Datalad Next

Release 1.3.0+74.g7e4c047.dirty

DataLad team

Apr 26, 2024

CONTENTS

1 Installation and usage 3

2 Functionality provided by DataLad NEXT 5

3 Developing with DataLad NEXT 253

4 Contributor information 255

5 Indices and tables 259

Python Module Index 261

Index 263

This DataLad extension can be thought of as a staging area for additional functionality, or for improved performance and user experience. Unlike other topical or more experimental extensions, the focus here is on functionality with broad applicability. This extension is a suitable dependency for other software packages that intend to build on this improved set of functionality.

CHAPTER

ONE

INSTALLATION AND USAGE

Install from PyPi or Github like any other Python package:

```
# create and enter a new virtual environment (optional)
$ virtualenv --python=python3 ~/env/dl-next
$ . ~/env/dl-next/bin/activate
# install from PyPi
$ python -m pip install datalad-next
```

Once installed, additional commands provided by this extension are immediately available. However, in order to fully benefit from all improvements, the extension has to be enabled for auto-loading by executing:

```
git config --global --add datalad.extensions.load next
```

Doing so will enable the extension to also alter the behavior the core DataLad package and its commands.

CHAPTER

TWO

FUNCTIONALITY PROVIDED BY DATALAD NEXT

The following table of contents offers entry points to the main components provided by this extension. The project README offers a more detailed summary in a different format.

2.1 High-level API commands

create_sibling_webdav(url, *[, dataset, ...])	Create a sibling(-tandem) on a WebDAV server
credentials([action, spec, name, prompt, ...])	Credential management and query
download(spec, *[, dataset, force, ...])	Download from URLs
ls_file_collection(type, collection, *[, hash])	Report information on files in a collection
next_status(*[, dataset, untracked, ...])	Report on the (modification) status of a dataset
tree([path, depth, recursive, ...])	Visualize directory and dataset hierarchies

2.1.1 datalad.api.create_sibling_webdav

datalad.api.**create_sibling_webdav**(*url*, *, *dataset=None*, *name=None*, *storage_name=None*, *mode='annex'*, *credential=None*, *existing='error'*, *recursive=False*, *recursion_limit=None*)

Create a sibling(-tandem) on a WebDAV server

WebDAV is a standard HTTP protocol extension for placing files on a server that is supported by a number of commercial storage services (e.g. 4shared.com, box.com), but also instances of cloud-storage solutions like Nextcloud or ownCloud. These software packages are also the basis for some institutional or public cloud storage solutions, such as EUDAT B2DROP.

For basic usage, only the URL with the desired dataset location on a WebDAV server needs to be specified for creating a sibling. However, the sibling setup can be flexibly customized (no storage sibling, or only a storage sibling, multi-version storage, or human-browsable single-version storage).

This command does not check for conflicting content on the WebDAV server!

When creating siblings recursively for a dataset hierarchy, subdataset exports are placed at their corresponding relative paths underneath the root location on the WebDAV server.

Collaboration on WebDAV siblings

The primary use case for WebDAV siblings is dataset deposition, where only one site is uploading dataset and file content updates. For collaborative workflows with multiple contributors, please make sure to consult the documentation on the underlying `datalad-annex::` Git remote helper for advice on appropriate setups: http://docs.datalad.org/projects/next/

Git-annex implementation details

Storage siblings are presently configured to NOT be enabled automatically on cloning a dataset. Due to a limitation of git-annex, this would initially fail (missing credentials). Instead, an explicit `datalad siblings enable --name <storage-sibling-name>` command must be executed after cloning. If necessary, it will prompt for credentials.

This command does not (and likely will not) support embedding credentials in the repository (see `embedcreds` option of the git-annex `webdav` special remote; https://git-annex.branchable.com/special_remotes/webdav), because such credential copies would need to be updated, whenever they change or expire. Instead, credentials are retrieved from DataLad's credential system. In many cases, credentials are determined automatically, based on the HTTP authentication realm identified by a WebDAV server.

This command does not support setting up encrypted remotes (yet). Neither for the storage sibling, nor for the regular Git-remote. However, adding support for it is primarily a matter of extending the API of this command, and passing the respective options on to the underlying git-annex setup.

This command does not support setting up chunking for webdav storage siblings (https://git-annex.branchable.com/chunking).

Examples

Create a WebDAV sibling tandem for storage of a dataset's file content and revision history. A user will be prompted for any required credentials, if they are not yet known.:

```
> create_sibling_webdav(url='https://webdav.example.com/myds')
```

Such a dataset can be cloned by DataLad via a specially crafted URL. Again, credentials are automatically determined, or a user is prompted to enter them:

```
> clone('datalad-annex::?type=webdav&encryption=none&url=https://webdav.example.com/
↪myds')
```

A sibling can also be created with a human-readable file tree, suitable for data exchange with non-DataLad users, but only able to host a single version of each file:

```
> create_sibling_webdav(url='https://example.com/browseable', mode='filetree')
```

Cloning such dataset siblings is possible via a convenience URL:

```
> clone('webdavs://example.com/browseable')
```

In all cases, the storage sibling needs to explicitly enabled prior to file content retrieval:

```
> siblings('enable', name='example.com-storage')
```

Parameters

- **`url`** -- URL identifying the sibling root on the target WebDAV server.
- **`dataset`** -- specify the dataset to process. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **`name`** -- name of the sibling. If none is given, the hostname-part of the WebDAV URL will be used. With *recursive*, the same name will be used to label all the subdatasets' siblings. [Default: None]

- **storage_name** -- name of the storage sibling (git-annex special remote). Must not be identical to the sibling name. If not specified, defaults to the sibling name plus '-storage' suffix. If only a storage sibling is created, this setting is ignored, and the primary sibling name is used. [Default: None]
- **mode** -- Siblings can be created in various modes: full-featured sibling tandem, one for a dataset's Git history and one storage sibling to host any number of file versions ('annex'). A single sibling for the Git history only ('git-only'). A single annex sibling for multi- version file storage only ('annex-only'). As an alternative to the standard (annex) storage sibling setup that is capable of storing any number of historical file versions using a content hash layout ('annex'|'annex-only'), the 'filetree' mode can used. This mode offers a human-readable data organization on the WebDAV remote that matches the file tree of a dataset (branch). However, it can, consequently, only store a single version of each file in the file tree. This mode is useful for depositing a single dataset snapshot for consumption without DataLad. The 'filetree' mode nevertheless allows for cloning such a single-version dataset, because the full dataset history can still be pushed to the WebDAV server. Git history hosting can also be turned off for this setup ('filetree- only'). When both a storage sibling and a regular sibling are created together, a publication dependency on the storage sibling is configured for the regular sibling in the local dataset clone. [Default: 'annex']
- **credential** -- name of the credential providing a user/password credential to be used for authorization. The credential can be supplied via configuration setting 'datalad.credential.<name>.user|secret', or environment variable DATALAD_CREDENTIAL_<NAME>_USER|SECRET, or will be queried from the active credential store using the provided name. If none is provided, the last-used credential for the authentication realm associated with the WebDAV URL will be used. Only if a credential name was given, it will be encoded in the URL of the created WebDAV Git remote, credential auto-discovery will be performed on each remote access. [Default: None]
- **existing** -- action to perform, if a (storage) sibling is already configured under the given name. In this case, sibling creation can be skipped ('skip') or the sibling (re-)configured ('reconfigure') in the dataset, or the command be instructed to fail ('error'). [Default: 'error']
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [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 tem-

plate (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']

2.1.2 datalad.api.credentials

datalad.api.**credentials**(*action='query'*, *spec=None*, ***, *name=None*, *prompt=None*, *dataset=None*)

Credential management and query

This command enables inspection and manipulation of credentials used throughout DataLad.

The command provides four basic actions:

QUERY

When executed without any property specification, all known credentials with all their properties will be yielded. Please note that this may not include credentials that only comprise of a secret and no other properties, or legacy credentials for which no trace in the configuration can be found. Therefore, the query results are not guaranteed to contain all credentials ever configured by DataLad.

When additional property/value pairs are specified, only credentials that have matching values for all given properties will be reported. This can be used, for example, to discover all suitable credentials for a specific "realm", if credentials were annotated with such information.

SET

This is the companion to 'get', and can be used to store properties and secret of a credential. Importantly, and in contrast to a 'get' operation, given properties with no values indicate a removal request. Any matching properties on record will be removed. If a credential is to be stored for which no secret is on record yet, an interactive session will prompt a user for a manual secret entry.

Only changed properties will be contained in the result record.

The appearance of the interactive secret entry can be configured with the two settings *datalad.credentials.repeat-secret-entry* and *datalad.credentials.hidden-secret-entry*.

REMOVE

This action will remove any secret and properties associated with a credential identified by its name.

GET (plumbing operation)

This is a *read-only* action that will never store (updates of) credential properties or secrets. Given properties will amend/overwrite those already on record. When properties with no value are given, and also no value for the

respective properties is on record yet, their value will be requested interactively, if a `prompt` text was provided too. This can be used to ensure a complete credential record, comprising any number of properties.

Details on credentials

A credential comprises any number of properties, plus exactly one secret. There are no constraints on the format or property values or the secret, as long as they are encoded as a string.

Credential properties are normally stored as configuration settings in a user's configuration ('global' scope) using the naming scheme:

datalad.credential.<name>.<property>

Therefore both credential name and credential property name must be syntax-compliant with Git configuration items. For property names this means only alphanumeric characters and dashes. For credential names virtually no naming restrictions exist (only null-byte and newline are forbidden). However, when naming credentials it is recommended to use simple names in order to enable convenient one-off credential overrides by specifying DataLad configuration items via their environment variable counterparts (see the documentation of the `configuration` command for details. In short, avoid underscores and special characters other than '.' and '-'.

While there are no constraints on the number and nature of credential properties, a few particular properties are recognized on used for particular purposes:

- 'secret': always refers to the single secret of a credential
- 'type': identifies the type of a credential. With each standard type, a list of mandatory properties is associated (see below)
- 'last-used': is an ISO 8601 format time stamp that indicated the last (successful) usage of a credential

Standard credential types and properties

The following standard credential types are recognized, and their mandatory field with their standard names will be automatically included in a 'get' report.

- 'user_password': with properties 'user', and the password as secret
- 'token': only comprising the token as secret
- 'aws-s3': with properties 'key-id', 'session', 'expiration', and the secret_id as the credential secret

Legacy support

DataLad credentials not configured via this command may not be fully discoverable (i.e., including all their properties). Discovery of such legacy credentials can be assisted by specifying a dedicated 'type' property.

