-01-31')

#... alternatively, if we wanted Erie, daily net basin supply, for the same time range in cubic feet
dataseries_feet = the_vault.withdraw(kind='nbs', units='ft3', intvl='daily', loc='erie', first='2000-01-
01', last='2000-01-31')




#
# PART 2: Two ways to deposit data:
#

# I) Deposit a pre-existing Dataseries object. (use a dataseries from PART 1)
the_vault.deposit(ds=dataseries3)

# II) Deposit data as a list and specify all the metadata "on the fly"
stClrFlows = [5.0, 6.0, 6.5, 6.0, 6.0, 6.0, 6.5, 5.0, 5.5, 5.5, 8.0, 9.0]
the_vault.deposit_data(kind='flow', units='cms', intvl='monthly', loc='stclr', first=1900-01-01',
last='1900-12-31', values=stClrFlows)


#
# PART 3: Write to file (withdraw and read_file are demonstrated above)
#
io.write_file(filename='myoutput.txt', file_format='column', ds=dataseries1)
```

4. Module Documentation:

Databank_io:

Databank_io (short for input/output) contains two functions that are intended to be called by the user: `read_file` and `write_file`. When using python in interactive mode, the user can call `help(databank_io)` to see the following docstrings which describes each function:

```
read_file(filename, kind=None, units=None, intvl=None, loc=None)
               Read in data and metadata from filename and return a dataseries object.

               Parameters
               ----------
               filename: string
                      The desired filename to read from.
               kind : string, optional
                      The kind of data ('precip') to be read in.  If filename does not contain kind
                      metadata (header info), kind must be set on the function call.
               units : string, optional
                      The units of data ('mm') to be read in.  If filename does not contain units metadata
                      (header info), units must be set.
               intvl : string, optional
                      The interval of data ('weekly') to be read in.  If filename does not contain intvl
                      metadata (header info), intvl must be set.
               loc : string, optional
                      The loc of data ('superior') to be read in.  If filename does not contain loc
                      metadata (header info), loc must be set.

               Returns
               -------
               dataseries : object
                      an object of class DataSeries with attributes dataKind, dataUnits, dataInterval,
                      dataLocation, startDate, endDate, and dataVals (the actual data)

               Notes
               -----
              If keywords kind, units, intvl, or loc are included in function call AND included in the
              metadata of filename, then they must match otherwise an exception will be raised.

              Classic CGLRRM files were NOT required to include loc or kind in the metadata (header info)
              so these keyword args will likely be mandatory when processing these files.

write_file(filename, file_format, dataseries, overwrite=False, width=9, prec=2)
               Write dataseries (metadata and datavals) to filename

               Parameters
               ----------
               filename: string
                      The desired filename to write to.
               file_format: string
                      The format of the output file ("table" or "column") note that weekly data only can be
                      written as column format
               dataseries: object
                       The dataseries object to be written to filename.
               overwrite: boolean, optional
                      Flag to indicate whether or not to overwrite existing file named "filename". default
                      is False (i.e don't overwrite files)
               width: string, optional
                      Used to specify the formatting (column width) of values written to filename. If set,
                      should be used in conjunction with prec. MINIMUM VALUE = 6 to properly print the fill
                      value of -999.9
               prec: string, optional
                      Used to specify the formatting (decimal precision) of values written to filename.
                      if set, it should be used in conjunction with width. MINIMUM VALUE = 1  to properly
                      print the fill value of -999.9
```

About file format used by databank_io:

Writing

Users are given two options for `file_format` when writing output files: "table" and "column", which are both comma separated (a trailing comma before the newline is optional). The specific formatting of files is described below. Data of each interval can be written in either format with the exception of "weekly" data which must be column format to avoid the usage of a text file with 53 columns! The optional arguments `width` and `precision` allow the user to specify the formatting of the actual data values when writing. For example, to write all data values to a file with formatting "123.56789", set `width=9` and `precision=5` (in C-style formatting this would be "%9.5f")

Reading

When reading data files, the format of the file need not be specified because `read_file` determines the format based on the combination of metadata, and number of columns. `read_file` accepts both "table" and "column" as well as "CGLRRM" formatted files which refers to data files from the classic fortran-based routing models. As indicated in the above docstring, "CGLRRM" formatted files typically did not contain `location` or `kind` metadata, so these optional parameters must be set when reading in CGLRRM data files.

Fill Values

Missing/unknown values should be set as -999.9. This value also serves as a placeholder in some table-formatted files. (e.g. the column for the 31st day in months that only have 30 days.)

General Format

