datalad$_{m}etaladDocumentation$

DataLad team

Jan 12, 2022

Contents

This software is a DataLad extension that equips DataLad with an alternative command suite for metadata handling (extraction, aggregation, reporting). It is backward-compatible with the metadata storage format in DataLad proper, while being substantially more performant (especially on large dataset hierarchies). Additionally, it provides new metadata extractors and improved variants of DataLad's own ones that are tuned for better performance and richer, JSON-LD compliant metadata reports.

CHAPTER 1

API

1.1 High-level API commands

These commands provide and improved and extended equivalent to the *metadata* and *aggregate_metadata* commands (and the primitive *extract-metadata* plugin) that ship with the DataLad core package.

meta_extract(extractorname, path, dataset, ...)	Run a metadata extractor on a dataset or file.
meta_aggregate([dataset, path])	Aggregate metadata of one or more sub-datasets for later reporting.
meta_dump([dataset, path, recursive])	Dump a dataset's aggregated metadata for dataset and file metadata

1.1.1 datalad.api.meta_extract

datalad.api.**meta_extract**(*extractorname: str, path: Optional[str] = None, dataset: Union[datalad.distribution.dataset.Dataset, str, None] = None, context: Union[str, Dict[str, str], None] = None, get_context: bool = False, extractorargs: Optional[List[str]] = None*)

Run a metadata extractor on a dataset or file.

This command distinguishes between dataset-level extraction and file-level extraction.

If no "path" argument is given, the command assumes that a given extractor is a dataset-level extractor and executes it on the dataset that is given by the current working directory or by the "-d" argument.

If a path is given, the command assumes that the path identifies a file and that the given extractor is a file-level extractor, which will then be executed on the specified file. If the file level extractor requests the content of a file that is not present, the command might "get" the file content to make it locally available. Path must not refer to a sub-dataset. Path must not be a directory.

Note: If you want to insert sub-dataset-metadata into the super-dataset's metadata, you currently have to do the following: first, extract dataset metadata of the sub-dataset using a dataset- level extractor, second add the

extracted metadata with sub-dataset information (i.e. dataset_path, root_dataset_id, root-dataset- version) to the metadata of the super-dataset.

The extractor configuration can be parameterized with key-value pairs given as additional arguments. Each key-value pair consists of two arguments, first the key, followed by the value. If no path is given, and you want to provide key-value pairs, you have to give the path "++", to prevent that the first key is interpreted as path.

The results are written into the repository of the source dataset or into the repository of the dataset given by the "-i" parameter. If the same extractor is executed on the same element (dataset or file) with the same configuration, any existing results will be overwritten.

The command can also take legacy datalad-metalad extractors and will execute them in either "content" or "dataset" mode, depending on the presence of the "path"-parameter.

Examples

Parameters

- **extractorname** – Name of a metadata extractor to be executed.
- **path** (*str or None, optional*) – Path of a file or dataset to extract metadata from. If this argument is provided, we assume a file extractor is requested, if the path is not given, or if it identifies the root of a dataset, i.e. "", we assume a dataset level metadata extractor is specified. [Default: None]
- **dataset** (*Dataset or None, optional*) – Dataset to extract metadata from. If no dataset is given, the dataset is determined by the current work directory. [Default: None]
- **context** (*Dataset or None, optional*) – Context, a JSON-serialized dictionary that provides constant data which has been gathered before, so meta-extract will not have re- gather this data. Keys and values are strings. meta-extract will look for the following key: 'dataset_version'. [Default: None]
- **get_context** (*bool, optional*) – Show the context that meta-extract determines with the given parameters and exit. The context can be used in subsequent calls to meta-extract with identical parameter, except from –get-context, to speed up the execution of meta-extract. [Default: False]
- **extractorargs** (*sequence of str or None, optional*) – Extractor arguments given as string arguments to the extractor. If dataset level extraction is performed, i.e. no path is required, specify '–' as path to prevent interpretation of the first extractor argument as path. [Default: None]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) – behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']
- **result_filter** (*callable or None, optional*) – if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]

- **result_renderer** (*{'default', 'json', 'json_pp', 'tailored'} or None, optional*) – format of return value rendering on stdout. [Default: 'tailored']
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) – if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) – return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

1.1.2 datalad.api.meta_aggregate

datalad.api.**meta_aggregate**(*dataset=None*, *path=None*)

Aggregate metadata of one or more sub-datasets for later reporting.

