dlisio Documentation

Release 0.1

Equinor

Sep 14, 2020

Contents

For help, examples and reference, type `help(function)` in your favourite python interpreter, or `pydoc function` in the unix console.

Before you get started we recommended that you familiarize yourself with some basic concepts of the dlis file format. Remember, dlis is a non-trivial format to work with. Even though dlisio aims to be simple-to-use library that abstracts away some of the pain of dlis, there are some concepts from the standard you need to be familiar with to be able to work with these files. Check out this short overview of need-to-know-concepts: *Digital Log Interchange Standard (DLIS)*.

CHAPTER 1

About the project

Welcome to dlisio. dlisio is a python package for reading Digital Log Interchange Standard (DLIS) v1. Version 2 exists, and has been around for quite a while, but it is our understanding that most dlis files out there are still version 1. Hence dlisio's focus is put on version 1[1], for now.

dlisio attempts to abstract away a lot of the pain of dlis and gives access to the data in a simple and easy-to-use manner. It gives the user the ability to work with dlis-files without having to know the details of the standard itself. Its main focus is making the data accessible while putting little assumptions on how the data is to be used.

dlisio is written and maintained by Equinor ASA as a free, simple, easy-to-use library to read well logs that can be tailored to our needs, and as a contribution to the open-source community.

Please note that dlisio is still in alpha, so expect breaking changes between versions.

[1] API RP66 v1, http://w3.energistics.org/RP66/V1/Toc/main.html

CHAPTER 2

Changelog

All notable changes to this project will be documented in this file, for a complete overview of changes, please refer to the git log.

The format is based on Keep a Changelog, but most notably, without sectioning changes into type-of-change.

2.1 0.2.4 - 2020.07.27

- fixes a bug in `dl::findoffsets` that caused an infinite loop for certain broken files.

2.2 0.2.3 - 2020.06.19

- Fixes a bug in `dlisio.load()` that caused it to leak open file handles when load failed.
- Added official support and distributed wheels for python 3.8.
- Better error message is reported when attempting to load files which do not exist.
- dlisio can now read files which contain empty logical records.
- The cli tool describe.cpp is removed as it has not been maintained and used.

2.3 0.2.2 - 2020.06.15

- Fixes a bug in `dlisio.load()` that caused it to leak an open file handle.

2.4 0.2.1 - 2020.06.05

- Fixes a bug in the build script that creates the macos wheels. The lfp library was not properly included, resulting in an import error when importing dlisio.

2.5 0.2.0 - 2020.06.04

- dlisio can now read files wrapped in Tape Image Format (tif).
- dlisio can now read files that do not contain a Storage Unit Label.
- The numpy array returned by `frame.curves()` can now be indexed with fingerprints in addition to the normal mnemonic indexing. Fingerprints are a more reliable indexing method as these are required to be unique by the standard, unlike mnemonics. This should mainly be of interest to automation pipelines where reliable indexing is key.
- dlisio can now read frames with duplicated channels. This behavior is explicitly forbidden by the spec. However, it is frequently violated. By default, `frame.curves()` still fails, but this can now be bypassed with `strict=False`.
- dlisio no longer accepts files where the last Visible Record is truncated, but the last Logical Record is intact. Support for such truncated files was never intended in the first place, but happened to work.
- `Channel.curves()` fails more gracefully when there is no recorded curve data.
- The documentation has been revamped and new sections focusing on understanding the content and structure of dlis-files are added.
- Fixes a bug that caused `channel.curves()` to use too much memory.
- Fixes a bug that causes `dlisio.load()` to fail if the file contained encrypted fdata record(s).
- Fixes a bug that caused `dlisio.load()` to fail if the obname of a fdata record spanned multiple Visible Records.
- Fixes a bug that re-read unknown objects from disk even if they were cached from previous reads.

2.6 0.1.16 - 2020.01.16

- Fixes a bug were `dlisio.load()` did not properly close the memory mapping to the file when loading failed.
- Fixes a bug where `dlis.match()` and `dlis.object()` returned the same object multiple times under certain circumstances.
- `dlis.describe()` again includes the object-count of each object-type.
- `dlisio.load()` now warns if a file contains `Update`-objects. The current lack of support for such objects means that dlisio may wrongfully present data in files with `Update`-objects.
- There is now a list of organization codes on readthedocs
- Fixes a bug in the Process-docs

2.7 0.1.15 - 2019.12.18

- Metadata objects are now parsed and loaded when needed, rather than all at once in `dlisio.load()`. This is not directly observable for the user, other than it improves performance for `dlisio.load()`. For files with a lot of metadata, the performance gain is huge.
- dlisio can now read even more curve-data. Specifically, where multiple FDATA (rows) are stored in the same IFLR.
- The array from `Frame.curves()` now includes FRAMENO as the first column. FRAMENO are the row numbers as represented in the file. It might happen that there are missing rows or that they are out-of-order in the file, that is now observable by inspecting FRAMENO.
- Better support for non-ascii strings. It is now possible to tell dlisio which string encodings to try if decoding with 'utf-8' fails. Supply a list of encodings o `set_encodings()` and dlisio will try them in order.
- `Frame.index` now returns the Channel mnemonic, not the `Channel`-object.
- `Channel.index` is removed.
- Validated types are now represented as tuples, not lists.
- Fixes a bug were microseconds in datetime objects were interpreted as milliseconds.
- Better error message when incomplete Channels objects cause parsing of curves to fail as a result.

2.8 0.1.14 - 2019.10.14

- dlisio has learned to read curves with variable length data types. Thus, every data-type that the standard allows for curves is now supported by dlisio.
- `Frame`- and `Channel`-objects now have an index-property. `index` returns the `Channel`-object that serves as the index-channel for the given Frame/Channel.

2.9 0.1.13 - 2019.10.3

- The sphinx documentation on readthedocs has a few new sections: About the project, an introduction to some dlis-concepts and a quick guide to help new users to get started with dlisio.
- API documentation has seen some improvements as well. The `dlis`-class documentation is revamped to better help users to work with logical files and accessing objects. `Frame` and `Channel` are more thoroughly documented, and more examples on how to work with curve data are provided.
- Direct access to specific objects has been made more convenient with `dlis.object()`.
- `dlis.match()` is no longer case sensitive.
- `dlis.fileheader` now returns the `Fileheader`-object directly, not wrapped as dict_values.
- `dlis.objects` has been removed
- CircleCI is added to the ci-pipeline for building and testing on linux
- Python test suite has seen some refactoring
- It is now possible to build the python module with `setup.py`, provided the core library is already installed on the system.

2.10 0.1.12 - 2019.08.15

- Output a readable summary of any metadata-object, logical file or batch-object with `.describe()`.
- Access to curves directly through `Frame`- and `Channel`-objects.
- dlisio has learned to read the following metadata-objects: `Process`, `Path`, `Splice`, `Well reference point`, `Group`, `Message`, `Comment`.
- `dlis.match()` lets you search for objects with a regular expression.
- dlisio now reads even more files. Restrictions such as number-of-objects in an object_set and missing representation codes in templates have been lifted.
- The parsing routine has seen some improvements. This includes giving the user more freedom to customize object-parsing.
- Multidimensional metadata attributes are handled correctly.
- `BasicObject.update_stash` has been removed.
- `dlis.getobjects()` has been removed.
- `dlis.object_set` has been renamed to `dlis.indexedobjects`.
- `Computation.source` is now a scalar, not vector.
- `BasicObject`'s `type` and `attic` is now attributes, not properties.
- Objects are allowed to have empty ids (name/mnemonic).
- The API documentation has seen some minor updates.
- dlisio uses endianness.h rather than its own implementation.
- Some of the binary test files have been simplified.
- core functionality such as `findfdata`, `findsul`, `findvrl`, `findoffsets` and `stream.at` are more thoroughly tested.
- Parts of the Python test suite have been refactored.
- Fixed a bug were long obnames were allocated insufficient memory.
- Fixed a bug were multi-dimensional fdata were interpreted incorrectly.
- Fixed a bug that caused incorrectly partitioning from physical- to logical file(s).
- Fixed a bug that caused parsing of a encrypted logical record to fail.

2.11 0.1.11 - 2019.06.04

- Support for logical files - dlisio now partitions the loaded physical file into logical files. This has resulted in a behavioral change were `dlisio.load()` now returns a tuple-like object of n-logical files.

CHAPTER 3

Digital Log Interchange Standard (DLIS)

dlis is a binary file format for well logs, developed by Schlumberger in the late 80's and published by the American Petroleum Institute (API) in 1991. In 1998 the stewardship was handed off to Petrotechnical Open Software Corporation (POSC), now known as energistics[1]

When developing dlis standard, the main emphasis was put on easy writing. The files are structured in such a way that data can be written directly while acquiring the logs. This is very handy for the producers of the files, and equally tedious for the consumer that wants to read them later on. Additionally it is a very tolerant standard. It specifies general data-structures within the file, such as channels and frames, but allows the producers to modify these to fit their needs. It also allows for completely new structures, not defined by the standard itself, such as vendor-specific object-types. This all sums up to a fairly complex standard with a lot of quirks and weirdness to it. It is safe to say that dlis a particularly difficult format to work with.

[1] POSC, https://www.energistics.org/

CHAPTER 4

Tape Image Format (TIF)

From version 0.2.0 dlisio supports Tape Image Format (TIF) files. TIF is a file format used to wrap other file formats, like DLIS, such that the file can be read by tape readers. A TIF'd DLIS-file is simply a regular DLIS-file plus some extra information that tape readers rely on in order to read the file. This information is of no use to the consumer of the file and **there is no semantic distinction between a regular DLIS-file and a TIF'd DLIS-file**.

