pyCDB Documentation

Release 0.5.0-alpha

J. Urban et al.

Jul 28, 2017

Contents

Contents:

CHAPTER 1

Description

The Compass DataBase (CDB) is a lightweight system designed for storing the COMPASS (IPP Prague) tokamak experimental data. It can equally well be used for any tokamak or generally any device that repeatedly produces experimental data.

CDB uses HDF5 (or NetCDF 4) files to store numerical data and a relational database, actually MySQL, to store metadata. The core application is implemented in Python (pyCDB). Cython is used to wrap the Python code in a C API. Matlab, IDL etc. clients can then be built using the C API.

There are several major advantages of this scheme:

- Vast of the required functionality is readily implemented and available for numerous operating systems and applications (high/low level data input/output, database functionality).
- Data can be stored on any file system (local or remote), no need for specific protocol.
- Rapid, platform independent development in Python.

CDB has a possibility to store the information about the data acquisition sources (channels) of the data. The database contains information about DAQ channels associacions to physical quantities.

An important point in CDB is its *never overwrite* design. Anything stored in CDB cannot be overwriten (at least using the standard API); instead, revisions are possible as corrective actions.

The data model

A relational database is used to store metadata of the numerical date. Metadata include physical quantities names, units, information about axes (which themselves are physical quantities and are the same entities as any other quantities).

generic_signals Describe physical quantities stored in the database. In particular, contain names, units, axes id's (axis are treated as any other signals), description, signal type (FILE or LINEAR), record numbers validity range, and data source.

data_sources Used as primary grouping criterion. Contain name, description and the directory name of the data files.

data_files List of all data files in the system. Each file can contain one or more signals. Data files are stored in subdirectories specified in the data source under the main CDB directory. **files_status** stetes whether a file is ready for reading.

data_signals In fact, data signals are instances od generic signals. A data signal either points to data in a data file or contains only coefficients for linear function (LINEAR signal). Data signal contains record number to which the data belong. Revisions can be created when a correction to a signal is needed.

Each data signal contains *offset* and *coefficient* used either for a linear signal construction or for linear tranformation of data stored in a file. See *Linear signals with get_signal_data* for details. *time0* specifies the time of the first data point for time-dependent signals.

shot_database Contains record numbers—unique numbers characterizing a data set. This is mostly a tokamak shot, can however be a simulation, a DAQ system test etc. Tokamak shots also have **shot_numbers**. Data files for a particualr shot are stored in **record_directories**.

FireSignal_Event_IDs CDB can be used as storage system for FireSignal. In this case, this table relates FireSignal id's and CDB record numbers.

Data acquisition management

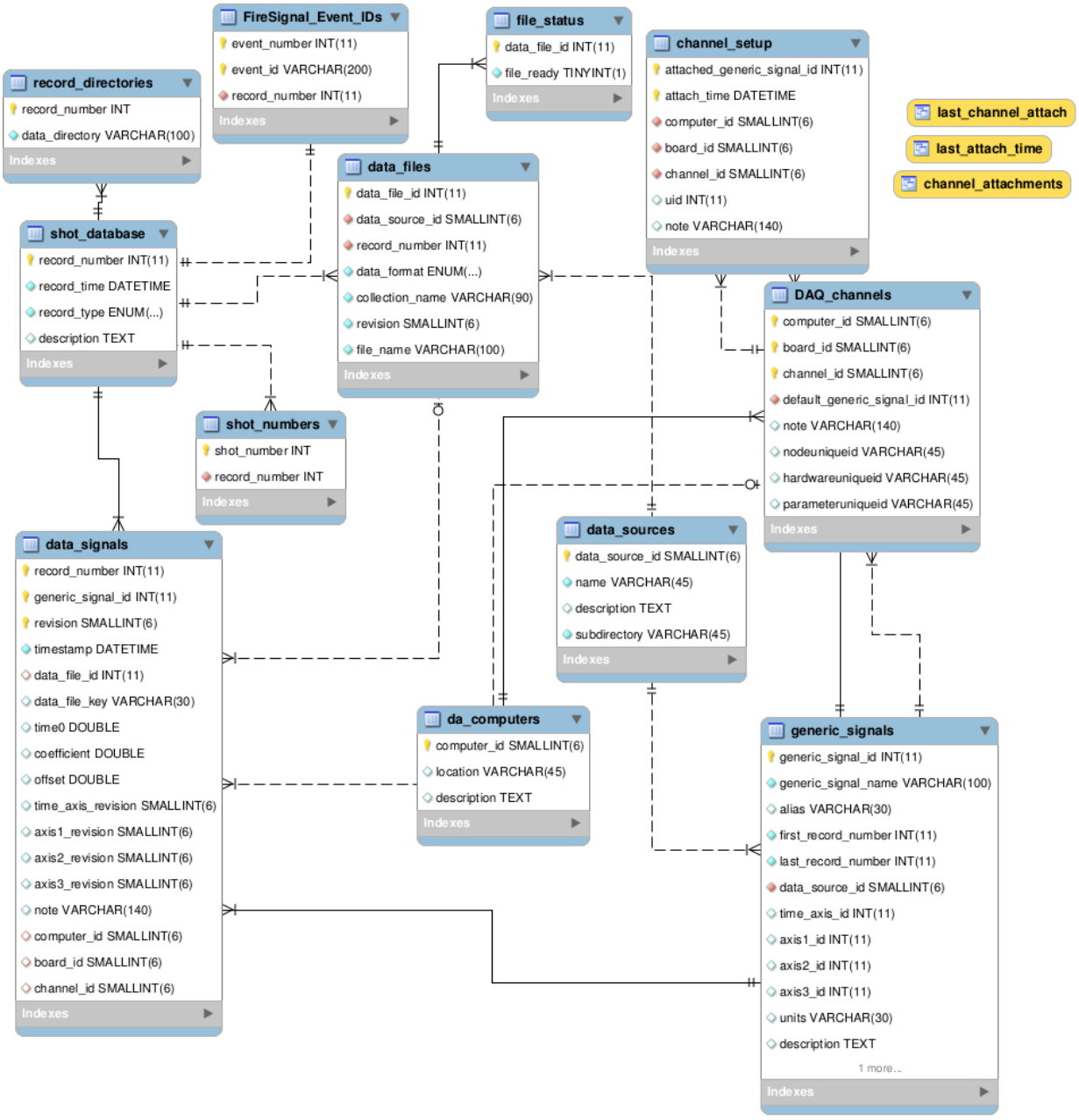
CDB has a possibility to track information about DAQ A/D channels. Each data acquisition system can be associated ("attached") with a physical quantity (generic_signal) it outputs.

DAQ_channels List of all DAQ channels available. Unique identification consist of *computer_id*, *board_id*, *channel_id*. In case CDB is used with FireSignal, *nodeuniquid*, *hardwareuniqueid* and *parameteruniqueid* relates the two databases.

Each DAQ channel has a *default_generic_signal_id*, which is the id of a (unique to the channel) generic signal associated with the channel if no other generic signal is associated. The reason for this is that, in CDB, every **data_signal** must have a *generic_sinal_id*.

channel_setup This table tells to which *generic_signal_id* is a DAQ channel associated (attached). It's an event-style table containing association events (date, time, user id and a note of a DAQ channel association to a generic signal.)

Database structure



CHAPTER 2

Installation

Dependencies

For pyCDB

- Python 2.7+
- MySQLdb or pymysql
- numpy
- h5py
- pint
- matplotlib (for the example only)

For JyCDB - Java, Matlab, IDL clients

- Jython

For cyCDB (the C-interface)

- Cython (tested with versions 0.14 and 0.15)

Clone CDB repository

CDB's Mercurial repository is hosted at bitbucket.org/compass-tokamak/cdb. Simply clone it via

```
hg clone https://bitbucket.org/compass-tokamak/cdb
```

pyCDB

Just place pyCDB on Python path and `import pyCDB`

If pyCDB is not on Python path, it is possible to modify the path:

```
export CDB_PATH=/path/to/cdb/src && export PYTHONPATH=$PYTHONPATH:/path/to/cdb/src
```

cyCDB

Run `make` in the src directory. This step is optional.

Database

CDB needs MySQL database. Import *database/mysql_setup.sql* to create the *CDB* database structure, run

```
mysql -u root -p < database/CDB_schema.sql
```

To create CDB users and the testing database (modify passwords for production use), run

```
mysql -u root -p < CDB_users.sql
```

CHAPTER 3

Configuration

CDB configuration parameters are sought in:

1. environment variables
2. ./.CDBrc
3. $HOME/.CDBrc

CDBrc file explanation, environment variables have the same names:

```
# CDB configuration file
#
# Edit and save to .CDBrc
#
# Possible directories, ordered by ascending priority
# - $CDB_PATH/..
# - user home directory
# - current directory
# - alternatively, CDB can be configured with environment variables

# Database configuration
[database]
# Root data directory
CDB_DATA_ROOT = /var/local/CDB

# MySQL host name or IP (:port)
CDB_HOST = localhost
# MySQL user name
CDB_USER = CDB_test
# MySQL password
CDB_PASSWORD = cdb_data
# MySQL database name
CDB_DB = CDB_test

# Logging configuration
[logging]
```

```
# Screen logging level: INFO, DEBUG, WARNING, ERROR, NOTSET
CDB_LOG_LEVEL = WARNING
# File log file name
CDB_LOG_FILE =
# File log level
CDB_FILE_LOG_LEVEL = INFO

# logbook configuration (optional)
[logbook]
CDB_LOGBOOK_HOST = localhost
CDB_LOGBOOK_USER = CDB
CDB_LOGBOOK_PASSWORD = cdb_data
CDB_LOGBOOK_DB = logbook
```

CHAPTER 4

Usage

Reading data

Basic example in Python

First import pyCDB and connect to the database:

```
from pyCDB.client import CDBClient
cdb = CDBClient()
```

Retrieve references for some generic signal:

```
generic_signal_name = 'electron density'
# get the full generic signal reference (all columns)
generic_signal_refs = cdb.get_generic_signal_references(generic_signal_name = generic_
↪signal_name)
# get signal references
signal_refs = cdb.get_signal_references(record_number=-1,generic_signal_id=generic_
↪signal_refs[0]['generic_signal_id'])
```

Now get the signal data, including description and axes, using *pyCDB.client.CDBClient.get_signal()*:

```
sig = cdb.get_signal(signal_ref=signal_refs[-1])
```

Matlab

Use *cdb_client* to instantiate a client:

```
cdb = cdb_client();
```

To get the signal data including axes use `cdb_client.cdb_get_signal()`

```
signal = cdb.get_signal(str_id)
```

Signal id's

Generic signal is uniquely identified either by

- numeric id (generic_signal_id)
- alias + record number
- name (generic_signal_name) + data source id + record number

CDB interface supports generic signal string id's in the following forms:

- *alias_or_name* - search by alias first, if no match is found search by name (>1 results possible)
- *name/data_source_id* - generic signal name followed by '/' and data_source_id (data source name or numeric id)
- *generic_signal_id* - numeric id of the generic signal

See also *pyCDB.client.decode_generic_signal_strid()*.

Data signal (single data set) is uniquely identified by

- generic signal id (numeric) + record number + revision

Data signal string id is in the form of:

```
id_type:generic_signal_string_id:record_number:revision[units]
```

For example, this refers to EFIT psi 2D, shot 4073, last revision:

```
CDB:psi_2D:4073:-1[default]
```

which is equivalent to:

```
psi_2D:4073
```

See also *pyCDB.client.decode_signal_strid()*.

Linear signals with get_signal_data

Described below is the logic for linear signals, implemented in *pyCDB.client.CDBClient.get_signal_data()*.