Note: Metadata storage is not forced to reside inside the datalad repository of the dataset. Metadata might be stored within the repository that is used by a dataset, but it might as well be stored in another repository (or a non-git backend, once those exist). To distinguish metadata storage from the dataset storage, we refer to metadata storage as metadata-store. For now, the metadata-store is usually the git-repository that holds the dataset.

Note: The distinction is the reason for the "double"-path arguments below. for each source metadata-store that should be integrated into the root metadata-store, we have to give the source metadata-store itself and the intra-dataset-path with regard to the root-dataset.

Metadata aggregation refers to a procedure that combines metadata from different sub-datasets into a root dataset, i.e. a dataset that contains all the sub-datasets. Aggregated metadata is "prefixed" with the intra-dataset-paths of the sub-datasets. The intra-dataset-path for a sub-dataset is the path from the top-level directory of the root dataset, i.e. the directory that contains the ".datalad"-entry, to the top-level directory of the respective sub-dataset.

Aggregate works on existing metadata, it will not extract meta data from data file. To create metadata, use the meta-extract command.

As a result of the aggregation, the metadata of all specified sub-datasets will be available in the root metadata-store. A datalad meta-dump command on the root metadata-store will therefore be able to process metadata from the root dataset, as well as all aggregated sub-datasets.

Examples

Parameters

- **dataset** (*Dataset or None, optional*) – Topmost dataset metadata will be aggregated into. If no dataset is specified, a dataset will be discovered based on the current

working directory. Metadata for aggregated datasets will contain a dataset path that is relative to the top-dataset. [Default: None]

- **path** (*sequence of str or None, optional*) – PATH to a sub-dataset whose metadata shall be aggregated into the topmost dataset (ROOT_DATASET). [Default: None]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) – behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']
- **result_filter** (*callable or None, optional*) – if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** (*{'default', 'json', 'json_pp', 'tailored'} or None, optional*) – format of return value rendering on stdout. [Default: None]
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) – if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) – return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

1.1.3 datalad.api.meta_dump

datalad.api.**meta_dump**(*dataset=None*, *path=''*, *recursive=False*)

Dump a dataset's aggregated metadata for dataset and file metadata

Two types of metadata are supported:

1. metadata describing a dataset as a whole (dataset-global metadata), and
2. metadata for files in a dataset (content metadata).

The DATASET_FILE_PATH_PATTERN argument specifies dataset and file patterns that are matched against the dataset and file information in the metadata. There are two format, UUID-based and dataset-tree based. The formats are:

> TREE: ["tree:"] [DATASET_PATH] ["@" VERSION-DIGITS] [":" [LOCAL_PATH]] UUID: "uuid:" UUID-DIGITS ["@" VERSION-DIGITS] [":" [LOCAL_PATH]]

(The tree-format is the default format and does not require a prefix).

Examples

Parameters

- **dataset** – Dataset for which metadata should be dumped. If no directory name is provided, the current working directory is used. [Default: None]
- **path** (*str or None, optional*) – path to query metadata for. [Default: ‘’]
- **recursive** (*bool, optional*) – If set, recursively report on any matching metadata based on given paths or reference dataset. Note, setting this option does not cause any recursion into potential subdatasets on the filesystem. It merely determines what metadata is being reported from the given/discovered reference dataset. [Default: False]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) – behavior to perform on failure: ‘ignore’ any failure is reported, but does not cause an exception; ‘continue’ if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; ‘stop’: processing will stop on first failure and an exception is raised. A failure is any result with status ‘impossible’ or ‘error’. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: ‘continue’]
- **result_filter** (*callable or None, optional*) – if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable’s return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** (*{'default', 'json', 'json_pp', 'tailored'} or None, optional*) – format of return value rendering on stdout. [Default: ‘tailored’]
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) – if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) – return value behavior switch. If ‘item-or-list’ a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: ‘list’]