Unlike other known DLIS-readers, dlisio does not require you to run the file through a tapemark-remover prior to opening it or to explicitly inform dlisio whether or not the passed file is DLIS or TIF. In practice this means that you as a consumer of dlis-files never need to know nor care whether your file is TIF'd or not, and can always open the file as if it was a regular DLIS:

```
with dlisio.load('tif.dlis') as files:
   pass
```

CHAPTER 5

Organization codes

Organizations are assigned their own organization codes by energistics. These organization codes typically pop up in the metadata of dlis files, e.g. to identify the producer.

The rp66v1 standard allows vendors to specify their own metadata objects. It also specifies that the type of such vendor-specific objects should always start with the organization code like so:

```
>>> f.unknowns
dict_keys(['440-FILE', '440-OP-TOOL', '440-CHANNEL'])
```

From he naming we can see that these objects are defined by Schlumberger. The semantic meaning of such objects may only be known to the producer.

Name (Code)	Description (Organization Name)
0	Subcommittee On Recommended Format For Digital Well Data, Basic Schema
1	Operator
2	Driller
3	Mud Logger
4	Abyssus Marine Services AS
6	ALT – Advanced Logic Technology (added 2014-09-18)
9	Amerada Hess
10	Analysts, The
12	ArenaPetro
15	Baker Hughes Inteq
20	Baroid
30	Birdwell
40	Reeves (1 Jan 99; formerly BPB)
50	Brett Exploration
58	Canrig (added 2009-09-09)
60	Cardinal
65	Center Line Data
66	Subcommittee On Recommended Format For Digital Well Data, DLIS Schema

Continued on next page

Table 1 – continued from previous page

Name (Code)	Description (Organization Name)
70	Century Geophysical
77	CGG Logging, Massey France
80	Charlene Well Surveying
85	China Oilfield Services Limited (COSL) (added 2019-02-11)
90	Compagnie de Services Numerique
95	Comprobe
100	Computer Data Processors
110	Computrex
115	COPGO Wood Group
120	Core Laboratories
125	CRC Wireline, Inc.
126	Crocker Data Processing Pty Ltd
127	Tucker Wireline Services (formerly Davis Great Guns Logging, Wichita, KS)
128	Datalog Technology (added 2009-09-09)
130	Digigraph
137	Tucker Technologies (formerly Digital Logging Inc.), Tulsa, OK.
140	Digitech
145	Deines Perforating
148	Drillog Petro-Dynamics Limited
150	Baker Atlas (formerly Dresser Atlas)
155	Dynamic Technologies (DTCC)
160	Earthworm Drilling
170	Electronic Logging Company
180	Elgen
190	El Toro
200	Empire
205	Encom Technology, Ltd.
206	Ensigh Geophysics, Ltd.
208	Epoch (merged with Canrig, Nov 2008; added 2009-09-09)
210	Frontier
213	GeoEnergy, Inc.
214	Geokinetics Inc.
215	Geolog
216	Geophysical Data Systems (added 2015-01-13)
217	Geoshare
218	GEO·X Systems Ltd.
220	G O International
225	GOWell Petroleum (added 2012-04-02)
230	Gravilog
240	Great Guns Servicing
250	Great Lakes Petroleum Services
260	GTS
268	Guardian Data Seismic Pty. Ltd.
270	Guns
280	Halliburton Logging
283	Harvey Rock Physics (added 2015-01-13)
285	Horizon Production Logging
290	Husky
293	INOVA Geophysical Equipment Limited (updated 2012-04-02)

Continued on next page

Table 1 – continued from previous page

Name (Code)	Description (Organization Name)
295	Input/Output, Inc.
297	iO Data AS
300	Jetwell
305	Landmark Graphics
310	Lane Wells
315	Logicom Computer Services (UK) Ltd
320	Magnolia
330	McCullough Tool
332	Mitchell Energy Corporation
333	MST Ltd. (Modern Seismic Technologies) – added 2014-08-15)
335	Paradigm Geophysical (formerly Mincom Pty Ltd)
337	DPTS Limited (formerly MR-DPTS Limited, changed as of 2008-08-12)
338	NRI On-Line Inc
339	Oilware, Inc.
340	Pan Geo Atlas
342	Pathfinder Energy Services
345	Perfco
350	Perfojet Services
360	Perforating Guns of Canada
361	Petcom, Inc.
362	CGG (FKA Petroleum Exploration Computer Consultants, Ltd).
363	Petrologic Limited
364	PetroMar Technologies
366	Phillips Petroleum Company
367	Phoenixdata Services Pty Ltd.
368	Petroleum Geo-Services (PGS)
370	Petroleum Information
380	Petrophysics
390	Pioneer
392	The Practical Well Log Standards Group
395	IHS Energy Log Services (formerly Q. C. Data Collectors)
400	Ram Guns
410	Riley's Datashare
418	RODE
420	Roke
430	Sand Surveys
440	Schlumberger
450	Scientific Software
455	Seismic Instruments, Inc.
460	Seismograph Service
462	SEGDEF
463	SEG Technical Standards High Density Media Format Subcommittee
464	Shell Services Company
465	Stratigraphic Systems, Inc.
466	Spectrum ASA
467	Sperry-Sun Drilling Services
468	SEPTCO
469	Sercel, Inc.
470	Triangle

Continued on next page

Table 1 – continued from previous page

Name (Code)	Description (Organization Name)
471	Thrubit Logging(added 2009-09-09)
472	TGS
475	Troika International
480	Welex
490	Well Reconnaissance
495	Wellsite Information Transfer Specification (WITS)
500	Well Surveys
510	Western
520	Westronics
525	Winters Wireline
530	Wireline Electronics
540	Worth Well
560	Z & S Consultants Limited
999	Reserved for local schemas
1000	Energistics (formerly POSC, changed as of 2006-11-06)

CHAPTER 6

Physical and logical files

The dlis standard distinguishes between physical files and logical files. A physical file would be a .dlis file. A logical file, on the other hand, is a collection of data that are logically connected. Typically this can be all measurements and metadata gathered at one run. A physical file may contain multiple logical files, where each logical file is independent of the other logical files:

```
                physical file (.dlis)
 ------------------------------------------------------
| logical file 0 | logical file 1 | ... | logical file n |
 ------------------------------------------------------
```

dlisio's main entrypoint, *dlisio.load()* takes a physical file and returns a tuple-like object with all the logical files.

Logical files can be thought of as a pool of metadata and curve-values that all are related in some way. Some common metadata objects are Fileheader, Origin, Frame, Channel, Tool, Parameter and Calibration. For a full list of all metadata objects, see *Object types*:

```
                logical file
 ----------------------------------------------------
| Fileheader | Origin | Frame1 | Channel | Frame2 | .. |
 ----------------------------------------------------
```

This is a simplified illustration of the logical file structure, but to work with dlis-files this is pretty much how you should think of logical files. To see how to access specific objects, check out *dlisio.dlis*

CHAPTER 7

Metadata

Together with the actual logs, there is often an abundance of metadata related to the acquisition of the logs. In a DLIS-file metadata is structured into different `dict`-like objects that describe certain pieces of information. RP66v1 defines over 20 object-types. In practice, only a handful see widespread use. Please refer to the *API Reference* to get a full[1] overview of the different object-types. Here are some examples of the frequently used ones:

Origin: Contains general information about the file, and the circumstances in which it was produced.

Channel: Description of a specific curve in the logical file.

Frame: A grouping of channels that all share the same index, typically a logpass. The actual curve-data are accessed through Frames.

Tool: Describes a physical tool that was used for the acquisition of the logs.

Parameter: Contains some parameter value and a description of it.

7.1 Identifying specific objects

Common for *all* objects are the four fields: *type*, *name* (mnemonic), *origin* and *copynumber*. Together, these form a unique identifier for the object. RP66v1 states that no two objects from the *same logical file* can have the same value for all four fields.

origin: The origin field states which origin the object is a part of. Its interger value implicitly refers to the *Origin*-object that has the same value in its origin field.

copynumber: The copynumber is used to distinguish two objects that otherwise have an identical signature. E.g. if there are two *Channel* objects with the same name/mnemonic and both belong to the same origin.

To access a specific object use *dlis.object()*. Or search for objects matching a regular expression with *dlis.match()*

[1] There is currently a handful of object-types (mostly from Chaper 7: semantics: dictionaries of RP66v1) that dlisio does not implement full support for. I.e. they are not translated into their own python class, but rather use the more generic and rough interface of *plumbing.BasicObject*. These rarely see the light of day, but if present they can be accessed through *dlis.unknowns*.

```
>>> channel = f.object('CHANNEL', 'GR', origin=1, copynumber=0)
>>> channel.long_name
'Gamma Ray'

>>> channels = f.match('.*GR.*', 'CHANNEL')
>>> channels
['GR', 'RGR']
```

Note: Note that *dlis.object()* allows you to ommit the origin and/or copynumber, but will raise if it's unable to uniquely identify the object. The documentation for *dlis.object()* and *dlis.match()* offers more examples.

7.2 Relationship between metadata objects

A key feature of RP66v1 is that objects refer to each other. Object-to-object referencing is used to establish a relationship between two objects. This relationship can serve as an implicit context to the object. Many objects rely on this context and make little sense without it.

Object-references are conveyed through the object's attributes. A concrete example is *Tool.channels*, which references all the channels that are produced directly by that tool. dlisio automatically resolves object references to make it easy to work with programmatically:

```
>>> tool = f.object('TOOL', 'USIT')
>>> tool.channels
[TDEP, BI, CBL, CBLF, CBSL, ..., CMCG, WF1, WF1N, WF2, WF2N]

>>> channel = tool.channels[1]
>>> channel.long_name
'Bond Index'
```

7.3 Multiple origins

RP66v1 allows for a single logical file to have multiple *Origin*-objects. Origin describes the source of the data, such as which field and well it's from. It's therefore theoretically possible that the data contained in a logical file stem from different sources when there are more than one *Origin*-objects. Such files obviously need special care. More precisely, the origin field of each object that is accessed needs to be examined in order to determine which origin it belongs to.