- number of axes > 1 –> **unresolved**
- number of axes = 1
  - axis is linear
    - axis is time_axis
      - time_limit provided: n_samples = int(ceil((time_limit - offset) / coefficient))
      - n_samples not provided –> **unresolved**
      - **result** = offset + coefficient*(axis_data[0..n_samples] - time0)
    - axis is not time_axis

            - n_samples not provided –> **unresolved**
            - **result** = offset + coefficient*axis_data[0..n_samples]
        - axis contains data
            - axis is time_axis
                - **result** = offset + coefficient*(axis_data - time0)
            - axis is not time_axis
                - **result** = offset + coefficient*axis_data
- number of axes = 0
    - n_samples and x0 provided (x0 is optional, defaults to 0)
        - **result** = x0 + [0,1, .., n_samples-1]*coefficient
    - x0 and x1 provided
        - **result** = [x0, x0+coefficient, x0+2*coefficient, .., x1]
    - otherwise
        - **result = function**: f(i) = offset + coefficient * i

Writing data

First create a new data file record (go to the last step for liner signals), providing data source id, record number and collection (a base name for the data file chosen by the user):

```
file_ref = cdb.new_data_file(collection_name, data_source_id =
           data_source_id, record_number = record_number, \
           file_format = "HDF5")
```

Now you can create the file and fill with data, e.g.:

```
import h5py
fh5 = h5py.File(file_ref['full_path'],'w')
grp_name = 'raw data'
data_file_key = grp_name + '/' + generic_signal_name
f_grp = fh5.create_group(grp_name)
f_grp.create_dataset(generic_signal_name, data=numpy_data)
fh5.close()
```

One has to say when the file is ready (for reading):

```
cdb.set_file_ready(file_ref['data_file_id'])
```

Finally, the CDB database must know what signals are stored in the file. Linear signals are created solely by this step:

```
cdb.store_signal(generic_signal_id, \
                    record_number=record_number, \
                    data_file_id=file_ref['data_file_id'], \
                    data_file_key = data_file_key,\
                    offset=0.5, coefficient=1.3, time0=-0.2, \
                    computer_id=1, board_id=1, channel_id=1, \
                    note='my first stored signal')
```

CHAPTER 5

CDB primer for (future) admins

This tutorial demonstrates some basics of CDB when starting from an empty database. This is useful for testing a new installation, particularly by (future) admins and developers.

```
In [1]: %matplotlib inline
```

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
```

Import CDB and connect

```
In [3]: from pyCDB.client import CDBClient
        cdb = CDBClient()
```

Populate SQL database

For simplicity, we will populate the dabase for this tutorial by SQL directly. Login to MySQL / MariaDB with enough persmisisons (e.g. as root) and execute:

```
INSERT INTO `da_computers` (`computer_id`, `computer_name`, `location`,␣
↪`description`) VALUES
(1, 'daq_comp_1', NULL, NULL);

INSERT INTO `data_sources` (`data_source_id`, `name`, `description`, `subdirectory`)␣
↪VALUES
(1, 'MAGNETICS', NULL, 'magnetics'),
(2, 'EFIT', NULL, 'efit'),
(3, 'DAQ', NULL, 'daq');

INSERT INTO `generic_signals` (`generic_signal_id`, `generic_signal_name`, `alias`,␣
↪`first_record_number`, `last_record_number`, `data_source_id`, `time_axis_id`,␣
↪`axis1_id`, `axis2_id`, `axis3_id`, `axis4_id`, `axis5_id`, `axis6_id`, `units`,␣
↪`description`, `signal_type`) VALUES
(1, 'daq_time', NULL, 1, -1, 3, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 's', NULL,
↪'LINEAR'),
```

```
(2, 'daq_comp_1_channel_1_1', NULL, 1, -1, 3, 1, NULL, NULL, NULL, NULL, NULL, NULL,␣
↪NULL, NULL, 'FILE'),
(3, 'I_plasma', 'Ip', 1, -1, 1, 1, NULL, NULL, NULL, NULL, NULL, NULL, 'A', NULL,
↪'FILE'),
(4, 't_efit', NULL, 1, -1, 2, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 's', NULL,
↪'FILE'),
(5, 'I_plasma', NULL, 1, -1, 2, 4, NULL, NULL, NULL, NULL, NULL, NULL, 'A', NULL,
↪'FILE');

INSERT INTO `DAQ_channels` (`computer_id`, `board_id`, `channel_id`, `default_generic_
↪signal_id`, `note`,
`nodeuniqueid`, `hardwareuniqueid`, `parameteruniqueid`) VALUES
(1, 1, 1, 2, NULL, NULL, NULL, NULL);

INSERT INTO `shot_database` (`record_number`, `record_time`, `record_type`,␣
↪`description`) VALUES
('1', NOW(), 'EXP', NULL);

INSERT INTO `record_directories` (`record_number`, `data_directory`) VALUES
('1', '1');
```

Work with CDB

List all generic signals

```
In [4]: gs_list = cdb.get_generic_signal_references()
        for gs in gs_list:
            ds = cdb.get_data_source_references(data_source_id=gs.data_source_id)[0]
            print("id: {generic_signal_id}, name/source: {generic_signal_name}/{data_source_name}".fo
                    **gs, data_source_name=ds.name))
```

```
id: 1, name/source: daq_time/DAQ
id: 2, name/source: daq_comp_1_channel_1_1/DAQ
id: 3, name/source: I_plasma/MAGNETICS
id: 4, name/source: t_efit/EFIT
id: 5, name/source: I_plasma/EFIT
```

```
In [5]: cdb.get_data_source_references(data_source_id=gs.data_source_id)
```

```
Out[5]: [OrderedDict([('data_source_id', 2),
                      ('name', 'EFIT'),
                      ('description', None),
                      ('subdirectory', 'efit')])]
```

Populate our first shot with data

```
In [6]: shot = cdb.last_record_number()
        print(shot)
```

```
1
```

```
In [7]: # example data + noise
        n = 100000
        dt = 1e-3
        t_data = np.r_[:n*dt:dt]
        Ip_data = 1e6 * (1 - np.linspace(-1, 1, n)**8)**0.5
        Ip_data_noisy = Ip_data * (1 + (0.05*np.random.rand(n) - 0.025))
```

Use `put_signal` to store data. `put_signal <http://cdb.readthedocs.io/en/latest/reference.html?highlight=put_signal#pyCDB.client.CDBClient.put_signal>`__ is a convenience wrapper around the full (and more flexible) store signal procedure, described in http://cdb.readthedocs.io/en/latest/usage.html#writing-data.

```
In [8]: gs_id = 3
        cdb.put_signal(gs_id, shot, Ip_data_noisy, time0=0, time_axis_coef=dt)
```

```
Out[8]: OrderedDict([('record_number', 1),
                     ('generic_signal_id', 3),
                     ('revision', 1),
                     ('variant', ''),
                     ('timestamp', datetime.datetime(2017, 7, 28, 11, 50, 24)),
                     ('data_file_id', 1),
                     ('data_file_key', 'I_plasma'),
                     ('time0', 0.0),
                     ('coefficient', 1.0),
                     ('offset', 0.0),
                     ('coefficient_V2unit', 1.0),
                     ('time_axis_id', 1),
                     ('time_axis_revision', 1),
                     ('axis1_revision', 1),
                     ('axis2_revision', 1),
                     ('axis3_revision', 1),
                     ('axis4_revision', 1),
                     ('axis5_revision', 1),
                     ('axis6_revision', 1),
                     ('time_axis_variant', ''),
                     ('axis1_variant', ''),
                     ('axis2_variant', ''),
                     ('axis3_variant', ''),
                     ('axis4_variant', ''),
                     ('axis5_variant', ''),
                     ('axis6_variant', ''),
                     ('note', ''),
                     ('computer_id', None),
                     ('board_id', None),
                     ('channel_id', None),
                     ('data_quality', 'UNKNOWN'),
                     ('deleted', 0)])
```

```
In [9]: cdb.put_signal(5, shot, Ip_data, time_axis_data=t_data)
```

```
Out[9]: OrderedDict([('record_number', 1),
                     ('generic_signal_id', 5),
                     ('revision', 1),
                     ('variant', ''),
                     ('timestamp', datetime.datetime(2017, 7, 28, 11, 50, 32)),
                     ('data_file_id', 2),
                     ('data_file_key', 'I_plasma'),
                     ('time0', 0.0),
                     ('coefficient', 1.0),
                     ('offset', 0.0),
                     ('coefficient_V2unit', 1.0),
                     ('time_axis_id', 4),
                     ('time_axis_revision', 1),
                     ('axis1_revision', 1),
                     ('axis2_revision', 1),
                     ('axis3_revision', 1),
                     ('axis4_revision', 1),
                     ('axis5_revision', 1),
```

```
                    ('axis6_revision', 1),
                    ('time_axis_variant', ''),
                    ('axis1_variant', ''),
                    ('axis2_variant', ''),
                    ('axis3_variant', ''),
                    ('axis4_variant', ''),
                    ('axis5_variant', ''),
                    ('axis6_variant', ''),
                    ('note', ''),
                    ('computer_id', None),
                    ('board_id', None),
                    ('channel_id', None),
                    ('data_quality', 'UNKNOWN'),
                    ('deleted', 0)])
```

Get the data back from CDB using the *string id*.

```
In [10]: str_id = 'Ip:{}'.format(shot)
         str_id

Out[10]: 'Ip:1'

In [11]: ip_sig = cdb.get_signal(str_id)
         ip_sig

Out[11]: { Generic signal name: I_plasma
          Generic signal alias: Ip
             Generic signal id: 3
                 Record number: 1
                      Revision: 1
                         Units: A
                          Data: shape = (100000,), dtype = float64
               More details in: ref, gs_ref, daq_attachment, file_ref}
```

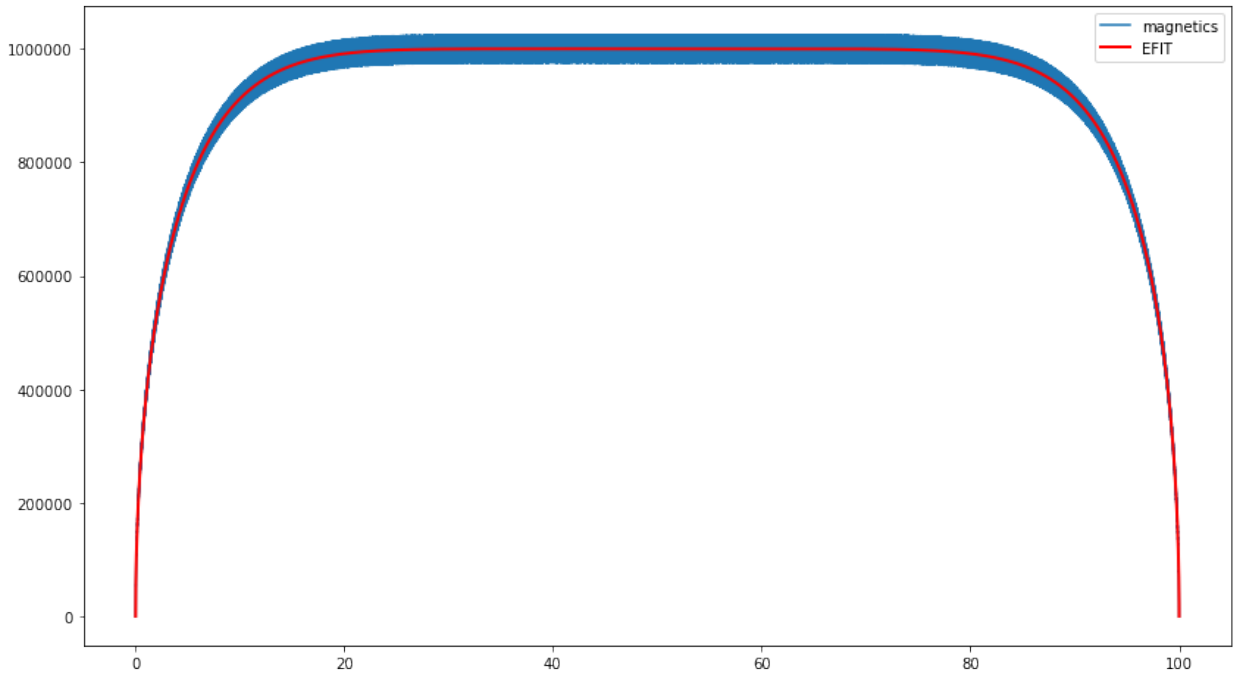
Now get the data using keyword arguments.