1.2 Metadata extractors

To use any of the contained extractors their names needs to be prefixed with *metalad_*, such that the *runprov* extractor is effectively named *metalad_runprov*.

core	Metadata extractor for Datalad’s own core storage
annex	Metadata extractor for Git-annex metadata
custom	Metadata extractor for custom (JSON-LD) metadata contained in a dataset

Continued on next page

Table 2 – continued from previous page

runprov	Metadata extractor for provenance information in DataLad's *run* records

1.2.1 datalad_metalad.extractors.core

Metadata extractor for Datalad's own core storage

class datalad_metalad.extractors.core.**DataladCoreExtractor**
Bases: datalad_metalad.extractors.base.MetadataExtractor

get_state(*dataset*)
Report on extractor-related state and configuration

Extractors can reimplement this method to report arbitrary information in a dictionary. This information will be included in the metadata aggregate catalog in each dataset. Consequently, this information should be brief/compact and limited to essential facts on a comprehensive state of an extractor that "fully" determines its behavior. Only plain key-value items, with simple values, such a string int, float, or lists thereof, are supported.

Any change in the reported state in comparison to a recorded state for an existing metadata aggregate will cause a re-extraction of metadata. The nature of the state change does not matter, as the entire dictionary will be compared. Primarily, this is useful for reporting per-extractor version information (such as a version for the extractor output format, or critical version information on external software components employed by the extractor), and potential configuration settings that determine the behavior of on extractor.

State information can be dataset-specific. The respective Dataset object instance is passed via the method's *dataset* argument.

datalad_metalad.extractors.core.**ri2url**(*ri*)

datalad_metalad.extractors.core.**whereis_file**(*self*, *path*)
Same as *whereis_file_()*, but for a single path and return-dict

datalad_metalad.extractors.core.**whereis_file_**(*self*, *paths*)

Parameters **paths** (*iterable*) – Paths of files to query for, either absolute paths matching the repository root (self.path), or paths relative to the root of the repository

Yields *dict* – A response dictionary to each query path with the following keys: 'path' with the queried path in the same form t was provided; 'status' {ok|error} indicating whether git annex was queried successfully for a path; 'key' with the annex key for the file; 'remotes' with a dictionary of remotes that have a copy of the respective file (annex UUIDs are keys, and values are dictionaries with keys: 'description', 'here', 'urls' (list) that contain the values of the respective 'git annex whereis' response.

1.2.2 datalad_metalad.extractors.annex

Metadata extractor for Git-annex metadata

This extractor only deals with the metadata that can be assigned to annexed files via git-annex's *metadata* command. It does not deal with other implicit git-annex metadata, such as file availability information. This is already handled by the *metalad_core* extractor.

There is no standard way to define a vocabulary that is used for this kind of metadata.

class datalad_metalad.extractors.annex.**AnnexMetadataExtractor**
Bases: datalad_metalad.extractors.base.MetadataExtractor

get_state(*dataset*)
Report on extractor-related state and configuration

Extractors can reimplement this method to report arbitrary information in a dictionary. This information will be included in the metadata aggregate catalog in each dataset. Consequently, this information should be brief/compact and limited to essential facts on a comprehensive state of an extractor that "fully" determines its behavior. Only plain key-value items, with simple values, such a string int, float, or lists thereof, are supported.

Any change in the reported state in comparison to a recorded state for an existing metadata aggregate will cause a re-extraction of metadata. The nature of the state change does not matter, as the entire dictionary will be compared. Primarily, this is useful for reporting per-extractor version information (such as a version for the extractor output format, or critical version information on external software components employed by the extractor), and potential configuration settings that determine the behavior of on extractor.

State information can be dataset-specific. The respective Dataset object instance is passed via the method's *dataset* argument.

1.2.3 datalad_metalad.extractors.custom

Metadata extractor for custom (JSON-LD) metadata contained in a dataset

One or more source files with metadata can be specified via the 'datalad.metadata.custom-dataset-source' configuration variable. The content of these files must be a JSON object, and a metadata dictionary is built by updating it with the content of the JSON objects in the order in which they are given.