Examples

Report all discoverable credentials:

```
> credentials()
```

Set a new credential mycred & input its secret interactively:

```
> credentials('set', name='mycred')
```

Remove a credential's type property:

```
> credentials('set', name='mycred', spec={'type': None})
```

Get all information on a specific credential in a structured record:

```
> credentials('get', name='mycred')
```

Upgrade a legacy credential by annotating it with a 'type' property:

```
> credentials('set', name='legacycred', spec={'type': 'user_password')
```

Set a new credential of type user_password, with a given user property, and input its secret interactively:

```
> credentials('set', name='mycred', spec={'type': 'user_password', 'user': '
↪<username>'})
```

Obtain a (possibly yet undefined) credential with a minimum set of properties. All missing properties and secret will be prompted for, no information will be stored! This is mostly useful for ensuring availability of an appropriate credential in an application context:

```
> credentials('get', prompt='Can I haz info plz?', name='newcred', spec={
↪'newproperty': None})
```

Parameters

- **action** -- which action to perform. [Default: 'query']
- **spec** -- specification of credential properties. Properties are given as name/value pairs. Properties with a *None* value indicate a property to be deleted (action 'set'), or a property to be entered interactively, when no value is set yet, and a prompt text is given (action 'get'). All property names are case-insensitive, must start with a letter or a digit, and may only contain '-' apart from these characters. Property specifications should be given a as dictionary, e.g., spec={'type': 'user_password'}. However, a CLI-like list of string arguments is also supported, e.g., spec=['type=user_password']. [Default: None]
- **name** -- name of a credential to set, get, or remove. [Default: None]
- **prompt** -- message to display when entry of missing credential properties is required for action 'get'. This can be used to present information on the nature of a credential and for instructions on how to obtain a credential. [Default: None]
- **dataset** -- specify a dataset whose configuration to inspect rather than the global (user) settings. [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']

2.1.3 datalad.api.download

datalad.api.**download**(*spec*, *, *dataset=None*, *force=None*, *credential=None*, *hash=None*)

Download from URLs

This command is the front-end to an extensible framework for performing downloads from a variety of URL schemes. Built-in support for the schemes 'http', 'https', 'file', and 'ssh' is provided. Extension packages may add additional support.

In contrast to other downloader tools, this command integrates with the DataLad credential management and is able to auto-discover credentials. If no credential is available, it automatically prompts for them, and offers to store them for reuse after a successful authentication.

Simultaneous hashing (checksumming) of downloaded content is supported with user-specified algorithms.

The command can process any number of downloads (serially). it can read download specifications from (command line) arguments, files, or STDIN. It can deposit downloads to individual files, or stream to STDOUT.

Implementation and extensibility

Each URL scheme is processed by a dedicated handler. Additional schemes can be supported by sub-classing datalad_next.url_operations.UrlOperations and implementing the *download()* method. Extension packages can register new handlers, by patching them into the *datalad_next.download._urlscheme_handlers* registry dict.

Examples

Download webpage to "myfile.txt":

```
> download({"http://example.com": "myfile.txt"})
```

Read download specification from STDIN (e.g. JSON-lines):

```
> download("-")
```

Simultaneously hash download, hexdigest reported in result record:

```
> download("http://example.com/data.xml", hash=["sha256"])
```

Download from SSH server:

```
> download("ssh://example.com/home/user/data.xml")
```

Parameters

- **spec** -- Download sources and targets can be given in a variety of formats: as a URL, or as a URL-path-pair that is mapping a source URL to a dedicated download target path. Any number of URLs or URL-path-pairs can be provided, either as an argument list, or read from a file (one item per line). Such a specification input file can be given as a path to an existing file (as a single value, not as part of a URL- path-pair). When the special path identifier '-' is used, the download is written to STDOUT. A specification can also be read in JSON-lines encoding (each line being a string with a URL or an object mapping a URL-string to a path-string). In addition, specifications can also be given as a list or URLs, or as a list of dicts with a URL to path mapping. Paths are supported in string form, or as *Path* objects.
- **dataset** -- Dataset to be used as a configuration source. Beyond reading configuration items, this command does not interact with the dataset. [Default: None]
- **force** -- By default, a target path for a download must not exist yet. 'force- overwrite' disabled this check. [Default: None]
- **credential** -- name of a credential to be used for authorization. If no credential is identified, the last-used credential for the authentication realm associated with the download target will be used. If there is no credential available yet, it will be prompted for. Once used successfully, a prompt for entering to save such a new credential will be presented. [Default: None]
- **hash** -- Name of a hashing algorithm supported by the Python 'hashlib' module, e.g. 'md5' or 'sha256'. [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']

2.1.4 datalad.api.ls_file_collection

datalad.api.**ls_file_collection**(*type: str*, *collection: CollectionSpec*, *, *hash: str | List[str] | None = None*)

Report information on files in a collection

This is a utility that can be used to query information on files in different file collections. The type of information reported varies across collection types. However, each result at minimum contains some kind of identifier for the collection ('collection' property), and an identifier for the respective collection item ('item' property). Each result also contains a `type` property that indicates particular type of file that is being reported on. In most cases this will be `file`, but other categories like `symlink` or `directory` are recognized too.

If a collection type provides file-access, this command can compute one or more hashes (checksums) for any file in a collection.

Supported file collection types are:

directory
: Reports on the content of a given directory (non-recursively). The collection identifier is the path of the directory. Item identifiers are the names of items within that directory. Standard properties like `size`, `mtime`, or `link_target` are included in the report. When hashes are computed, an `fp` property with a file-like is provided. Reading file data from it requires a `seek(0)` in most cases. This file handle is only open when items are yielded directly by this command (`return_type='generator`) and only until the next result is yielded.

gittree
: Reports on the content of a Git "tree-ish". The collection identifier is that tree-ish. The command must be executed inside a Git repository. If the working directory for the command is not the repository root (in case of a non-bare repository), the report is constrained to items underneath the working directory. Item identifiers are the relative paths of items within that working directory. Reported properties include `gitsha` and `gittype`; note that the `gitsha` is not equivalent to a SHA1 hash of a file's content, but is the SHA-type blob identifier as reported and used by Git. Reporting of content hashes beyond the `gitsha` is presently not supported.

gitworktree
: Reports on all tracked and untracked content of a Git repository's work tree. The collection identifier is a path of a directory in a Git repository (which can, but needs not be, its root). Item identifiers are the relative paths of items within that directory. Reported properties include `gitsha` and `gittype`; note that the `gitsha` is not equivalent to a SHA1 hash of a file's content, but is the SHA-type blob identifier as reported and used by Git. When hashes are computed, an `fp` property with a file-like is provided. Reading file data from it requires a `seek(0)` in most cases. This file handle is only open when items are yielded directly by this command (`return_type='generator`) and only until the next result is yielded.

annexworktree
: Like `gitworktree`, but amends the reported items with git-annex information, such as `annexkey`,

annexsize, and annnexobjpath.

tarfile
Reports on members of a TAR archive. The collection identifier is the path of the TAR file. Item identifiers are the relative paths of archive members within the archive. Reported properties are similar to the directory collection type. When hashes are computed, an fp property with a file-like is provided. Reading file data from it requires a seek(0) in most cases. This file handle is only open when items are yielded directly by this command (return_type='generator') and only until the next result is yielded.

zipfile
Like tarfile for reporting on ZIP archives.

Examples

Report on the content of a directory:

```
> records = ls_file_collection("directory", "/tmp")
```

Report on the content of a TAR archive with MD5 and SHA1 file hashes:

```
> records = ls_file_collection("tarfile", "myarchive.tar.gz", hash=["md5", "sha1"])
```

List annex keys of all files in the working tree of a dataset:

```
> [r['annexkey'] \
  for r in ls_file_collection('annexworktree', '.') \
  if 'annexkey' in r]
```

Parameters

- **type** -- Name of the type of file collection to report on.
- **collection** -- identifier or location of the file collection to report on. Depending on the type of collection to process, the specific nature of this parameter can be different. A common identifier for a file collection is a path (to a directory, to an archive), but might also be a URL. See the documentation for details on supported collection types.
- **hash** -- One or more names of algorithms to be used for reporting file hashes. They must be supported by the Python 'hashlib' module, e.g. 'md5' or 'sha256'. Reporting file hashes typically implies retrieving/reading file content. This processing may also enable reporting of additional properties that may otherwise not be readily available. [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']

2.1.5 datalad.api.next_status

datalad.api.**next_status**(*, *dataset=None*, *untracked='normal'*, *recursive='repository'*, *eval_subdataset_state='full'*) → Generator[*StatusResult*, None, None] | list[*StatusResult*]

Report on the (modification) status of a dataset

Note: This is a preview of an command implementation aiming to replace the DataLad `status` command.

For now, expect anything here to change again.

This command provides a report that is roughly identical to that of `git status`. Running with default parameters yields a report that should look familiar to Git and DataLad users alike, and contain the same information as offered by `git status`.

The main difference to `git status` are:

- Support for recursion into submodule. `git status` does that too, but the report is limited to the global state of an entire submodule, whereas this command can issue detailed reports in changes inside a submodule (any nesting depth).
- Support for directory-constrained reporting. Much like `git status` limits its report to a single repository, this command can optionally limit its report to a single directory and its direct children. In this report subdirectories are considered containers (much like) submodules, and a change summary is provided for them.
- Support for a "mono" (monolithic repository) report. Unlike a standard recursion into submodules, and checking each of them for changes with respect to the HEAD commit of the worktree, this report compares a submodule with respect to the state recorded in its parent repository. This provides an equally comprehensive status report from the point of view of a queried repository, but does not include a dedicated item

on the global state of a submodule. This makes nested hierarchy of repositories appear like a single (mono) repository.

- Support for "adjusted mode" git-annex repositories. These utilize a managed branch that is repeatedly rewritten, hence is not suitable for tracking within a parent repository. Instead, the underlying "corresponding branch" is used, which contains the equivalent content in an un-adjusted form, persistently. This command detects this condition and automatically check a repositories state against the corresponding branch state.

Presently missing/planned features

- There is no support for specifying paths (or pathspecs) for constraining the operation to specific dataset parts. This will be added in the future.
- There is no reporting of git-annex properties, such as tracked file size. It is undetermined whether this will be added in the future. However, even without a dedicated switch, this command has support for datasets (and their submodules) in git-annex's "adjusted mode".

Differences to the ``status`` command implementation prior DataLad v2

- Like `git status` this implementation reports on dataset modification, whereas the previous `status` also provided a listing of unchanged dataset content. This is no longer done. Equivalent functionality for listing dataset content is provided by the `ls_file_collection` command.
- The implementation is substantially faster. Depending on the context the speed-up is typically somewhere between 2x and 100x.
- The implementation does not suffer from the limitation re type change detection.
- Python and CLI API of the command use uniform parameter validation.

