datalad$_{m}etaladDocumentation$

Release 0.4.17

DataLad team

May 16, 2023

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

Commands and 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.

1.1.1 Commandline reference

datalad meta-add

Synopsis

```
datalad meta-add [-h] [-d DATASET] [-o] [-u] [-i] [--json-lines] [-b] [--version]␣
↪METADATA [ADDITIONAL_VALUES]
```

Description

Add metadata to a dataset.

This command reads metadata from a source and adds this metadata to a dataset. The source can either be a file, or standard input. The metadata format is a string with the JSON-serialized dictionary, or list of dictionaries (or individual dictionaries in JSON-Lines format) that describes the metadata.

In case of an API-call metadata can also be provided in a python dictionary or a list of dictionaries.

If metadata is read from a source, additional parameter can overwrite or amend information that is provided by the source.

The ADDITIONAL_VALUES arguments can be pre-fixed by '@', in which case the pre-fixed argument is interpreted as a file-name and the argument value is read from the file.

The metadata key "dataset-id" must be identical to the ID of the dataset that receives the metadata, unless -i or –allow-id-mismatch is provided.

Examples

Add metadata stored in the file "metadata-123.json" to the dataset in the current directory.:

```
% datalad meta-add metadata-123.json
```

Add metadata stored in the JSON lines file "metadata-entries.jsonl" to the dataset in the current directory.:

```
% datalad meta-add --json-lines metadata-123.json
```

Add metadata stored in the file "metadata-123.json" to the dataset "/home/user/dataset_0":

```
% datalad meta-add -d /home/user/dataset_0 metadata-123.json
```

Add metadata stored in the file "metadata-123.json" to the dataset in the current directory and overwrite the "dataset_id" value provided in "metadata-123.json":

```
% datalad meta-add -d /home/user/dataset_0 -i metadata-123.json '{"dataset_id":
↪"00010203-1011-2021-3031-404142434445"}'
```

Add metadata read from standard input to the dataset in the current directory:

```
% datalad meta-add -
```

Add metadata stored in the file "metadata-123.json" to the dataset in the current directory and overwrite the valuesfrom "metadata-123.json" with the values stored in "extra-info.json":

```
% datalad meta-add metadata-123.json @extra-info.json
```

Options

METADATA

path of the file that contains the metadata that should be added to the metadata store (metadata records must be provided as a JSON-serialized metadata dictionary). The file may contain a single metadata-record or a JSON-array with multiple metadata-records. If –json-lines is given, the file may also contain one dictionary per line. If the path is "-", the metadata file is read from standard input. The dictionary must contain the following keys: 'type' 'extractor_name' 'extractor_version' 'extraction_parameter' 'extraction_time' 'agent_name' 'agent_email' 'dataset_id' 'dataset_version' 'extracted_metadata' If the metadata is associated with a file, the following key indicates the file path: 'path' If the metadata should refer to a sub-dataset element (that means if an "aggregated" record should be stored (see meta-aggregate for more info), the following key indicates the path of the sub-dataset from the root of the "containing dataset": 'dataset_path' The containing dataset, aka. the "root dataset", is the dataset version, specified by dataset-id and version that contains the given sub-dataset version at the path 'dataset_path'. If the containing dataset, aka. the "root dataset", is known, it can be specified by providing the following keys: 'root_dataset_id' 'root_dataset_version' If the version of the root dataset that contains the given subdataset at the given path is not known, the "root_dataset_*"-keys can be omitted. Such a situation might arise, if the sub-dataset metadata was extracted in an old version of the sub- dataset, and the relation of this old version to the root-dataset is not known (we assume that the root-dataset would be the dataset to which the metadata is added.) In this case the metadata is added with an "anonymous" root dataset, but with the given sub-dataset-path. (This makes sense, if the sub-dataset at the given path contains a version that is defined in the metadata, i.e. in dataset_version. The metadata will then be added to all versions that meet this condition.). Constraints: value must be a string

ADDITIONAL_VALUES

A string that contains a JSON serialized dictionary of key value-pairs. These key values-pairs are used in addition to the key value pairs in the metadata dictionary to describe the metadata that should be added. If an additional key is already present in the metadata, an error is raised, unless -o, –allow- override is provided. In this case, the additional values will override the value in metadata and a warning is issued. NB! If multiple records are provided in METADATA, the additional values will be applied to all of them. Constraints: value must be a string or value must be NONE

-h, –help, –help-np

show this help message. –help-np forcefully disables the use of a pager for displaying the help message

-d *DATASET*, –dataset *DATASET*

"dataset to which metadata should be added. If not provided, the dataset is assumed to be given by the current directory. Constraints: Value must be a Dataset or a valid identifier of a Dataset (e.g. a path) or value must be NONE

-o, –allow-override

Allow the additional values to override values given in metadata.

-u, –allow-unknown

Allow unknown keys. By default, unknown keys generate an errors. If this switch is True, unknown keys will only be reported. For processing unknown keys will be ignored.

-i, –allow-id-mismatch

Allow insertion of metadata, even if the "dataset-id" in the metadata source does not match the ID of the target dataset.

–json-lines

Interpret metadata input as JSON lines. i.e. expect one metadata record per line. This is the format used by commands like "datalad meta-dump".

-b, –batch-mode

Enable batch mode. In batch mode metadata-records are read from stdin, one record per line, and a result is written to stdout, one result per line. Batch mode can be exited by sending an empty line that just consists of a newline. Meta-add will return an empty line that just consists of a newline to confirm the exit request. When this flag is given, the metadata file name should be set to "-" (minus).

–version

show the module and its version which provides the command

Authors

datalad is developed by The DataLad Team and Contributors <team@datalad.org>.

datalad meta-extract

Synopsis

```
datalad meta-extract [-h] [-d DATASET] [-c CONTEXT] [--get-context] [--force-dataset-
↪level] [--version] EXTRACTOR_NAME [FILE] [EXTRACTOR_ARGUMENTS [EXTRACTOR_ARGUMENTS .
↪..]]
```

Description

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 dataset level extraction should be performed and you want to provide extractor arguments, you have to specify '–force_dataset_level' to ensure dataset-level extraction. i.e. to prevent interpretation of the key of the first extractor argument as path for a file-level extraction.

The command can also take legacy datalad-metalad extractors and will execute them in either "content" or "dataset" mode, depending on the whether file-level- or dataset-level extraction is requested.

Examples

Use the metalad_example_file-extractor to extract metadatafrom the file "subdir/data_file_1.txt". The dataset is given by the current working directory:

```
% datalad meta-extract metalad_example_file subdir/data_file_1.txt
```

Use the metalad_example_file-extractor to extract metadata from the file "subdir/data_file_1.txt" in the dataset /home/datasets/ds0001:

```
% datalad meta-extract -d /home/datasets/ds0001 metalad_example_file subdir/data_file_
↪1.txt
```

Use the metalad_example_dataset-extractor to extract dataset-level metadata from the dataset given by the current working directory:

```
% datalad meta-extract metalad_example_dataset
```

Use the metalad_example_dataset-extractor to extract dataset-level metadata from the dataset in /home/datasets/ds0001:

```
% datalad meta-extract -d /home/datasets/ds0001 metalad_example_dataset
```

Options

EXTRACTOR_NAME

Name of a metadata extractor to be executed.

FILE

Path of a file or dataset to extract metadata from. The path should be relative to the root of the dataset. 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. You might provide an absolute file path, but it has to contain the dataset path as prefix. Constraints: value must be a string or value must be NONE

EXTRACTOR_ARGUMENTS

Extractor arguments given as string arguments to the extractor. The extractor arguments are interpreted as key-value pairs. The first argument is the name of the key, the next argument is the value for that key, and so on. Consequently, there should be an even number of extractor arguments. If dataset level extraction should be performed and you want to provide extractor arguments. you have tp specify '–force-dataset-level' to ensure dataset-level extraction. i.e. to prevent interpretation of the key of the first extractor argument as path for a file-level extraction. Constraints: value must be a string or value must be NONE

-h, –help, –help-np

show this help message. –help-np forcefully disables the use of a pager for displaying the help message

-d *DATASET*, –dataset *DATASET*

Dataset to extract metadata from. If no dataset is given, the dataset is determined by the current work directory. Constraints: Value must be a Dataset or a valid identifier of a Dataset (e.g. a path) or value must be NONE

-c *CONTEXT*, –context *CONTEXT*

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'. Constraints: Value must be a Dataset or a valid identifier of a Dataset (e.g. a path) or value must be NONE

–get-context

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.