```
In [12]: ip_efit = cdb.get_signal(generic_signal_id=5, shot=shot)
         ip_efit

Out[12]: { Generic signal name: I_plasma
          Generic signal alias: None
             Generic signal id: 5
                 Record number: 1
                      Revision: 1
                         Units: A
                          Data: shape = (100000,), dtype = float64
               More details in: ref, gs_ref, daq_attachment, file_ref}

In [13]: fig, ax = plt.subplots(figsize=(14, 8))
         ax.plot(ip_sig.time_axis.data, ip_sig.data, label="magnetics")
         ax.plot(ip_efit.time_axis.data, ip_efit.data, c='r', lw=2, label="EFIT")
         ax.legend(loc='best')

Out[13]: <matplotlib.legend.Legend at 0x7fed7ace5128>
```


magnetics
EFIT
1000000
800000
600000
400000
200000
0
0
20
40
60
80
100

In [ ]:

CHAPTER 6

pyCDB reference

pyCDB.client

COMPASS database main module

Contains functions to connect, retrieve and store data using the COMPASS database.

class pyCDB.client.**CDBClient**(*host=None*, *user=None*, *passwd=None*, *db=None*, *port=3306*, *log_level=None*, *data_root=None*)

Fundamental connector class for CDB

CDB2FS_ref(*computer_id*, *board_id*, *channel_id*)

Return the CDB DAQ_channel reference of a CDB channel

FS2CDB_ref(*nodeuniqueid*, *hardwareuniqueid*, *parameteruniqueid*)

Return the CDB DAQ_channel reference of a FireSignal channel

add_unit(*unit_system_id*, *unit_name*, *auto_dim=False*, ***dimensions*)

Add unit to unit system.

Parameters **auto_dim** – if true, dimensions are inferred automatically.

Returns unit_id of the unit

Return type int

apply_unit_factor_tree(*signal*, *unit_tree*)

Apply unit conversion of signal data tree.

Parameters

- **signal** – a CDBSignal object with dependent axes and fully read data.
- **unit_tree** – a tree of units to convert to as returned by get_unit_factor_tree.

create_generic_signal(*generic_signal_name=None*, *alias=None*, *first_record_number=1*, *last_record_number=-1*, *data_source_id=None*, *time_axis_id=None*, *axis1_id=None*, *axis2_id=None*, *axis3_id=None*, *axis4_id=None*, *axis5_id=None*, *axis6_id=None*, *units=None*, *description=None*, *signal_type='FILE'*, ***kwargs*)

Creates a generic signal in database.

Returns the id of the new generic signal.

create_record(*record_type='EXP'*, *record_time=None*, *description=''*, *data_root=None*, *FS_event_number=0*, *FS_event_id=''*, *record_number=None*, *autonumber=False*)

Creates a new record in the database

Parameters

- **record_type** – 'EXP', 'VOID' or 'MODEL'
- **record_time** – default current, can be integer time stamp or a datetime.datetime obejct
- **description** – record description
- **data_root** – CDB data root path, see *get_data_root_path()*
- **FS_event_number** – FireSignal event number (optional) # TODO: Remove?
- **FS_event_id** – FireSignal event id (optional) # TODO: Remove?
- **record_number** – new record number (None for first available)
- **autonumber** – automatic numbering (effective only if record_number is not None)

Return type new record number

delete_signal(*str_id=''*, ***kwargs*)

Delete a signal

Parameters

- **str_id** – string id of the signal
- **kwargs** – alternative signal id (signal_reference)

kwargs - fields from data signal reference:

Parameters

- **timestamp** –
- **note** –

find_generic_signals(*alias_or_name=''*, *regexp=False*)

Find generic signals by a name or alias

Parameters

- **alias_or_name** – search string
- **regexp** – use regular expressions (MySQL REGEXP)

Return type tuple of signal references (like get_generic_signal_references)

The search criterion in MySQL is "SELECT ... WHERE generic_signal_name LIKE %%alias_or_name%% OR alias LIKE %%alias_or_name%%"

get_FS_signal_reference(*nodeuniqueid*, *hardwareuniqueid*, *parameteruniqueid*)

Get the currently attached generic signal reference for a FireSignal channel.

Returns None if not found.

get_attachment_table(*timestamp=None*, *computer_id=None*, *board_id=None*, *channel_id=None*, *generic_signal_id=None*, *record_number=None*)
Returns tuple of rows from a "view" corresponding to channel attachments based on specified criteria.

Parameters

- **timestamp** – reference time (default current)
- **computer_id** – DAQ computer id
- **board_id** – DAQ board id
- **channel_id** – DAQ channel id
- **generic_signal_id** – attached generic signal id
- **record_number** – timestamp of this record number will be used

get_channel_references(*computer_name=None*, *computer_id=None*, *board_id=None*, *channel_id=None*)
Get DAQ channel reference(s)

get_coefficient_V2unit(*generic_signal_id*)
Get coefficient_V2unit (needed for FireSignal operation)

get_coefficient_lev2V(*generic_signal_id*)
Get coefficient_lev2V (needed for FireSignal operation)

get_computer_id(*computer_name=None*, *description=None*)
Returns computer ID by computer name or description

get_data_file_reference(***kwargs*)
Returns a tuple of dictionaries containing complete file references containing the signal. Returns one reference for each file revision.

Typical parameters: data_file_id: unique id generic_signal_name, data_source_id (name), data_source_id, generic_signal_id record_number

If data_source_id is not specified the signal is first sought in generic signal names.

get_data_root_path(*data_root=None*)
Return the CDB data root path

Parameters **data_root** – user specified root directory, if None given by CDB_DATA_ROOT

Return type the full path with expanded variables

get_data_source_id(*data_source_name*)
Returns data source id of a given data source name

get_data_source_references(***kwargs*)
Returns a tuple with references to data sources

Typical parameters: data_source_name, data_source_id, description, record_number

get_data_source_subdir(*data_source_id*)
Returns the data storage subdirectory for a given data source id

get_generic_signal_id(*generic_signal_name*, *data_source_id=None*)
Returns generic signal id given the signal name

get_generic_signal_references(*str_id=''*, ***kwargs*)
Returns a tuple of references to generic signal, based on specified criteria

Parameters **str_id** – generic signal string id - see see *decode_generic_signal_strid()*

Keyword parameters: generic_signal_id (or signal_id): id of the generic signal generic_signal_ref or generic_signal_reference generic_signal_name + (optionally) data_source_id or data_source_name + record_number (for validity) alias_or_name: first find by alias; if no match find by name generic_signal_strid: deprecated - use str_id

classmethod **get_max_revisions**(*signals*, *deleted=False*)
From a set of signals, take only those that have maximum revision number for a combination of (record_number, generic_signal_id).

Parameters **deleted** – If true, keep deleted, otherwise filter them out.

get_record_datetime(***kwargs*)
Return record datetime (timestamp)

kwargs: :param record_number: record number

get_record_path(***kwargs*)
Returns data storage path (from the database) for a given record number

get_signal(*str_id=None*, *squeeze=True*, *deleted=False*, *dtype=None*, ***kwargs*)
Get signal information and data.

Parameters

- **str_id** – signal string id, see decode_signal_strid
- **squeeze** – remove singleton dimensions in the received data
- **deleted** – return or not deleted signals (this will not work anyway ...)
- **dtype** – force a certain numpy data type

Keyword parameters: :param units: raw / dav / default (case insensitive, conflict with str_id) :param signal_ref: signal reference :param generic_signal_ref: generic signal reference :param n_samples: number of samples :param time_limit: time limit (instead of n_samples) :param **slice_** standard Python slice :rtype: CDBSignal instance

If str_id argument is not present, keyword arguments are used in the search.

get_signal_base_tree(*signal_ref*)
Create a reference tree of dependent signals.

Parameters **signal_ref** – A valid signal reference

It reads parameters from the database and sets some signal properties as well.

get_signal_calibration(*str_id=None*, *sig_ref=None*, *units=None*, *gs_ref=None*)
Calculate signal offset and coefficient based on signal reference

Parameters

- **str_id** – signal string id
- **sig_ref** – signal reference
- **units** – signal units, None uses default

The calibration transformations is: (data + offset) * coefficient

If str_id is input, all the references are fetched from the database. Note: Applies only to FILE signals, LINEAR cannot be transformed this way

get_signal_data(*signal_ref=None*, *generic_signal_ref=None*, *signal=None*, *squeeze=True*, *raw=False*, ***kwargs*)
Get signal data.

Parameters **squeze** – Reduce N... x 1 x...M data_sets into N... x ...M (numpy squeeze function).

For FILE data signals, it returns the data without any modifications (apart from squeeze or slicing). For LINEAR data signals, it returns: - if n_samples is set: an array of linear values - else: a linear function

kwargs:

Parameters **x0** – if set, it is used instead of signal_ref["offset"]

Parameters: time_limit, n_samples x0,x1 - axis limits raw - if True, data won't be scaled

get_signal_data_tree(*signal*, *units='default'*, *n_samples=None*, *squeeze=True*, *x0=None*, *slice_=Ellipsis*, *is_this_axis=False*, *dtype=None*)
Recursively read data for signal and all its axes.

Parameters

- **signal** – the signal object to set data to.
- **x0** – additive constant

get_signal_parameters(*str_id=''*, *parse_json=True*, ***kwargs*)
Get data signal parameters from data_signal_parameters table

Parameters

- **str_id** – string id
- **kwargs** – keyword arguments for signal identification
- **parse_json** – if True, the method tries to parse fields as JSON.

get_signal_references(*str_id=''*, *deleted=False*, *limit=10000*, *dereference=False*, ***kwargs*)
Get signal references by specified criteria

Parameters

- **str_id** – signal string id (see *decode_signal_strid()*)
- **deleted** – return or not deleted references for revision=-1
- **limit** – limit the number of results
- **dereference** – return explicit gs_ref, data_file_ref and data_source_ref

keyword arguments offer an alternative to str_id: :param record_number: record number :param revision: revision :param signal_ref: signal reference :param signal_reference: the same as signal_ref

get_signal_setup(*gs_str_id*, *record_number=None*, *timestamp=None*)
Get signal setup for a generic signal

Parameters **gs_str_id** – generic signal alias, name or numeric id

get_unit(***kwargs*)
Get a unit.

kwargs:

Parameters

- **unit_id** – Direct ID of the unit.
- **(or unit_system_idor unit_system_name)** (*unit_system*) – The unit system to use.
- **construct** – whether to construct unit from basic units (default: True)

For unit system, specify any of the following keyword arguments: m, kg, s, A, K, mol, cd (as dimensionality)

You can specify either unit_id OR unit_system (unit_system_id/unit_system_name) and dimensionality.

get_unit_factor_tree(*signal*, *units=None*, *unit_system=None*)
Get tree of factors and unit names for a signal object.

Parameters

- **signal** – CDBSignal + its axes signals in a tree
- **units** – name of the unit or unit system to convert to
- **unit_system** – unit system to convert to (takes precedence)

Works recursively but only unit system is passed along the recursion. Example return value: { 'units': 'mm', 'factor' : 0.001}

get_unit_system(*unit_system_name=None*, *unit_system_id=None*, *load_parent=True*, ***ignored_args*)
Get a unit system.

Parameters

- **unit_system_name** – Name of the system.
- **unit_system_id** – The numerical ID of the system.
- **load_parent** – if true, automatically load parent unit systems.

You have to specify either the system name (takes precedence) or unit_system_id.

is_file_ready(*data_file_id*)
Set file_ready to True

last_record_number(*record_type='EXP'*)
Returns the "record number" of the last available record

Parameters **record_type** – record type: EXP, VOID, MODEL or empty string (any type)

Return type record number (integer)

last_shot_number()
The same as last_record_number for record_type = 'EXP'

Return type record number (integer)

new_data_file(*collection_name*, *file_format='HDF5'*, *file_extension=None*, *create_subdir=None*, *data_root=None*, ***kwargs*)
Creates a record in the COMPASS database for a new data file. Returns the created data file reference.

Parameters

- **collection_name** – signal collection name (used for file name)