Parameters

- **`dataset`** -- Dataset to be used as a configuration source. Beyond reading configuration items, this command does not interact with the dataset. [Default: None]
- **`untracked`** -- Determine how untracked content is considered and reported when comparing a revision to the state of the working tree. 'no': no untracked content is considered as a change; 'normal': untracked files and entire untracked directories are reported as such; 'all': report individual files even in fully untracked directories. In addition to these git-status modes, 'whole-dir' (like normal, but include empty directories), and 'no-empty-dir' (alias for 'normal') are understood. [Default: 'normal']
- **`recursive`** -- Mode of recursion for status reporting. With 'no' the report is restricted to a single directory and its direct children. With 'repository', the report comprises all repository content underneath current working directory or root of a given dataset, but is limited to items directly contained in that repository. With 'datasets', the report also comprises any content in any subdatasets. Each subdataset is evaluated against its respective HEAD commit. With 'mono', a report similar to 'datasets' is generated, but any subdataset is evaluate with respect to the state recorded in its parent repository. In contrast to the 'datasets' mode, no report items on a joint submodule are generated. [Default: 'repository']
- **`eval_subdataset_state`** -- Evaluation of subdataset state (modified or untracked content) can be expensive for deep dataset hierarchies as subdataset have to be tested recursively for uncommitted modifications. Setting this option to 'no' or 'commit' can substantially boost performance by limiting what is being tested. With 'no' no state is evaluated and subdataset are not investigated for modifications. With 'commit' only a discrepancy of the HEAD commit gitsha of a subdataset and the gitsha recorded in the superdataset's record is evaluated. With 'full' any other modifications are considered too. [Default: 'full']

- **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']

2.1.6 datalad.api.tree

datalad.api.**tree**(*path='.'*, ***, *depth=None*, *recursive=False*, *recursion_limit=None*, *include_files=False*, *include_hidden=False*)

Visualize directory and dataset hierarchies

This command mimics the UNIX/MS-DOS 'tree' utility to generate and display a directory tree, with DataLad-specific enhancements.

It can serve the following purposes:

1. Glorified 'tree' command
2. Dataset discovery
3. Programmatic directory traversal

Glorified 'tree' command

The rendered command output uses 'tree'-style visualization:

```
/tmp/mydir
├── [DS~0] ds_A/
│   └── [DS~1] subds_A/
└── [DS~0] ds_B/
    ├── dir_B/
    │   ├── file.txt
    │   ├── subdir_B/
    │   └── [DS~1] subds_B0/
    └── [DS~1] (not installed) subds_B1/

5 datasets, 2 directories, 1 file
```

Dataset paths are prefixed by a marker indicating subdataset hierarchy level, like `[DS~1]`. This is the absolute subdataset level, meaning it may also take into account superdatasets located above the tree root and thus not included in the output. If a subdataset is registered but not installed (such as after a non-recursive `datalad clone`), it will be prefixed by `(not installed)`. Only DataLad datasets are considered, not pure git/git-annex repositories.

The 'report line' at the bottom of the output shows the count of displayed datasets, in addition to the count of directories and files. In this context, datasets and directories are mutually exclusive categories.

By default, only directories (no files) are included in the tree, and hidden directories are skipped. Both behaviours can be changed using command options.

Symbolic links are always followed. This means that a symlink pointing to a directory is traversed and counted as a directory (unless it potentially creates a loop in the tree).

Dataset discovery

Using the `recursive` or `recursion_limit` option, this command generates the layout of dataset hierarchies based on subdataset nesting level, regardless of their location in the filesystem.

In this case, tree depth is determined by subdataset depth. This mode is thus suited for discovering available datasets when their location is not known in advance.

By default, only datasets are listed, without their contents. If `depth` is specified additionally, the contents of each dataset will be included up to `depth` directory levels (excluding subdirectories that are themselves datasets).

Tree filtering options such as `include_hidden` only affect which directories are reported as dataset contents, not which directories are traversed to find datasets.

Performance note: since no assumption is made on the location of datasets, running this command with the `recursive` or `recursion_limit` option does a full scan of the whole directory tree. As such, it can be significantly slower than a call with an equivalent output that uses `depth` to limit the tree instead.

Programmatic directory traversal

The command yields a result record for each tree node (dataset, directory or file). The following properties are reported, where available:

"path"
: Absolute path of the tree node

"type"
: Type of tree node: "dataset", "directory" or "file"

"depth"
: Directory depth of node relative to the tree root

"exhausted_levels"
Depth levels for which no nodes are left to be generated (the respective subtrees have been 'exhausted')

"count"
Dict with cumulative counts of datasets, directories and files in the tree up until the current node. File count is only included if the command is run with the `include_files` option.

"dataset_depth"
Subdataset depth level relative to the tree root. Only included for node type "dataset".

"dataset_abs_depth"
Absolute subdataset depth level. Only included for node type "dataset".

"dataset_is_installed"
Whether the registered subdataset is installed. Only included for node type "dataset".

"symlink_target"
If the tree node is a symlink, the path to the link target

"is_broken_symlink"
If the tree node is a symlink, whether it is a broken symlink

Examples

Show up to 3 levels of subdirectories below the current directory, including files and hidden contents:

```
> tree(depth=3, include_files=True, include_hidden=True)
```

Find all top-level datasets located anywhere under `/tmp`:

```
> tree('/tmp', recursion_limit=0)
```

Report all subdatasets recursively and their directory contents, up to 1 subdirectory deep within each dataset:

```
> tree(recursive=True, depth=1)
```

Parameters

- **path** -- path to directory from which to generate the tree. Defaults to the current directory. [Default: '.']
- **depth** -- limit the tree to maximum level of subdirectories. If not specified, will generate the full tree with no depth constraint. If paired with `recursive` or `recursion_limit`, refers to the maximum directory level to output below each dataset. [Default: None]
- **recursive** (*bool, optional*) -- produce a dataset tree of the full hierarchy of nested subdatasets. *Note*: may have slow performance on large directory trees. [Default: False]
- **recursion_limit** -- limit the dataset tree to maximum level of nested subdatasets. 0 means include only top-level datasets, 1 means top-level datasets and their immediate subdatasets, etc. *Note*: may have slow performance on large directory trees. [Default: None]
- **include_files** (*bool, optional*) -- include files in the tree. [Default: False]
- **include_hidden** (*bool, optional*) -- include hidden files/directories in the tree. This option does not affect which directories will be searched for datasets when specifying `recursive` or `recursion_limit`. For example, datasets located underneath the hidden folder *.datalad* will be reported even if `include_hidden` is omitted. [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']

2.2 Command line reference

2.2.1 datalad create-sibling-webdav

Synopsis

```
datalad create-sibling-webdav [-h] [-d DATASET] [-s NAME] [--storage-name NAME] [--mode␣
↪MODE] [--credential NAME] [--existing EXISTING] [-r] [-R LEVELS] [--version] URL
```

Description

Create a sibling(-tandem) on a WebDAV server

WebDAV is a standard HTTP protocol extension for placing files on a server that is supported by a number of commercial storage services (e.g. 4shared.com, box.com), but also instances of cloud-storage solutions like Nextcloud or ownCloud. These software packages are also the basis for some institutional or public cloud storage solutions, such as EUDAT B2DROP.

For basic usage, only the URL with the desired dataset location on a WebDAV server needs to be specified for creating a sibling. However, the sibling setup can be flexibly customized (no storage sibling, or only a storage sibling, multi-version storage, or human-browsable single-version storage).

This command does not check for conflicting content on the WebDAV server!

When creating siblings recursively for a dataset hierarchy, subdataset exports are placed at their corresponding relative paths underneath the root location on the WebDAV server.

Collaboration on WebDAV siblings

The primary use case for WebDAV siblings is dataset deposition, where only one site is uploading dataset and file content updates. For collaborative workflows with multiple contributors, please make sure to consult the documentation on the underlying `datalad-annex::` Git remote helper for advice on appropriate setups: http://docs.datalad.org/projects/next/

Git-annex implementation details

Storage siblings are presently configured to NOT be enabled automatically on cloning a dataset. Due to a limitation of git-annex, this would initially fail (missing credentials). Instead, an explicit `datalad siblings enable --name <storage-sibling-name>` command must be executed after cloning. If necessary, it will prompt for credentials.

This command does not (and likely will not) support embedding credentials in the repository (see `embedcreds` option of the git-annex `webdav` special remote; https://git-annex.branchable.com/special_remotes/webdav), because such credential copies would need to be updated, whenever they change or expire. Instead, credentials are retrieved from DataLad's credential system. In many cases, credentials are determined automatically, based on the HTTP authentication realm identified by a WebDAV server.

This command does not support setting up encrypted remotes (yet). Neither for the storage sibling, nor for the regular Git-remote. However, adding support for it is primarily a matter of extending the API of this command, and passing the respective options on to the underlying git-annex setup.

This command does not support setting up chunking for webdav storage siblings (https://git-annex.branchable.com/chunking).

Examples

Create a WebDAV sibling tandem for storage of a dataset's file content and revision history. A user will be prompted for any required credentials, if they are not yet known.:

```
% datalad create-sibling-webdav "https://webdav.example.com/myds"
```

Such a dataset can be cloned by DataLad via a specially crafted URL. Again, credentials are automatically determined, or a user is prompted to enter them:

```
% datalad clone "datalad-annex::?type=webdav&encryption=none&url=https://webdav.example.
↪com/myds"
```

A sibling can also be created with a human-readable file tree, suitable for data exchange with non-DataLad users, but only able to host a single version of each file:

```
% datalad create-sibling-webdav --mode filetree "https://example.com/browsable"
```

Cloning such dataset siblings is possible via a convenience URL:

```
% datalad clone "webdavs://example.com/browsable"
```

In all cases, the storage sibling needs to explicitly enabled prior to file content retrieval:

```
% datalad siblings enable --name example.com-storage
```

Options

URL

URL identifying the sibling root on the target WebDAV server.

-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*

specify the dataset to process. If no dataset is given, an attempt is made to identify the dataset based on the current working directory.

-s NAME, --name NAME

name of the sibling. If none is given, the hostname-part of the WebDAV URL will be used. With RECURSIVE, the same name will be used to label all the subdatasets' siblings.

--storage-name NAME

name of the storage sibling (git-annex special remote). Must not be identical to the sibling name. If not specified, defaults to the sibling name plus '-storage' suffix. If only a storage sibling is created, this setting is ignored, and the primary sibling name is used.

--mode *MODE*

Siblings can be created in various modes: full-featured sibling tandem, one for a dataset's Git history and one storage sibling to host any number of file versions ('annex'). A single sibling for the Git history only ('git-only'). A single annex sibling for multi-version file storage only ('annex-only'). As an alternative to the standard (annex) storage sibling setup that is capable of storing any number of historical file versions using a content hash layout ('annex'|'annex-only'), the 'filetree' mode can used. This mode offers a human- readable data organization on the WebDAV remote that matches the file tree of a dataset (branch). However, it can, consequently, only store a single version of each file in the file tree. This mode is useful for depositing a single dataset snapshot for consumption without DataLad. The 'filetree' mode nevertheless allows for cloning such a single-version dataset, because the full dataset history can still be pushed to the WebDAV server. Git history hosting can also be turned off for this setup ('filetree-only'). When both a storage sibling

and a regular sibling are created together, a publication dependency on the storage sibling is configured for the regular sibling in the local dataset clone. [Default: 'annex']

--credential NAME

name of the credential providing a user/password credential to be used for authorization. The credential can be supplied via configuration setting 'datalad.credential.<name>.user|secret', or environment variable DATALAD_CREDENTIAL_<NAME>_USER|SECRET, or will be queried from the active credential store using the provided name. If none is provided, the last-used credential for the authentication realm associated with the WebDAV URL will be used. Only if a credential name was given, it will be encoded in the URL of the created WebDAV Git remote, credential auto-discovery will be performed on each remote access.

--existing *EXISTING*

action to perform, if a (storage) sibling is already configured under the given name. In this case, sibling creation can be skipped ('skip') or the sibling (re-)configured ('reconfigure') in the dataset, or the command be instructed to fail ('error'). [Default: 'error']

-r, --recursive

if set, recurse into potential subdatasets.