–force-dataset-level

–version

show the module and its version which provides the command

Authors

datalad is developed by The DataLad Team and Contributors <team@datalad.org>.

datalad meta-aggregate

Synopsis

```
datalad meta-aggregate [-h] [-d ROOT_DATASET] [--version] SUB_DATASET_PATH [SUB_
↪DATASET_PATH ...]
```

Description

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

NOTE

> MetadataRecord storage is not forced to reside inside the datalad repository of the dataset. MetadataRecord 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.

MetadataRecord 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

For example, if the root dataset path is '/home/root_ds', the following command can be used to aggregate metadata of two sub- datasets, e.g. '/home/root_ds/sub_ds1' and '/home/root_ds/sub_ds2', into the root dataset:

```
% datalad meta-aggregate -d /home/root_ds /home/root_ds/sub_ds1 /home/root_ds/sub_ds2
```

Options

SUB_DATASET_PATH

SUB_DATASET_PATH is a path to a sub-dataset whose metadata shall be aggregated into the topmost dataset (ROOT_DATASET). The sub-dataset must be located within the directory of the topmost dataset. Note: if SUB_DATASET_PATH is relative, it is resolved against the current working directory, not against the path of the topmost dataset. Constraints: value must be a string or value must be NONE

-h, –help, –help-np

show this help message. –help-np forcefully disables the use of a pager for displaying the help message

-d ROOT_DATASET, –dataset ROOT_DATASET

Topmost dataset metadata will be aggregated into. If no dataset is specified, a dataset will be discovered based on the current working directory. MetadataRecord for aggregated datasets will contain a dataset path that is relative to the top-dataset. Constraints: Value must be a Dataset or a valid identifier of a Dataset (e.g. a path) or value must be NONE

–version

show the module and its version which provides the command

Authors

datalad is developed by The DataLad Team and Contributors <team@datalad.org>.

datalad meta-dump

Synopsis

```
datalad meta-dump [-h] [-d DATASET] [-r] [--version] [DATASET_FILE_PATH_PATTERN]
```

Description

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

Dump the metadata of the file "dataset_description.json" in the dataset "simon". (The queried dataset is determined based on the current working directory):

```
% datalad meta-dump simon:dataset_description.json
```

Sometimes it is helpful to get metadata records formatted in a more accessible form, here as pretty-printed JSON:

```
% datalad -f json_pp meta-dump simon:dataset_description.json
```

Same query as above, but specify that all datasets should be queried for the given path:

```
% datalad meta-dump :somedir/subdir/thisfile.dat
```

Dump any metadata record of any dataset known to the queried dataset:

```
% datalad meta-dump -r
```

Dump any metadata record of any dataset known to the queried dataset and output pretty-printed JSON:

```
% datalad -f json_pp meta-dump -r
```

Show metadata for all files ending in *.json* in the root directories of all datasets:

```
% datalad meta-dump *:*.json -r
```

Show metadata for all files ending in *.json* in all datasets by not specifying a dataset at all. This will start dumping at the top-level dataset.:

```
% datalad meta-dump :*.json -r
```

Options

DATASET_FILE_PATH_PATTERN

path to query metadata for. Constraints: value must be a string or value must be NONE [Default: ”]

-h, –help, –help-np

show this help message. –help-np forcefully disables the use of a pager for displaying the help message

-d DATASET, –dataset DATASET

Dataset for which metadata should be dumped. If no directory name is provided, the current working directory is used.

-r, –recursive

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 sub-datasets on the filesystem. It merely determines what metadata is being reported from the given/discovered reference dataset.

–version

show the module and its version which provides the command

Authors

datalad is developed by The DataLad Team and Contributors <team@datalad.org>.

datalad meta-conduct

Synopsis

```
datalad meta-conduct [-h] [-m MAX_WORKERS] [-p {process|thread|sequential}] [--
↪pipeline-help] [--version] CONFIGURATION [ARGUMENTS [ARGUMENTS ...]]
```

Description

Conduct the execution of a processing pipeline

A processing pipeline is a metalad-specific application of the Unix shell philosophy, have a number of small programs that do one thing, but that one thing, very well.

Processing pipelines consist of:

- A provider, that provides data that should be processed
- A list of processors. A processor reads data, either from the previous processor or the provider and performs computations on the data and return a result that is processed by the next processor. The computation may have side-effect, e.g. store metadata.

The provider is usually executed in the current processes' main thread. Processors are usually executed in concurrent processes, i.e. workers. The maximum number of workers is given by the parameter MAX_WORKERS.

Which provider and which processors are used is defined in an "configuration", which is given as JSON-serialized dictionary.

Examples

Run 'metalad_example_dataset' extractor on the top dataset and all subdatasets. Add the resulting metadata in aggregatedmode. This command uses the provided pipelinedefinition 'extract_metadata'.:

```
% datalad meta-conduct extract_metadata traverser.top_level_dir=<dataset path>␣
↪traverser.item_type=dataset traverser.traverse_sub_datasets=True extractor.
↪extractor_type=dataset extractor.extractor_name=metalad_example_dataset adder.
↪aggregate=True
```

Run metalad_example_file extractor on all files of the root dataset and the subdatasets. Automatically get the content, if it is not present. Drop content that was automatically fetched after its metadata has been added.:

```
% datalad meta-conduct extract_metadata_autoget_autodrop traverser.top_level_dir=
↪<dataset path> traverser.item_type=file traverser.traverse_sub_datasets=True␣
↪extractor.extractor_type=file extractor.extractor_name=metalad_example_file adder.
↪aggregate=True
```

Options

CONFIGURATION

Path to a file with contains the pipeline configuration as JSON-serialized object. If the path is "-", the configuration is read from standard input.

ARGUMENTS

Constructor arguments for pipeline elements, i.e. provider, processors, and consumer. The arguments have to be prefixed with the name of the pipeline element, followed by ".", the keyname, a "=", and the value. The pipeline element arguments are identified by the pattern "<name>.<key>=<value>".

-h, –help, –help-np

show this help message. –help-np forcefully disables the use of a pager for displaying the help message

-m MAX_WORKERS, –max-workers MAX_WORKERS

maximum number of workers. Constraints: value must be convertible to type 'int' or value must be NONE

-p {process|thread|sequential}, –processing-mode {process|thread|sequential}

Specify how elements are executed, either in subprocesses, in threads, or sequentially in the main thread. The respective values are "process", "thread", and "sequential", (default: "process"). Constraints: value must be one of ('process', 'thread', 'sequential') [Default: 'process']

–pipeline-help

Show documentation for the elements in the pipeline and exit.

–version

show the module and its version which provides the command

Authors

datalad is developed by The DataLad Team and Contributors <team@datalad.org>.

datalad meta-filter

Synopsis

```
datalad meta-filter [-h] [-d DATASET] [-r] [--version] FILTER_NAME METADATA_URL␣
↪[METADATA_URL ...] [FILTER_ARGUMENTS [FILTER_ARGUMENTS ...]]
```

Description

Run a metadata filter on a set of metadata elements.

Take a number of metadata elements and run a filter on it.

The result of the filter operation will be written to stdout and can, for example, be passed to meta-add.

The filter can be configured by passing key-value pairs given as additional arguments. Each key-value pair consists of two arguments, first the key, then the value. The key value pairs have to be separated by '++' from the metadata coordinates

Examples

Use the provided 'metalad_demofilter' to build a 'histogram' of keys and their content in the metadata of the dataset 'root-dataset', iterating over the sub-datasets 'sub-a' and 'sub-b'.:

```
% datalad meta-filter metalad_demofilter -d root-dataset
  sub-a sub-b
```

Apply 'metalad_demofilter' to all directories/sub-datasets of the dataset in the current working directory that start with 'subject'.:

```
% datalad meta-filter metalad_demofilter subject*
```

Options

FILTER_NAME

Name of the filter that should be executed. Constraints: value must be a string

METADATA_URL

MetadataRecord URL(s). A list of at least one metadata URL. The filter will receive a list of iterables, that contains one iterable for each metadata URL. The iterables will yields all metadata-entries that match the respective metadata URL. Constraints: value must be a string

FILTER_ARGUMENTS

Extractor arguments given as string arguments to the extractor. Filter arguments have to be separated from the list of metadata coordinates by '++'. Constraints: value must be a string or value must be NONE

-h, –help, –help-np

show this help message. –help-np forcefully disables the use of a pager for displaying the help message

-d *DATASET*, –dataset *DATASET*

Git repository that contains datalad metadata. If no repository path is given, the metadata store is determined by the current work directory. All metadata URLs (see below) are relative to this dataset. Constraints: Value must be a Dataset or a valid identifier of a Dataset (e.g. a path) or value must be NONE [Default: '.']