- **create_subdir** – automatically create the file subdirectory (default if data_root is not set)
- **data_root** – use user-supplied data_root instead of the default one
- **kwargs** – keyword parameters * record_number: resolve_record_number * data_source_id, data_source_ref, or data_source (name)

First looks for a record with the input record number, data source and collection. If found, a new revision is created. Otherwise a new id with revision 1 is created.

put_signal(*gs_alias_or_id*, *record_number*, *data*, *variant=''*, *time0=None*, *data_coef=None*, *data_offset=None*, *data_coef_V2unit=None*, *note=None*, *no_debug=None*, *data_quality='UNKNOWN'*, *time_axis_data=None*, *time_axis_coef=None*, *time_axis_offset=None*, *time_axis_note=None*, *time_axis_id=None*, *axis1_data=None*, *axis1_coef=None*, *axis1_offset=None*, *axis1_note=None*, *axis2_data=None*, *axis2_coef=None*, *axis2_offset=None*, *axis2_note=None*, *axis3_data=None*, *axis3_coef=None*, *axis3_offset=None*, *axis3_note=None*, *axis4_data=None*, *axis4_coef=None*, *axis4_offset=None*, *axis4_note=None*, *axis5_data=None*, *axis5_coef=None*, *axis5_offset=None*, *axis5_note=None*, *axis6_data=None*, *axis6_coef=None*, *axis6_offset=None*, *axis6_note=None*, *gs_parameters=None*, *daq_parameters=None*)

Write CDB signal to HDF5 and database

cdb_signal = cdb_put_signal(gs_alias_or_id, record_number, data, ...) writes signal data into CDB, including HDF5 file.

Parameters

- **gs_parameters** (*JSON string or dict*) – generic signal parameters
- **daq_parameters** (*JSON string or dict*) – data acquistion channel parameters

record_exists(*record_number=None*, ***kwargs*)

Returns True if a record exists, False otherwise.

Parameters **kwargs** – named parameters interpretable by resolve_record_number (see)

Return type bool

resolve_record_number(*argv*)

Resolves record number from input parameters dictionary argv

Parameters **argv** – dictionary with parameters

Return type record number

Recognized argv keys:

•record_number (-1 returns last record number)

•record_type

set_file_ready(*data_file_id*, *chmod_ro=True*)

Set file_ready to True

Parameters

- **data_file_id** – data file id
- **chmod_ro** – make the file read-only (strongly recommended)

classmethod signal_dim(*signal_ref*)

Signal dimensionality from signal dict reference

If the signal has no axes, it is assumed to be an axis and expected to have dimension 1. Otherwise the expected dimension is equal to the number of signal axes.

store_channel_data_as_signal(*computer_id*, *board_id*, *channel_id*, *generic_signal_id*, *record_numbers*, *time_axis_id=None*, *time0=None*, *offset=None*, *coefficient=None*, *coefficient_V2unit=None*, *delete=True*, *note=None*, *enable_deleted=False*, *restore_default=True*)

Store data signal(s) from a particular DAQ channel as a given generic signal

Parameters

- **computer_id** – numeric id or computer name
- **board_id** –
- **channel_id** –
- **generic_signal_id** –
- **record_numbers** – list (or any iterable) of record numbers (or a single record number)
- **time_axis_id** – time axis (generic signal) id, latest revision will be used
- **time0** – time0, None keeps the last value
- **offset** – set offset (in raw levels), None keeps the last value
- **coefficient** – DAQ levels to Volts coefficient, None keeps the last value
- **coefficient_V2unit** – DAQ Volts to units coefficient, None keeps the last value
- **delete** – delete the previous data signal
- **enable_deleted** – do it even if there is no generic signal that is not deleted.
- **restore_default** – if another data signal belongs to the g.s., it is restored to default g.s.

store_signal (*generic_signal_id*, ***kwargs*)

Insert new signal into SQL database. Does not create/write to the data file!

Parameters **generic_signal_id** – generic signal id

kwargs - fields from data signal reference:

Parameters

- **record_number** – must be specified
- **data_file_id** –
- **data_file_key** –
- **time0** –
- **coefficient** –
- **offset** –
- **coefficient_V2unit** –
- **variant** –
- **time_axis_id** –
- **time_axis_revision** –
- **axis1_revision** –
- **axis2_revision** –
- **axis3_revision** –
- **axis4_revision** –
- **axis5_revision** –
- **axis6_revision** –
- **time_axis_variant** –
- **axis1_variant** –

- **axis2_variant** –
- **axis3_variant** –
- **axis4_variant** –
- **axis5_variant** –
- **axis6_variant** –
- **note** – text note (new revision reason etc.)
- **computer_id** – specify originating A/D
- **computer_name** – alternative to computer_id
- **board_id** –
- **channel_id** –
- **data_quality** – enum('UNKNOWN', 'POOR', 'GOOD', 'VALIDATED')
- **deleted** –
- **daq_parameters** – DAQ parameters (setup)
- **gs_parameters** – generic signal parameters (setup)
- **ignore_if_exists** – Will not write if such signal exists (useful when multiple sources want to write it at the same time)

Attachment information (computer, board & channel) is automatically filled in if not specified (using *get_attachment_table* method).

update_signal(*str_id=''*, *sig_ref=None*, ***kwargs*)
Create a new revision of a signal with some properties changed.

Use ether str_id or sig_ref as signal identifier.

Parameters

- **str_id** – signal string id
- **sig_ref** – signal reference (dict)
- **kwargs** – new values of the fields

class pyCDB.client.**CDBSignal**(*data=None*, *name=''*, *units=''*, *axes=None*, *description=''*, *ref=None*, *gs_ref=None*, *file_ref=None*, *daq_attachment=None*)
CDB signal class, contains description and data

get_axes_info(*remove_empty_dim=True*)
Return list of dicts describing each viable axis

get_log_level()
Get logging level

info()
Information string about the signal

log_level
Get logging level

plot(*fig=None*, *subplot=111*, *mplbackend=None*, *down_sample=1*, *start_time=-1*, *stop_time=-1*, *y_start=-1*, *y_stop=-1*, *xmin=None*, *xmax=None*, *color_set=None*, *plot_kwargs={}*, *contour=True*, *colormap=True*, ***kwargs*)
Plots the signal

Parameters

- **fig** – existing figure to add plot to
- **subplot** – subplot to plot to
- **mplbackend** – matplotlib backend to use (None - use default), use 'Agg' for no screen output
- **color_set** – Color set of the figure, currently supported None (default) and logbook
- **plot_kwargs** – dictionary with keyword arguments for the matplotlib plotting command
- **(only 2D)** (*colormap*) – show contour map
- **(only 2D)** – show color map

set_log_level(*value*)
Set logging level

url
WebCDB URL.

pyCDB.client.**apply_calibration**(*data*, *calibration*)
Apply calibration to RAW data.

Parameters

- **data** – numpy array (or anything that can be multiplied and added)
- **calibration** – dictionary having keys [offset, coefficient] (signal_ref can be used)

Equation: (data + offset) * coefficient

pyCDB.client.**decode_generic_signal_strid**(*generic_signal_strid*)
Decode generic signal string reference and return a dictionary

Parameters **generic_signal_strid** – is defined as 'alias or generic_signal_name' or 'generic_signal_name/data_source_name' or 'generic_signal_name/data_source_id'

Return type dictionary containing 'alias_or_name' or ('generic_signal_name' and ('data_source_id' or 'data_source_name'))

pyCDB.client.**decode_signal_strid**(*str_id*)
Decode signal string id

Parameters **str_id** – string identifier, defined as id_type:identification[units_or_slice][units_or_slice]

units_or_slice can be either units or slice specification. If two brackets are supplied, the second section in brackets has to differ from the first one. id_type := CDB | FS | DAQ

identification := (based on id_type)

- CDB -> generic_signal_strid:record_number:variant:revision
- FS -> nodeuniqueid/hardwareuniqueid/parameteruniqueid:record_number:variant:revision
- DAQ -> computer_id/board_id/channel_id:record_number:variant:revision * if computer_id is a string (contains non-numeric characters), it's understood as computer_name * alternatively, computer_id and channel_id can be prefixed by "board_" and "channel_" * in DAQ and FS, you can use "." instead of "/"

special units:

- [RAW] or [raw] - get raw signal

- [DAV] - signal in Data Acquistion Volts (applied get_coefficient_lev2V)
- [] or [DEFAULT] or [default] - default units
- [unit_name] or [unit_system_name] - conversion to unit stored in the database

certain parts can be omitted, the defaults are:

- id_type = CDB
- revision = -1
- variant = ‘’
- record_number = -1
- default units

i.e., “ne/thomson:100” == “CDB:ne/thomson:100:-1”

pyCDB.client.**format_record_path**(*record_number*, *record_type*)
Return data path for a given record number

Return type string with the subdirectory name

pyCDB.client.**get_h5_dataset**(*file_path=’‘*, *file_key=’‘*, *file_format=’HDF5’*, *slice_=Ellipsis*)
Get the HDF5 dataset.

pyCDB.client.**get_h5_dataset_shape**(*file_path=’‘*, *file_key=’‘*, *file_format=’HDF5’*)
Get the HDF5 dataset shape.

pyCDB.client.**read_file_data**(*file_path=’‘*, *file_key=’‘*, *file_format=’HDF5’*, *slice_=Ellipsis*)
Read data from a file

Parameters

- **file_path** – file path of the file
- **file_key** – data location in the file (groupA/groupB/.../dataset for HDF5)
- **file_format** – data format of the file
- **file_handle** – file handle if file is already open
- **slice** – slice specification

pyCDB.client.**validate_variant**(*variant*)
Check that the supplied variant follows the rules.

pyCDB.DAQClient

@package DAQClient COMPASS Data AcQuisition module

Contains class for DAQ management.

class pyCDB.DAQClient.**CDBDAQClient**(*host=None*, *user=None*, *passwd=None*, *db=None*, *port=3306*, *log_level=None*, *data_root=None*)
CDB data acquisition class - extends CDBClient

attach_channel(*pc_id*, *board_id*, *channel_id*, *generic_signal_id*, *offset=None*, *coefficient_lev2V=None*, *coefficient_V2unit=None*, *uid=None*, *note=’‘*, *parameters=None*, *timestamp=None*, *force_detach=False*, ***kwargs*)
Attach a channel to a generic signal (specified by id)

Parameters that are not specified (default values None) will be inherited from previous channel attachment record.

parameters is a JSON text that describes channel parameters

attached_channel_reference(*pc_id*, *board_id*, *channel_id*)
Checks whether a physical channel is attached

change_calibration(*generic_signal_id=None*, *computer_id=None*, *board_id=None*, *channel_id=None*, ***kwargs*)
Shortcut method for changing the calibration of channel / generic signal)

Specify channel / generic signal by either choice (overspecification will be rejected): 1) generic_signal_id 2) computer_id, board_id, channel_id

Arguments that are specified, will be changed, other won't.

create_DAQ_channels(*nodeuniqueid*, *hardwareuniqueid*, *parameteruniqueid*, *data_source_id=None*, *note=''*, *time_axis_id=None*, *axis1_id=None*, *axis2_id=None*, *axis3_id=None*, *axis4_id=None*, *axis5_id=None*, *axis6_id=None*, ***kwargs*)
Creates a new DAQ channel.

It enables automatic creation of new channels and boards (if requested). New computers cannot be created automatically. Use *create_computer* method in such case.

Parameters **data_source_id** – Not required only if you specify generic_signal_id

Keyword arguments:

Parameters

- **default_generic_signal_id** – If set, it is used as default generic signal instead of creating a new one.
- **computer_id** – Valid computer id that will be used if there is no channel for the computer yet.
- **board_id** – Board id for a non-existent board (if found, stored board_id is always used). If set to -1, the number will be evaluated automatically. If not set, exception is thrown. Please note that two hardwareuniqueid's can be used as one board (although not recommended).
- **channel_id** – If set, it will be used. Otherwise (or value of -1), a new one will be created.
- **coefficient_lev2V** – coefficient_lev2V for the default channel attachment.
- **offset** – offset for the default channel attachment.

create_computer(*computer_name*, *description=None*, *location=None*)
Create new computer in the database.

detach_channel(*pc_id*, *board_id*, *channel_id*, *timestamp=None*)
Detach channel in the channel setup

find_daq_signals(*record_number=-1*)
Find all data signals that originate in any DAQ channel.

find_duplicate_signals(**record_numbers*)
Find duplicate signals from DAQ channels in a set of records.

Example: cdb.find_duplicate_signals(6228, 6229)

Parameters ***record_numbers** – two or more record numbers

get_FS_attachement(*nodeuniqueid*, *hardwareuniqueid*, *parameteruniqueid*)