The majority of files do *not* contain multiple origins, which makes it safe to ignore the origin field. However, it is considered a good practice to check the origin count when opening a new file, e.g. by issuing a warning if there are more than one:

```
import dlisio
import logging

with dlisio.load(path) as (f, *tail):
   if len(f.origins) > 1: logging.warning('File contains multiple origins')
```

7.4 Vendor-specific metadata

In addition to the many object-structures defined by RP66v1 itself, vendors are free to specify their own metadata objects. However, with often cryptic naming and minimal explanation, such objects can be challenging to decipher without any external explanation of the intent of these objects.

The vendor-specific objects are reachable through `f.unknowns`:

```
with dlisio.load(path) as (f, *tail):
   f.unknowns
```

CHAPTER 8

Curves

The primary data of a dlis-file are curves, also referred to as channels. Typically a curve is the recorded measurements taken along the borehole, indexed against an axis, like depth or time. But a curve can also be a computation of some sort, e.g. a calibrated version of a measured curve.

In dlisio, curves are accessed through *Frame*- or *Channel*-objects, by calling their `curves()` methods. The primary data type for curves is structured numpy.ndarray. This enables quick and easy mathematical operations on the data you care about.

8.1 Frame-objects

Curves are organized in frames. Conceptually, a frame can be seen as a table of data where each column is a curve, like in the time-indexed frame below. Frames almost always have an index channel that provides the position in e.g. depth or time at which the rest of the values in the row were measured. Each frame usually corresponds to a log run, but otherwise frames impose little structure except grouping channels that have a common index. There is no restrictions on the number of channels per frame, or the number of frames in a file.

Frames are described by *Frame*-objects. These contain a list of *Channel*-objects, which describe the individual curves in more detail. Frame objects also contain additional information about the index channel, such as its min/max values, spacing, direction and type-of-index. See *Frame* for a full list of its attributes.

Table 1: A TIME indexed frame

FRAME NO	TIME	TDEP	ETIM	LMVL	UMVL	CFLA	OCD
1	16677259	852606.0	0.	585	635	18	6789.05
2	16677659	852606.0	0.4	585	635	18	6789.05
3	16678059	852606.0	0.8	585	635	0	6789.05
4	16678459	852606.0	1.2	585	635	0	6789.05
5	16678859	852606.0	1.6	585	635	0	6789.05

Note: FRAMENO: The first column of every Frame is always FRAMENO, which is the row number. Generally FRAMENO is not that interesting, but it can aid in debugging strange-looking curve-data. For example if you are suspecting that some of the data is missing (or even is out-of-order).

8.2 Channel-objects

Curve-metadata is recorded in *Channel*-objects. Each *Channel*-object describes a single curve, e.g. TDEP, GR, VDL, WF1, etc. The metadata includes a description, the dimension of each sample and their units, which *Frame* the curve belongs to and the source of the curve. The source is the entity that produced the curve. For measured curves, that is typically a *Tool*. For computed curves, the source might be a *Computation*. Additionally, *Channel* contains a list of property indicators. These indicate the general intrinsic properties of the Channel. Some examples are MEASURED, COMPUTED, DEPTH-MATCHED and AVERAGED. Appendix C of RP66v1 defines the full list of property indicators with descriptions.

8.3 N-dimensional curve samples

By far the most common case is that each channel has scalar samples i.e. a single measured numerical value per row, however RP66v1 allows each sample to be a multi- dimensional array. For example, this is common for ultrasonic logs, where some channels contain a one-dimensional array of values per row, representing measurements made at different azimuthal angles. For other channels each sample may even be a 2- or 3-dimensional array.

RP66v1 imposes no limit on the dimensionality that a channel-sample can have, nor does dlisio.

Note: Thanks to numpy's structured numpy.ndarray, dlisio is able to support n-dimensional curves of any shape and form. However, other popular formats such as Pandas DataFrame or CSV are only able to handle tabular data, i.e. frames where all channels have scalar sample values. Hence, conversion to these formats may not always be possible.

8.4 Channels with no data

It is not uncommon for files to have *Channel*-objects describing curves that are not present in the file. I.e. there is metadata describing a curve, but the curve itself is missing.

This typically happens because metadata like *Tool*-objects and all the curves it can produce is recorded prior to the acquisition. However when acquisition starts only a subset of the channels is actually recorded, but the metadata for the unused channels are never removed.

Note: Calls to *Channel.curves()* and *Channel.frame* returns `None` when there is no recorded curve data.

CHAPTER 9

Strings and encodings

set_encodings(*encodings*)

Set codepages to use for decoding strings

RP66 specifies that all strings should be in ASCII, meaning 7-bit. Strings in ASCII have identical bitwise representation in UTF-8, and python strings are in UTF-8. However, a lot of files contain strings that aren't ASCII, but encoded in some way - a common is the degree symbol[1], but plenty of files use other encodings too.

This function sets the code pages that dlisio will try *in order* when decoding strings (IDENT, ASCII, UNITS, and when they appear as members in e.g. ATTREF). UTF-8 will always be tried first, and is always correct if the file behaves according to spec.

Available encodings can be found in the Python docs[2].

If none of the encodings succeed, all strings will be returned as a bytes object.

Parameters **encodings** (*list of str*) – Ordered list of encodings to try

Warns **UnicodeWarning** – When no decode was successful, and a bytes object is returned

Warning: There is no place in the DLIS spec to put or look for encoding information, decoding is a wild guess. Plenty of strings are valid in multiple encodings, so there's a high chance that decoding with the wrong encoding will give a valid string, but not the one the writer intended.

Warning: It is possible to change the encodings at any time. However, only strings created after the change will use the new encoding. Having strings that are out of sync w.r.t encodings might lead to unexpected behaviour. It is recommended that the file is reloaded after changing the encodings to ensure that all strings use the same encoding.

See also:

[1] https://stackoverflow.com/questions/8732025/why-degree-symbol-differs
[2] https://docs.python.org/3/library/codecs.html#standard-encodings

get_encodings() currently set encodings

Notes

Strings are decoded using Python's bytes.decode(errors = 'strict').

References

Examples

Decoding of the same string under different encodings

```
>>> dlisio.set_encodings([])
>>> with dlisio.load('file.dlis') as (f, *_):
...     print(getchannel(f).units)
b'custom unit\xb0'
>>> dlisio.set_encodings(['latin1'])
>>> with dlisio.load('file.dlis') as (f, *_):
...     print(getchannel(f).units)
'custom unit°'
>>> dlisio.set_encodings(['utf-16'])
>>> with dlisio.load('file.dlis') as (f, *_):
...     print(getchannel(f).units)
''
```

get_encodings()

Get codepages to use for decoding strings

Get the currently set codepages used when decoding strings.

Returns **encodings**

Return type list

See also:

set_encodings()

CHAPTER 10

Open and Load

load(*path*)

Loads a file and returns one filehandle pr logical file.

The dlis standard have a concept of logical files. A logical file is a group of related logical records, i.e. curves and metadata and is independent from any other logical file. Each physical file (.dlis) can contain 1 to n logical files. Layouts of physical- and logical files:

Physical file:

```
 ------------------------------------------------------
| Logical File 1 | Logical File 2 | ... | Logical File n |
 ------------------------------------------------------
```

Logical File:

```
 ------------------------------------------------------
| Fileheader |  Origin  |  Frame  |  Channel  | curvedata |
 ------------------------------------------------------
```

This means that dlisio.load() will return 1 to n logical files.

Parameters **path** (*str_like*) –

Examples

Read the fileheader of each logical file

```
>>> with dlisio.load(filename) as files:
...     for f in files:
...         header = f.fileheader
```

Automatically unpack the first logical file and store the remaining logical files in tail

```
>>> with dlisio.load(filename) as (f, *tail):
...     header = f.fileheader
...     for g in tail:
...         header = g.fileheader
```

Note that the parentheses are needed when unpacking directly in the with- statement. The asterisk allows an arbitrary number of extra logical files to be stored in tail. Use len(tail) to check how many extra logical files there are.

Returns **dlis**

Return type tuple(*dlisio.dlis*)

open(*path*)

Open a file

Open a low-level file handle. This is not intended for end-users - rather, it's an escape hatch for very broken files that dlisio cannot handle.

Parameters **path** (*str_like*) –

Returns **stream**

Return type dlisio.core.stream

See also:

dlisio.load()

CHAPTER 11

Logical files

class dlis

Logical file

A logical file is a collection of objects - in lack of a better word, that are logically connected in some way. Think of a logical file as a pool of objects. Some examples of objects are Channel, Frame, Fileheader, Tool. They are all accessible through their own attribute.

object() and *match()* let you access specific objects or search for objects matching a regular expression. Unknown objects such as vendor-specific objects are accessible through the 'unknown'-attribute.

Uniquely identifying an object within a logical file can be a bit cumbersome, and confusing. It requires no less than 4 attributes! Namely, the object's type, name, origin and copynumber. The standard allows for two objects of the same type to have the same name, as long as either their origin or copynumber differ. E.g. there may exist two channels with the same name/mnemonic.

types = {}

Dispatch-table for turning `dlisio.core.basic_objects` into type-specific python objects like Channel and Frame. This is mainly intended for internal use so the typical user can safely ignore this attribute.

Object-types not present in the table are considered as unknowns. They can still be reached through *object()* and *match()* but lack the syntactic sugar added by the type-specific classes.