-r, –recursive

If set, the metadata URL iterables will yield all metadata recursively from the matching metadata URLs.

–version

show the module and its version which provides the command

Authors

datalad is developed by The DataLad Team and Contributors <team@datalad.org>.

1.1.2 Python module reference

meta_add(metadata, int, float, bool, None, ...)	Add metadata to a dataset.
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
meta_conduct(configuration, int, float, ...)	Conduct the execution of a processing pipeline
meta_filter(filtername, metadataurls, ...)	Run a metadata filter on a set of metadata elements.

datalad.api.meta_add

datalad.api.**meta_add**(*metadata: Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]]]]]], additionalvalues: Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]]]]]] = None, dataset: Union[datalad.distribution.dataset.Dataset, str, None] = None, allow_override: bool = False, allow_unknown: bool = False, allow_id_mismatch: bool = False, json_lines: bool = False, batch_mode: bool = False*)

Add metadata to a dataset.

This command reads metadata from a source and adds this metadata to a dataset. The source can either be a file, or standard input. The metadata format is a string with the JSON-serialized dictionary, or list of dictionaries (or individual dictionaries in JSON-Lines format) that describes the metadata.

In case of an API-call metadata can also be provided in a python dictionary or a list of dictionaries.

If metadata is read from a source, additional parameter can overwrite or amend information that is provided by the source.

The ADDITIONAL_VALUES arguments can be pre-fixed by '@', in which case the pre-fixed argument is interpreted as a file-name and the argument value is read from the file.

The metadata key "dataset-id" must be identical to the ID of the dataset that receives the metadata, unless -i or –allow-id-mismatch is provided.

Examples

Parameters

- **metadata** (*str*) – path of the file that contains the metadata that should be added to the metadata store (metadata records must be provided as a JSON- serialized metadata dictionary). The file may contain a single metadata-record or a JSON-array with multiple metadata-records. If –json-lines is given, the file may also contain one dictionary per line. If the path is "-", the metadata file is read from standard input. The dictionary must contain the following keys: 'type' 'extractor_name' 'extractor_version' 'extraction_parameter' 'extraction_time' 'agent_name' 'agent_email' 'dataset_id' 'dataset_version' 'extracted_metadata' If the metadata is associated with a file, the following key indicates the file path: 'path' If the metadata should refer to a sub-dataset element (that means if an "aggregated" record should be stored (see meta-aggregate for more info), the following key indicates the path of the sub-dataset from the root of the "containing dataset": 'dataset_path' The containing dataset, aka. the "root dataset", is the dataset version, specified by dataset-id and version that contains the given sub-dataset version at the path 'dataset_path'. If the containing dataset, aka. the "root dataset", is known, it can be specified by providing the following keys: 'root_dataset_id' 'root_dataset_version' If the version of the root dataset that contains the given subdataset at the given path is not known, the "root_dataset_*"-keys can be omitted. Such a situation might arise, if the sub-dataset metadata was extracted in an old version of the sub-dataset, and the relation of this old version to the root-dataset is not known (we assume that the root-dataset would be the dataset to which the metadata is added.) In this case the metadata is added with an "anonymous" root dataset, but with the given sub-dataset-path. (This makes sense, if the sub-dataset at the given path contains a version that is defined in the metadata, i.e. in dataset_version. The metadata will then be added to all versions that meet this condition.).
- **additionalvalues** (*str or None, optional*) – A string that contains a JSON serialized dictionary of key value- pairs. These key values-pairs are used in addition to the key value pairs in the metadata dictionary to describe the metadata that should be added. If an additional key is already present in the metadata, an error is raised, unless -o, –allow-override is provided. In this case, the additional values will override the value in metadata and a warning is issued. NB! If multiple records are provided in METADATA, the additional values will be applied to all of them. [Default: None]
- **dataset** (*Dataset or None, optional*) – "dataset to which metadata should be added. If not provided, the dataset is assumed to be given by the current directory. [Default: None]
- **allow_override** (*bool, optional*) – Allow the additional values to override values given in metadata. [Default: False]
- **allow_unknown** (*bool, optional*) – Allow unknown keys. By default, unknown keys generate an errors. If this switch is True, unknown keys will only be reported. For processing unknown keys will be ignored. [Default: False]

- **allow_id_mismatch** (*bool, optional*) – Allow insertion of metadata, even if the "dataset-id" in the metadata source does not match the ID of the target dataset. [Default: False]
- **json_lines** (*bool, optional*) – Interpret metadata input as JSON lines. i.e. expect one metadata record per line. This is the format used by commands like "datalad metadump". [Default: False]
- **batch_mode** (*bool, optional*) – Enable batch mode. In batch mode metadata-records are read from stdin, one record per line, and a result is written to stdout, one result per line. Batch mode can be exited by sending an empty line that just consists of a newline. Meta-add will return an empty line that just consists of a newline to confirm the exit request. When this flag is given, the metadata file name should be set to "-" (minus). [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** – select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'generic']
- **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']

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, force_dataset_level: 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 dataset level extraction should be performed and you want to provide extractor arguments, you have to specify '–force_dataset_level' to ensure dataset-level extraction. i.e. to prevent interpretation of the key of the first extractor argument as path for a file-level extraction.

The command can also take legacy datalad-metalad extractors and will execute them in either "content" or "dataset" mode, depending on the whether file-level- or dataset-level extraction is requested.

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. The path should be relative to the root of the dataset. 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. You might provide an absolute file path, but it has to contain the dataset path as prefix. [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]

- **force_dataset_level** (*bool, optional*) – [Default: False]
- **extractorargs** (*sequence of str or None, optional*) – Extractor arguments given as string arguments to the extractor. The extractor arguments are interpreted as key-value pairs. The first argument is the name of the key, the next argument is the value for that key, and so on. Consequently, there should be an even number of extractor arguments. If dataset level extraction should be performed and you want to provide extractor arguments. you have tp specify '– force-dataset-level' to ensure dataset-level extraction. i.e. to prevent interpretation of the key of the first extractor argument as path for a file-level extraction. [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** – select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [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']

datalad.api.meta_aggregate

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

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

Note: MetadataRecord storage is not forced to reside inside the datalad repository of the dataset. MetadataRecord 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.

MetadataRecord 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. MetadataRecord for aggregated datasets will contain a dataset path that is relative to the top-dataset. [Default: None]
- **path** (*non-empty sequence of str or None, optional*) – SUB_DATASET_PATH is a path to a sub-dataset whose metadata shall be aggregated into the topmost dataset (ROOT_DATASET). The sub-dataset must be located within the directory of the topmost dataset. Note: if SUB_DATASET_PATH is relative, it is resolved against the current working directory, not against the path of the topmost 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** – select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [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']

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 sub-datasets 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** – select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [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']

datalad.api.meta_conduct

datalad.api.**meta_conduct**(*configuration: Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]], List[Union[str, int, float, bool, None, Dict[str, Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]]]]]]]], arguments: List[str], max_workers: Optional[int] = None, processing_mode: str = 'process', pipeline_help: bool = False*)

Conduct the execution of a processing pipeline

A processing pipeline is a metalad-specific application of the Unix shell philosophy, have a number of small programs that do one thing, but that one thing, very well.

Processing pipelines consist of:

- A provider, that provides data that should be processed
- A list of processors. A processor reads data, either from the previous processor or the provider and performs computations on the data and return a result that is processed by the next processor. The computation may have side-effect, e.g. store metadata.

The provider is usually executed in the current processes' main thread. Processors are usually executed in concurrent processes, i.e. workers. The maximum number of workers is given by the parameter *max_workers*.

Which provider and which processors are used is defined in an "configuration", which is given as JSON-serialized dictionary.

Examples

Parameters

- **configuration** – Path to a file with contains the pipeline configuration as JSON- serialized object. If the path is "-", the configuration is read from standard input.
- **arguments** – Constructor arguments for pipeline elements, i.e. provider, processors, and consumer. The arguments have to be prefixed with the name of the pipeline element, fol-

lowed by ".", the keyname, a "=", and the value. The pipeline element arguments are identified by the pattern "<name>.<key>=<value>".

- **max_workers** (*int or None, optional*) – maximum number of workers. [Default: None]
- **processing_mode** (*{'process', 'thread', 'sequential'}, optional*) – Specify how elements are executed, either in subprocesses, in threads, or sequentially in the main thread. The respective values are "process", "thread", and "sequential", (default: "process"). [Default: 'process']
- **pipeline_help** (*bool, optional*) – Show documentation for the elements in the pipeline and exit. [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** – select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [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']

datalad.api.meta_filter

datalad.api.**meta_filter**(*filtername: str, metadataurls: List[Union[str, datalad_metalad.pathutils.metadataurlparser.MetadataURL]], dataset: Union[datalad.distribution.dataset.Dataset, str, None] = '.', filterargs: Optional[List[str]] = None, recursive: bool = False*) → Iterable[T_co]

Run a metadata filter on a set of metadata elements.