Return the currently attached generic signal reference and the corresponding line from the channel_attachments view for a FireSignal channel

get_board_id(*description=''*, *description2=''*)
Returns board ID by computer and board name

get_record_number_from_FS_event(*event_number*, *event_id='0x0000'*)
Return record number for a specific FireSignal event.

Event_number Number of the FS event.

Event_id string with the ID of the event. Default is "0x0000", shot event.

is_channel_attached(*pc_id*, *board_id*, *channel_id*)
Checks whether a physical channel is attached

is_generic_signal_attached(*generic_signal_id*)
Checks whether a generic signal is attached to a physical channel

nodiff_signal_data(*signal1*, *signal2*)
Compare data of two signals. Returns True for identical signals.

Parameters

- **signal1** – first signal str_id or reference
- **signal2** – second signal str_id or reference

signal_setup(*generic_signal_id*, *parameters=''*, *timestamp=None*, *uid=None*)
Write generic signal parameters (setup) into signal_setup

Parameters

- **generic_signal_id** – numerical generic_signal_id
- **parameters** – signal parameters (typically JSON/YAML formatted text)
- **timestamp** – time, default is now
- **uid** – user id

switch_channels(*pc_id_1*, *board_id_1*, *channel_id_1*, *pc_id_2*, *board_id_2*, *channel_id_2*, *uid=None*, *note=''*, *timestamp=None*)
Switch 2 channels attached generic signals

Keeps the offset and coefficient from the original channel, i.e., the offset and the coefficient are switched as well.

pyCDB.pyCDBBase

Created on Jun 13, 2012

@author: jurban

exception pyCDB.pyCDBBase.**CDBException**
A generic CDB exception class

class pyCDB.pyCDBBase.**OrderedDict**(**args*, ***kwargs*)
Customized ordered dictionary

diff(*other*, *mode='norm'*)
Find differences to another OrderedDict, ignoring the keys order

Parameters **other** – OrderedDict object to compare to

classmethod load_h5 (*filename*)
Deserialize from an HDF5 file (created by save_h5)

Parameters **filename** – input file name

save_h5 (*filename*, *mode='w'*)
Serialize to an HDF5 file

Parameters

- **filename** – output file name
- **mode** – file open mode, typically 'w' or 'a'

pyCDB.pyCDBBase.**cdb_base_check** (*func*)
Basic checking (database connection in particular) before calling CDB functions.

To be used as decorator for client methods. Raises an exception if problem occurs.

pyCDB.pyCDBBase.**isintlike** (*obj*)
Is object int-like?

In Python 2, this means int or long. In Python 3, this means just int.

pyCDB.pyCDBBase.**isstringlike** (*obj*)
Is object string-like?

In Python 2, this means string or unicode. In Python 3, this means just string.

class pyCDB.pyCDBBase.**pyCDBBase** (*host=None*, *user=None*, *passwd=None*, *db=None*, *port=3306*, *log_level=None*, *data_root=None*)
PyCDB base class for other pyCDB classes

check_connection (*reconnect=True*)
Checks if connection is open, optionally (re)connects.

close ()
Close all connections (SQL)

db
Connection to the database.

•each thread has its local connection

•connection is automatically opened

delete (*table_name*, *fields*)
Delete a row from the table.

Parameters

- **table_name** – name of table to insert to
- **fields** – dictionary of fields identifying the row

Deletes at most one row: First, it checks for its existence in the table. If more than one row is found, raises exception. Returns True is something was deleted, False if not.

error (*exception*)
Log an error and throw the exception

Parameters **exception** – can be an exception or string with the message.

file_log_level
Get logging level

classmethod **filter_sql_str**(*sql_str*, *extra_chars='_'*, *quiet=False*)
Filter a string intended to be stored to SQL

Parameters **sql_str** – input string

Keep only white listed characters: letters, digits, _.

classmethod **format_file_name**(*collection_name*, *revision*, *file_format='HDF5'*, *file_extension=None*)
Returns generic file name

classmethod **get_conf_value**(*var_name*, *section='database'*, *required=False*)
Get configuration value for a given variable name

First looks for environment variables, then to ./.CDBrc, then to ~/.CDBrc

Parameters

- **var_name** – one of CDB_HOST, CDB_USER, CDB_PASSWD, CDB_DB
- **required** – if True, exception is raised telling the user that he/she has to set the option.

Return type variable value

get_file_log_level()
Get logging level

get_log_level()
Get logging level

get_table_struct(*table_name*)
Get a table structure (as a dict)

insert(*table_name*, *fields*, *return_inserted_id=False*, *check_structure=True*)
Insert a new row to a table.

Parameters

- **table_name** – name of table to insert to
- **fields** – dictionary of (field name, value) to insert
- **return_inserted_id** – if true, returns the id of the inserted row
- **check_structure** – if true, checks, whether fields agree with table layout,

Values are processed to be DB-friendly by mysql_values (see).

classmethod **is_json**(*text*)
Checks whether text has a valid JSON syntax

Parameters **text** – text (string) to check

log_level
Get logging level

classmethod **makedirs**(*path*[, *mode=2047*])
Super-mkdir; create a leaf directory and all intermediate ones. Works like mkdir, except that any intermediate path segment (not just the rightmost) will be created if it does not exist. This is recursive.

classmethod **mysql_str**(*val*, *float_format='%.17g'*, *datetime_format='%Y-%m-%d %H:%M:%S'*, *quote=True*)
Convert Python value into MySQL string and quote if necessary

Use as the first character of MySQL builtin functions to avoid quoting

Lists and tuples are converted into parentheses-wrapped lists (using recursion) to work with IN operator.

classmethod **mysql_values**(*val_list*, *float_format='%.17g'*, *datetime_format='%Y-%m-%d %H:%M:%S'*, *quote=True*)
Construct whole VALUES MySQL INSERT construct for a list of Python values

query(*sql*, *error_if_empty=False*, *warn_if_empty=False*, *error_if_multiple=False*)
Execute SQL query and returns a list of rows as dictionaries.

Parameters

- **sql** – valid SQL query
- **error_if_empty** – raises exception if the result set is empty
- **warn_if_empty** – issues warning if the result set is empty but successfully returns it
- **error_if_multiple** – the result has to include at maximum 1 row

classmethod **row_as_dict**(*row*, *description*)
Convert SQL row tuple to python dict

set_file_log_level(*value*)
Set logging level

set_log_level(*value*)
Set logging level

classmethod **sql_to_python_type**(*sql_type_str*)
Transforms MySQL type definition to Python type and its length

update(*table_name*, *id_field*, *id_value*, *fields*)
Update a row in a table.

Parameters

- **table_name** – name of table to insert to
- **id_field** – name of the unique key field
- **id_value** – value of the unique key
- **fields** – dictionary of (field name, value) to insert

pyCDB.logbook

Created on Jun 13, 2012

@author: jurban

class pyCDB.logbook.**logbook**(*host=None*, *user=None*, *passwd=None*, *db=None*, *port=3306*, *log_level=None*)
logbook client

campaign_name(*cid*)
Translates cid to the campaign name

get_shot_comments(*shot_number*)
Get shot comments

get_shot_info(*shot_number*)
Get shot info

get_shot_linear_profile_value(*shot_number*, *prof_name*)
Get a profile as an array

get_shot_linear_profiles(*shot_number*)
Get all shot parameters (in a dictionary)

get_shot_param_value(*shot_number*, *param_name*)
Get the parameter's value

get_shot_params(*shot_number*)
Get all shot parameters (in a dictionary)

get_tags(*shot_number*)
Get shot tags

has_tag(*shot_number*, *tag*)
Check for a tag

shot_param_name(*param_id*)
Translate param_id to param name

uid_name(*uid*)
Translates uid to (LDAP) name

pyCDB.CodeGeneration

pyCDB.CodeGeneration.**generate_pxi**(*filename='cyCDB_api.pxi'*, *conn=None*)
Generates .pxi file with C-api definitions

pyCDB.CodeGeneration.**get_table_struct**(*table_name*, *conn=None*)
Reads table structure and returns a dict containing row_name : type

CHAPTER 7

Matlab CDB reference

class cdb_client

CDB client class.

get_signal (*str_id*)

Parameters **str_id** – CDB string id (see *Signal id's*)

Get CDB signal by string id.

Returns a structure withthe CDB signal, including the data and the description (references).

Matlab errors

- `CDB:JyCDBError`
- `CDB:SignalError`
- `CDB:InputError`
- `CDB:StoreError`

CHAPTER 8

Installation of FireSignal with pyCDB

Installation steps

INSTALLATION OF POSTGRES DB:

Exactly follow the instuction at IPFN IST web page: http://metis.ipfn.ist.utl.pt/index.php?title=CODAC/FireSignal/Databases/PostgreSQL_%2b_FileSystem

- Useful commands for wokr witl POSTGRES:
- sudo -u postgres psql
- \d
- \l
- \c genericdb
- SELECT * FROM events;
- \q

HOW TO CREATE NEW CDB DATABASE?

mysql -u root -p

GRANT ALL PRIVILEGES ON . TO 'CDB'@'localhost' WITH GRANT OPTION; CREATE USER 'CDB'@'localhost' IDENTIFIED BY 'cmpsSQLdata' ;

mysql -h localhost -u CDB -p < localhost.sql with password in ".CDBrc" file

TO CHECK IT: connect CDB show tables;

copy file .CDBrc to your home directory

mkdir /home/rrr/CDB_data

FINAL CHECK of SUCCESFULL OPERATION OF pyCDB: python pyCDB.py → should some nice draw graph

INSTALLATION OF FIRESIGNAL:

Source: svn co svn://baco.ipfn.ist.utl.pt/scad/trunk

/trunk/java/dist$ sudo ./firesignal-1.1-linux-installer.bin

IN INSTALLER: installed to: /home/rrr/FS_PT locate javac /home/rrr/FS/jdk1.6.0_27/jre/ ..better to use sun's implementation 10.136.245.226 1050 PostgreSQL + filesystem /home/rrr/CDB_myDATA Database(Posgres) genericdbadmin xxx genericdb localhost 5432

TO RUN CENTRAL SERVER:

sudo ./fsignal start

etc. (e.g sudo ./fsignal_test_node start)

INSTALLATION OF FIRESIGNAL GUI CLIENT:

sudo ./firesignaljws-1.1-linux-installer.bin

/usr/share/mini-httpd/html/XXX

http://localhost/XXX – improve setting of http server 127.0.0.1 or localhost 1050 ok to the rest

TO RUN IT:

/home/rrr/FS/jdk1.6.0_27/jre/bin/javaws FireSignalClient.jnlp

sudo ./fsignal_test_node start Burn button → OK → you should see the data in the "Data" panel

HDF5 plugin:

jep-2.4 compilovat s original java

change scripts : /home/rrr/firesignal-1.1/dbcontroller/DBScript /home/rrr/firesignal-1.1/server/FSServerScript

> 4x change IP 2x export LD_PRELOAD=/usr/lib/libpython2.6.so.1.0 (P by which was compiled jep, ldd..) 2x FS_PT path 1x #:../libs/libjhdf5.so Xx parth firesignal-1.1 ...

copy libraries: sudo cp /home/rrr/firesignal-1.1/libs/jhdf5.jar /home/rrr/FS_PT/libs/ sudo cp /home/rrr/firesignal-1.1/libs/libjhdf5.so /home/rrr/FS_PT/libs/ sudo cp /home/rrr/firesignal-1.1/libs/hdf5test.jar /home/rrr/FS_PT/libs/ → fsPqsqlHDF5.jar in mail 19.10.2011 sudo cp /home/rrr/firesignal-1.1/libs/jep.jar /home/rrr/FS_PT/libs/

rrr@rrr-laptop:~/FS_PT/init$ sudo ./fsignal start

TO GET INFOS ABOUT START OF FS: either see the logs

or

sudo gedit ../server/StartFSServer & comment lines: /home/rrr/FS_PT/server/FSServerScript \ #1>>/dev/null \ #2>>/dev/null

> sudo gedit ../dbcontroller/StartDBController &

Nota Bene: How many instances of FSs are running?

If you experience some very strange/unexpected behaviout of FS mybe the error can be cause by that more than one instance of firesignal are sunnig.

ps aux | grep java

CHAPTER 9

Compilation of FireSignal:

in NetBeans IDE -> new project from resources -> choose all java structure

You will neeed many libraries. Usually they are common and publicly accesible.

SDAS Core Libraries (SDAS.jar) and SDAS Client (SDASClient.jar) you can find here: http://metis.ipfn.ist.utl.pt/CODAC/SDAS/Download

Example: LAST RECORD NUMBER IN GUI CLIENT

export LD_PRELOAD=/usr/lib/libpython2.6.so.1.0 ..needs (at least) pymsql export PYTHONPATH=$PYTHONPATH:/home/rrr/workspace/pyCDB/src/

java -Djava.library.path=/home/rrr/FS_PT/libs/:/usr/local/lib/:/usr/lib -jar ./dist/FS.jar ..location of c-libs

NOTE: Done just in FS.jar X not as FireSignalClient.jpnl

CHAPTER 10

JyCDB

Javadoc

cz.cas.ipp.compass.jycdb

CDBClient

public class **`CDBClient`** extends *PythonAdapter*

Jython-based adapter of python CDBClient. Threading: Each thread should acquire an instance of this class using getInstance(). Multiple calls in one thread result in returning the same object. There is no need of closing the connections, it happens automatically in garbage collector (this is forced when we run out of connections). However, you should not pass reference to one instance among different threads (if you cannot assure the threads will not use it concurrently). The methods map to Client/DAQClient python methods as closely as possible. Several methods have overloaded versions: 1) a totally generic one that accepts ParameterList (a "dictionary" of values) 2,...) more specific versions with frequently used sets of parameters. See pyCDB documentation.