It is possible to monkey-patch the dispatch-table with your own custom classes. However this is considered to be a fairly advanced usage and it's then the users responsibility of ensuring correctness for the custom class.

Type dict

close()

Close the file handle

It is not necessary to call this method if you're using the *with* statement, which will close the file for you. Calling methods on a previously-closed file will raise *IOError*.

fileheader

Return the Fileheader

Returns fileheader

Return type *Fileheader*

axes = None

calibrations = None

channels = None

coefficients = None

comments = None

computations = None

equipments = None

frames = None

groups = None

longnames = None

measurements = None

messages = None

origins = None

parameters = None

paths = None

processes = None

splices = None

tools = None

wellrefs = None

zones = None

unknowns

Return all objects that are unknown to dlisio.

Unknown objects are object-types that dlisio does not know about. By default, any metadata object not defined by rp66v1 [1]. The are all parsed as `dlisio.plumbing.Unknown`, that implements a dict interface.

[1] http://w3.energistics.org/rp66/v1/Toc/main.html

Notes

Adding a custom python class for an object-type to dlis.types will in-effect remove all objects of that type from unknowns.

Returns **objects** – A defaultdict index by object-type

Return type defaultdict(dict)

match(*pattern*, *type='CHANNEL'*)

Filter channels by mnemonics

Returns all objects of given type with mnemonics matching a regex [1]. By default only matches pattern against Channel objects. Use the type parameter to match against other object types. Note that type support

regex as well. Pattern and type are not case-sensitive, i.e. match("TDEP") and match("tdep") yield the same result.

[1] https://docs.python.org/3.7/library/re.html

Parameters

- **pattern** (*str*) – Regex to match object mnemonics against.
- **type** (*str*) – Extend the targeted object-set to include all objects that have a type which matches the supplied type. type may be a regex. To see available types, refer to dlis.types.keys()

Yields **objects** (*generator of objects*)

Notes

Some common regex characters:

Regex	Description
'.'	Any character
'^'	Starts with
'$'	Ends with
'*'	Zero or more occurrences
'+'	One or more occurrences
'|'	Either or
'[]'	Set of characters

Please bear in mind that any special character you include will have special meaning as per regex

Examples

Return all channels which have mnemonics matching 'AIBK':

```
>>> channels = f.match('AIBK')
```

Return all objects which have mnemonics matching the regex 'AIBK', targeting all object-types starting with 'CHANNEL':

```
>>> channels = f.match('AIBK', type='^CHANNEL')
```

Return all CHANNEL objects where the mnemonic matches 'AI':

```
>>> channels = f.match('AI.*')
```

Return all CUSTOM-FRAME objects where the mnemonic includes 'PR':

```
>>> frames = f.match('.*RP.*', 'custom-frame')
```

Remember, that special characters always have their regex meaning:

```
>>> for o in f.match("CHANNEL.23"):
...     print(o)
Channel(CHANNEL.23)
Channel(CHANNEL123)
```

object(*type*, *name*, *origin=None*, *copynr=None*)
Direct access to a single object. dlis-objects are uniquely identifiable by the combination of type, name, origin and copynumber of the object. However, in most cases type and name are sufficient to identify a specific object. If origin and/or copynr is omitted in the function call, and there are multiple objects matching the type and name, a ValueError is raised.

Parameters

- **type** (*str*) – type as specified in RP66
- **name** (*str*) – name
- **origin** (*number, optional*) – number specifying which origin object the current object belongs to
- **copynr** (*number, optional*) – number specifying the copynumber of current object

Returns **obj**

Return type searched object

Examples

```
>>> ch = f.object("CHANNEL", "MKAP", 2, 0)
>>> print(ch.name)
MKAP
```

describe(*width=80*, *indent=''*)
Printable summary of the logical file

Parameters

- **width** (*int*) – maximum width of each line.
- **indent** (*str*) – string that will be prepended to each line.

Returns **summary** – A printable summary of the logical file

Return type Summary

load()
Force load all objects - mainly indended for debugging

promote(*objects*)
Enrich instances of the generic core.basicobject into type-specific objects like Channel, Frame, etc...

storage_label()
Return the storage label of the physical file

Notes

This method is mainly intended for internal use.

CHAPTER 12

Object types

12.1 Basic Object

class BasicObject

A Basic object that all other object-types derive from

BasicObject is mainly an implementation detail. Its the least common denominator of all object-types.

When working with dlisio you need not care too much about the BasicObject. However, keep in mind that all other object-types inherit BasicObject's attributes and methods and they can be called directly on the objects. Hence it might be a good idea to somewhat familiarize yourself with its features.

When reading the documentation keep in mind that the attributes defined on the BasicObject are **not** documented again on the derived object.

An object is uniquely identifiable within a logical file by the combination of its type, name, origin and copynumber. I.e. no two objects can have the same type, name, origin AND copynumber. This means that the standard allows for e.g. two Channels to have the same name/mnemonic, which can easily become a bit confusing.

type

Type of the object, e.g. rp66 object-types as CHANNEL or FRAME

Type str

name

Mnemonic / name

Type str

origin

Defines which origin the object belongs to

Type int

copynumber

There may exist several copies of the same object, the copynumber is used to distinguish them

Type int

attic
Attic refers the underlying basic_object, which is a dict-representation of the data on disk. The attic can be None if this particular instance was not loaded from disk.

Type dict_like

attributes = {}
Parsing guide for native dlis objects. The typical user can safely ignore this attribute. It is mainly intended for internal use and advanced customization of dlisio's parsing routine.

In short, this dictionary tells dlisio parse each attribute for a given object. By default it implements rules defined by the dlis-spec. However it can easily be customized if you'd like dlisio to parse your file in a different way.

Examples

Lets take a look at Channel.attributes, it looks like this:

```
attributes = {
    'LONG-NAME'           : scalar,
    'REPRESENTATION-CODE': scalar,
    'UNITS'               : scalar,
    'PROPERTIES'          : vector,
    'DIMENSION'           : reverse,
    'AXIS'                : reverse,
    'ELEMENT-LIMIT'       : reverse,
    'SOURCE'              : scalar,
}
```

The keys are from the dlis-spec, i.e. this is how the attributes of a CHANNEL-object are named in the file. The values tells dlisio how to interpret these keys. In the dlis-spec it's defined that 'UNITS' only contains a single value. This is communicated to dlisio with the 'scalar'-keyword. I.e. Channels __getitem__ will return a single value:

```
>>> channel['UNITS']
'm/s'
```

The __getitem__ is not intended for direct use. However it is called internally from all properties of the Channel-object. I.e. you observe the same here:

```
>>> channel.units
'm/s'
```

The following example might be a bit absurd, but keep in mind that this approach can be applied to _any_ attribute of _any_ object-type, even Unknown object-types.

But now let's say that you are in possession of a file that you know is structured differently from what the standard specifies. It contains some weird Channel's where there are multiple units per Channel. Simply update the attribute-dict before loading the file and dlisio will parse 'UNITS' as a list:

```
>>> from dlisio.plumbing import vector
>>> Channel.attributes['UNITS'] = vector
>>> with dlisio.load('file.dlis') as (f, *_):
...     ch = f.object('CHANNEL', 'TDEP')
...     ch.units
['m/s', 'rad/s']
```

The same can be achieved for a _specific_ object. Forcing a copy of the attribute-dict for a given object before altering it makes sure your changes only apply _that_ object

```
>>> ch = f.object('CHANNEL', 'TDEP')
>>> ch.attributes = dict(ch.attributes)
>>> Channel.attributes['UNITS'] = vector
>>> ch.units
['m/s', 'rad/s']
```

References

[1] http://w3.energistics.org/RP66/V1/Toc/main.html

Type dict

linkage = {}

Defines which attributes contain references to other objects. The typical user can safely ignore this attribute. It is mainly intended for internal use and advanced customization of dlisio's parsing routine.

Object-to-object references often contains implicit information. E.g. Frame.channels implicitly reference Channel object, so the type 'CHANNEL' is not specified in the file. Hence dlisio needs to be told about this.

Like for attributes, this behavior is customizable.

Examples

Change how dlisio parses Channel.source:

```
>>> from dlisio.plumbing import obname
>>> Channel.linkage['SOURCE'] = obname('PARAMETER')
>>> with dlisio.load('file.dlis') as (f, *_):
...     ch = f.object('channel', 'TDEP')
...     ch.source
Parameter('2000T')
```

The same can be achieved for a _specific_ object. Forcing a copy of the linkage-dict makes sure your changes only apply to that specific object:

```
>>> ch = f.object('channel', 'TDEP')
>>> ch.linkage = dict(ch.linkage)
>>> ch.linkage['SOURCE'] = dlisio.plumbing.parse.obname('PARAMETER')
>>> ch.source
Parameter('2000T')
```

Type dict

fingerprint

Object fingerprint

Return the fingerprint, a unique identifier, for this object. This is basically an objref type from the RP66 standard, but with a pythonic flavour, and suitable for keys in dicts.

Returns **fingerprint**

Return type str

stash
Attributes unknown to dlisio

It is not uncommon for objects to have 'extra' attributes that are not defined by the standard. Because such attributes are unknown to dlisio, they cannot be reach through normal attributes.

Returns **stash** – all attributes not defined in *attributes*

Return type dict

describe(*width=80*, *indent=''*, *exclude='er'*)
Printable summary of the object

Parameters

- **width** (*int*) – the maximum width of each line.
- **indent** (*str*) – string that will be prepended to each line.
- **exclude** (*str*) – exclude certain parts of the object in the summary.