Take a number of metadata elements and run a filter on it.

The result of the filter operation will be written to stdout and can, for example, be passed to meta-add.

The filter can be configured by passing key-value pairs given as additional arguments. Each key-value pair consists of two arguments, first the key, then the value. The key value pairs have to be separated by '++' from the metadata coordinates

Examples

Parameters

- **filtername** (*str*) – Name of the filter that should be executed.
- **metadataurls** (*non-empty sequence of str*) – MetadataRecord URL(s). A list of at least one metadata URL. The filter will receive a list of iterables, that contains one iterable for each metadata URL. The iterables will yields all metadata- entries that match the respective metadata URL.
- **dataset** (*Dataset or None, optional*) – Git repository that contains datalad metadata. If no repository path is given, the metadata store is determined by the current work directory. All metadata URLs (see below) are relative to this dataset. [Default: '.']
- **filterargs** (*sequence of str or None, optional*) – Extractor arguments given as string arguments to the extractor. Filter arguments have to be separated from the list of metadata coordinates by '++'. [Default: None]
- **recursive** (*bool, optional*) – If set, the metadata URL iterables will yield all metadata recursively from the matching metadata URLs. [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** – select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result

rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [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*.

1.2.1 Next generation extractors

core	MetadataRecord extractor for Datalad's own core storage
annex	MetadataRecord extractor for Git-annex metadata
custom	MetadataRecord extractor for custom (JSON-LD) metadata contained in a dataset
runprov	MetadataRecord extractor for provenance information in DataLad's *run* records
studyminimeta	

datalad_metalad.extractors.core

MetadataRecord 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.

datalad_metalad.extractors.annex

MetadataRecord 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.

datalad_metalad.extractors.custom

MetadataRecord 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.

datalad_metalad.extractors.runprov

MetadataRecord 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",
      "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*)

datalad_metalad.extractors.studyminimeta

1.2.2 Legacy extractors

A number of legacy extractors are kept in the source code of this repository.

annex	Metadata extractor for Git-annex metadata
datacite	Extractor for datacite xml records, currently for CRCNS datasets
datalad_core	Metadata extractor for DataLad's own core storage
datalad_rfc822	Extractor for RFC822-based metadata specifications

Continued on next page

Table 3 – continued from previous page

frictionless_datapackage	Extractor for friction-less data packages (http://specs.frictionlessdata.io/data-packages)
image	generic image metadata extractor

datalad_metalad.extractors.legacy.annex

Metadata extractor for Git-annex metadata

class datalad_metalad.extractors.legacy.annex.**AnnexMetadataExtractor**(*ds*, *paths*)

Bases: datalad_metalad.extractors.base.BaseMetadataExtractor

NEEDS_CONTENT = False

datalad_metalad.extractors.legacy.datacite

Extractor for datacite xml records, currently for CRCNS datasets

class datalad_metalad.extractors.legacy.datacite.**DataciteMetadataExtractor**(*ds*, *paths*)

Bases: datalad_metalad.extractors.base.BaseMetadataExtractor

datalad_metalad.extractors.legacy.datalad_core

Metadata extractor for DataLad's own core storage

class datalad_metalad.extractors.legacy.datalad_core.**DataladCoreMetadataExtractor**(*ds*, *paths*)

Bases: datalad_metalad.extractors.base.BaseMetadataExtractor

NEEDS_CONTENT = False

datalad_metalad.extractors.legacy.datalad_rfc822

Extractor for RFC822-based metadata specifications

This is inspired by (and very similar to) Debian's package metadata format. The main difference is that information spread across multiple files in Debian packages, is concentrated in one file.

The main advantage of this format is that it is proven to be hand-editable, i.e. can be composed from scratch, by hand, in an editor – with a good chance of producing syntax-compliant content with the first attempt.

class datalad_metalad.extractors.legacy.datalad_rfc822.**DataladRFC822MetadataExtractor**(*ds*, *paths*)

Bases: datalad_metalad.extractors.base.BaseMetadataExtractor

datalad_metalad.extractors.legacy.frictionless_datapackage

Extractor for friction-less data packages (http://specs.frictionlessdata.io/data-packages)

class datalad_metalad.extractors.legacy.frictionless_datapackage.**FRDPMetadataExtractor**(*ds*, *paths*

Bases: datalad_metalad.extractors.base.BaseMetadataExtractor

metadatasrc_fname = 'datapackage.json'

datalad_metalad.extractors.legacy.image

generic image metadata extractor

class datalad_metalad.extractors.legacy.image.**ImageMetadataExtractor**(*ds*, *paths*)

Bases: datalad_metalad.extractors.base.BaseMetadataExtractor

get_metadata(*dataset*, *content*)

Returns Dataset metadata dict, dictionary of filepath regexes with metadata, dicts, each return value could be None if there is no such metadata

Return type dict or None, dict or None

1.3 User guide

1.3.1 User guide

This chapter contains user guides describing various aspects of MetaLad.

Meet Metalad, an extension to datalad that supports metadata handling

This chapter will show you how to install `metalad`, create a datalad dataset and how to work with metadata. Working with metadata means, adding metadata, viewing metadata, and aggregating metadata.

Below you will find examples and instructions on how to install metalad, how to create an example dataset and how to work with metadata. You will learn how to add metadata, how to display metadata that is stored locally or remotely, and how to combine (aka aggregate) metadata from multiple datasets.

Preparations

Install metalad and create a datalad dataset

Install metalad

Create a virtual python environment and activate it (Linux example shown).

```
> python3 -m venv venv/datalad-metalad
> source venv/datalad-metalad/bin/activate
```

Install the latest version of metalad

```
> pip install --upgrade datalad-metalad
```

Create a datalad dataset

The following command creates a datalad dataset that stores text-files in git and non-text files in git-annex.

```
> datalad create -c text2git example-ds
```

Add a text-file to the dataset

```
> cd example-ds
> echo some content > file1.txt
> datalad save
```

Add a binary file to the dataset

```
> dd if=/dev/zero of=file.bin count=1000
> datalad save
```

Working with metadata

This chapter provides an overview of commands in metalad. If you want to continue with the hands-on examples, skip to next chapter (and probably come back here later).

Metalad allows you to *associate* metadata with datalad datasets or files in datalad-datasets. More specifically, metadata can be associated with: datasets, sub-datasets, and files in a dataset or sub-dataset.

Metalad can associate an arbitrary amount of individual metadata *instances* with a single element (dataset or file). Each metadata instance is identified by a *type-name* that specifies the type of the data contained in the metadata instance. For example: `metalad_core`, `bids`, etc.

Note: Implementation side note: The metadata associations can in principle be stored in any git-repository, but are by default stored in the git-repository of a root dataset.

The plumbing

Metalad has a few basic commands, aka plumbing commands, that perform essential elementary metadata operations:

- `meta-add` add metadata to an element (dataset, sub-dataset, or file)
- `meta-dump` show metadata stored in a local or remote dataset

The porcelain

To simplify working with metadata, metalad provides a number of higher level functions and tools that implement typical use cases.

- `meta-extract` run an extractor (see below) on an existing dataset, sub-dataset, or file and emit metadata that can be fed to `meta-add`.
- `meta-conduct` run pipelines of extractors and adders on locally available datatasets, sub-datasets and files, in order to automatate metadata extraction and adding tasks
- `meta-aggregate` combine metadata from number of sub-datasets into the root-dataset.
- `meta-filter` walk through metadata from multiple stores, apply a filter, and output new metadata

Metadata extractors

Datalad supports pluggable metadata extractors. Metadata extractors can perform arbitrary operations on the given element (dataset, sub-dataset, or file) and return arbitrary metadata in JSON-format. Meta-extract will associate the metadata with the metadata element.

Metalad comes with a number of extractors. Some extractors are provided by metalad, some are inherited from datalad. The provided extractors generate provenance records for datasets and data, or they extract metadata from specific files or data-structures, e.g. BIDS. In principle any processing is possible. There is also a generic extractor, which allows to invoke external commands to generate metadata.