Author pipek

Methods

FS2CDB_ref

public void **`FS2CDB_ref`** (String *nodeUniqueId*, String *hardwareUniqueId*, String *parameterUniqueId*)

Return the CDB DAQ_channel reference of a FireSignal channel

Parameters

- **`nodeUniqueId`** –
- **`hardwareUniqueId`** –
- **`parameterUniqueId`** –

checkConnection

public boolean **checkConnection**()

createComputer

public Long **createComputer**(String *computerName*, String *description*, String *location*)

createComputer

public Long **createComputer**(String *computerName*, String *description*)

createComputer

public Long **createComputer**(String *computerName*)

createDAQChannel

public void **createDAQChannel**(String *nodeId*, String *hardwareId*, String *parameterId*, long *dataSourceId*, Long[] *axisIds*)

createDAQChannel

public void **createDAQChannel**(*ParameterList* *parameters*)

createGenericSignal

public void **createGenericSignal**(String *name*, String *alias*, long *dataSourceId*, Long[] *axisIds*, String *units*, String *signalType*)

createGenericSignal

public void **createGenericSignal**(*ParameterList* *parameters*)

createGenericSignalWhichReturnsLong

public Long **createGenericSignalWhichReturnsLong**(String *name*, String *alias*, long *dataSourceId*, Long[] *axisIds*, String *units*, String *signalType*)

createGenericSignalWhichReturnsLong

public Long **createGenericSignalWhichReturnsLong**(*ParameterList* *parameters*)

createRecord

public long **createRecord**(Map<String, PyObject> *parameters*)

createRecord

public long **createRecord**(String *recordType*)

createRecord

public long **createRecord**(long *fsEventNumber*, String *fsEventID*)
Creates new FireSignal record with forced record number equivalent to fsEventNumber.

Parameters

- **fsEventID** – event identification "0x0000" -
- **fsEventNumber** – firesignal shot number

createRecord

public long **createRecord**(long *fsEventNumber*, String *fsEventID*, String *recordType*)

deleteSignal

public void **deleteSignal**(*ParameterList parameters*)

finalize

public void **finalize**()
Even if finalize is not always called, try to close the connection.

findGenericSignals

public List<*GenericSignal*> **findGenericSignals**(String *aliasOrName*, boolean *useRegexp*)

getAttachment

public *ChannelAttachment* **getAttachment**(long *genericSignalId*)

getAttachmentTable

public List<*ChannelAttachment*> **getAttachmentTable**(*ParameterList parameters*)

getAttachmentTable

public *ChannelAttachment* **getAttachmentTable** (long *computerId*, long *boardId*, long *channelId*)

getComputerId

public Long **getComputerId** (String *computerName*, String *description*)

getComputerId

public Long **getComputerId** (String *computerName*)

getDataFile

public *DataFile* **getDataFile** (*ParameterList parameters*)

getDataFile

public *DataFile* **getDataFile** (long *dataFileId*)

getDataSourceId

public Long **getDataSourceId** (String *dataSourceName*)

getFSSignal

public *GenericSignal* **getFSSignal** (String *nodeId*, String *hardwareId*, String *parameterId*)
Get current generic signal attached to node, board & parameter.

Returns null if not found. Use Firesignal names.

getGenericSignal

public *GenericSignal* **getGenericSignal** (long *genericSignalId*)

getGenericSignal

public *GenericSignal* **getGenericSignal** (String *strid*)

getGenericSignals

public List<*GenericSignal*> **getGenericSignals** (*ParameterList parameters*)

getGenericSignals

public List<*GenericSignal*> **getGenericSignals** (String *strid*)

getInstance

public static synchronized *CDBClient* **getInstance** ()

Get CDB client specific for current thread. It can serve a few hundred connections before problem arises. However, at that moment, there is a slight pause during which we try to close all unused connections (by hinting garbage connection) and obtain the connection for the second time. Only after this it eventually fails. Note: you should not pass CDBClient instances among threads or undefined behaviour results. But it is possible if you know what you are doing (e.g. in FS database controller).

Returns null if not connected.

getInstance

public static synchronized *CDBClient* **getInstance** (*ParameterList parameters*)

Get CDB Client with specific construction parameters.

Parameters

- **parameters** – (host, user, passwd, db, port, log_level, data_root). If any of the parameters is not specified, default value (from env. variables, etc.) is taken. Warning: This version of constructor is not recommended for general use. It doesn't use any clever method of preserving connections, memory, thread safety etc.

getRecordNumberFromFSEvent

public long **getRecordNumberFromFSEvent** (long *eventNumber*, String *eventId*)

getSignal

public *Signal* **getSignal** (String *strid*, String *variant*)

getSignal

public *Signal* **getSignal** (String *strid*)

getSignal

public *Signal* **getSignal** (long *recordNumber*, long *genericSignalId*)

getSignal

public *Signal* **getSignal** (*ParameterList parameters*)

getSignalCalibration

public *SignalCalibration* **getSignalCalibration** (String *strId*)

getSignalCalibration

public *SignalCalibration* **getSignalCalibration** (*ParameterList parameters*)

getSignalParameters

public *SignalParameters* **getSignalParameters** (*ParameterList parameters*)

getSignalReference

public SignalReference **getSignalReference** (long *recordNumber*, long *genericSignalId*, long *revision*)

getSignalReference

public SignalReference **getSignalReference** (long *recordNumber*, long *genericSignalId*)

getSignalReference

public SignalReference **getSignalReference** (String *strid*)

getSignalReferences

public List<SignalReference> **getSignalReferences** (*ParameterList parameters*)

insert

public Long **insert** (String *tableName*, *ParameterList fields*, Boolean *returnInsertedId*, Boolean *checkStructure*)

insert

public Long **insert** (String *tableName*, *ParameterList fields*, Boolean *returnInsertedId*)

insert

public Long **insert** (String *tableName*, *ParameterList fields*)

lastRecordNumber

public long **lastRecordNumber** (String *recordType*)

lastShotNumber

public long **lastShotNumber**()
Last shot (or experimental record) number.

newDataFile

public *DataFile* **newDataFile**(*ParameterList parameters*)

newDataFile

public *DataFile* **newDataFile**(String *collectionName*, long *recordNumber*, long *dataSourceId*)

recordExists

public boolean **recordExists**(long *recordNumber*)
Check whether a record exists.

setFileReady

public void **setFileReady**(long *fileId*)

storeLinearSignal

public SignalReference **storeLinearSignal**(long *genericSignalId*, long *recordNumber*, double *time0*, double *coefficient*, double *offset*)
Store signal of type LINEAR.

storeSignal

public SignalReference **storeSignal**(*ParameterList parameters*)

storeSignal

public SignalReference **storeSignal**(long *genericSignalId*, long *recordNumber*, String *dataFileKey*, long *dataFileId*, double *time0*, double *coefficient*, double *offset*, double *coefficient_V2unit*)
Store signal of type FILE. Underlying Python method ensures that computer, board and channel ids are correctly set.

storeSignal

public SignalReference **storeSignal**(long *genericSignalId*, long *recordNumber*, String *dataFileKey*, long *dataFileId*, double *time0*)
Store signal of type FILE. Underlying Python method ensures that computer, board and channel ids are correctly set.

updateSignal

public void **updateSignal** (*ParameterList parameters*)
Update signal.

Parameters

- **parameters** – As few parameters as needed.

CDBConnectionException

public class **CDBConnectionException** extends *CDBException*
The connection to database does not work. This means that the pyCDB was correctly loaded but there is a problem inside it or between it and MySQL.

Author pipek

Constructors

CDBConnectionException

public **CDBConnectionException** (String *message*)

CDBException

public class **CDBException** extends Exception
Some problem with CDB. Common parent for more specific exception classes.

Author pipek

Constructors

CDBException

public **CDBException** (String *message*)

ChannelAttachment

public class **ChannelAttachment** extends *DictionaryAdapter*

Methods

create

public static *ChannelAttachment* **create** (PyObject *object*)

getAttachedGenericSignalId

public long **getAttachedGenericSignalId** ()

getCoefficientLev2V

public double **getCoefficientLev2V**()

getCoefficientV2Unit

public double **getCoefficientV2Unit**()

getParameters

public Object **getParameters**()

getParametersString

public String **getParametersString**()

DataFile

public class **DataFile** extends *DictionaryAdapter*
Wrapper around rows of *data_files* table.

Author honza

Methods

create

public static *DataFile* **create**(PyObject *object*)

getCollectionName

public String **getCollectionName**()

getFileName

public String **getFileName**()

getFullPath

public String **getFullPath**()
Full path to the file containing data.

getId

public long **getId**()

FSNodeWriter

public class **FSNodeWriter**

Methods for FireSignal nodes. It is mostly a facade over CDBClient methods to simplify development of nodes that communicate directly with CDB. How to write a signal? 1) For each record, create an instance of FSNodeWriter (you have to know generic signal id as well). 2) Create a file using createDataFile(). (see variants of this method) 3) Write data (as many times as you want). a] Using writeData(). Please, notice that you have to be aware of the data offset in HDF5 (or write them in one step). b] If you have an existing HDF5 file, use copyHdf5(). 4) Tell DB that the file is ready using setFileReady(). (if you changed it) 5) Write all axes using writeAxis(index). index=0 for time axis, index=1,2,3,... for other axes. (can be called multiple times for the same axis). 6) Write signal using writeSignal().

Author pipek

Fields

gson

Gson **gson**

Constructors

FSNodeWriter

public **FSNodeWriter**(long *genericSignalId*, long *recordNumber*, String *collectionName*, String *fileKey*)

Parameters

- **collectionName** – Name of the file without prefixes.
- **fileKey** – Name of the dataset in HDF5 file.

Throws

- **cz.cas.ipp.compass.jycdb.CDBException** –

FSNodeWriter

public **FSNodeWriter**(long *computerId*, long *boardId*, long *channelId*, long *recordNumber*, String *collectionName*, String *fileKey*)

Methods

addDaqParameter

public void **addDaqParameter**(String *key*, Object *value*)

addGenericSignalParameter

public void **addGenericSignalParameter**(String *key*, Object *value*)

copyHdf5

public void **copyHdf5**(String *path*)
Copy an existing HDF5 file to the correct destination.

Parameters

- **path** – Local path of the HDF5 file.

Throws

- **IOException** – If the destination exist, copying is prevented and exception thrown.

createDataFile

public void **createDataFile**()
Create new data file (row in *data_files*).

Throws

- ***CDBException*** – In this default version, an already existing file with requested properties is taken as error.

createDataFile

public boolean **createDataFile**(boolean *okIfExists*, boolean *createIfExists*)
Create new data file (row in *data_files*).

Parameters

- **okIfExists** – If false, an exception is thrown if a file with the same collection name, data source and record number exists
- **createIfExists** – If true and a file already exists, create a new one (otherwise the existing one is returned).

Throws

- ***CDBException*** –

Returns true if a file was created, false if nothing happens.

getDataFile

public *DataFile* **getDataFile**()

getFileKey

public String **getFileKey**()

getGenericSignal

public *GenericSignal* **getGenericSignal**()

getRecordNumber

public long **getRecordNumber**()

setFileReady

public void **setFileReady**()
Tell CDB that you have finished writing data. After this, you cannot write more data.

setRequireChannelAttachment

public void **setRequireChannelAttachment**(boolean *require*)
Set whether writing should succeed (assuming value 1.0 for coefficients) when channel attachment not found. It has to be specified explicitely because it is quite probable that non-existence marks a bug.