By default a single file is read: '.metadata/dataset.json'

class datalad_metalad.extractors.custom.**CustomMetadataExtractor**
Bases: datalad_metalad.extractors.base.MetadataExtractor

get_required_content(*dataset*, *process_type*, *status*)
Report records for dataset content that must be available locally

Any implementation can yield records in the given *status* that correspond to dataset content that must be available locally for an extractor to perform its work. It is acceptable to not yield such a record, or no records at all. In such case, the extractor is expected to handle the case of non-available content in some sensible way internally.

The parameters are identical to those of *MetadataExtractor.__call__()*.

Any content corresponding to a yielded record will be obtained automatically before metadata extraction is initiated. Hence any extractor reporting accurately can expect all relevant content to be present locally.

Instead of a status record, it is also possible to return custom dictionaries that must contain a 'path' key, containing the absolute path to the required file within the given dataset.

Example implementation:

```
for s in status:
    if s['path'].endswith('.pdf'):
        yield s
```

get_state(*dataset*)
Report on extractor-related state and configuration

Extractors can reimplement this method to report arbitrary information in a dictionary. This information will be included in the metadata aggregate catalog in each dataset. Consequently, this information should

be brief/compact and limited to essential facts on a comprehensive state of an extractor that "fully" determines its behavior. Only plain key-value items, with simple values, such a string int, float, or lists thereof, are supported.

Any change in the reported state in comparison to a recorded state for an existing metadata aggregate will cause a re-extraction of metadata. The nature of the state change does not matter, as the entire dictionary will be compared. Primarily, this is useful for reporting per-extractor version information (such as a version for the extractor output format, or critical version information on external software components employed by the extractor), and potential configuration settings that determine the behavior of on extractor.

State information can be dataset-specific. The respective Dataset object instance is passed via the method's *dataset* argument.

1.2.4 datalad_metalad.extractors.runprov

Metadata extractor for provenance information in DataLad's *run* records

Concept

- Find all the commits with a run-record encoded in them
- the commit SHA provides @id for the "activity"
- pull out the author/date info for annotation purposes
- pull out the run record (at the very least to report it straight up, but there can be more analysis of the input/output specs in the context of the repo state at that point)
- pull out the diff: this gives us the filenames and shasums of everything touched by the "activity". This info can then be used to look up which file was created by which activity and report that in the content metadata

Here is a sketch of the reported metadata structure:

```
{
  "@context": "http://openprovenance.org/prov.jsonld",
  "@graph": [
    # agents
    {
      "@id": "Name_Surname<email@example.com>",
      "@type": "agent"
    },
    ...
    # activities
    {
      "@id": "<GITSHA_of_run_record>",
      "@type": "activity",
      "atTime": "2019-05-01T12:10:55+02:00",
      "rdfs:comment": "[DATALAD RUNCMD] rm test.png",
      "prov:wasAssociatedWith": {
        "@id": "Name_Surname<email@example.com>",
      }
    },
    ...
    # entities
    {
      "@id": "SOMEKEY",
      "@type": "entity",
```

(continues on next page)

(continued from previous page)

```
      "prov:wasGeneratedBy": {"@id": "<GITSHA_of_run_record>"}
    }
    ...
  ]
}
```

class datalad_metalad.extractors.runprov.**RunProvenanceExtractor**

Bases: datalad_metalad.extractors.base.MetadataExtractor

datalad_metalad.extractors.runprov.**yield_run_records**(*ds*)

CHAPTER 2

Acknowledgments

DataLad development is being performed as part of a US-German collaboration in computational neuroscience (CRCNS) project "DataGit: converging catalogues, warehouses, and deployment logistics into a federated 'data distribution'" (Halchenko/Hanke), co-funded by the US National Science Foundation (NSF 1429999) and the German Federal Ministry of Education and Research (BMBF 01GQ1411). Additional support is provided by the German federal state of Saxony-Anhalt and the European Regional Development Fund (ERDF), Project: Center for Behavioral Brain Sciences, Imaging Platform

DataLad is built atop the git-annex software that is being developed and maintained by Joey Hess.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index