Metadata extraction examples

Extract dataset-level metadata

Extract dataset-level metadata with the `datalad` command `meta-extract`. It takes a number of optional arguments and one required argument, the name of the metadata extractor that should be used. We use `metalad_core` for now.

```
> datalad meta-extract metalad_core
```

The extracted metadata will be written to stdout and will look similar to this (times, names, and UUIDs will be different for you):

```
{"type": "dataset", "dataset_id": "853d9356-fc2e-459e-96bc-02414a1fef93", "dataset_
↪version": "8d6d0e50a27b7540717360e21332b1ad0c924415", "extractor_name": "metalad_
↪core", "extractor_version": "1", "extraction_parameter": {}, "extraction_time":␣
↪1637921555.282522, "agent_name": "Your Name", "agent_email": "you@example.com",
↪"extracted_metadata": {"@context": {"@vocab": "http://schema.org/", "datalad":
↪"http://dx.datalad.org/"}, "@graph": [{"@id": "59286713dacabfbce1cecf4c865fff5a",
↪"@type": "agent", "name": "Your Name", "email": "you@example.com"}, {"@id":
↪"8d6d0e50a27b7540717360e21332b1ad0c924415", "identifier": "853d9331-fc2e-459e-96bc-
↪02414a1fef93", "@type": "Dataset", "version": "0-3-g8d6d0e5", "dateCreated": "2021-
↪11-26T11:03:25+01:00", "dateModified": "2021-11-26T11:09:27+01:00", "hasContributor
↪": {"@id": "59286713dacabfbce1cecf4c865fff5a"}}]}}
```

The output is a JSON-serialized object. You can use ` jq <https://stedolan.github.io/jq/>‘__ to get a nicer formatting of the JSON-object. For example the command:

```
> datalad meta-extract metalad_core|jq .
```

would result in an output similar to:

```
{
  "type": "dataset",
  "dataset_id": "853d9356-fc2e-459e-96bc-02414a1fef93",
  "dataset_version": "ee512961b878a674c8068e54656e161d40566d9b",
  "extractor_name": "metalad_core",
  "extractor_version": "1",
  "extraction_parameter": {},
  "extraction_time": 1637923596.9511302,
  "agent_name": "Your Name",
  "agent_email": "you@example.com",
  "extracted_metadata": {
    "@context": {
```

(continues on next page)

(continued from previous page)

```
      "@vocab": "http://schema.org/",
      "datalad": "http://dx.datalad.org/"
    },
    "@graph": [
      {
        "@id": "59286713dacabfbce1cecf4c865fff5a",
        "@type": "agent",
        "name": "Your Name",
        "email": "you@example.com"
      },
      {
        "@id": "ee512961b878a674c8068e54656e161d40566d9b",
        "identifier": "853d9356-fc2e-459e-96bc-02414a1fef93",
        "@type": "Dataset",
        "version": "0-4-gee51296",
        "dateCreated": "2021-11-26T11:03:25+01:00",
        "dateModified": "2021-11-26T11:13:58+01:00",
        "hasContributor": {
          "@id": "59286713dacabfbce1cecf4c865fff5a"
        }
      }
    ]
  }
```

Extract file-level metadata

The `datalad` command `meta-extract` also support the extraction of file-level metadata. File-level metadata extraction requires a second argument, besides the extractor-name, to `datalad meta-extract`. The second argument identifies the file for which metadata should be extracted.

NB: you must specify an extractor that supports file-level extraction if a file-name is passed to `datalad meta-extract`, and an extractor that supports dataset-level extraction if no file-name is passed to `datalad meta-extract`. The extractor `metalad_core` supports both metadata levels.

To extract metadata for the file `file1.txt`, execute the following command:

```
> datalad meta-extract metalad_core file1.txt
```

which will lead to an output similar to:

```
{"type": "file", "dataset_id": "853d9331-fc2e-459e-96bc-02414a1fef93", "dataset_
↪version": "ee512961b878a674c8068e54656e161d40566d9b", "path": "file1.txt",
↪"extractor_name": "metalad_core", "extractor_version": "1", "extraction_parameter":
↪{}, "extraction_time": 1637927097.2165475, "agent_name": "Your Name", "agent_email
↪": "you@example.com", "extracted_metadata": {"@id": "datalad:SHA1-s13--
↪2ef267e25bd6c6a300bb473e604b092b6a48523b", "contentbytesize": 13}}
```

Add metadata

You can add extracted metadata to the dataset (metadata will be stored in a special area of the git-repository and not interfere with your data in the dataset).

To add metadata you use the `datalad` command `meta-add`. The `meta-add` command takes on required argument, the name of a file that contains metadata in JSON-format. It also supports reading JSON-metadata from stdin, if you

provided - as the file name. That mean you can pipe the output of `meta-extract` directly into `meta-add` by specifying - as metadata file-name like this:

```
> datalad meta-extract metalad_core |datalad meta-add -
```

`meta-add` supports files that contain lists of JSON-records in "JSON Lines"-format (see jsonlines.org).

Let's add the file-level metadata for `file1.txt` and `file.bin` to the metadata of the dataset by executing the two commands:

```
> datalad meta-extract metalad_core file1.txt |datalad meta-add -
```

and

```
> datalad meta-extract metalad_core file.bin |datalad meta-add -
```

Display (retrieve) metadata

To view the metadata that has been stored in a dataset, you can use the `datalad` command `meta-dump`. The following command will show all metadata that is stored in the dataset. Metadata is displayed in JSON Lines-format (aka newline-delimited JSON), which is a number of lines where each line contains a serialized JSON object.

```
datalad meta-dump -r
```

Its execution will generate a result similar to:

```
{"type": "dataset", "dataset_id": "853d9356-fc2e-459e-96bc-02414a1fef93", "dataset_
↪version": "ee512961b878a674c8068e54656e161d40566d9b", "extraction_time": 1637924361.
↪8114567, "agent_name": "Your Name", "agent_email": "you@example.com", "extractor_
↪name": "metalad_core", "extractor_version": "1", "extraction_parameter": {},
↪"extracted_metadata": {"@context": {"@vocab": "http://schema.org/", "datalad":
↪"http://dx.datalad.org/"}, "@graph": [{"@id": "59286713dacabfbce1cecf4c865fff5a",
↪"@type": "agent", "name": "Your Name", "email": "you@example.com"}, {"@id":
↪"ee512961b878a674c8068e54656e161d40566d9b", "identifier": "853d9356-fc2e-459e-96bc-
↪02414a1fef93", "@type": "Dataset", "version": "0-4-gee51296", "dateCreated": "2021-
↪11-26T11:03:25+01:00", "dateModified": "2021-11-26T11:13:58+01:00", "hasContributor
↪": {"@id": "59286713dacabfbce1cecf4c865fff5a"}}]}}
{"type": "file", "path": "file1.txt", "dataset_id": "853d9356-fc2e-459e-96bc-
↪02414a1fef93", "dataset_version": "ee512961b878a674c8068e54656e161d40566d9b",
↪"extraction_time": 1637927239.2590044, "agent_name": "Your Name", "agent_email":
↪"you@example.com", "extractor_name": "metalad_core", "extractor_version": "1",
↪"extraction_parameter": {}, "extracted_metadata": {"@id": "datalad:SHA1-s13--
↪2ef267e25bd6c6a300bb473e604b092b6a48523b", "contentbytesize": 13}}
{"type": "file", "path": "file.bin", "dataset_id": "853d9356-fc2e-459e-96bc-
↪02414a1fef93", "dataset_version": "ee512961b878a674c8068e54656e161d40566d9b",
↪"extraction_time": 1637927246.2115273, "agent_name": "Your Name", "agent_email":
↪"you@example.com", "extractor_name": "metalad_core", "extractor_version": "1",
↪"extraction_parameter": {}, "extracted_metadata": {"@id": "datalad:MD5E-s512000--
↪816df6f64deba63b029ca19d880ee10a.bin", "contentbytesize": 512000}}
```

Adding a lot of metadata with `meta-conduct`

To extract and add metadata from a large number of files or from all files of a dataset you can use `meta-conduct`. Meta-conduct can be configured to execute a number of `meta-extract` and `meta-add` commands automatically

in parallel. The operations that `meta-conduct` should perform are defined in pipeline definitions. A few pipeline definitions are provided with metalad, and we will use the `extract_metadata` pipeline.

Adding dataset-level metadata

Execute the following command:

```
datalad meta-conduct extract_metadata traverser:`pwd` traverser:dataset␣
↪extractor:dataset extractor:metalad_core
```

You will get an output which is similar to:

```
meta_conduct(ok): <...>/gist/example-ds
```

What happened?

You just ran the `extract_metadata` pipeline and specified that you want to traverse the current directory (`traverser:`pwd``), and that you want to operate on all datasets that are encountered (`traverser:Dataset`). You also specified that, for each element found during traversal, you would like to execute a dataset-level extractor (`extractor:dataset`) with the name `metalad_core` (`extractor:metalad_core`).