```
HEADER LINES CONTAINING METADATA
# Any user comments
# COLUMN TITLE LINE
<DATE1>,  <VALUE1>, <VALUE2>, …, <VALUEN>
<DATE2>,  <VALUE1>, <VALUE2>, …, <VALUEN>
<DATE3>,  <VALUE1>, <VALUE2>, …, <VALUEN>
<DATE4>,  <VALUE1>, <VALUE2>, …, <VALUEN>
```

Below is a snippet of data files for each combination of `interval` and `file_format`:

DAILY/TABLE: (columns for days 7-25 are omitted so each line fits in this document)

```
KIND:nbs
UNITS:m3
INTERVAL:dy
LOCATION:er
# file created: 2018-05-22 22:20:54
#YYYY-MM,    D1,     D3,     D4,     D6,  ...,    D26,    D27,    D28,    D29,    D30,    D31
1990-01, -999.9, -999.9, -999.9,   74.0,  ..., -100.0,   12.0,  -86.0,   43.0,   93.0,   62.0
1990-02,   57.0,  -78.0,   97.0,    9.0,  ...,  -50.0,  -56.0,  -36.0, -999.9, -999.9, -999.9
1990-03,  -21.0,  -23.0,   46.0,  -20.0,  ...,   66.0,   -4.0,   51.0,  -39.0,  -64.0,   13.0
1990-04,  -21.0,   29.0,  -80.0,  -18.0,  ...,  -87.0,   76.0,   -9.0,   29.0,   73.0, -999.9
1990-05,  -59.0,   80.0,   25.0,   72.0,  ...,  -58.0,   61.0,   98.0,  -66.0,   74.0,   32.0
1990-06,   43.0,   29.0,   95.0,  -67.0,  ...,   95.0,   79.0,  -69.0,  -94.0,  -49.0, -999.9
1990-07,   83.0,  -49.0,   85.0,   48.0,  ...,   78.0,  -15.0,  -36.0,  -38.0,   99.0,  -59.0
1990-08,  -69.0,   57.0,  -37.0, -100.0,  ...,  -98.0,  -87.0,   38.0,  -46.0,  -76.0,  -87.0
1990-09,   41.0,   -8.0,  -34.0,    5.0,  ...,  -92.0,  -22.0,   40.0,  -49.0,  -99.0, -999.9
```

DAILY/COLUMN:

```
KIND:nbs
UNITS:m3
INTERVAL:dy
LOCATION:er
# file created: 2018-05-29 01:04:31
#YYYY-MM-DD,      VAL
1990-01-01,   -999.90
1990-01-02,   -999.90  # Comments can go here too!
1990-01-03,   -999.90  # Note: missing data for first 4 days.
1990-01-04,   -999.90
```

```
1990-01-05,     29.00
1990-01-06,     74.00
1990-01-07,    -92.00
1990-01-08,    -43.00
1990-01-09,    -88.00
1990-01-10,    -45.00
1990-01-11,     62.00
1990-01-12,     -4.00
```

WEEKLY: (based on CGLRRM convention, date is the day of the preceding Friday for a given week)

```
KIND:prc
UNITS:10cms
INTERVAL:wk
LOCATION:det
# file created: 2018-05-22 22:49:39
#YYYY-MM-DD,     VAL
1922-10-06,    29.00
1922-10-13,    53.00
1922-10-20,    58.00
1922-10-27,    60.00
1922-11-03,    59.00
1922-11-10,    57.00
1922-11-17,    57.00
1922-11-24,    56.00
1922-12-01,    57.00
1922-12-08,    52.00
1922-12-15,    55.00
1922-12-22,    55.00
1922-12-29,    55.00
1923-01-05,    56.00
1923-01-12,    56.00
1923-01-19,    56.00
1923-01-26,    56.00
1923-02-02,    55.00
1923-02-09,    54.00
```

QUARTER-MONTHLY/TABLE: (this example uses a non-default width/precision float formatting)