-R LEVELS, --recursion-limit LEVELS

limit recursion into subdatasets to the given number of levels. Constraints: value must be convertible to type 'int' 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>.

2.2.2 datalad credentials

Synopsis

```
datalad credentials [-h] [--prompt PROMPT] [-d DATASET] [--version] [action] [[name]␣
↪[:]property[=value] ...]
```

Description

Credential management and query

This command enables inspection and manipulation of credentials used throughout DataLad.

The command provides four basic actions:

QUERY

When executed without any property specification, all known credentials with all their properties will be yielded. Please note that this may not include credentials that only comprise of a secret and no other properties, or legacy credentials for which no trace in the configuration can be found. Therefore, the query results are not guaranteed to contain all credentials ever configured by DataLad.

When additional property/value pairs are specified, only credentials that have matching values for all given properties will be reported. This can be used, for example, to discover all suitable credentials for a specific "realm", if credentials were annotated with such information.

SET

This is the companion to 'get', and can be used to store properties and secret of a credential. Importantly, and in contrast to a 'get' operation, given properties with no values indicate a removal request. Any matching properties on record will be removed. If a credential is to be stored for which no secret is on record yet, an interactive session will prompt a user for a manual secret entry.

Only changed properties will be contained in the result record.

The appearance of the interactive secret entry can be configured with the two settings *datalad.credentials.repeat-secret-entry* and *datalad.credentials.hidden-secret-entry*.

REMOVE

This action will remove any secret and properties associated with a credential identified by its name.

GET (plumbing operation)

This is a *read-only* action that will never store (updates of) credential properties or secrets. Given properties will amend/overwrite those already on record. When properties with no value are given, and also no value for the respective properties is on record yet, their value will be requested interactively, if a `--prompt` text was provided too. This can be used to ensure a complete credential record, comprising any number of properties.

Details on credentials

A credential comprises any number of properties, plus exactly one secret. There are no constraints on the format or property values or the secret, as long as they are encoded as a string.

Credential properties are normally stored as configuration settings in a user's configuration ('global' scope) using the naming scheme:

datalad.credential.<name>.<property>

Therefore both credential name and credential property name must be syntax-compliant with Git configuration items. For property names this means only alphanumeric characters and dashes. For credential names virtually no naming restrictions exist (only null-byte and newline are forbidden). However, when naming credentials it is recommended to use simple names in order to enable convenient one-off credential overrides by specifying DataLad configuration items via their environment variable counterparts (see the documentation of the `configuration` command for details. In short, avoid underscores and special characters other than '.' and '-'.

While there are no constraints on the number and nature of credential properties, a few particular properties are recognized on used for particular purposes:

- 'secret': always refers to the single secret of a credential

- 'type': identifies the type of a credential. With each standard type, a list of mandatory properties is associated (see below)
- 'last-used': is an ISO 8601 format time stamp that indicated the last (successful) usage of a credential

Standard credential types and properties

The following standard credential types are recognized, and their mandatory field with their standard names will be automatically included in a 'get' report.

- 'user_password': with properties 'user', and the password as secret
- 'token': only comprising the token as secret
- 'aws-s3': with properties 'key-id', 'session', 'expiration', and the secret_id as the credential secret

Legacy support

DataLad credentials not configured via this command may not be fully discoverable (i.e., including all their properties). Discovery of such legacy credentials can be assisted by specifying a dedicated 'type' property.

Examples

Report all discoverable credentials:

```
% datalad credentials
```

Set a new credential mycred & input its secret interactively:

```
% datalad credentials set mycred
```

Remove a credential's type property:

```
% datalad credentials set mycred :type
```

Get all information on a specific credential in a structured record:

```
% datalad -f json credentials get mycred
```

Upgrade a legacy credential by annotating it with a 'type' property:

```
% datalad credentials set legacycred type=user_password
```

Set a new credential of type user_password, with a given user property, and input its secret interactively:

```
% datalad credentials set mycred type=user_password user=<username>
```

Obtain a (possibly yet undefined) credential with a minimum set of properties. All missing properties and secret will be prompted for, no information will be stored! This is mostly useful for ensuring availability of an appropriate credential in an application context:

```
% datalad credentials --prompt 'can I haz info plz?' get newcred :newproperty
```

Options

action

which action to perform. [Default: 'query']

[name] [:]property[=value]

specification ofa credential name and credential properties. Properties are either given as name/value pairs or as a property name prefixed by a colon. Properties prefixed with a colon indicate a property to be deleted (action 'set'), or a property to be entered interactively, when no value is set yet, and a prompt text is given (action 'get'). All property names are case-insensitive, must start with a letter or a digit, and may only contain '-' apart from these characters.

-h, --help, --help-np

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

--prompt *PROMPT*

message to display when entry of missing credential properties is required for action 'get'. This can be used to present information on the nature of a credential and for instructions on how to obtain a credential.

-d *DATASET*, --dataset *DATASET*

specify a dataset whose configuration to inspect rather than the global (user) settings.

--version

show the module and its version which provides the command

Authors

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

2.2.3 datalad download

Synopsis

```
datalad download [-h] [-d DATASET] [--force {overwrite-existing}] [--credential NAME] [--
↪hash ALGORITHM] [--version] <path>|<url>|<url-path-pair> [<path>|<url>|<url-path-pair>␣
↪...]
```

Description

Download from URLs

This command is the front-end to an extensible framework for performing downloads from a variety of URL schemes. Built-in support for the schemes 'http', 'https', 'file', and 'ssh' is provided. Extension packages may add additional support.

In contrast to other downloader tools, this command integrates with the DataLad credential management and is able to auto-discover credentials. If no credential is available, it automatically prompts for them, and offers to store them for reuse after a successful authentication.

Simultaneous hashing (checksumming) of downloaded content is supported with user-specified algorithms.

The command can process any number of downloads (serially). it can read download specifications from (command line) arguments, files, or STDIN. It can deposit downloads to individual files, or stream to STDOUT.

Implementation and extensibility

Each URL scheme is processed by a dedicated handler. Additional schemes can be supported by sub-classing datalad_next.url_operations.UrlOperations and implementing the *download()* method. Extension packages can register new handlers, by patching them into the *datalad_next.download._urlscheme_handlers* registry dict.

Examples

Download webpage to "myfile.txt":

```
% datalad download "http://example.com myfile.txt"
```

Read download specification from STDIN (e.g. JSON-lines):

```
% datalad download -
```

Simultaneously hash download, hexdigest reported in result record:

```
% datalad download --hash sha256 http://example.com/data.xml"
```

Download from SSH server:

```
% datalad download "ssh://example.com/home/user/data.xml"
```

Stream a download to STDOUT:

```
% datalad -f disabled download "http://example.com -"
```

Options

<path>|<url>|<url-path-pair>

Download sources and targets can be given in a variety of formats: as a URL, or as a URL-path-pair that is mapping a source URL to a dedicated download target path. Any number of URLs or URL-path-pairs can be provided, either as an argument list, or read from a file (one item per line). Such a specification input file can be given as a path to an existing file (as a single value, not as part of a URL-path-pair). When the special path identifier '-' is used, the download is written to STDOUT. A specification can also be read in JSON-lines encoding (each line being a string with a URL or an object mapping a URL-string to a path-string).

-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 be used as a configuration source. Beyond reading configuration items, this command does not interact with the dataset.

--force {overwrite-existing}

By default, a target path for a download must not exist yet. 'force-overwrite' disabled this check.

--credential NAME

name of a credential to be used for authorization. If no credential is identified, the last-used credential for the authentication realm associated with the download target will be used. If there is no credential available yet, it will be prompted for. Once used successfully, a prompt for entering to save such a new credential will be presented.

--hash ALGORITHM

Name of a hashing algorithm supported by the Python 'hashlib' module, e.g. 'md5' or 'sha256'. This option can be given more than once.

--version

show the module and its version which provides the command

Authors

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

2.2.4 datalad ls-file-collection

Synopsis

```
datalad ls-file-collection [-h] [--hash ALGORITHM] [--version] {directory,tarfile,
↪zipfile,gittree,gitworktree,annexworktree} ID/LOCATION
```

Description

Report information on files in a collection

This is a utility that can be used to query information on files in different file collections. The type of information reported varies across collection types. However, each result at minimum contains some kind of identifier for the collection ('collection' property), and an identifier for the respective collection item ('item' property). Each result also contains a `type` property that indicates particular type of file that is being reported on. In most cases this will be `file`, but other categories like `symlink` or `directory` are recognized too.

If a collection type provides file-access, this command can compute one or more hashes (checksums) for any file in a collection.

Supported file collection types are:

`directory`
: Reports on the content of a given directory (non-recursively). The collection identifier is the path of the directory. Item identifiers are the names of items within that directory. Standard properties like `size`, `mtime`, or `link_target` are included in the report.

`gittree`
: Reports on the content of a Git "tree-ish". The collection identifier is that tree-ish. The command must be executed inside a Git repository. If the working directory for the command is not the repository root (in case of a non-bare repository), the report is constrained to items underneath the working directory. Item identifiers are the relative paths of items within that working directory. Reported properties include `gitsha` and `gittype`; note that the `gitsha` is not equivalent to a SHA1 hash of a file's content, but is the SHA-type blob identifier as reported and used by Git. Reporting of content hashes beyond the `gitsha` is presently not supported.

`gitworktree`
: Reports on all tracked and untracked content of a Git repository's work tree. The collection identifier is a path of a directory in a Git repository (which can, but needs not be, its root). Item identifiers are the relative paths of items within that directory. Reported properties include `gitsha` and `gittype`; note that the `gitsha` is not equivalent to a SHA1 hash of a file's content, but is the SHA-type blob identifier as reported and used by Git.

`annexworktree`
: Like `gitworktree`, but amends the reported items with git-annex information, such as `annexkey`, `annexsize`, and `annnexobjpath`.

`tarfile`
: Reports on members of a TAR archive. The collection identifier is the path of the TAR file. Item identifiers are the relative paths of archive members within the archive. Reported properties are similar to the `directory` collection type.

`zipfile`
: Like `tarfile` for reporting on ZIP archives.

Examples

Report on the content of a directory:

```
% datalad -f json ls-file-collection directory /tmp
```

Report on the content of a TAR archive with MD5 and SHA1 file hashes:

```
% datalad -f json ls-file-collection --hash md5 --hash sha1 tarfile myarchive.tar.gz
```

Register URLs for files in a directory that is also reachable via HTTP. This uses `ls-file-collection` for listing files and computing MD5 hashes, then using `jq` to filter and transform the output (just file records, and in a JSON array), and passes them to `addurls`, which generates annex keys/files and assigns URLs. When the command finishes, the dataset contains no data, but can retrieve the files after confirming their availability (i.e., via *git annex fsck*):

```
% datalad -f json ls-file-collection directory wwwdir --hash md5 \
  | jq '. | select(.type == "file")' \
  | jq --slurp . \
  | datalad addurls --key 'et:MD5-s{size}--{hash-md5}' - 'https://example.com/{item}'
```

List annex keys of all files in the working tree of a dataset:

```
% datalad -f json ls-file-collection annexworktree . \
  | jq '. | select(.annexkey) | .annexkey'
```

Options

{directory,tarfile,zipfile,gittree,gitworktree,annexworktree}

Name of the type of file collection to report on.