The pipeline found one dataset in the current directory and added the metadata to it. Since you have done that already before using `meta-extract` and `meta-add`, you have the same number of metadata entries in the metadata store. That means `datalad meta-dump -r` will give you three results. But you might notice that the `extraction-time` of the dataset-level entry has changed.

Metalad comes with different pre-built pipelines. Some allow to automatically fetch an annexed file and automatically drop said file, after is has been processed.

Adding file-level metadata

You can also add file-metadata using `meta-conduct`. Execute the following command:

```
datalad meta-conduct extract_metadata traverser:`pwd` traverser:file extractor:file␣
↪extractor:metalad_core
```

You will get an output which is similar to:

```
meta_conduct(ok): <...>/example-ds/file1.txt
meta_conduct(ok): <...>/example-ds/file.bin
action summary:
  meta_conduct (ok: 2)
```

What happened here?

The traverser found two elements that fitted your description (`traverser:Dataset`), executed the specified extractor on them (`extractor:metalad_core`), and added the results to the metadata storage.

Again, you can verify this with the value of `extraction_time` in the output of `datalad meta-dump -r`.

Joining metadata from multiple datasets with `meta-aggregate`

Let's have a look at `meta-aggregate`. The command `meta-aggregate` copies metadata from sub-datasets into the metadata store of the root dataset.

Subdataset creation

To see `meta-aggregate` in action we first create a sub-datasets:

```
> datalad create -d . -c text2git subds1
```

This command will yield an output similar to:

```
[INFO   ] Creating a new annex repo at <...>/example-ds/subds1
[INFO   ] Running procedure cfg_text2git
[INFO   ] == Command start (output follows) =====
[INFO   ] == Command exit (modification check follows) =====
add(ok): subds1 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
create(ok): subds1 (dataset)
action summary:
  add (ok: 2)
  create (ok: 1)
  save (ok: 1)
```

Create some content and save it:

```
> cd subds1
> echo content of subds1/file_subds1.1.txt > file_subds1.1.txt
> datalad save
```

Now run the file level extractor in the subdataset:

```
> datalad meta-conduct extract_metadata traverser:`pwd` traverser:file extractor:file␣
↪extractor:metalad_core
```

and the dataset-level extractor:

```
> datalad meta-conduct extract_metadata traverser:`pwd` traverser:dataset␣
↪extractor:dataset extractor:metalad_core
```

If you want you can view the added metadata in the subdataset with the command `datalad meta-dump -r`.

Since we modified the subdataset, we should also save the root dataset:

```
> cd ..
> datalad save
```

Aggregating

After all the above commands are executed, we have metadata stored in two datasets (more precisely, in the metadata stores of the datasets which are the git repositories). In the metadata store of `example-ds` we have the following information:

And in the metadata store of `subds1` we have:

Now let us aggregate the subdataset metadata into the root dataset with the command `meta-aggregate`:

```
> datalad meta-aggregate -d . subds1
```

And display the result:

```
> datalad meta-dump -r
```

The output will contain five JSON records (in 5 lines), three from the top-level datasets and two from the subdataset. It will look similar to this:


```
{"type": "dataset", "dataset_id": "ceeb844a-c6e8-4b2f-bb7c-62b7ae449a9f", "dataset_
↪version": "bcf9cfde4a599d26094a58efbe4369e0878cb9c8", "extraction_time": 1638357863.
↪4242253, "agent_name": "Your Name", "agent_email": "you@example.com", "extractor_
↪name": "metalad_core", "extractor_version": "1", "extraction_parameter": {},
↪"extracted_metadata": {"@context": {"@vocab": "http://schema.org/", "datalad":
↪"http://dx.datalad.org/"}, "@graph": [{"@id": "59286713dacabfbce1cecf4c865fff5a",
↪"@type": "agent", "name": "Your Name", "email": "you@example.com"}, {"@id":
↪"bcf9cfde4a599d26094a58efbe4369e0878cb9c8", "identifier": "ceeb844a-c6e8-4b2f-bb7c-
↪62b7ae449a9f", "@type": "Dataset", "version": "0-4-gbcf9cfd", "dateCreated": "2021-
↪12-01T12:24:17+01:00", "dateModified": "2021-12-01T12:24:19+01:00", "hasContributor
↪": {"@id": "59286713dacabfbce1cecf4c865fff5a"}}]}}
{"type": "file", "path": "file1.txt", "dataset_id": "ceeb844a-c6e8-4b2f-bb7c-
↪62b7ae449a9f", "dataset_version": "bcf9cfde4a599d26094a58efbe4369e0878cb9c8",
↪"extraction_time": 1638357864.5259314, "agent_name": "Your Name", "agent_email":
↪"you@example.com", "extractor_name": "metalad_core", "extractor_version": "1",
↪"extraction_parameter": {}, "extracted_metadata": {"@id": "datalad:SHA1-s13--
↪2ef267e25bd6c6a300bb473e604b092b6a48523b", "contentbytesize": 13}}
{"type": "file", "path": "file.bin", "dataset_id": "ceeb844a-c6e8-4b2f-bb7c-
↪62b7ae449a9f", "dataset_version": "bcf9cfde4a599d26094a58efbe4369e0878cb9c8",
↪"extraction_time": 1638357864.5327883, "agent_name": "Your Name", "agent_email":
↪"you@example.com", "extractor_name": "metalad_core", "extractor_version": "1",
↪"extraction_parameter": {}, "extracted_metadata": {"@id": "datalad:MD5E-s512000--
↪816df6f64deba63b029ca19d880ee10a.bin", "contentbytesize": 512000}}
{"type": "dataset", "root_dataset_id": "<unknown>", "root_dataset_version":
↪"7228f027171f7b8949a47812a651600412f2577e", "dataset_path": "subds1", "dataset_id":
↪"4e3422f4-b606-4cf9-818a-a3bb840e3396", "dataset_version":
↪"ddf2a2758fd6773a1171a6fbae4afe48cc982773", "extraction_time": 1638357869.7052076,
↪"agent_name": "Your Name", "agent_email": "you@example.com", "extractor_name":
↪"metalad_core", "extractor_version": "1", "extraction_parameter": {}, "extracted_
↪metadata": {"@context": {"@vocab": "http://schema.org/", "datalad": "http://dx.
↪datalad.org/"}, "@graph": [{"@id": "59286713dacabfbce1cecf4c865fff5a", "@type":
↪"agent", "name": "Your Name", "email": "you@example.com"}, {"@id":
↪"ddf2a2758fd6773a1171a6fbae4afe48cc982773", "identifier": "4e3422f4-b606-4cf9-818a-
↪a3bb840e3396", "@type": "Dataset", "version": "0-3-gddf2a27", "dateCreated": "2021-
↪12-01T12:24:25+01:00", "dateModified": "2021-12-01T12:24:27+01:00", "hasContributor
↪": {"@id": "59286713dacabfbce1cecf4c865fff5a"}}]}}
{"type": "file", "path": "file_subds1.1.txt", "root_dataset_id": "<unknown>", "root_
↪dataset_version": "7228f027171f7b8949a47812a651600412f2577e", "dataset_path":
↪"subds1", "dataset_id": "4e3422f4-b606-4cf9-818a-a3bb840e3396", "dataset_version":
↪"ddf2a2758fd6773a1171a6fbae4afe48cc982773", "extraction_time": 1638357868.706351,
↪"agent_name": "Your Name", "agent_email": "you@example.com", "extractor_name":
↪"metalad_core", "extractor_version": "1", "extraction_parameter": {}, "extracted_
↪metadata": {"@id": "datalad:SHA1-s36--9ce18068eb4126c23235d965c179b2a53546d104",
↪"contentbytesize": 36}}
```


Todo: Upcoming: how to delete metadata, how to filter metadata, and how to export metadata.

Writing custom extractors

Metalad supports automated extraction of metadata through a common single interface which allows execution of metadata extractors. An extractor, in MetaLad sense, is a class derived from one of the base extractor classes defined in `datalad_metalad.extractors.base` (`DatasetMetadataExtractor` or `FileMetadataExtractor`). It needs to implement several required methods, most notably `extract()`.

Example extractors can be found in MetaLad source code:

- Dataset-level extractor example
- File-level extractor example

Base class

There are two primary types of extractors, dataset-level extractors and file-level extractors.

Dataset-level extractors, by inheritance from the `DatasetMetadataExtractor` class, can access the dataset on which they operate as `self.dataset`. Extractor functions may use this object to call any DataLad dataset methods. They can perform whatever operations they deem necessary to extract metadata from the dataset, for example, they could count the files in the dataset or look for a file named `CITATION.cff` in the root directory of the dataset and return its content.