Returns **summary** – A printable summary of the object

Return type Summary

Notes

The exclude parameter gives the option to omit parts of the summary. The table below states the different modes available.

option	Description
'h'	header
'a'	known attributes
's'	attributes from stash
'i'	attributes that violates the standard [1]
'e'	attributes with empty values (default) [2]

[1] Only applicable to attributes that should be interpreted in a specific way, such as Parameter.values. If not applicable, it is ignored.

[2] Do not print attributes that have no value.

describe_attr(*buf*, *width*, *indent*, *exclude*)
Describe the attributes of the object.

This method is intended to be called internally from describe()

12.2 Axis

class Axis(*BasicObject*)
Axis

The Axis object describes the coordinate axis of an array, e.g. the sample array of channels. One axis object describes only one coordinate axis. I.e a three dimensional array is described by three Axis objects.

axis_id
Axis identifier

Type str

coordinates
Explicit coordinate value along the axis

Type list

spacing
Constant, signed spacing along the axis between successive coordinates

See also:

BasicObject The basic object that Axis is derived from

Notes

The Axis object reflects the logical record type AXIS, defined in rp66. AXIS objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.3.1 - Static and Frame Data, Axis Objects.

12.3 Calibration

class Calibration(*BasicObject*)
Calibration objects are a collection of measurements and coefficients that defines the calibration process of channel objects.

method
Computational method used to calibrate the channels

Type str

calibrated
Calibrated channels

Type list(*Channel*)

uncalibrated
Uncalibrated channels. I.e. the channels as they where before calibration

Type list(*Channel*)

coefficients
Coefficients

Type list(*Coefficient*)

measurements
Measurements

Type list(*Measurement*)

parameters
Parameters containing numerical and textual information assosiated with the calibration process.

Type list(*Parameter*)

See also:

BasicObject The basic object that Calibration is derived from

Notes

The Calibration reflects the logical record type CALIBRATION, defined in rp66. CALIBRATION records are listen in Appendix A.2 - Logical Record Types and described detail in Chapter 5.8.7.3 - Static and Frame Data, CALIBRATION objects.

12.4 Channel

class Channel(*BasicObject*)

A channel is a sequence of measured or computed samples that are indexed against some physical quantity e.g. depth or time. The standard supports multi-dimensional samples. Each sample can be a scalar or a n-dimensional array. In addition to giving access to the actual curve-data, the Channel object contains metadata about the curve.

All Channels are a part of one, and only one, Frame. The parent Frame can be reached directly through *frame*.

Refer to the *curves()* to see some examples on how to access the curve-data.

long_name

Descriptive name of the channel.

Type str or *Longname*

reprc

Representation code

Type int

units

Physical units of each element in the channel's sample arrays

Type str

properties

Property indicators that summarizes the characteristics of the channel and the processing that have produced it.

Type list(str)

dimension

Dimensions of the samples

Type list(int)

axis

Coordinate axes of the samples

Type list(*Axis*)

element_limit

The maximum size of the sample dimensions

Type list(int)

source

The source of the channel. Returns the source object, if any

frame

Frame to which channel belongs to

Type *Frame*

See also:

BasicObject The basic object that Channel is derived from

Notes

The Channel object reflects the logical record type CHANNEL, defined in rp66. CHANNEL records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.5.1 - Static and Frame Data, CHANNEL objects.

curves()

Returns a numpy ndarray with the curves-values.

Returns **curves**

Return type np.ndarray or None

See also:

Frame.curves()

Notes

This method should only be used if there is only *one* channel of interest in a particular frame.

Due to the memory-layout of dlis-files, reading a single channel from disk and reading the entire frame is almost equally fast. That means reading channels from the same frame one-by-one with this method is _way_ slower than reading the entire frame with *Frame.curves()* and then indexing on the channels-of-interest.

Examples

Read the full curve

```
>>> curve = channel.curves()
>>> curve
array([1.1, 2.2, 3.3, 4.4])
```

The returned array supports common slicing operations

```
>>> curve[::2]
array([1.1, 3.3])
```

Read the full curve from a multidimensional channel

```
>>> curve = multichannel.curves()
>>> curve
array([[[  1,  2,  3],
        [  4,  5,  6]],
       [[  7,  8,  9],
        [ 10, 11,  12]]])
```

This curve has two samples, that both are of size 2x3. From the 1st sample, read the element located in the 2nd row, 3rd column

```
>>> curve[0][1][2]
6
```

dtype
data-type of each sample in the channel's sample array. The dtype-label is *channel.name.id*.

Returns **dtype**

Return type np.dtype

12.5 Coefficent

class Coefficient(*BasicObject*)
Records of measurements, references, and tolerances used in the calibration of channels.

label
Identify the coefficient-role in the calibration process

Type str

coefficients
Coefficients corresponding to the label

Type list

references
Nominal values for each coefficient

Type list

plus_tolerance
Maximum value that a sample can exceed the reference and still be "within tolerance"

Type list

minus_tolerance
Maximum value that a sample can fall below the reference and still be "within tolerance"

Type list

See also:

BasicObject The basic object that Coefficient is derived from

Notes

The Coefficient object reflects the logical record type CALIBRATION-COEFFICIENT, defined in rp66. CALIBRATION-COEFFICIENT records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.7.2 - Static and Frame Data, CALIBRATION-COEFFICIENT objects.

12.6 Comment

class Comment(*BasicObject*)
Comment objects contains arbitrary messages that may be interesting for the consumer e.g. a drilling report.

text
Textual comments

Type list(str)

See also:

`BasicObject` The basic object that Comment is derived from

Notes

The Comment object reflects the logical record type COMMENT, described in rp66. COMMENT objects are defined in Appendix A.2 - Logical Record Types, described in detail in Chapter 6.1.2 - Transient Data, Comment objects.

12.7 Computation

class Computation(*BasicObject*)

Results of computations that are more appropriately expressed as static information rather than as channels.

The computation value(s) may be scalars or an array. In the later case, the structure of the array is defined in the dimension attribute. The zones attribute specifies which zones the computations is defined. If there are no zones the computation is defined everywhere.

The axis attribute, if present, defines axis labels for multidimensional value(s).

long_name
Descriptive name of the computation

Type str or *Longname*

properties
Property indicators that summarizes the characteristics of the computation and the processing that has occurred to produce it

Type list(str)

dimension
Array structure of a single value

Type list(int)

axis
Coordinate axes of the values

Type list(*Axis*)

zones
Mutually disjoint zones over which the value of the current computation is constant

Type list(*Zone*)

source
The immediate source of the Computation

See also:

`BasicObject` The basic object that Computation is derived from

Notes

The Computation object reflects the logical record type COMPUTATION, defined in rp66. COMPUTATION objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.8.6 - Static and Frame Data, COMPUTATION objects.

values
Computation values

Computation value(s) may be scalar or array's. The size/dimensionallity of each value is defined in the dimensions attribute.

Each value may or may not be zoned, i.e. it is only defined in a certain zone. If this is the case the first zone, computation.zones[0], will correspond to the first value, computation.values[0] and so on. If there is no zones, there should only be one value, which is said to be unzoned, i.e. it is defined everywere.

Raises `ValueError` – Unable to structure the values based on the information available.

Returns **values**

Return type structured np.ndarray

Notes

If dlisio is unable to structure the values due to insufficient or contradictory information in the object, an ValueError is raised. The raw array can still be accessed through attic, but note that in this case, the semantic meaning of the array is undefined.

Examples

First value:

```
>>> computation.values[0]
[10, 20, 30]
```

Zone (if any) where that parameter value is valid:

```
>>> computation.zones[0]
Zone('ZONE-A')
```

12.8 Equipment

class Equipment(*BasicObject*)
Equipment objects contains information about individual pieces of surface and downhole equipment used in the acquistion of the data. Typically, tools (specified by the Tool object) is a composition of equipment.

trademark_name
The producer's name for the equipment

Type str

status
Operational status

Type bool

generic_type
Generic type

Type str

serial_number
Serial number

Type str

location
General location of equipment during acqusition

Type str

height
Heigth

length
Length

diameter_min
Minimum diameter

diameter_max
Maximum diameter

volume
Volume

weight
Weight

hole_size
Hole size

pressure
Pressure

temperature
Temperature

vertical_depth
Vertical depth

radial_drift
Radial drift

angular_drift
Angular drift

See also:

BasicObject The basic object that Equipment is derived from

Notes

The Equipment object reflects the logical record type EQUIPMENT, defined in rp66. EQUIPMENT records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.3 - Static and Frame Data, EQUIPMENT objects.

12.9 Fileheader

class Fileheader (*BasicObject*)

The Fileheader is an identifier for the Logical File. Below follows a description of the relationship between a DLIS-file, Logical File, File Set, and Storeage Set:

DLIS-file - single dlis-file may or may not consists of multiple logical files:

```
 ---------------------------------------
| Logical File 1 | ... | Logical File n |
 ---------------------------------------
```

Logical File - Each Logical File has exactly one Fileheader, but can have mutiple origins:

```
 -----------------------------------------------
| Fileheader | Origin | Frame | Channel | ... |
 -----------------------------------------------
```

File Set - A File set consists of multiple Logical Files which may spand across multiple DLIS-files. Logical Files are grouped into File Sets by producer defined criterias:

```
 ---------------------------------------
| Logical File 1 | ... | Logical File n |
 ---------------------------------------
```

Storage Set - A Storage Set consist of multiple DLIS-files. A Storage Set may or may not coincide with a File Set:

```
 ---------------------------------
| DLIS-file 1 | ... | DLIS-File n |
 ---------------------------------
```

sequencenr

Sequential position of the logical file in a storage set

Type str

id

Descriptive identification of the logical file

Type str

See also:

BasicObject The basic object that Fileheader is derived from

Notes

The Fileheader object reflects the logical record type FILE-HEADER, defined in rp66. FILE-HEADER records are listed in Appendix A.2 - Logical Record Types and described in Chapter 5.1 - Static and Frame Data, File Header Logical Record (FHLR).