ID/LOCATION

identifier or location of the file collection to report on. Depending on the type of collection to process, the specific nature of this parameter can be different. A common identifier for a file collection is a path (to a directory, to an archive), but might also be a URL. See the documentation for details on supported collection types.

-h, --help, --help-np

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

--hash ALGORITHM

One or more names of algorithms to be used for reporting file hashes. They must be supported by the Python 'hashlib' module, e.g. 'md5' or 'sha256'. Reporting file hashes typically implies retrieving/reading file content. This processing may also enable reporting of additional properties that may otherwise not be readily available. This option can be given more than once.

--version

show the module and its version which provides the command

Authors

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

2.2.5 datalad next-status

Synopsis

```
datalad next-status [-h] [-d DATASET] [--untracked {no,whole-dir,no-empty-dir,normal,all}
↪] [-r [{no,repository,datasets,mono}]] [-e {no,commit,full}] [--version]
```

Description

Report on the (modification) status of a dataset

NOTE

> This is a preview of an command implementation aiming to replace the DataLad `status` command.
>
> For now, expect anything here to change again.

This command provides a report that is roughly identical to that of `git status`. Running with default parameters yields a report that should look familiar to Git and DataLad users alike, and contain the same information as offered by `git status`.

The main difference to `git status` are:

- Support for recursion into submodule. `git status` does that too, but the report is limited to the global state of an entire submodule, whereas this command can issue detailed reports in changes inside a submodule (any nesting depth).
- Support for directory-constrained reporting. Much like `git status` limits its report to a single repository, this command can optionally limit its report to a single directory and its direct children. In this report subdirectories are considered containers (much like) submodules, and a change summary is provided for them.
- Support for a "mono" (monolithic repository) report. Unlike a standard recursion into submodules, and checking each of them for changes with respect to the HEAD commit of the worktree, this report compares a submodule with respect to the state recorded in its parent repository. This provides an equally comprehensive status report from the point of view of a queried repository, but does not include a dedicated item on the global state of a submodule. This makes nested hierarchy of repositories appear like a single (mono) repository.
- Support for "adjusted mode" git-annex repositories. These utilize a managed branch that is repeatedly rewritten, hence is not suitable for tracking within a parent repository. Instead, the underlying "corresponding branch" is used, which contains the equivalent content in an un-adjusted form, persistently. This command detects this condition and automatically check a repositories state against the corresponding branch state.

Presently missing/planned features

- There is no support for specifying paths (or pathspecs) for constraining the operation to specific dataset parts. This will be added in the future.
- There is no reporting of git-annex properties, such as tracked file size. It is undetermined whether this will be added in the future. However, even without a dedicated switch, this command has support for datasets (and their submodules) in git-annex's "adjusted mode".

Differences to the ``status`` command implementation prior DataLad v2

- Like `git status` this implementation reports on dataset modification, whereas the previous `status` also provided a listing of unchanged dataset content. This is no longer done. Equivalent functionality for listing dataset content is provided by the `ls_file_collection` command.
- The implementation is substantially faster. Depending on the context the speed-up is typically somewhere between 2x and 100x.

- The implementation does not suffer from the limitation re type change detection.
- Python and CLI API of the command use uniform parameter validation.

Examples

Options

-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 be used as a configuration source. Beyond reading configuration items, this command does not interact with the dataset.

--untracked {no,whole-dir,no-empty-dir,normal,all}

Determine how untracked content is considered and reported when comparing a revision to the state of the working tree. 'no': no untracked content is considered as a change; 'normal': untracked files and entire untracked directories are reported as such; 'all': report individual files even in fully untracked directories. In addition to these git-status modes, 'whole-dir' (like normal, but include empty directories), and 'no-empty-dir' (alias for 'normal') are understood. [Default: 'normal']

-r [{no,repository,datasets,mono}], --recursive [{no,repository,datasets,mono}]

Mode of recursion for status reporting. With 'no' the report is restricted to a single directory and its direct children. With 'repository', the report comprises all repository content underneath current working directory or root of a given dataset, but is limited to items directly contained in that repository. With 'datasets', the report also comprises any content in any subdatasets. Each subdataset is evaluated against its respective HEAD commit. With 'mono', a report similar to 'datasets' is generated, but any subdataset is evaluate with respect to the state recorded in its parent repository. In contrast to the 'datasets' mode, no report items on a joint submodule are generated. If no particular value is given with this option the 'datasets' mode is selected. [Default: 'repository']

-e {no,commit,full}, --eval-subdataset-state {no,commit,full}

Evaluation of subdataset state (modified or untracked content) can be expensive for deep dataset hierarchies as subdataset have to be tested recursively for uncommitted modifications. Setting this option to 'no' or 'commit' can substantially boost performance by limiting what is being tested. With 'no' no state is evaluated and subdataset are not investigated for modifications. With 'commit' only a discrepancy of the HEAD commit gitsha of a subdataset and the gitsha recorded in the superdataset's record is evaluated. With 'full' any other modifications are considered too. [Default: 'full']

--version

show the module and its version which provides the command

Authors

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

2.2.6 datalad tree

Synopsis

```
datalad tree [-h] [-L DEPTH] [-r] [-R LEVELS] [--include-files] [--include-hidden] [--
↪version] [path]
```

Description

Visualize directory and dataset hierarchies

This command mimics the UNIX/MS-DOS 'tree' utility to generate and display a directory tree, with DataLad-specific enhancements.

It can serve the following purposes:

1. Glorified 'tree' command
2. Dataset discovery
3. Programmatic directory traversal

Glorified 'tree' command

The rendered command output uses 'tree'-style visualization:

```
/tmp/mydir
├── [DS~0] ds_A/
│   └── [DS~1] subds_A/
└── [DS~0] ds_B/
    ├── dir_B/
    │   ├── file.txt
    │   ├── subdir_B/
    │   └── [DS~1] subds_B0/
    └── [DS~1] (not installed) subds_B1/

5 datasets, 2 directories, 1 file
```

Dataset paths are prefixed by a marker indicating subdataset hierarchy level, like `[DS~1]`. This is the absolute subdataset level, meaning it may also take into account superdatasets located above the tree root and thus not included in the output. If a subdataset is registered but not installed (such as after a non-recursive `datalad clone`), it will be prefixed by `(not installed)`. Only DataLad datasets are considered, not pure git/git-annex repositories.

The 'report line' at the bottom of the output shows the count of displayed datasets, in addition to the count of directories and files. In this context, datasets and directories are mutually exclusive categories.

By default, only directories (no files) are included in the tree, and hidden directories are skipped. Both behaviours can be changed using command options.

Symbolic links are always followed. This means that a symlink pointing to a directory is traversed and counted as a directory (unless it potentially creates a loop in the tree).

Dataset discovery

Using the `--recursive` or `--recursion-limit` option, this command generates the layout of dataset hierarchies based on subdataset nesting level, regardless of their location in the filesystem.

In this case, tree depth is determined by subdataset depth. This mode is thus suited for discovering available datasets when their location is not known in advance.

By default, only datasets are listed, without their contents. If `--depth` is specified additionally, the contents of each dataset will be included up to `--depth` directory levels (excluding subdirectories that are themselves datasets).

Tree filtering options such as `--include-hidden` only affect which directories are reported as dataset contents, not which directories are traversed to find datasets.

Performance note: since no assumption is made on the location of datasets, running this command with the `--recursive` or `--recursion-limit` option does a full scan of the whole directory tree. As such, it can be significantly slower than a call with an equivalent output that uses `--depth` to limit the tree instead.

Programmatic directory traversal

The command yields a result record for each tree node (dataset, directory or file). The following properties are reported, where available:

"path"
: Absolute path of the tree node

"type"
: Type of tree node: "dataset", "directory" or "file"

"depth"
: Directory depth of node relative to the tree root

"exhausted_levels"
: Depth levels for which no nodes are left to be generated (the respective subtrees have been 'exhausted')

"count"
: Dict with cumulative counts of datasets, directories and files in the tree up until the current node. File count is only included if the command is run with the `--include-files` option.

"dataset_depth"
: Subdataset depth level relative to the tree root. Only included for node type "dataset".

"dataset_abs_depth"
: Absolute subdataset depth level. Only included for node type "dataset".

"dataset_is_installed"
: Whether the registered subdataset is installed. Only included for node type "dataset".

"symlink_target"
: If the tree node is a symlink, the path to the link target

"is_broken_symlink"
: If the tree node is a symlink, whether it is a broken symlink

Examples

Show up to 3 levels of subdirectories below the current directory, including files and hidden contents:

```
% datalad tree -L 3 --include-files --include-hidden
```

Find all top-level datasets located anywhere under `/tmp`:

```
% datalad tree /tmp -R 0
```

Report all subdatasets recursively and their directory contents, up to 1 subdirectory deep within each dataset:

```
% datalad tree -r -L 1
```

Options

path

path to directory from which to generate the tree. Defaults to the current directory. [Default: '.']

-h, --help, --help-np

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

-L *DEPTH*, --depth *DEPTH*

limit the tree to maximum level of subdirectories. If not specified, will generate the full tree with no depth constraint. If paired with `--recursive` or `--recursion-limit`, refers to the maximum directory level to output below each dataset.

-r, --recursive

produce a dataset tree of the full hierarchy of nested subdatasets. *Note*: may have slow performance on large directory trees.

-R LEVELS, --recursion-limit LEVELS

limit the dataset tree to maximum level of nested subdatasets. 0 means include only top-level datasets, 1 means top-level datasets and their immediate subdatasets, etc. *Note*: may have slow performance on large directory trees.

--include-files

include files in the tree.

--include-hidden

include hidden files/directories in the tree. This option does not affect which directories will be searched for datasets when specifying `--recursive` or `--recursion-limit`. For example, datasets located underneath the hidden folder *.datalad* will be reported even if `--include-hidden` is omitted.

--version

show the module and its version which provides the command

Authors

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

2.3 Python tooling

`datalad-next` comprises a number of more-or-less self-contained mini-packages providing particular functionality. These implementations are candidates for a migration into the DataLad core package, and are provided here for immediate use. If and when components are migrated, transition modules will be kept to prevent API breakage in dependent packages.

archive_operations	Handler for operations on various archive types
commands	Essential tooling for implementing DataLad commands
config	Configuration query and manipulation
constraints	Data validation, coercion, and parameter documentation
consts	Common constants
credman	Credential management
datasets	Representations of DataLad datasets built on git/git-annex repositories
exceptions	Special purpose exceptions
iterable_subprocess	Context manager to communicate with a subprocess using iterables
itertools	Various iterators, e.g., for subprocess pipelining and output processing
iter_collections	Iterators for particular types of collections
repo_utils	Common repository operations
runners	Execution of subprocesses
shell	A persistent shell connection
tests	Tooling for test implementations
tests.fixtures	Collection of fixtures for facilitation test implementations
types	Custom types and dataclasses
uis	UI abstractions for user communication
url_operations	Handlers for operations on various URL types and protocols
utils	Assorted utility functions

2.3.1 datalad_next.archive_operations

Handler for operations on various archive types

All handlers implement the API defined by `ArchiveOperations`.