File-level extractors, by inheritance from the `FileMetadataExtractor`, contain a `Dataset`-object in the property `self.dataset` and a `FileInfo`-object in the property `self.file_info`. `FileInfo` is a dataclass with the properties `type`, `git_sha_sum`, `byte_size`, `state`, `path`, and `intra_dataset_path` fields. File-level extractors should return metadata that describes the file that is referenced by `FileInfo`.

Required methods

Any extractor is expected to implement a number of abstract methods that define the interface through which MetaLad communicates with extractors.

`get_id()`

This function should return a `UUID` object containing a UUID which is not used by another metalad-extractor or metalad-filter. Since it is meant to uniquely identify the extractor and the type of data it returns, its return value should be hardcoded, or generated in a deterministic way.

A UUID can be generated in several ways, e.g. with Python (`import uuid; print(uuid.uuid4())`), or online generators (e.g. https://www.uuidgenerator.net/ or by searching 'UUID' in DuckDuckGo).

While version 4 (random) UUIDs are probably sufficient, version 5 UUIDs (generated based on a given namespace and name) can also be used. For example, extractors in packages developed under the datalad organisation could use `<extractor_name>.datalad.org`. Example generation with Python: `uuid.uuid5(uuid.NAMESPACE_DNS, "bids_dataset.extractors.datalad.org")`.

Example:

```
def get_id(self) -> UUID:
    return UUID("0c26f218-537e-5db8-86e5-bc12b95a679e")
```

get_version()

This function should return a version string representing the extractor's version.

The extractor version is meant to be included with the extracted metadata. It might be a version number, or a reference to a particular schema. While there are no requirements on the versioning scheme, it should be chosen carefully, and rules for version updating need to be considered. For example, when using semantic versioning style, one might decide to guarantee that attribute type and name will never change within a major release, while new attributes may be added with a minor release.

Example:

```
def get_version(self) -> str:
    return "0.0.1"
```

get_data_output_category()

This function should return a `DataOutputCategory` object, which tells MetaLad what kind of (meta)data it is dealing with. The following output categories are available in the enumeration `datalad_metalad.extractors.base.DataOutputCategory`:

```
IMMEDIATE
FILE
DIRECTORY
```

Output categories declare how the extractor delivers its results. If the output category is `IMMEDIATE`, the result is returned by the `extract()`-call. In case of `FILE`, the extractor deposits the result in a file. This ie especially useful for extractors that are external programs. The category `DIRECTORY` is not yet supported. If you write an extractor in Python, you would usually use the output category `IMMEDIATE` and return extractor results from the `extract()`-call.

Example:

```
def get_data_output_category(self) -> DataOutputCategory:
    return DataOutputCategory.IMMEDIATE
```

get_required_content()

This function is used in dataset-level extractors only. It will be called by MetaLad prior to metadata extraction. Its purpose is to allow the extractor to ensure that content that is required for metadata extraction is present (relevant, for example, if some of files to be inspected may be annexed).

The function should either return a boolean value (`True | False`) or return a `Generator` with DataLad result records. In the case of a boolean value, the function should return `True` if it has obtained the required content, or confirmed its presence. If it returns `False`, metadata extraction will not proceed. Alternatively, yielding result records provides extractors with the capability to signal more expressive messages or errors. If a result record is yielded with a failure status (i.e. with `status` equal to `impossible` or `error`) metadata extraction will not proceed.

This function can be a place to call `dataset.get()`. It is advisable to disable result rendering (`result_renderer="disabled"`), because during metadata extraction, users will typically want to redirect standard output to a file or another command.

Example 1:

```
def get_required_content(self) -> bool:
    result = self.dataset.get("CITATION.cff", result_renderer="disabled")
    return result[0]["status"] in ("ok", "notneeded")
```

Example 2:

```
from typing import Generator
def get_required_content(self) -> Generator:
    yield self.dataset.get("CITATION.cff", result_renderer="disabled")
```

Example 3:

```
from typing import Generator
def get_required_content(self) -> Generator:
    result = self.dataset.get('CITATION.cff', result_renderer='disabled')
    failure_count = 0
    result_dict = dict(
        path=self.dataset.path,
        type='dataset',
    )
    for r in res:
        if r['status'] in ['error', 'impossible']:
            failure_count += 1
    if failure_count > 0:
        result_dict.update({
            'status': 'error'
            'message': 'could not retrieve required content'
        })
    else:
        result_dict.update({
            'status': 'ok'
            'message': 'required content retrieved'
        })
    yield result_dict
```

is_content_required()

This function is used in file-extractors only. It is a file-extractor counterpart to `get_required_content()`. Its purpose is to tell MetaLad whether the file content is required or not (relevant for annexed files - extraction may depend on file content, or require only annex key). If the function returns `True`, MetaLad will get the file content. If it returns `False`, the get operation will not be performed.

extract()

This function is used for actual metadata extraction. It has one parameter called `output_location`. If the output category of the extractor is `DataOutputCategory.IMMEDIATE`, this parameter will be `None`. If the output category is `DataOutputCategory.FILE`, this parameter will contain either a file name or a `file`-object into which the extractor can write its output.

The function should return an `datalad_metalad.extractors.base.ExtractorResult` object. The `ExtractorResult` is a dataclass object, containing the following fields:

- `extractor_version`: a version string representing the extractor's version.

- `extraction_parameter`: a dictionary containing parameters passed to the extractor by the calling command; can be obtained with: `self.parameter or {}`.
- `extraction_success`: either `True` or `False`.
- `datalad_result_dict`: a dictionary with entries added to the DataLad result record produced by a MetaLad calling command. Result records are used by DataLad to inform generic error handling and decisions on how to proceed with subsequent operations. MetaLad commands always set the mandatory result record fields `action` and `path`; the minimally useful set of fields which should by the extractor is `"status"` (one of: `"ok"`, `"notneeded"`, `"impossible"`, `"error"`) and `"type"` (`"dataset"` or `"file"`).
- `immediate_data` (a dictionary, optional): if the output category of the extractor is `IMMEDIATE`, then the `immediate_data` field should contain the result of the extraction process as a dictionary with freely-chosen keys. Contents of this dictionary should be JSON-serializable, because `datalad meta-extract` will print the JSON-serialized extractor result to standard output.

Example:

```
def extract(self, _=None) -> ExtractorResult:
    # Returns citation file content as metadata, altering only date

    # load file, guaranteed to be present
    with open(Path(self.dataset.path) / "CITATION.cff") as f:
        yamlContent = yaml.safe_load(f)

    # iso-format dates (nonexhaustive - publications have them too)
    if "date-released" in yamlContent:
        isodate = yamlContent["date-released"].isoformat()
        yamlContent["date-released"] = isodate

    return ExtractorResult(
        extractor_version=self.get_version(),
        extraction_parameter=self.parameter or {},
        extraction_success=True,
        datalad_result_dict={
            "type": "dataset",
            "status": "ok",
        },
        immediate_data=yamlContent,
    )
```

Passing runtime parameter to extractors

When an extractor is executed via `meta-extract`, you can pass runtime parameter to it. The runtime parameters are given as key-value pairs after the `EXTRACTOR_NAME`-parameter in dataset level extraction commands, or after the `FILE`-parameter in file-level extraction commands. Each key-value pair consists of two arguments, first the key, followed by the value.

The parameters are provided to dataset-level or file-level extractors in the extractor property `self.parameter`. The property contains a dictionary that holds the given key-value pairs.

For example, the following call:

```
datalad meta-extract -d . metalad_example_file README.md key1 value1 key2 value2
```

Will place the following dictionary in the `parameter` property of the extractor instance:

```
{'key1': 'value1', 'key2': 'value2'}
```

Please not, if dataset level extraction should be performed and you want to provide extractor parameter, you have to provide the `--force-dataset-level` parameter to ensure dataset-level extraction. i.e. to prevent `meta-extract` from interpreting the key of the first extractor argument as file name for a file-level extraction.

Please note also that only extractors that are derive from the classes `FileMetadataExtractor` or `DatasetMetadataExtractor` have a `parameter`-property and are able to read the parameters that are provided in the command line.

Use external programs for metadata extraction

Consider the situation where you have an external program, that is able to extract metadata from a dataset or a file. There might be many reasons, why you cannot create an equivalent extractor in Python. For example, the algorithm is unknown and you only have a binary version, a Python version might be too slow, you cannot afford the effort.