12.10 Frame

class Frame (*BasicObject*)

Frame

A Frame is a logical gathering of multiple Channels (curves), typically Channels from the same run. A Frame containing three Channels would look something like this:

Usually, the first channel in the channel list is considered to be the index-channel of the Frame. See:attr:*index_type* for more information about the index channels.

All Channels belonging to a Frame are directly accessible through `channels`. A full table of all the curve-data can be accessed with `curves`.

description
Textual description of the Frame.

Type str

channels
Channels in the frame

Type list(*Channel*)

index_type
The measurement of the index, e.g. borehole-depth. If **not** None, the first channel is considered to be an index channel. If index_type is None, then the Frame has no index channel and is implicitly indexed by samplenumber i.e. 0, 1, . . . , n.

Type str

direction
Direction of the index (Increasing or decreasing)

Type str

spacing
Constant spacing in the index

index_min
Minimum value of the index

index_max
Maximum value of the index

encrypted
If the frame was encrypted

Type bool

See also:

BasicObject The basic object that Frame is derived from

Channel Channel objects

Notes

The Frame object reflects the logical record type FRAME, defined in rp66. FRAME objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.7.1 - Static and Frame Data, FRAME objects.

curves (*strict=True*)

All curves belonging to this frame

Get all the curves in this frame as a structured numpy array. The frame includes the frame number (FRAMENO), to detect errors such as missing entries and out-of-order frames.

Parameters **strict** (*boolean, optional*) – By default (strict=True) curves() raises a ValueError if there are multiple channel with the same values for both name, origin and copynumber. This would be a clear violation of the dlis-spec. Setting strict=False lifts this restriction and dlisio will append numerical values (i.e. 0, 1, 2 ..) to the labels used for column-names in the returned array.

Returns **curves** – curves with dtype = self.dtype

Return type np.ndarray

Raises `ValueError` – If there multiple channels with identical name, origin, copynumber in Frame.channels. This can be suppressed by passing strict=False

See also:

Channel.curves() Access the curve-data directly through the Channel objects

Frame.dtype() dtype of the array

Examples

The returned array supports both horizontal- and vertical slicing. Do a vertical slice by specifying a single Channel

```
>>> curves = frame.curves()
>>> curves['CHANN1']
array([16677259., 852606., 16678259., 852606.])
```

Access a subset of Channels, note the double-bracket syntax

```
>>> curves[['CHANN2', 'CHANN3']]
array([
    (16677259., 852606.),
    (16678259., 852606.),
    (16679259., 852606.),
    (16680259., 852606.)
])
```

Do a horizontal slice of all Channels, i.e. read a subset of samples from all channels

```
>>> curves[0:2]
array([
    (16677259., 852606., 2233., 852606.),
    (16678259., 852606., 2237., 852606.)])
```

Horizontal and vertical slicing can be combined

```
>>> curves['CHANN2'][0]
16677259.0
```

And here the subscription order is irrelevant

```
>>> curves[0]['CHANN2']
16677259.0
```

Some curves, like image curves, have multi-dimensional samples. Accessing a single sample from a 2-dimensional curve

```
>>> curves = frame.curves()
>>> sample = curves['MULTI_D_CHANNEL'][0]
>>> sample
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11,  12]])
```

This sample is a 2-dimensional array of size 4x3. We can continue to slice this sample. Note that now the subscription order here **does** matter. Here we read the two last rows

```
>>> sample[-2:]
array([[  7,  8,  9],
       [ 10, 11, 12]])
```

Lets read every second column

```
>>> sample[:,::2]
array([[ 1, 3],
       [ 4, 6],
       [ 7, 9],
       [10, 12]])
```

Note the syntax. Within the brackets, everything before the ',' is row- operations and everything after are column-operations. Read it as: keep all rows (:) and then from first to last column, keep every 2nd column (::2). The comma syntax for indexing different axes extends to array's with higher orders as well.

Combine the two to read a specific element

```
>>> sample[0,0]
1
```

If you prefer to work with pandas over numpy, the conversion is trivial

```
>>> import pandas as pd
>>> curves = pd.DataFrame(frame.curves())
```

Note that pandas (and CSV) *only* supports scalar sample values. I.e. frames containing one or more channels that have none-scalar sample values cannot be converted to pandas.DataFrame or CSV directly.

If the Frame contains an index Channel, use that as index in the DataFrame

```
>>> curves = frame.curves()
>>> pdcurves = pd.DataFrame(curves, index=curves[frame.index])
>>> pdcurves.index.name = frame.index
```

By default the returned np.ndarray have column-names that reflects the mnemonics of each Channel

```
>>> frame.channels
[Channel(TDEP), Channel(TIME), Channel(GR)]
```

```
>>> curves = frame.curves()
>>> curves.dtype.names
('FRAMENO', 'TDEP', 'TIME', 'GR')
```

Sometimes a Frame can contain multiple Channels with the same name/mnemonic, in that case the labels are augmented with the origin and copynumber:

```
>>> frame.channels
[Channel(TDEP), Channel(TDEP), Channel(GR)]
```

```
>>> curves = frame.curves()
>>> curves.dtype.names
('FRAMENO', 'TDEP.0.0', 'TDEP.0.1', 'GR')
```

This augmentation is customizable by changing `dtype_fmt`. See *dtype*. Some frames have multiple instances of the same channel or mulitple channels where name, origin and copynumber are identical. This is a clear violation of the dlis spec and *curves()* will raise an ValueError by default.

```
>>> frame.channels
[Channel(TDEP), Channel(TDEP), Channel(GR)]
```

```
>>> Frame.curves()
ValueError: field 'TDEP.0.0' occurs more than once
```

However, *curves()* offers an escape-hatch to get the underlying data. By passing strict=False dlisio appends numerical values to the identical channels

```
>>> curves = frame.curves(strict=False)
>>> curves.dtype.names
('FRAMENO', 'TDEP.0.0(0)', 'TDEP.0.0(1)', 'GR')
```

dtype (*strict=True*)

data-type of each frame, i.e. the sum of channel.dtype of each channel in self.channels. The first column is always FRAMENO.

If all curve mnemonics are unique, then dtype.names == ['FRAMENO'] + [ch.name for ch in self.channels]. If there are more than one channel with the same name for this frame, all duplicated mnemonics are enriched with origin and copynumber.

Consider a frame with the channels mnemonics [('TIME', 0, 0), ('TDEP', 0, 0), ('TIME, 1, 0)]: dtype.names for this frame would be ('FRAMENO', 'TIME.0.0', 'TDEP', 'TIME.1.0').

Duplicated mnemonics are formatted by the dtype_fmt attribute. To use a custom format for a specific frame instance, set dtype_fmt for the Frame object. If you want to have some other formatting for *all* dtypes, set the dtype_format class attribute. It has to be a 3-element format-string taking a string and two ints. Custom formatting is particularly useful for peculiar files where the full stop (.) appears in the mnemonic itself, and a consistent way of parsing origin and copynumber are needed.

In addition to the customizable dtype.names, ch.fingerprint is always used as field title, which serves as an alias for the name.

Returns **dtype**

Return type np.dtype

Examples

A frame with two TIME channels:

```
>>> dtype = frame.dtype()
>>> dtype.names
dtype([('FRAMENO', '<i4'),
       (('T.CHANNEL-I.TIME-O.0-C.0','TIME.0.0'), '<f4'),
       (('T.CHANNEL-I.TDEP-O.0-C.0','TDEP'), '<i2'),
       (('T.CHANNEL-I.TIME-O.1-C.0','TIME.1.0'), '<i2')])
```

Override instance-specific mnemonic formatting

```
>>> frame.dtype().names
(FRAMENO', 'TIME.0.0', 'TDEP', 'TIME.1.0')
>>> frame.dtype_fmt = '{:s}-{:d}-{:d}'
>>> frame.dtype()
(FRAMENO','TIME-0-0', 'TDEP','TIME-1-0')
```

index
Mnemonic of the channel all channels in this Frame are indexed against, if any. See *Frame.index_type* for definition of existing index channel.

Returns **mnemonic**

Return type str

12.11 Group

class Group(*BasicObject*)
Group Objects indicate logical groupings of other Objects. A Group is defined by the producer by any associations deemed useful.

description

Type str

object_type
Specifies the type of object that is referenced in the object list attribute.

Type str

object_list
References to arbitrary objects.

Type list

group_list
Reference to other Group objects

Type list(*Group*)

Notes

The Group object reflects the logical record type Group, defined in rp66. Group records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.8 - Static and Frame Data, Group objects.

12.12 Longname

class Longname(*BasicObject*)

Structured names of other objects.

modifier
General modifier

Type list(str)

quantity
Something that is measureable E.g. the diameter of a pipe

Type str

quantity_mod
Specialization of a quantity

Type list(str)

altered_form
Altered form of the quantity. E.g. standard deviation is an altered form of a temperature quantity.

Type str

entity
The entity of which the quantity is measured. E.g. entity = borehole, quantity = diameter

Type str

entity_mod
Specialization of an entity

Type list(str)

entity_nr
Distinguishes multiple instances of the same entity

Type str

entity_part
Part of an entity

Type str

entity_part_nr
Distinguishes multiple instances of the same entity part

Type str

generic_source
The source of the information

Type str

source_part
A specific part of the source information. E.g. “transmitter”

Type list(str)

source_part_nr
Distinguishes multiple instances of the same source part

Type list(str)

conditions
Conditions applicable at the time the information was acquired or generated

Type list(str)

standard_symbol
Industry-standardized symbolic name by which the information is known. The possible values are specified by POSC

Type str

private_symbol
Association between the recorded information and corresponding records or objects of the Producer's internal or corporate database

Type str

See also:

BasicObject The basic object that Longname is derived from

Notes

The Longname object reflects the logical record type LONG-NAME, defined in rp66. LONG-NAME objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.4.1 - Static and Frame Data, Long-Name Objects.