Available handlers:

TarArchiveOperations(location, *[, cfg])	Handler for a TAR archive on a local file system
ZipArchiveOperations(location, *[, cfg])	Handler for a ZIP archive on a local file system

datalad_next.archive_operations.TarArchiveOperations

class datalad_next.archive_operations.**TarArchiveOperations**(*location: Path*, *, *cfg:* ConfigManager | *None = None*)

Bases: `ArchiveOperations`

Handler for a TAR archive on a local file system

Any methods that take an archive item/member name as an argument accept a POSIX path string, or any *PurePath* instance.

close() → None

Closes any opened TAR file handler

open(*item: str | PurePosixPath*) → Generator[IO | None]

Get a file-like for a TAR archive item

The file-like object allows to read from the archive-item specified by *item*.

Parameters
item (*str | PurePath*) -- The identifier must be a POSIX path string, or a *PurePath* instance.

Returns
A file-like object to read bytes from the item, if the item is a regular file, else *None*. (This is returned by the context manager that is created via the decorator @*contextmanager*.)

Return type
IO | None

Raises
KeyError -- If no item with the name *item* can be found in the tar-archive

property tarfile: TarFile

Returns *TarFile* instance, after creating it on-demand

The instance is cached, and needs to be released by calling `.close()` if called outside a context manager.

datalad_next.archive_operations.ZipArchiveOperations

class datalad_next.archive_operations.**ZipArchiveOperations**(*location: Path*, *, *cfg:* ConfigManager | *None = None*, ***kwargs*)

Bases: `ArchiveOperations`

Handler for a ZIP archive on a local file system

close() → None

Calls *.close()* on the underlying `zipfile.ZipFile` instance

open(*item: str | PurePosixPath | ZipInfo*, ***kwargs*) → Generator[IO | None, None, None]

Context manager, returning an open file for a member of the archive.

The file-like object will be closed when the context-handler exits.

This method can be used in conjunction with `__iter__` to read any file from an archive:

```
with ZipArchiveOperations(archive_path) as zf:
    for item in zf:
        if item.type != FileSystemItemType.file:
            continue
        with zf.open(item.name) as fp:
            ...
```

Parameters

- **item** (*str | PurePosixPath | zipfile.ZipInfo*) -- Name, path, or ZipInfo-instance that identifies an item in the zipfile
- **kwargs** (*dict*) -- Keyword arguments that will be used for ZipFile.open()

Returns
A file-like object to read bytes from the item or to write bytes to the item.

Return type
IO

property zipfile: ZipFile

Access to the wrapped ZIP archive as a `zipfile.ZipFile`

2.3.2 datalad_next.commands

Essential tooling for implementing DataLad commands

This module provides the advanced command base class *ValidatedInterface*, for implementing commands with uniform argument validation and structured error reporting.

Beyond that, any further components necessary to implement command are imported in this module to offer a one-stop-shop experience. This includes `build_doc`, `datasetmethod`, and `eval_results`, among others.

CommandResult(action, status, path[, ...])	Base data class for result records emitted by DataLad commands.
CommandResultStatus(value)	Enumeration of possible statuses of command results
status.StatusResult(action, status, path[, ...])	

datalad_next.commands.CommandResult

class datalad_next.commands.**CommandResult**(*action: str*, *status:* CommandResultStatus, *path: str | Path*, *message: str | tuple | None = None*, *exception:* CapturedException *| None = None*, *error_message: str | tuple | None = None*, *type: str | None = None*, *logger: logging.Logger | None = None*, *refds: str | Path |* Dataset *= None*)

Bases: `object`

Base data class for result records emitted by DataLad commands.

Historically, such results records have taken the form of a Python `dict`. This class provides some API for its instances to be compatible with legacy code that expects a `dict`.

See also:

https://docs.datalad.org/design/result_records.html

action: str

A string label identifying which type of operation a result is associated with. Labels must not contain white space. They should be compact, and lower-cases, and use _ (underscore) to separate words in compound labels.

error_message: str | tuple | None = None

exception: *CapturedException* | None = None

get(*key*, *default=None*)

items()

logger: logging.Logger | None = None

message: str | tuple | None = None

path: str | Path

An *absolute* path describing the local entity a result is associated with (the subject of the result record). Paths must be platform-specific (e.g., Windows paths on Windows, and POSIX paths on other operating systems). When a result is about an entity that has no meaningful relation to the local file system (e.g., a URL to be downloaded), the `path` value should be determined with respect to the potential impact of the result on any local entity (e.g., a URL downloaded to a local file path, a local dataset modified based on remote information).

pop(*key*, *default=None*)

refds: str | Path | *Dataset* = None

status: *CommandResultStatus*

This field indicates the nature of a result in terms of four categories, identified by a *CommandResultStatus* value. The result status is used by user communication, but also for decision making on the overall success or failure of a command operation.

type: str | None = None

datalad_next.commands.CommandResultStatus

class datalad_next.commands.**CommandResultStatus**(*value*)

Bases: Enum

Enumeration of possible statuses of command results

error = 'error'

impossible = 'impossible'

notneeded = 'notneeded'

ok = 'ok'

datalad_next.commands.status.StatusResult

class datalad_next.commands.status.**StatusResult**(*action: 'str'*, *status: 'CommandResultStatus'*, *path: 'str | Path'*, *message: 'str | tuple | None' = None*, *exception: 'CapturedException | None' = None*, *error_message: 'str | tuple | None' = None*, *type: 'str | None' = None*, *logger: 'logging.Logger | None' = None*, *refds: 'str | Path | Dataset' = None*, *diff_state: 'GitDiffStatus | None' = None*, *gittype: 'GitTreeItemType | None' = None*, *prev_gittype: 'GitTreeItemType | None' = None*, *modification_types: 'tuple[GitContainerModificationType] | None' = None*)

Bases: *CommandResult*

diff_state: *GitDiffStatus* | None = None

The `status` of the underlying `GitDiffItem`. It is named "_state" to emphasize the conceptual similarity with the legacy property 'state'

gittype: *GitTreeItemType* | None = None

The `gittype` of the underlying `GitDiffItem`.

modification_types: tuple[*GitContainerModificationType*] | None = None

Qualifiers for modification types of container-type items (directories, submodules).

prev_gittype: *GitTreeItemType* | None = None

The `prev_gittype` of the underlying `GitDiffItem`.

property prev_type: str

property state: StatusState

A (more or less legacy) simplified representation of the subject state. For a more accurate classification use the `diff_status` property.

property type: str | None

property type_src: str | None

Backward-compatibility adaptor

class datalad_next.commands.**ValidatedInterface**

Bases: Interface

Alternative base class for commands with uniform parameter validation

Note: This interface is a draft. Usage is encouraged, but future changes are to be expected.

Commands derived from the traditional Interface class have no built-in input parameter validation beyond CLI input validation of individual parameters. Consequently, each command must perform custom parameter validation, which often leads to complex boilerplate code that is largely unrelated to the purpose of a particular command.

This class is part of a framework for uniform parameter validation, regardless of the target API (Python, CLI, GUI). The implementation of a command's __call__ method can focus on the core purpose of the command, while validation and error handling can be delegated elsewhere.

A validator for all individual parameters and the joint-set of all parameters can be provided through the *get_parameter_validator()* method.

To transition a command from Interface to ValidatedInterface, replace the base class declaration and declare a _validator_ class member. Any constraints= declaration for Parameter instances should either be removed, or moved to the corresponding entry in _validator_.

classmethod get_parameter_validator() → *EnsureCommandParameterization* | None

Returns a validator for the entire parameter set of a command

If parameter validation shall be performed, this method must return an instance of *EnsureCommandParameterization*. All parameters will be passed through this validator, and only the its output will be passed to the underlying command's __call__ method.

Consequently, the core implementation of a command only needs to support the output values of the validators declared by itself.

Factoring out input validation, normalization, type coercion etc. into a dedicated component also makes it accessible for upfront validation and improved error reporting via the different DataLad APIs.

If a command does not implement parameter validation in this fashion, this method must return None.

The default implementation returns the _validator_ class member.

2.3.3 datalad_next.config

Configuration query and manipulation

This modules provides the central ConfigManager class.

ConfigManager([dataset, overrides, source])	Thin wrapper around *git-config* with support for a dataset configuration.

datalad_next.config.ConfigManager

class datalad_next.config.**ConfigManager**(*dataset=None*, *overrides=None*, *source='any'*)

Bases: object

Thin wrapper around *git-config* with support for a dataset configuration.

The general idea is to have an object that is primarily used to read/query configuration option. Upon creation, current configuration is read via one (or max two, in the case of the presence of dataset-specific configuration) calls to *git config*. If this class is initialized with a Dataset instance, it supports reading and writing configuration from `.datalad/config` inside a dataset too. This file is committed to Git and hence useful to ship certain configuration items with a dataset.

The API aims to provide the most significant read-access API of a dictionary, the Python ConfigParser, and GitPython's config parser implementations.

This class is presently not capable of efficiently writing multiple configurations items at once. Instead, each modification results in a dedicated call to *git config*. This author thinks this is OK, as he cannot think of a situation where a large number of items need to be written during normal operation.

Each instance carries a public *overrides* attribute. This dictionary contains variables that override any setting read from a file. The overrides are persistent across reloads.

Any DATALAD_* environment variable is also presented as a configuration item. Settings read from environment variables are not stored in any of the configuration files, but are read dynamically from the environment at each *reload()* call. Their values take precedence over any specification in configuration files, and even overrides.

Parameters

- **dataset** (*Dataset, optional*) -- If provided, all *git config* calls are executed in this dataset's directory. Moreover, any modifications are, by default, directed to this dataset's configuration file (which will be created on demand)
- **overrides** (*dict, optional*) -- Variable overrides, see general class documentation for details.
- **source** (*{'any', 'local', 'branch', 'branch-local'}, optional*) -- Which sources of configuration setting to consider. If 'branch', configuration items are only read from a dataset's persistent configuration file in current branch, if any is present (the one in `.datalad/config`, not `.git/config`); if 'local', any non-committed source is considered (local and global configuration in Git config's terminology); if 'branch-local', persistent configuration in current dataset branch and local, but not global or system configuration are considered; if 'any' all possible sources of configuration are considered. Note: 'dataset' and 'dataset-local' are deprecated in favor of 'branch' and 'branch-local'.

add(*var*, *value*, *scope='branch'*, *reload=True*)

Add a configuration variable and value

Parameters

- **var** (*str*) -- Variable name including any section like *git config* expects them, e.g. 'core.editor'
- **value** (*str*) -- Variable value
- **scope** (*{'branch', 'local', 'global', 'override'}, optional*) -- Indicator which configuration file to modify. 'branch' indicates the persistent configuration in .datalad/config of a dataset; 'local' the configuration of a dataset's Git repository in .git/config; 'global' refers to the general configuration that is not specific to a single repository (usually in $USER/.gitconfig); 'override' limits the modification to the ConfigManager instance,

and the assigned value overrides any setting from any other source. Note: 'dataset' is being DEPRECATED in favor of 'branch'.