writeAxis

public void **writeAxis**(int *axis*, double *coefficient*, double *offset*)
Write one of the axes.

Parameters

- **axis** – Index of the axis (0=time, 1,2,3,...=other axes)

writeData

public void **writeData**(*Data* *data*, long[] *dataOffset*)
Write data (with possible offset to append).

Parameters

- **data** – Data to be written.
- **dataOffset** – Zero-based index of the first data element. Size is calculated automatically. If data Offset is null, starts from beginning (fails with existing file and dataset).

Throws

- ***cz.cas.ipp.compass.jycdb.util.Hdf5WriteException*** –

writeNotAttachedSignal

public void **writeNotAttachedSignal**(double *time0*, double *coefficient*)
Write signal info to CDB for signals that have no DAQ attachment. Usage is similar to writeSignal.

writeSignal

public void **writeSignal**(double *time0*)
Write signal info to CDB. Call this after you have finished with writing data and axes. Can be used only for generic signal with valid DAQ attachment.

FSWriter

public class **FSWriter** extends *PythonAdapter*

Java wrapper for quick storing of signal data to CDB in database controller. See: fs_writer.py Warning: This class is meant to be used only in FireSignal database controller! Not elsewhere, even the nodes. Unexpected behaviour may result.

Author pipek

Constructors

FSWriter

public **FSWriter**(*CDBClient client*, String *nodeId*, String *hardwareId*, String *parameterId*, long *recordNumber*, double *time0*, double *sampleLength*)

Methods

getFileKey

public String **getFileKey**()

Get the name of the dataset to write to.

getFilePath

public String **getFilePath**()

Get the full path where to write the file.

readSignalInfo

public void **readSignalInfo**()

1st step - read all information we need for file storage.

signalExists

public boolean **signalExists**()

Is the signal already stored in the database?

storeSignalAndFile

public void **storeSignalAndFile**()

2nd step - write all that is to be written to database. This step should be undertaken only if signal does not exist.

GenericSignal

public class **GenericSignal** extends *DictionaryAdapter*
Wrapper around rows of *generic_signals* table.

Author honza

Fields

FILE

public static final String **FILE**

LINEAR

public static final String **LINEAR**

MAX_AXES

public static int **MAX_AXES**

VALID_TYPES

public static final String[] **VALID_TYPES**

Methods

create

public static *GenericSignal* **create**(PyObject *object*)

getAlias

public String **getAlias**()

getAxis

public *GenericSignal* **getAxis**(int *index*)

getAxisId

public long **getAxisId**(int *index*)
Get id of one of the axes (time or spatial).

Parameters

- **index** – Number of the axis (starting with 1). If 0, time axis is returned.

Throws

- *CDBException* –

Returns 0 if there is no axis.

getDataSourceId

public long **getDataSourceId**()

getDescription

String **getDescription**()

getId

public long **getId**()

getLastRecordNumber

public long **getLastRecordNumber**()

getName

public String **getName**()

getSignalType

public String **getSignalType**()

getTimeAxis

public *GenericSignal* **getTimeAxis**()

getTimeAxisId

public long **getTimeAxisId**()

getUnits

public String **getUnits**()

isFile

public boolean **isFile**()

isLinear

public boolean **isLinear**()

isValidType

public static boolean **isValidType**(String *signalType*)
Check if a signal type is valid. Only "LINEAR" and "FILE" are accepted now.

Signal

public class **Signal** extends *PythonAdapter*
Class mimicking the CDBSignal Python class. In a few aspects, it is more object-oriented and lazy-evaluated to simplify manipulation in MATLAB/IDL.

Author pipek

Constructors

Signal

public **Signal**(SignalReference *signalRef*, String *units*)

Signal

protected **Signal**(*Signal parent*, int *nthChild*, *PythonAdapter signalTree*, *DictionaryAdapter unitFactorTree*)
Constructor for dependent objects from signal tree with unit factor.

Parameters

- **parent** – Signal that uses this as an axis.
- **nthChild** – The order of this signal among axes in the parent signal.
- **signalTree** – Signal subtree.
- **unitFactorTree** – Unit factor subtree.

Signal

protected **Signal**(*Signal parent*, int *nthChild*, *PythonAdapter signalTree*, String *units*)
Constructor for dependent objects from signal tree without unit factor.

Parameters

- **parent** – Signal that uses this as an axis.
- **nthChild** – The order of this signal among axes in the parent signal.
- **signalTree** – Signal subtree.
- **units** – Most times, this will be "default", but any other value can used.

Methods

getAxes

public SortedMap<String, *Signal*> **getAxes**()
A map of all existing axes.

Throws

- ***CDBException*** – Non-existent axes are omitted.

Returns A dictionary of [axis_name, Signal object] values.

getAxis

public *Signal* **getAxis**(int *i*)
An axis as Signal instance.

Parameters

- **i** – 0-time axis, N-axisN

Throws

- ***CDBException*** –

Returns null if not found

getData

public synchronized *Data* **getData**()
Numerical data of the signal. Note: this method is not used in bindings using JyCDB (IDL/MATLAB) due to efficiency. Try using native implementations of file reading.

getDataFile

public *DataFile* **getDataFile**()

getDataFileId

public long **getDataFileId**()

getDescription

public String **getDescription**()
Generic signal description.

getGenericSignal

public *GenericSignal* **getGenericSignal**()

getGenericSignalId

public long **getGenericSignalId**()

getGenericSignalReference

public *GenericSignal* **getGenericSignalReference**()

getName

public String **getName**()
Generic signal name.

getParameters

public *SignalParameters* **getParameters**()

getPythonObject

public PyObject **getPythonObject**()
Return the Python object of CDBSignal class.

Returns null if CDBSignal not found. Note: This method catches exceptions thus hiding problems.

getRecordNumber

public long **getRecordNumber**()

getSignalCalibration

public *SignalCalibration* **getSignalCalibration**(boolean *includeUnitFactor*)
Signal calibration for default/DAV/RAW.

Throws

- ***CDBException*** –

getSignalReference

public SignalReference **getSignalReference**()

getTimeAxis

public *Signal* **getTimeAxis**()
Time axis as Signal instance. Just shortcut to getAxis method.

getUnit

public String **getUnit**()
Unit name.

Throws

- ***CDBException*** – RAW/DAQ/default GS units or units from factor tree (if requested).

getUnitFactor

public double **getUnitFactor**()
Unit factor, by which the default signal has to be multiplied.

Throws

- ***CDBException*** – If DAV/RAW/default are selected, this factor is 1.0. If another unit/unit system is selected, this is a conversion from the default unit to the selected one. Internally, this reflects unit_factor_tree returned from Python.

hasUnitFactor

public boolean **hasUnitFactor**()
Whether the signal has associated unit factor tree. This is true only if unit or unit system were requested.

isDAV

public boolean **isDAV**()

isFile

public boolean **isFile**()

isLinear

public boolean **isLinear**()

isRaw

public boolean **isRaw**()

SignalCalibration

public class **SignalCalibration** extends *DictionaryAdapter*
Signal calibration. Meaning: value = (data + offset) * coefficient

Author pipek

Fields

DAQ_VOLTS

public static final String **DAQ_VOLTS**

RAW

public static final String **RAW**

Methods

apply

public double[] **apply**(double[] *array*)
If you have an array of data, apply calibration to it. Returns new array.

create

public static *SignalCalibration* **create**(PyObject *object*)

getCoefficient

public double **getCoefficient**()

getOffset

public double **getOffset**()

getUnits

public String **getUnits**()

SignalParameters

public class **SignalParameters** extends *DictionaryAdapter*
Signal parameters (both GS & DAQ-based). These are strings interpreted as JSON. Strings are accessible using getDAQParametersString() & getGenericSignalParametersString(), Parsed JSON objects (as trees) are accessible using getDAQParameters() & getGenericSignalParameters().

Author pipek

Methods

create

public static *SignalParameters* **create**(PyObject *object*)

getDAQParameters

public Object **getDAQParameters**()

getDAQParametersString

public String **getDAQParametersString**()

getGenericSignalParameters

public Object **getGenericSignalParameters**()

getGenericSignalParametersString

public String **getGenericSignalParametersString**()

WrongSignalTypeException

public class **WrongSignalTypeException** extends *CDBException*

Signal type is not valid for its intended use. It can be either LINEAR or FILE with different operations available.

Author pipek

Constructors

WrongSignalTypeException

public **WrongSignalTypeException**(String *message*)

cz.cas.ipp.compass.jycdb.plotting

DataPlotter

public class **DataPlotter** extends JFrame

Constructors

DataPlotter

public **DataPlotter**()

Methods

appendChart

public static JFreeChart **appendChart** (JFreeChart *oldChart*, *Signal signal*)

createChart

public static JFreeChart **createChart** (*Signal signal*)

main

public static void **main** (String[] *args*)

cz.cas.ipp.compass.jycdb.test

JyCDBTest

public class **JyCDBTest**

Author pipek

Methods

fsWriterTest

public static void **fsWriterTest** ()

instanceTest

public static void **instanceTest** ()

main

public static void **main** (String[] *args*)

cz.cas.ipp.compass.jycdb.util

ArrayUtils

public class **ArrayUtils**

Utils for quick manipulation with (numeric) arrays. Warning: most methods work only on rectangular arrays.

Author pipek

Methods

add

public static double[] **add**(double[] *array*, double *offset*)
Add a constant to all elements of an array.

Returns a new array

asDoubleArray

public static double[] **asDoubleArray**(Object *array*)
Cast all values of an array to double (runtime version).

Parameters

- **array** – Array of allowed type.

Throws

- ***UnknownDataTypeException*** –

asDoubleArray

public static double[] **asDoubleArray**(long[] *array*)
Cast all values of an array to double.

asDoubleArray

public static double[] **asDoubleArray**(int[] *array*)
Cast all values of an array to double.

asDoubleArray

public static double[] **asDoubleArray**(short[] *array*)
Cast all values of an array to double.

asDoubleArray

public static double[] **asDoubleArray**(float[] *array*)
Cast all values of an array to double.

asIntArray

public static int[] **asIntArray**(long[] *array*)
Cast all values of an array to int.

asLongArray

public static long[] **asLongArray** (int[] *array*)
Cast all values of an array to long.

dimensions

public static long[] **dimensions** (Object *array*)
Get the dimensions along all axes of an array. Works only on rectangular arrays.

get

public static Object **get** (Object *array*, int[] *index*)
Get an element from a multidimensional array. Works on all arrays.

linearTransform

public static double[] **linearTransform** (double[] *array*, double *multiplyBy*, double *add*)
Apply a linear transformation y = ax + b on an array.

Parameters

- **multiplyBy** – Multiplication constant
- **add** – Addition constant

Returns a new array

linearTransform

public static Object **linearTransform** (Object *array*, double *multiplyBy*, double *add*)
A general linear transformation of multidimensional array.

Parameters

- **array** –
  - N-dimensional rectangular array of double/long/int/short/float
- **multiplyBy** –
  - multiplicative factor
- **add** –
  - additive factor

Returns

- N-dimensional array of the same shape as input

max

public static double **max** (double[] *array*)
Maximum value found in the array.

min

public static double **min** (double[] *array*)
Minimum value found in the array.

multiply

public static double[] **multiply** (double[] *array*, double *coefficient*)
Multiply all elements of an array by a constant.

Returns a new array

product

public static long **product** (int[] *array*)

product

public static long **product** (long[] *array*)

range

public static int[] **range** (int *xmin*, int *xmax*)

rank

public static int **rank** (Object *array*)
Get the number of dimensions of an array. Works only on rectangular arrays.

reshape

public static Object **reshape** (Object *array*, int[] *newDimensions*)
Reshape array. Create a new array with the same total size but with different dimensions. Last index is moving fastest, while the elements are copied one after another. Works only on rectangular arrays.