12.13 Measurement

class Measurement(*BasicObject*)
Records of measurements, references, and tolerances used to compute calibration coefficients.

phase
In what phase of the overall job sequence the measurement as acquired

Type str

source
Source the measurement

mtype
Type of measurement

Type str

dimension
Structure of the sample array

Type list(int)

axis
Coordinate axis of the sample array

Type list(*Axis*)

samplecount
Number of samples used to compute the max/std_deviation

Type int

begin_time
Time of the sample acquisition. Either an absolute time represented by datetime or elapsed time from file-creation (see *Origin.creation_time*).

duration
Time duration of the sample acquisition

standard
Measurable quantity of the calibration standard used to produce the sample

See also:

BasicObject The basic object that Measurement derived from

Notes

The Measurement object reflects the logical record type CALIBRATION-MEASUREMENT, defined in rp66. CHANNEL records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.7.1 - Static and Frame Data, CALIBRATION-MEASUREMENT objects.

max_deviation
Maximum deviation

Only applicable when the sample attribute contains mean values. In that case, this is maximum deviation from the mean of any value used to compute the mean.

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

minus_tolerance
The maximum value that any sample (in samples) can fall below the reference and still be 'within tolerance'. Should be all non-negative numbers. If this attribute is empty, the minus tolerance is implicity infinite.

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

plus_tolerance
The maximum value that any sample (in samples) can exceed the reference and still be 'within tolerance'. Should be all non-negative numbers. If this attribute is empty, the plus tolerance is implicity infinite.

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

reference
The nominal value of each sample in the samples attribute

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

samples
Measurment samples

The type of measurment is described by the type attribute. Each sample may be either a scalar or ndarray

std_deviation
Standard deviation

Only applicable when the sample attribute contains mean values. In that case, this is the standard deviation of the samples used to compute the mean.

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

12.14 Message

class Message(*BasicObject*)

Textual messages tied to other data by means of time and/or position of tool zero point when the message was recorded

message_type
source and purpose of the message.

Type str

time
time the message was issued. Either an absolute time represented by datetime or elapsed time from file-creation (see *Origin.creation_time*).

borehole_drift
borehole drift of the tool zero point when message was issued.

vertical_depth
vertical depth of the tool zero point when message was issued.

radial_drift
radial drift of the tool zero point when message was issued.

angular_drift
angular drift of the tool zero point when message was issued.

text
message(s).

Type list(str)

See also:

BasicObject The basic object that Message is derived from

Notes

The Message object reflects the logical record type MESSAGE, described in rp66. MESSAGE objects are defined in Appendix A.2 - Logical Record Types, described in detail in Chapter 6.1.1 - Transient Data, message objects.

12.15 Origin

class Origin(*BasicObject*)

Describes the creation of the logical file.

Origin objects is an unique indentifier for a Logical File and it describes the circumstances under which the file was created. The Origin object also spesify the Logical File's relation to a DLIS-file and to which Logical Set it belongs.

A logical file may have several Origin objects, whereas the first Origin object is the Defining object. No two logical files should have identical Defining Origins.

file_id
An exact copy of Fileheader.id

Type str

file_set_name
The name of the File Set that the Logical File is a part of

Type str

file_set_nr
The number of the File Set that the Logical File is a part of

Type int

file_nr
The file number of the Logical File within a File Set

Type int

file_type
A producer spesified File-Type that signifies the content of the DLIS-file

Type str

product
Name of the software product that produced the DLIS-file

Type str

version
The version of the software product that created the DLIS-file

Type str

programs
Other programs and services that was a part of the software that created the DLIS-file

Type list(str)

creation_time
Date and time at which the DLIS-File was created

Type datetime

order_nr
An unique accounting number assosiated with the creation of the DLIS-File

Type str

descent_nr
The meaning of this number must be obtained directly from the producer

run_nr
The meaning of this number must be obtained directly from the company

well_id
Id of the well at which the measurements where acquired

well_name
Name of the well at which the measurements where acquired

Type str

field_name
The field to which the well belongs

Type str

producer_code
The producer's identifying code

Type int

producer_name
The producer's name

Type str

company
The name of the client company which the log was produced for

Type str

namespace_name
(DLIS internal) A producer-defined namespace for which the object names for this origin are defined under

Type str

namespace_version
(DLIS internal) The version of the namespace.

Type int

See also:

BasicObject The basic object that Origin is derived from

Fileheader Fileheader

Notes

The Origin object reflects the logical record type ORIGIN, defined in rp66. ORIGIN records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.1 - Static and Frame Data, Origin objects.

12.16 Parameter

class Parameter(*BasicObject*)
Parameter

A parameter object describes a parameter used in the acquisition and processing of data. The parameter value(s) may be scalars or an array. In the later case, the structure of the array is defined in the dimension attribute. The zones attribute specifies which zones the parameter is defined. If there are no zones the parameter is defined everywhere.

The axis attribute, if present, defines axis labels for multidimensional value(s).

long_name
Descriptive name of the channel.

Type *Longname*

dimension
Dimensions of the parameter values

Type list(int)

axis
Coordinate axes of the parameter values

Type list(*Axis*)

zones
Mutually disjoint intervals where the parameter values is constant

Type list(*Zone*)

See also:

BasicObject The basic object that Parameter is derived from

Notes

The Parameter object reflects the logical record type PARAMETER, described in rp66. PARAMETER objects are defined in Appendix A.2 - Logical Record Types, described in detail in Chapter 5.8.2 - Static and Frame Data, PARAMETER objects.

values
Parameter values

Parameter value(s) may be scalar or array's. The size/dimensionallity of each value is defined in the dimensions attribute.

Each value may or may not be zoned, i.e. it is only defined in a certain zone. If this is the case the first zone, parameter.zones[0], will correspond to the first value, parameter.values[0] and so on. If there is no zones, there should only be one value, which is said to be unzoned, i.e. it is defined everywere.

Raises `ValueError` – Unable to structure the values based on the information available.

Returns **values**

Return type structured np.ndarray

Notes

If dlisio is unable to structure the values due to insufficient or contradictory information in the object, an ValueError is raised. The raw array can still be accessed through attic, but note that in this case, the semantic meaning of the array is undefined.

Examples

First value:

```
>>> parameter.values[0]
[10, 20, 30]
```

Zone (if any) where that parameter value is valid:

```
>>> parameter.zones[0]
Zone('ZONE-A')
```

12.17 Path

class Path (*BasicObject*)

Path Objects defines Channels in the Data Frames of a given Frame Type and are combined to define part or all of a Data Path, and variation in the alignment.

frame_type
The frame in which the channel's of the current path are recorded.

Type *Frame*

well_reference_point
Well Reference Point

Type *Wellref*

value
Value Channel for the current Path.

Type list(*Channel*)

borehole_depth
Specifies the constant Borehole Depth coordinate for the current Path, It may or may not be described by a channel object.

vertical_depth
Specifies the constant Vertical Depth coordinate for the current Path, It may or may not be described by a channel object.

radial_drift
Specifies the constant Radial Drift coordinate for the current Path, It may or may not be described by a channel object.

angular_drift
Specifies the constant Angular Drift coordinate for the current Path, It may or may not be described by a channel object.

time
Specifies the constant Time coordinate for the current Path, It may or may not be described by a channel object. Either an absolute time represented by datetime or elapsed time from file-creation (see *Origin.creation_time*).

depth_offset
Specifies the Depth Offset, which indicates how much the *value* is "off depth".

measure_point_offset
Specifies a Measure Point Offset, which indicates a fixed distance along Borehole Depth from the Value Channel's Measure Point to a Data Reference Point.

tool_zero_offset
Distance of the Data Reference Point for the current Path above the tool string's Tool Zero Point.

See also:

BasicObject The basic object that Parameter is derived from

Notes

The Path object reflects the logical record type PATH, defined in rp66. PATH records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.7.2 - Static and Frame Data, PATH objects.

12.18 Process

class Process(*BasicObject*)

Process objects describes a specific process or computation applied to input objects to get output objects.

description

Type str

trademark_name

Trademark name refers to the process and its products.

Type str

version

Software version.

Type str

properties

Properties that applies to the output of the process, as a result of the process.

Type list(str)

status

Indicated the status of the process. It's typically updated to indicate when the process is completed or aborted.

Type str

input_channels

Channels that are used directly by this Process.

Type list(*Channel*)

output_channels

Channels that are produced directly by this Process.

Type list(*Channel*)

input_computations

Computations that are used directly by this Process.

Type list(*Computation*)

output_computations

Computations that are produced directly by this Process.

Type list(*Computation*)

parameters

Parameters that are used by the Process or that directly affect the operation of the Process.

Type list(*Parameter*)

comments

Comments contains information specific to the particular execution of the process.