- **where** (*{'branch', 'local', 'global', 'override'}, optional*) -- DEPRECATED, use 'scope'.
- **reload** (*bool*) -- Flag whether to reload the configuration from file(s) after modification. This can be disable to make multiple sequential modifications slightly more efficient.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

Parameters

- **default** (*optional*) -- Value to return when key is not present. *None* by default.
- **get_all** (*bool, optional*) -- If True, return all values of multiple identical configuration keys. By default only the last specified value is returned.

get_from_source(*source*, *key*, *default=None*)

Like get(), but a source can be specific.

If *source* is 'branch', only the committed configuration is queried, overrides are applied. In the case of 'local', the committed configuration is ignored, but overrides and configuration from environment variables are applied as usual.

get_value(*section*, *option*, *default=None*)

Like *get()*, but with an optional default value

If the default is not None, the given default value will be returned in case the option did not exist. This behavior imitates GitPython's config parser.

getbool(*section*, *option*, *default=None*)

A convenience method which coerces the option value to a bool

Values "on", "yes", "true" and any int!=0 are considered True Values which evaluate to bool False, "off", "no", "false" are considered False TypeError is raised for other values.

getfloat(*section*, *option*)

A convenience method which coerces the option value to a float

getint(*section*, *option*)

A convenience method which coerces the option value to an integer

has_option(*section*, *option*)

If the given section exists, and contains the given option

has_section(*section*)

Indicates whether a section is present in the configuration

items(*section=None*)

Return a list of (name, value) pairs for each option

Optionally limited to a given section.

keys()

Returns list of configuration item names

obtain(*var*, *default=None*, *dialog_type=None*, *valtype=None*, *store=False*, *scope=None*, *reload=True*, ***kwargs*)

Convenience method to obtain settings interactively, if needed

A UI will be used to ask for user input in interactive sessions. Questions to ask, and additional explanations can be passed directly as arguments, or retrieved from a list of pre-configured items.

Additionally, this method allows for type conversion and storage of obtained settings. Both aspects can also be pre-configured.

Parameters

- **var** (*str*) -- Variable name including any section like *git config* expects them, e.g. 'core.editor'
- **default** (*any type*) -- In interactive sessions and if *store* is True, this default value will be presented to the user for confirmation (or modification). In all other cases, this value will be silently assigned unless there is an existing configuration setting.
- **dialog_type** (*{'question', 'yesno', None}*) -- Which dialog type to use in interactive sessions. If *None*, pre-configured UI options are used.
- **store** (*bool*) -- Whether to store the obtained value (or default)
- **scope** (*{'branch', 'local', 'global', 'override'}, optional*) -- Indicator which configuration file to modify. 'branch' indicates the persistent configuration in .datalad/config of a dataset; 'local' the configuration of a dataset's Git repository in .git/config; 'global' refers to the general configuration that is not specific to a single repository (usually in $USER/.gitconfig); 'override' limits the modification to the ConfigManager instance, and the assigned value overrides any setting from any other source. Note: 'dataset' is being DEPRECATED in favor of 'branch'.
- **where** (*{'branch', 'local', 'global', 'override'}, optional*) -- DEPRECATED, use 'scope'.
- **reload** (*bool*) -- Flag whether to reload the configuration from file(s) after modification. This can be disable to make multiple sequential modifications slightly more efficient.
- ****kwargs** -- Additional arguments for the UI function call, such as a question *text*.

options(*section*)

Returns a list of options available in the specified section.

reload(*force=False*)

Reload all configuration items from the configured sources

If *force* is False, all files configuration was previously read from are checked for differences in the modification times. If no difference is found for any file no reload is performed. This mechanism will not detect newly created global configuration files, use *force* in this case.

remove_section(*sec*, *scope='branch'*, *reload=True*)

Rename a configuration section

Parameters

- **sec** (*str*) -- Name of the section to remove.
- **scope** (*{'branch', 'local', 'global', 'override'}, optional*) -- Indicator which configuration file to modify. 'branch' indicates the persistent configuration in .datalad/config of a dataset; 'local' the configuration of a dataset's Git repository in .git/config; 'global' refers to the general configuration that is not specific to a single repository (usually in $USER/.gitconfig); 'override' limits the modification to the ConfigManager instance, and the assigned value overrides any setting from any other source. Note: 'dataset' is being DEPRECATED in favor of 'branch'.

- **where** (*{'branch', 'local', 'global', 'override'}, optional*) -- DEPRECATED, use 'scope'.
- **reload** (*bool*) -- Flag whether to reload the configuration from file(s) after modification. This can be disable to make multiple sequential modifications slightly more efficient.

rename_section(*old*, *new*, *scope='branch'*, *reload=True*)

Rename a configuration section

Parameters

- **old** (*str*) -- Name of the section to rename.
- **new** (*str*) -- Name of the section to rename to.
- **scope** (*{'branch', 'local', 'global', 'override'}, optional*) -- Indicator which configuration file to modify. 'branch' indicates the persistent configuration in .datalad/config of a dataset; 'local' the configuration of a dataset's Git repository in .git/config; 'global' refers to the general configuration that is not specific to a single repository (usually in $USER/.gitconfig); 'override' limits the modification to the ConfigManager instance, and the assigned value overrides any setting from any other source. Note: 'dataset' is being DEPRECATED in favor of 'branch'.
- **where** (*{'branch', 'local', 'global', 'override'}, optional*) -- DEPRECATED, use 'scope'.
- **reload** (*bool*) -- Flag whether to reload the configuration from file(s) after modification. This can be disable to make multiple sequential modifications slightly more efficient.

rewrite_url(*url*)

Any matching 'url.<base>.insteadOf' configuration is applied

Any URL that starts with such a configuration will be rewritten to start, instead, with <base>. When more than one insteadOf strings match a given URL, the longest match is used.

Parameters

- **cfg** (ConfigManager *or dict*) -- dict-like with configuration variable name/value-pairs.
- **url** (*str*) -- URL to be rewritten, if matching configuration is found.

Returns
Rewritten or unmodified URL.

Return type
str

sections()

Returns a list of the sections available

set(*var*, *value*, *scope='branch'*, *reload=True*, *force=False*)

Set a variable to a value.

In opposition to *add*, this replaces the value of *var* if there is one already.

Parameters

- **var** (*str*) -- Variable name including any section like *git config* expects them, e.g. 'core.editor'
- **value** (*str*) -- Variable value

- **force** (*bool*) -- if set, replaces all occurrences of *var* by a single one with the given *value*. Otherwise raise if multiple entries for *var* exist already
- **scope** (*{'branch', 'local', 'global', 'override'}, optional*) -- Indicator which configuration file to modify. 'branch' indicates the persistent configuration in .datalad/config of a dataset; 'local' the configuration of a dataset's Git repository in .git/config; 'global' refers to the general configuration that is not specific to a single repository (usually in $USER/.gitconfig); 'override' limits the modification to the ConfigManager instance, and the assigned value overrides any setting from any other source. Note: 'dataset' is being DEPRECATED in favor of 'branch'.
- **where** (*{'branch', 'local', 'global', 'override'}, optional*) -- DEPRECATED, use 'scope'.
- **reload** (*bool*) -- Flag whether to reload the configuration from file(s) after modification. This can be disable to make multiple sequential modifications slightly more efficient.

unset(*var*, *scope='branch'*, *reload=True*)

Remove all occurrences of a variable

Parameters

- **var** (*str*) -- Name of the variable to remove
- **scope** (*{'branch', 'local', 'global', 'override'}, optional*) -- Indicator which configuration file to modify. 'branch' indicates the persistent configuration in .datalad/config of a dataset; 'local' the configuration of a dataset's Git repository in .git/config; 'global' refers to the general configuration that is not specific to a single repository (usually in $USER/.gitconfig); 'override' limits the modification to the ConfigManager instance, and the assigned value overrides any setting from any other source. Note: 'dataset' is being DEPRECATED in favor of 'branch'.
- **where** (*{'branch', 'local', 'global', 'override'}, optional*) -- DEPRECATED, use 'scope'.
- **reload** (*bool*) -- Flag whether to reload the configuration from file(s) after modification. This can be disable to make multiple sequential modifications slightly more efficient.

2.3.4 datalad_next.constraints

Data validation, coercion, and parameter documentation

This module provides a set of uniform classes to validate and document particular aspects of inputs. In a nutshell, each of these *Constraint* class:

- focuses on a specific aspect, such as data type coercion, or checking particular input properties
- is instantiated with a set of parameters to customize such an instance for a particular task
- performs its task by receiving an input via its `__call__()` method
- provides default auto-documentation that can be customized by wrapping an instance in *WithDescription*

Individual `Constraint` instances can be combined with logical AND (*AllOf*) and OR (*AnyOf*) operations to form arbitrarily complex constructs.

On (validation/coercion) error, instances raise *ConstraintError*) via their `raise_for()` method. This approach to error reporting helps to communicate standard (yet customizable) error messages, aids structured error reporting, and is capable of communication the underlying causes of an error in full detail without the need to generate long textual descriptions.

EnsureCommandParameterization is a particular variant of a `Constraint` that is capable of validating a complete parameterization of a command (or function), for each parameter individually, and for arbitrary combinations of parameters. It puts a particular emphasis on structured error reporting.

Constraint()	Base class for value coercion/validation.
AllOf(*constraints)	Logical AND for constraints.
AnyOf(*constraints)	Logical OR for constraints.
NoConstraint()	A constraint that represents no constraints
WithDescription(constraint, *[, ...])	Constraint that wraps another constraint and replaces its description
ConstraintError(constraint, value, msg[, ctx])	Exception type raised by constraints when their conditions are violated
CommandParametrizationError(exceptions)	Exception type raised on violating any command parameter constraints
ParameterConstraintContext(parameters[, ...])	Representation of a parameter constraint context
EnsureDataset([installed, purpose, require_id])	Ensure an absent/present *Dataset* from any path or Dataset instance
DatasetParameter(original, ds)	Utility class to report an original and resolve dataset parameter value
EnsureBool()	Ensure that an input is a bool.
EnsureCallable()	Ensure an input is a callable object
EnsureChoice(*values)	Ensure an input is element of a set of possible values
EnsureFloat()	Ensure that an input (or several inputs) are of a data type 'float'.
EnsureHashAlgorithm()	Ensure an input matches a name of a `hashlib` algorithm
EnsureDType(dtype)	Ensure that an input (or several inputs) are of a particular data type.
EnsureInt()	Ensure that an input (or several inputs) are of a data type 'int'.
EnsureKeyChoice(key, values)	Ensure value under a key in an input is in a set of possible values
EnsureNone()	Ensure an input is of value *None*
EnsurePath(*, path_type, is_format, lexists, ...)	Ensures input is convertible to a (platform) path and returns a *Path*
EnsureStr([min_len, match])	Ensure an input is a string of some min.
EnsureStrPrefix(prefix)	Ensure an input is a string that starts with a given prefix.
EnsureRange([min, max])	Ensure an input is within a particular range
EnsureValue(value)	Ensure an input is a particular value
EnsureIterableOf(iter_type, item_constraint)	Ensure that an input is a list of a particular data type
EnsureListOf(item_constraint[, min_len, max_len])	
EnsureTupleOf(item_constraint[, min_len, ...])	
EnsureMapping(key, value[, delimiter, ...])	Ensure a mapping of a key to a value of a specific nature
EnsureGeneratorFromFileLike(item_constraint)	Ensure a constraint for each item read from a file-like.
EnsureJSON()	Ensures that string is JSON formatted and can be deserialized.
EnsureURL([required, forbidden, match])	Ensures that a string is a valid URL with a select set of components
EnsureParsedURL([required, forbidden, match])	Like *EnsureURL*, but returns a parsed URL
EnsureGitRefName([allow_onelevel, ...])	Ensures that a reference name is well formed
EnsureRemoteName([known, dsarg])	Ensures a valid remote name, and optionally if such a remote is known

continues on next page

Table 1 – continued from previous page

EnsureSiblingName([known, dsarg])	Identical to EnsureRemoteName, but used the term "sibling"
EnsureCommandParameterization(...[, ...])	Base class for *ValidatedInterface* parameter validators

datalad_next.constraints.Constraint

class datalad_next.constraints.**Constraint**

Bases: object

Base class for value coercion/validation.