Metalad provides specific extractors that invoke external programs to perform extraction, i.e. `metalad_external_file` and `metalad_external_dataset`. Those extractors interact with external programs via standard input and standard output in order to query them, for example, for their UUID and their output category. The external programs are expected to support execution with one of the following parameters:

```
--get-uuid
--get-version
--get-data-output-category
```

In addition external file-level extractor programs must support:

```
--is-content-required
--extract <dataset-path> <dataset-ref-commit> <file-path> <dataset-relative-file-path>
```

and the external dataset-level extractor programs must support:

```
--get-required
--extract <dataset-path> <dataset-ref-commit>
```

Usually the external extractor has to be wrapped into a thin layer that provides the interface that is outlined above.

Making extractors discoverable

To be discovered by `meta_extract`, an extractor should be part of a DataLad extension. In addition, to make it discoverable, you need to declare an entry point in the extension's `setup.cfg` file. You can define the entrypoint name, and specify which extractor class it should point to. It is recommended to give the extractor name a prefix, to reduce the risk of name collisions.

Example:

```
[options.entry_points]
# (...)
datalad.metadata.extractors =
  hello_cff = datalad_helloworld.extractors.basic_dataset:CffExtractor
```

Tips

Using git methods to discover contents efficiently

Dataset-level extractors may need to check specific files to obtain information about specific files. If the files need to be listed, it may be more efficient to call git-ls-files or git-ls-tree instead of using pathlib methods (this limits the listing to files tracked by the dataset and helps avoid costly indexing if the *.git* directory). For example, a list of files with a given extension (including those in subfolders) can be created with:

```
files = list(self.dataset.repo.call_git_items_(["ls-files", "*.xyz"]))
```

1.4 Concepts and technologies

1.4.1 Design

The chapter describes the design of particular subsystems in DataLad.

Conduct

Specification scope and status

This specification provides an overview over meta-conduct concepts and its current implementation.

Purpose and Design

Meta-conduct allows to execute a number of metadata operations in a pipeline. Pipelines composed of:

- one *Provider*
- a number of *Processors*
- and an optional *Consumer*

The task of the provider is to supply the elements that the processors should operate on, for example, all files in a dataset. Processors perform certain operations on the provided elements, for example, extract metadata. And consumer will consume the data generated by processors. A consumer could for example add metadata to the metadata store of a dataset.

The provider, processors, and the consumer are described by a *Pipeline Definition* provided to conduct during invocation. The pipeline definition is a JSON-encoded description of the provider, processors and consumers that should be used.

If desired conduct will parallelize the execution of multiple processor pipelines. More precisely, it allows to execute processors in concurrent processes or threads (using python's concurrent module). Provider must yield elements that can be processed independently. Conduct will execute all providers in the order in which they are defined in the pipeline description on each element yielded by the provider. Conduct will parallelize the execution of those individual process-pipelines by default. Conduct will then hand the results to the consumer, of which only one exists, if any. That means the consumer will aggregate all results that were generated by the concurrently executed processor-pipelines.

(Note: you don't have to use a consumer to process results. An alternative would be to use a processor that finalizes the data processing, for example, by storing metadata in metadata stores.)

Data Handling

All results that are generated by the pipeline elements are collected in a *Pipeline Result*, indexed by the name of the pipeline elements that created them. Downstream elements are responsible for selecting the correct names.

Datatypes

Specification scope and status

This specification provides an overview over data types that are represented by classes in metalad.

Motivation for data types

Datalad, and historically metalad as well, use dictionaries to pass results internally and to report results externally (by converting the dictionaries into JSON-strings). Datalad code historically uses strings as dictionary keys, not constants. This approach has a few drawbacks:

1. Without proper documentation the expected and optional values are difficult to determine.
2. The direct access to elements through *string* is error prone.
3. It is not easy to determine which arguments a method accepts, or which keys a dictionary should have without additional documentation.

Action

Metalad therefore started to define a number of classes that represent the internally used objects in a more rigid way. Metalad uses custom classes or tools like `dataclasses`, `attr`, or `pydantic`.

This metadata-type classes are defined in `datalad_metalad.metadatatypes`. They are used for internal objects as well as for objects that are emitted as result of datalad operations like `meta-filter`. Currently the following metadata-type classes are defined:

- `Result`: a baseclass for all results, that are yielded in datalad API calls
- `MetadataResult`: a subclass of `Result`, that represents the result of a datalad API call that yields metadata
- `MetadataRecord`: a dataclass, which represents a metadata record in a metadata result

MetaLad development history and backward compatibility

Functionality related to metadata has been a part of the DataLad ecosystem from the very start. However, it underwent several evolutions, and this extension is the most recent state of it. If you have been an early adopter of the metadata functionalities of DataLad or MetaLad, this section provides an overview of past systems and notable changes for you to assess upgrades and backward-compatibility to legacy metadata.

First-generation metadata

The first generation of metadata commands was implemented in the main `datalad` Python package, but barely saw the light of day. Very early users of DataLad might have caught a glimpse of it.

In the 1st-gen metadata implementation, metadata of a dataset had two levels. The first one contained the metadata about the actual content of a dataset (generated by DataLad or other processes), the second one was metadata about the dataset itself (generated by DataLad). The metadata was represented in RDF.

Second-generation metadata

The second generation of metadata commands came to life when the main `datalad` package was a few years old already. It brought the concept of dedicated _extractors_, including the legacy extractors that are supported to this day. It also provided a range of dedicated metadata subcommands of a `datalad metadata` command such as `aggregate` and `extract`, as well as a dedicated `datalad search` command. Extracted metadata was stored in a dataset in (compressed) files using a JSON stream format, separately for metadata describing a dataset as a whole, and metadata describing individual files in a dataset.

The 2nd-gen metadata implementation was moved into the datalad-deprecated extension in 2022.

Third-generation metadata

The third generation of metadata commands was developed as the datalad-extension MetaLad. Initially, until version `0.2.1`, it was the continuation of developing 2nd generation metadata functionality. Afterwards, beginning with `0.3x` series, the metadata model and command set was once more revised into the current state 3rd-gen metadata implementation. This implementation came with an entirely new metadata model.

Gen 2 versus gen 3 metadata

This paragraph is important if you have used `datalad-metalad` prior to the `0.3.0` release.

Overview of changes

The new system in `0.3.0` is quite different from the previous release in a few ways:

1. Leaner commands with unix-style behavior, i.e. one command for one operation, and commands are chainable (use results from one command as input for another command, e.g. meta-extract|meta-add).
2. MetadataRecord modifications does not alter the state of the datalad dataset. In previous releases, changes to metadata have altered the version (commit-hash) of the repository although the primary data did not change. This is not the case in the new system. The new system does provide information about the primary data version, i.e. commit-hash, from which the individual metadata elements were created.
3. The ability to support a wide range of metadata storage backends in the future (this is facilitated by the [datalad-metadata-model](https://github.com/datalad/metadata-model)) which is developed alongside metalad), which separates the logical metadata model used in metalad from the storage backends, by abstracting the storage backend), Currently git-repository storage is supported.
4. The ability to transport metadata independently of the data in the dataset. The new system introduces the concept of a *metadata-store* which is usually the git-repository of the datalad dataset that is described by the metadata. But this is not a mandatory configuration, metadata can be stored in almost any git-repository.
5. The ability to report a subset of metadata from a remote metadata store without downloading the complete remote metadata. In fact only the minimal necessary information is transported from the remote metadata store. This ability is available to all metadata-based operations, for example, also to filtering.
6. A new simplified extractor model that distinguishes between two extractor-types: dataset-level extractors and file-extractors. The former are executed with a view on a dataset, the latter are executed with specific information

about a single file-path in the dataset. The previous extractors (datalad, and datalad-metalad<=0.2.1) are still supported.

7. A built-in pipeline mechanism that allows parallel execution of metadata operations like metadata extraction, and metadata filtering. (Still in early stage)
8. A new set of commands that allow operations that map metadata to metadata. Those operations are called filtering and are implemented by MetadataFilter-classes. Filter are dynamically loaded and custom filter are supports, much like extractors. (Still in early stage)

Backward-compatibility

Certain versions of MetaLad metadata are temporarily incompatible.

Note: Incompability of 0.3.0 and 0.2.x

Please note that the metadata storage format introduced in release `0.3.0` is incompatible with the metadata storage format in previous versions, i.e. *0.2.x*, and those in `datalad-deprecated`. Both storage formats can coexist in storage, but version `0.3.0` of MetaLad will not be able to read metadata that was stored by the previous version and vice versa. Eventually there will be an importer that will pull old-version metadata into the new metadata storage.

1.4.2 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 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

N

R

W

Y