Type list(str)

See also:

BasicObject The basic object that Parameter is derived from

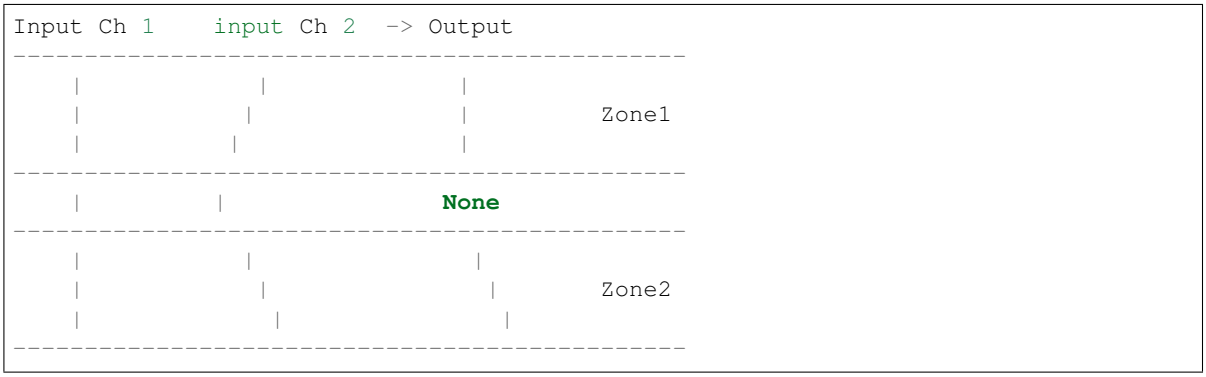
Notes

The Process object reflects the logical record type Process, defined in rp66. PROCESS records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.5 - Static and Frame Data, Process objects.

12.19 Splice

class Splice(*BasicObject*)

Splice describes the process of concatinating multiple channels into one. The concatination is defined by the zone objects, where the first zone object corresponds to the first input channel and so on. The zones must be mutually disjoint but the ordering is arbitrary:



output_channel

Concatination of all input channels

Type *Channel*

input_channels

Channels that where used to create the output channel

Type list(*Channel*)

Zones

Zones of each input channel that is used in the concatination process

Type list(*Zone*)

See also:

BasicObject The basic object that Splice is derived from

Notes

The Splice object reflects the logical record type SPLICE, defined in rp66. SPLICE records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.9 - Static and Frame Data, SPLICE objects.

12.20 Tool

class Tool(*BasicObject*)

A tool is an ensembly of equiptment that as a whole provide measurements or services. The list of equiptment that makes up the tool can be found in *tool.parts*. Tools objects also keep a list of all channels that where produced by the tool in *tool.channels*.

description
Textual description of the tool

Type str

trademark_name
The producer's name for the tool

Type str

generic_name
The name generally used by the industry to describe such a tool

Type str

status
If the tool is enabled to provide information to the acquisition system

Type bool

parts
Equipments that makes up the tool

Type list(*Equipment*)

channels
Channels that are produced by this tool

Type list(*Channel*)

parameters
Parameters that directly affect or reflect the operation of the tool

Type list(*Parameter*)

See also:

BasicObject The basic object that Tool is derived from

Notes

The tool object reflects the logical record type TOOL defined in rp66. TOOL objects are listed in Appendix A.2 - Logical Record Types, described in detail in Chapter 5.8.4 - Static and Frame Data, TOOL objects.

12.21 Wellref

class Wellref(*BasicObject*)

Well reference defines origin of well with coordinates.

permanent_datum
Level from where vertical distance is measured

Type str

vertical_zero
Vertical zero is an entity that corresponds to zero depth.

Type str

permanent_datum_elevation
Permanent datum, structure or entity from which the vertical distance can be measured.

above_permanent_datum
Distance of permanent Datum above mean sea level. Negative values indicates that the Permanent datum is below mean sea level

magnetic_declination
Angle between the line of direction to geographic north and the line of direction to magnetic north. This defines angle with vertex at well reference point.

coordinate
Independent spatial coordinates. Typically, latitude, longitude and elevation

Type dict

See also:

BasicObject The basic object that Wellref is derived from

Notes

The Well Reference object reflects the well reference point of a well, defined in rp66. Well reference records are listed in Appendix A.2 - Logical Record Types are described in detail in Chapter 5.2.2 - Static and Frame Data, Well reference objects.

12.22 Zone

class Zone(*BasicObject*)
A Zone objects specifies a single interval in depth or time. Other objects use zones to define spesific regions in wells or time intervals where the object data is valid.

description
Description of the zone

Type str

domain
Type of interval, e.g. borhole-depth, time or vertical-depth

Type str

maximum
Latest time or deepest point, not inclusive

minimum
Earliest time or shallowest point, inclusive

See also:

BasicObject The basic object that Zone is derived from

Notes

The Zone object reflects the logical record type ZONE, defined in rp66. ZONE objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.8.1 - Static and Frame Data, Zone Objects.

12.23 Unknown

class Unknown(*BasicObject*)

The unknown object is intended as a fall-back object if the object-type is not recognized by dlisio, e.g. vendor spesific object types

See also:

BasicObject The basic object that Unknown is derived from

CHAPTER 13

A quick guide

This is a quick guide to get you started with dlisio. Note that all classes and functions are more thoroughly documented under *API Reference*. Please refer there for more information about them.

The same documentation is also available directly in your favorite python interpreter and in the unix console, just type `help(function)` or `pydoc function`, respectably. In the interpreter, help can be used directly on class instances. E.g: `help(frame)` or `help(frame.curves)`

13.1 Opening files

Load all *Logical files*:

```
>>> with dlisio.load(filename) as files:
...     for f in files:
...         pass
```

The returned `files` can be iterated over and operations can be applied to each logical file.

If you only want to work with one logical file at the time, *`dlisio.load()`* supports automatic unpacking of logical files. The following syntax unpacks the first logical file into `f` and stores the rest (0-n) logical files into `*tail`.

```
>>> with dlisio.load(filename) as (f, *tail):
...     pass
```

Or, if the number of logical files is known:

```
>>> with dlisio.load(filename) as (f1, f2, f3):
...     pass
```

When a file is loaded, you can output some basic information about the physical file:

```
>>> import dlisio
>>> with dlisio.load(filename) as files:
```

(continues on next page)

(continued from previous page)

```
...     files.describe()
----------------------
Batch of Logical Files
----------------------
Number of Logical Files : 3

Description : dlis(DDBC1)
Frames      : 0
Channels    : 0

Description : dlis(DDBC2)
Frames      : 2
Channels    : 22

Description : dlis(DDBC3)
Frames      : 2
Channels    : 160
```

Or about a logical file:

```
>>> import dlisio
>>> with dlisio.load(filename) as (f, *tail):
...     f.describe()
------------
Logical File
------------
Description  : dlis(MSCT_200LTP)
Frames       : 2
Channels     : 104

Known objects
--
FILE-HEADER             : 1
ORIGIN                  : 3
CALIBRATION-COEFFICIENT : 8
CHANNEL                 : 104
FRAME                   : 2

Unknown objects
--
440-CHANNEL             : 93
440-OP-CORE_TABLES      : 17
440-OP-CHANNEL          : 101
```

13.2 Accessing objects

Think of *Logical files* as pools of objects with different types. All objects of a type can be reached by name, e.g. channels or coefficients:

```
>>> for ch in f.channels:
...     pass
```

See *Logical files* for a full list of all object types.

`dlisio.dlis.object()` lets you access a specific object:

```
>>> obj = f.object('CHANNEL', 'TDEP')
```

Objects can also be searched for with *dlisio.dlis.match()*:

```
>>> objs = f.match('T.*')
```

Inspect an object with the `.describe()`-method:

```
>>> obj.describe()
-----
Frame
-----
name   : 800T
origin : 2
copy   : 0

Channel indexing
--
Indexed by       : TIME
Interval         : [33354518, 35194520]
Direction        : INCREASING
Constant spacing : 800
Index channel    : Channel(TIME)

Channels
--
TIME TDEP ETIM LMVL UMVL CFLA OCD  RCMD RCPP CMRT
RCNU DCFL DFS  DZER RHMD HMRT RHV  RLSW MNU  S1CY
S2CY RSCU RSTS UCFL CARC CMDV CMPP CNU  HMDV HV
LSWI SCUR SSTA RCMP RHPP RRPP CMPR HPPR RPPV SMSC
CMCU HMCU CMLP
```

13.3 Frames and Channels

See *Curves* for information about the relationship between Channels and Frames. Have a look at *Channel* and *Frame*, they contain some useful metadata in addition to the curve-values!

Channels belonging to a Frame can be accessed directly through *dlisio.plumbing.Frame.channels*:

```
>>> frame.channels[0]
Channel(TDEP)
```

Likewise, the parent-frame of a Channel can be accessed through the channel:

```
>>> ch.frame
Frame(800T)
```

The actual curve data of a Channel is accessed by *dlisio.plumbing.Channel.curves()*, which returns a structured numpy array that support common slicing operations:

```
>>> curve = ch.curves()
>>> curve[0:5]
array([852606., 852606., 852606., 852606., 852606.], dtype=float32)
```

Note that its almost always considerably faster to read curves-data with *dlisio.plumbing.Frame.curves()*. Please refer to *dlisio.plumbing.Channel.curves()* for further elaboration on why this is.

Access all curves in a frame with *dlisio.plumbing.Frame.curves()*. The returned structured numpy array can be indexed by Channel mnemonics and/or sliced by samples:

```
>>> curves = fr.curves()
>>> curves[[fr.index, 'TENS_SL']][0:5]
array([(16677259., 2233.), (16678259., 2237.), (16679259., 2211.),
       (16680259., 2193.), (16681259., 2213.)])
```

Note that double brackets are needed in order to access muliple channels at once.

As long as the frame only contains channels with scalar samples, it can be trivially converted to a pandas DataFrame:

```
>>> import pandas as pd
>>> curves = pd.DataFrame(frame.curves())
```

For more examples of how to work with the curve-data, please refer to *dlisio.plumbing.Frame.curves()* and *dlisio.plumbing.Channel.curves()*

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Index

Q

R

S

T

U

V

W

Z