Parameters

- **array** –
- **newDimensions** –

Returns new array

set

public static void **set** (Object *array*, int[] *index*, Object *value*)
Set an element in a multidimensional array. Works on all arrays.

size

public static long **size** (Object *array*)
Total number of items in array. Works only on rectangular arrays.

Data

public class **Data**
Wrapper around data. It enables us to live with just one copy of most methods and pass data around independently of its dimension and numerical type. It can be constructed using: 1) one of the 6x3 constructors with explicit array types 2) a general constructor using Object and dimensionality data specified by you.

Author pipek

Fields

DOUBLE

public static final int **DOUBLE**

FLOAT

public static final int **FLOAT**

INT16

public static final int **INT16**

INT32

public static final int **INT32**

INT64

public static final int **INT64**

INT8

public static final int **INT8**

MAX_RANK

public static final int **MAX_RANK**

Constructors

Data

public **Data**(byte[] *data*)

Data

public **Data**(byte[][] *data*)

Data

public **Data**(byte[][][] *data*)

Data

public **Data**(short[] *data*)

Data

public **Data**(short[][] *data*)

Data

public **Data**(short[][][] *data*)

Data

public **Data**(int[] *data*)

Data

public **Data**(int[][] *data*)

Data

public **Data**(int[][][] *data*)

Data

public **Data**(long[] *data*)

Data

public **Data**(long[][] *data*)

Data

public **Data**(long[][][] *data*)

Data

public **Data**(double[] *data*)

Data

public **Data**(double[][] *data*)

Data

public **Data**(double[][][] *data*)

Data

public **Data**(float[] *data*)

Data

public **Data**(float[][] *data*)

Data

public **Data**(float[][][] *data*)

Data

public **Data**(Object *data*, int *type*, long[] *dimensions*)

Generic constructor.

Parameters

- **data** – The correct array object (or null)
- **type** – Type of data in terms of this class constants.
- **dimensions** – Length along all dimensions. Gets copied. This is useful if we obtain data from external source, we know its properties but we don't want to cast them unnecessarily (like read from HDF5). Data needn't be specified (in such case a new array is created.)

Methods

asDoubleArray1D

public double[] **asDoubleArray1D**()
Get a 1-D double array representation of data. Works for 1-D data. For doubles, it simply returns, for others, it converts them using ArrayUtils.asDoubleArray() (see).

Throws

- ***WrongDimensionsException*** – if data are not 1-D.
- ***UnknownDataTypeException*** – if conversion to double is not supported by underlying procedure.

createRawObject

public static Object **createRawObject**(int *type*, long[] *dimensions*)
Create a multidimensional array object of a selected type and dimensions.

flatten

public boolean **flatten**()
Remove dimensions that have length 1. Preserves 1D arrays. Makes changes only if there is a trivial dimension.

Returns true if there was a change, false otherwise. Motivation: Our HDF5 file sometimes have Nx1 arrays instead of N arrays.

from1DArray

public static *Data* **from1DArray**(Object *array*, int *type*, long[] *dimensions*)
Reshapes 1-D array to a multidimensional array of correct dimensions. Motivation: HDF5 library returns only 1-D arrays. In C order.

getDimensions

public long[] **getDimensions**()
Get length of the data along all dimensions.

getRank

public int **getRank**()
Get the number of dimensions. i.e. double[] => 1, short[][] => 2 etc.

getRawData

public Object **getRawData**()
Get the original array used for the initialization of this object. No type information, you have to cast it yourself.

getType

public int **getType**()
Get the type of stored array (in terms of constants defined in this class). Our data types differ from HDF5 so that this class is not dependent on HDF5 library. Hdf5Utils class contains conversion routines.

linearTransform

public *Data* **linearTransform**(double *mul*, double *add*)

DebugUtils

public class **DebugUtils**
Utilities helping with debugging.

Author pipek

Methods

logWithDuration

public static void **logWithDuration**(String *message*)
Print a message alongside with the number of nanoseconds since last call of this method.

Parameters

- **message** – Message to log

logWithDuration

public static void **logWithDuration**(String *message*, Logger *logger*)
Print a message alongside with the number of nanoseconds since last call of this method.

Parameters

- **message** – Message to log
- **logger** – Logger to use

DictionaryAdapter

public class **DictionaryAdapter** extends *PythonAdapter*
Object that is represented by a python dictionary (or OrderedDict) and usually obtained as a row from database. Each of the classes should implement static method create(PyObject) which returns null if underlying Python object is None. There should be no public constructor in child classes.

Constructors

DictionaryAdapter

public **DictionaryAdapter**(PyObject *object*)

Methods

asMap

public SortedMap **asMap**()
Map python dictionary to java map. Motivation: Binding for MATLAB.

get

public PyObject **get**(String *key*)
Get dictionary value indexed by key as python object.

getDictKeys

public String[] **getDictKeys**()
Get list of dictionary keys. Motivation: Binding for IDL.

Returns Array of keys in the order they are stored in by Python.

getDictValue

public Object **getDictValue**(String *key*)
Get dictionary value indexed by key as "Java native" object. Motivation: Binding for IDL.

See also: *PythonUtils.asNative(org.python.core.PyObject)*

getDictValueAsDouble

public double **getDictValueAsDouble**(String *key*)
Get dictionary value indexed by key as double. Motivation: Binding for IDL.

getDictValueAsLong

public long **getDictValueAsLong**(String *key*)
Get dictionary value indexed by key as long. Motivation: Binding for IDL.

getDictValueAsString

public String **getDictValueAsString**(String *key*)
Get dictionary value indexed by key as string. Motivation: Binding for IDL.

Hdf5ReadException

public class **Hdf5ReadException** extends *CDBException*

Author pipek

Constructors

Hdf5ReadException

public **Hdf5ReadException** (String *message*)

Hdf5Utils

public class **Hdf5Utils**

Utilities for reading/writing HDF5 files. Methods readData, writeData and writeData

Author pipek

Methods

readFileData

public static *Data* **readFileData** (String *filePath*, String *fileKey*)

Read data from file. Up to 3-D arrays are supported. (Higher rank would require reimplementation of Data class)

Parameters

- **filePath** – Path to the file.
- **fileKey** – Name of the dataset in the file.

writeData

public static void **writeData** (String *filePath*, String *fileKey*, *Data data*)

Write (from beginning) new HDF5 dataset.

Parameters

- **filePath** – Path to the file (may or may not exist).
- **fileKey** – Name of the dataset (error if it exists).
- **data** – Wrapped data (see class Data).

writeData

public static void **writeData** (String *filePath*, String *fileKey*, *Data data*, long[] *offset*)

Write (or append) data to an existing HDF5 dataset.

Parameters

- **filePath** – Path to the file.
- **fileKey** – Name of the (existing) dataset.
- **data** – Wrapped data (see class Data).
- **offset** – Offset (in dimensions) from which start with writing data. If offset == null, new data set is created. You have to be sure that your offset is correct. The dataset is enlarged if possible but you can overwrite existing data if you are not careful.

writeData

public static void **writeData**(String *filePath*, String *fileKey*, int *dataType*, long[] *dimensions*, Object *rawData*, long[] *offset*)

Write data to a HDF5 dataset (generic version). This generic version is available as a public method, still it is more convenient (and practical) to use writeData(String, String, Data) or writeData.

Parameters

- **filePath** – Path to the file (may or may not exist).
- **fileKey** – Name of the dataset.
- **dataType** – HDF5 data type.
- **dimensions** – The dimensions of the data array.
- **rawData** – The data array of any format.
- **offset** – Offset of data (null => create new dataset)

Hdf5WriteException

public class **Hdf5WriteException** extends *CDBException*

An error when writing to HDF5.

Author pipek

Constructors

Hdf5WriteException

public **Hdf5WriteException**(String *message*)

JsonUtils

public class **JsonUtils**

Utility to read and write data from/to JSON format.

Author pipek

Methods

asNative

public static Object **asNative**(JsonElement *element*)

Transform JSON element into native object. Works recursively: - double, boolean, string - native - arrays => Object[] - objects => TreeMap

parseNative

public static Object **parseNative**(String *source*)

Take JSON string and turn it into native objects.

ParameterList

public class **ParameterList** extends HashMap<String, PyObject>

List of parameters for python methods that accepts (via put method) some of the basic types in addition to default PyObject. Various put methods just simplify adding of native objects.

Methods

put

public String **put** (String *key*, String *value*)

put

public long **put** (String *key*, long *value*)

put

public int **put** (String *key*, int *value*)

put

public Date **put** (String *key*, java.sql.Date *value*)

put

public boolean **put** (String *key*, boolean *value*)

put

public double **put** (String *key*, double *value*)

put

public Timestamp **put** (String *key*, Timestamp *value*)

put

public short **put** (String *key*, short *value*)

put

public *ParameterList* **put** (String *key*, *ParameterList* *value*)

PythonAdapter

public class **PythonAdapter**

Adapter of a PyObject that offers some shortcut methods for invoking etc. Motivation: The Jython API is not as elegant as would be desirable. However, with proper understanding of PyObject, this class would not be necessary.

Author pipek

Fields

pyObject

protected PyObject **pyObject**

Constructors

PythonAdapter

public **PythonAdapter**(PyObject *object*)

Methods

getAttribute

public PyObject **getAttribute**(String *name*)

Get the attribute of PyObject.

Parameters

- **name** – Name of the attribute.

Returns The attribute as PyObject or null (if attribute not present).

getPythonObject

public PyObject **getPythonObject**()

Get the unwrapped PyObject.

hasAttribute

public boolean **hasAttribute**(String *name*)

invoke

public PyObject **invoke**(String *method*, Map<String, PyObject> *parameters*)

Call a method on PyObject with parameters.

invoke

public PyObject **invoke** (String *method*)
Call a method on PyObject without parameters.

Parameters

- **method** – Name of the method (callable attribute) to call on the object.

Returns Result of the call as PyObject

PythonUtils

public class **PythonUtils**
Utilities for easier manipulation with Python through Jython.

Author pipek

Methods

asNative

public static Object **asNative** (PyObject *object*)
Interpret the python object as a native Java one.

Returns Object of the correct type. It has to be cast to be useful. Numbers converts to numbers (long or double). Dicts converts to TreeMap. Tuples and lists converts to Object[]. Dates converts to Date. NoneType converts to null. In case it does not know a type, it returns its string representation along with ! and type name. (e.g. "!representation of my weird type{WeirdType}")

getInterpreter

public static synchronized PythonInterpreter **getInterpreter** ()
Get a single copy of python interpreter. More of them should not be needed.

invoke

public static PyObject **invoke** (PyObject *object*, String *method*, Map<String, PyObject> *parameters*)
Invoke a method on a Python object with named parameters. It calls *invoke* method present in Jython, it only allows easier manipulation with parameters (enabling this to be called with ParameterList).

newObject

public static PyObject **newObject** (String *className*, Map<String, PyObject> *parameters*)
Create a new instance of a named python class.

UnknownDataTypeException

public class **UnknownDataTypeException** extends *CDBException*

Author pipek

Constructors

UnknownDataTypeException

public **`UnknownDataTypeException`** (String *message*)

WrongDimensionsException

public class **`WrongDimensionsException`** extends *CDBException*

Author pipek

Constructors

WrongDimensionsException

public **`WrongDimensionsException`** (String *message*)

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

U

V

W