```
KIND:prc
UNITS:mm
INTERVAL:qm
LOCATION:er
# file created: 2018-05-22 22:54:46
#YYYY-MM,             Q1,             Q2,             Q3,             Q4
1900-01,         1.23000,        1.78000,        1.60000,        1.02000
1900-02,         0.98000,        0.98000,        1.03000,        1.05000
1900-03,         1.04000,        1.00000,        0.98000,        0.96000
1900-04,         1.56000,        1.83000,        1.49000,        0.98000
1900-05,         0.97000,        0.99000,        1.03000,        1.04000
1900-06,         1.03000,        1.00000,        0.98000,        0.99000
1900-07,         1.68000,        1.74000,        1.33000,        0.98000
```

QUARTER-MONTHLY/COLUMN:

```
KIND:prc
UNITS:mm
INTERVAL:qm
LOCATION:er
# file created: 2018-05-22 22:52:57
#YYYY-MM-QQ,      VAL
1900-01-01,     1.2300
1900-01-02,     1.7800
1900-01-03,     1.6000
1900-01-04,     1.0200
1900-02-01,     0.9800
1900-02-02,     0.9800
1900-02-03,     1.0300
1900-02-04,     1.0500
1900-03-01,     1.0400
```

MONTHLY/TABLE:

```
KIND:nbs
UNITS:cms
INTERVAL:mn
LOCATION:mi
# file created: 2018-05-24 01:07:20
#YYYY,    M1,     M2,     M3,     M4,     M5,     M6,     M7,     M8,     M9,    M10,    M11,    M12
1999, -999.9, -999.9,   17.0,  -54.0,  -28.0,  -70.0,   93.0,   46.0,   12.0,  -14.0,   57.0,   58.0
2000,  -58.0,   25.0,  -51.0,   38.0,  -79.0,   50.0,  -76.0,   71.0,  -87.0,  -22.0,   19.0,   59.0
2001,  -71.0,    0.0,  -94.0,  -83.0,   17.0,   35.0,   95.0,   37.0,  -97.0,  -91.0,   20.0,  -55.0
2002,   89.0,  -11.0,  -96.0,  -36.0,  -26.0,  -17.0,   95.0,  -80.0,   28.0,   94.0,  -18.0, -999.9
```

MONTHLY/COLUMN:

```
KIND:nbs
UNITS:cms
INTERVAL:mn
LOCATION:mi
# file created: 2018-05-24 01:09:56
#YYYY-MM,   VAL
1999-01, -999.9
1999-02, -999.9
1999-03,   17.0
1999-04,  -54.0
1999-05,  -28.0
1999-06,  -70.0
1999-07,   93.0
1999-08,   46.0
1999-09,   12.0
1999-10,  -14.0
1999-11,   57.0
1999-12,   58.0
2000-01,  -58.0
2000-02,   25.0
```

Databank_util:

Databank_util is a module that contains many utilities that are used by databank and databank_io. Some of these utilities (methods and functions) may also be useful to the user and have three major categories:

I. Basic unit conversion (“feet” to “meters”)
II. Dimensional conversion (a flow rate to a total volume between two dates)
III. Date utilities (# of days in a given month or quarter-month)

In interactive mode, the user can call help(databank_util) to access the full docstrings of all utilities.

Utilities in databank_util:

```
days_in_month
getFridayDate
last_day_of_month
date_from_entry
days_in_qtr_mon
getQtrMonthStartEnd
convertValues
linearConvert
arealConvert
cubicConvert
rateConvert
valueLinearToRate
valueCubicToRate
valueRateToLinear
valueRateToCubic
trimDataValues
linearToRate
rateToLinear
cubicToRate
rateToCubic
```

Databank:

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,
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 the primary name, "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 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. The word "type" was avoided 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.

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. The 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.