These classes are also meant to be able to generate appropriate documentation on an appropriate parameter value.

__repr__()

Rudimentary repr to avoid default scary to the user Python repr

__str__()

Rudimentary self-description

for_dataset(*dataset:* DatasetParameter) → *Constraint*

Return a constraint-variant for a specific dataset context

The default implementation returns the unmodified, identical constraint. However, subclasses can implement different behaviors.

property input_description: str

Returns full description of valid input for a constraint

Like input_synopsis this information is user-facing. In contrast, to the synopsis there is length/line limit. Nevertheless, the information should be presented in a compact fashion that avoids needless verbosity. If possible, a single paragraph is a good format. If multiple paragraphs are necessary, they should be separated by a single, empty line.

Rendering code may indent, or rewrap the text, so no line-by-line formatting will be preserved.

If possible, the synopsis should be written in a UI/API-agnostic fashion. However, if this is impossible or leads to imprecisions or confusion, it should focus on use within Python code and with Python data types. Tailored documentation can be provided via the WithDescription wrapper.

property input_synopsis: str

Returns brief, single line summary of valid input for a constraint

This information is user-facing, and to be used in any place where space is limited (tooltips, usage summaries, etc).

If possible, the synopsis should be written in a UI/API-agnostic fashion. However, if this is impossible or leads to imprecisions or confusion, it should focus on use within Python code and with Python data types. Tailored documentation can be provided via the WithDescription wrapper.

long_description()

This method is deprecated. Use input_description instead

raise_for(*value*, *msg*, ***ctx*) → None

Convenience method for raising a ConstraintError

The parameters are identical to those of ConstraintError. This method merely passes the Constraint instance as self to the constructor.

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.AllOf

class datalad_next.constraints.**AllOf**(**constraints*)

Bases: `_MultiConstraint`

Logical AND for constraints.

An arbitrary number of constraints can be given. They are evaluated in the order in which they were specified. The return value of each constraint is passed an input into the next. The return value of the last constraint is the global return value. No intermediate exceptions are caught.

Documentation is aggregated for all constraints.

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.AnyOf

class datalad_next.constraints.**AnyOf**(**constraints*)

Bases: `_MultiConstraint`

Logical OR for constraints.

An arbitrary number of constraints can be given. They are evaluated in the order in which they were specified. The value returned by the first constraint that does not raise an exception is the global return value.

Documentation is aggregated for all alternative constraints.

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.NoConstraint

class datalad_next.constraints.**NoConstraint**

Bases: *Constraint*

A constraint that represents no constraints

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.WithDescription

class datalad_next.constraints.**WithDescription**(*constraint:* Constraint, *, *input_synopsis: str | None = None*, *input_description: str | None = None*, *error_message: str | None = None*, *input_synopsis_for_ds: str | None = None*, *input_description_for_ds: str | None = None*, *error_message_for_ds: str | None = None*)

Bases: *Constraint*

Constraint that wraps another constraint and replaces its description

Whenever a constraint's self-description does not fit an application context, it can be wrapped with this class. The given synopsis and description of valid inputs replaces those of the wrapped constraint.

property constraint: *Constraint*

Returns the wrapped constraint instance

for_dataset(*dataset:* DatasetParameter) → *Constraint*

Wrap the wrapped constraint again after tailoring it for the dataset

property input_description

Returns full description of valid input for a constraint

Like `input_synopsis` this information is user-facing. In contrast, to the synopsis there is length/line limit. Nevertheless, the information should be presented in a compact fashion that avoids needless verbosity. If possible, a single paragraph is a good format. If multiple paragraphs are necessary, they should be separated by a single, empty line.

Rendering code may indent, or rewrap the text, so no line-by-line formatting will be preserved.

If possible, the synopsis should be written in a UI/API-agnostic fashion. However, if this is impossible or leads to imprecisions or confusion, it should focus on use within Python code and with Python data types. Tailored documentation can be provided via the `WithDescription` wrapper.

property input_synopsis

Returns brief, single line summary of valid input for a constraint

This information is user-facing, and to be used in any place where space is limited (tooltips, usage summaries, etc).

If possible, the synopsis should be written in a UI/API-agnostic fashion. However, if this is impossible or leads to imprecisions or confusion, it should focus on use within Python code and with Python data types. Tailored documentation can be provided via the `WithDescription` wrapper.

long_description() → str

This method is deprecated. Use `input_description` instead

short_description() → str

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.ConstraintError

exception datalad_next.constraints.**ConstraintError**(*constraint*, *value: Any*, *msg: str*, *ctx: Dict[str, Any] | None = None*)

Exception type raised by constraints when their conditions are violated

A primary purpose of this class is to provide uniform means for communicating information on violated constraints.

datalad_next.constraints.CommandParametrizationError

exception datalad_next.constraints.**CommandParametrizationError**(*exceptions: Dict[str, ConstraintError] | Dict[ParameterConstraintContext, ConstraintError]*)

Exception type raised on violating any command parameter constraints

See also:

EnsureCommandParameterization

datalad_next.constraints.ParameterConstraintContext

class datalad_next.constraints.**ParameterConstraintContext**(*parameters: Tuple[str]*, *description: str | None = None*)

Bases: `object`

Representation of a parameter constraint context

This type is used for the keys in the error map of. `ParametrizationErrors`. Its purpose is to clearly identify which parameter combination (and its nature) led to a *ConstraintError*.

An error context comprises to components: 1) the names of the parameters that were considered, and 2) a description of how the parameters were linked or combined. In the simple case of an error occurring in the context of a single parameter, the second component is superfluous. Otherwise, it can be thought of as an operation label, describing what aspect of the set of parameters is being relevant in a particular context.

Example:

A command has two parameters *p1* and *p2*. The may also have respective individual constraints, but importantly they 1) must not have identical values, and 2) their sum must be larger than 3. If the command is called with `cmd(p1=1, p2=1)`, both conditions are violated. The reporting may be implemented using the following `ParameterConstraintContext` and `ConstraintError` instances:

```
ParameterConstraintContext(('p1', 'p2'), 'inequality):
  ConstraintError(EnsureValue(True), False, <EnsureValue error>)

ParameterConstraintContext(('p1', 'p2'), 'sum):
  ConstraintError(EnsureRange(min=3), False, <EnsureRange error>)
```

where the `ConstraintError` instances are generated by standard `Constraint` implementation. For the second error, this could look like:

```
EnsureRange(min=3)(params['p1'] + params['p2'])
```

description: str | None = None

get_label_with_parameter_values(*values: dict*) → str

Like `.label` but each parameter will also state a value

property label: str

A concise summary of the context

This label will be a compact as possible.

parameters: Tuple[str]

datalad_next.constraints.EnsureDataset

class datalad_next.constraints.**EnsureDataset**(*installed: bool | None = None*, *purpose: str | None = None*, *require_id: bool | None = None*)

Bases: *Constraint*

Ensure an absent/present *Dataset* from any path or Dataset instance

Regardless of the nature of the input (*Dataset* instance or local path) a resulting instance (if it can be created) is optionally tested for absence or presence on the local file system.

Due to the particular nature of the *Dataset* class (the same instance is used for a unique path), this constraint returns a *DatasetParameter* rather than a *Dataset* directly. Consuming commands can discover the original parameter value via its *original* property, and access a *Dataset* instance via its *ds* property.

In addition to any value representing an explicit path, this constraint also recognizes the special value *None*. This instructs the implementation to find a dataset that contains the process working directory (PWD). Such a dataset need not have its root at PWD, but could be located in any parent directory too. If no such dataset can be found, PWD is used directly. Tests for `installed` are performed in the same way as with an explicit dataset location argument. If *None* is given and `installed=True`, but no dataset is found, an exception is raised (this is the behavior of the `required_dataset()` function in the DataLad core package). With `installed=False` no exception is raised and a dataset instances matching PWD is returned.

short_description() → str

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.DatasetParameter

class datalad_next.constraints.**DatasetParameter**(*original*, *ds*)

Bases: `object`

Utility class to report an original and resolve dataset parameter value

This is used by *EnsureDataset* to be able to report the original argument semantics of a dataset parameter to a receiving command. It is consumed by any `Constraint.for_dataset()`.

The original argument is provided via the *original* property. A corresponding *Dataset* instance is provided via the *ds* property.

datalad_next.constraints.EnsureBool

class datalad_next.constraints.**EnsureBool**

Bases: *Constraint*

Ensure that an input is a bool.

A couple of literal labels are supported, such as: False: '0', 'no', 'off', 'disable', 'false' True: '1', 'yes', 'on', 'enable', 'true'

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureCallable

class datalad_next.constraints.**EnsureCallable**

Bases: *Constraint*

Ensure an input is a callable object

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureChoice

class datalad_next.constraints.**EnsureChoice**(**values*)

Bases: *Constraint*

Ensure an input is element of a set of possible values

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureFloat

class datalad_next.constraints.**EnsureFloat**

Bases: *EnsureDType*

Ensure that an input (or several inputs) are of a data type 'float'.

datalad_next.constraints.EnsureHashAlgorithm

class datalad_next.constraints.**EnsureHashAlgorithm**

Bases: *EnsureChoice*

Ensure an input matches a name of a `hashlib` algorithm

Specifically the item must be in the `algorithms_guaranteed` collection.

datalad_next.constraints.EnsureDType

class datalad_next.constraints.**EnsureDType**(*dtype*)

Bases: *Constraint*

Ensure that an input (or several inputs) are of a particular data type. .. rubric:: Examples

```
>>> c = EnsureDType(float)
>>> type(c(8))
float
>>> import numpy as np
>>> c = EnsureDType(np.float64)
>>> type(c(8))
numpy.float64
```

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureInt

class datalad_next.constraints.**EnsureInt**

Bases: *EnsureDType*

Ensure that an input (or several inputs) are of a data type 'int'.

datalad_next.constraints.EnsureKeyChoice

class datalad_next.constraints.**EnsureKeyChoice**(*key*, *values*)

Bases: *EnsureChoice*

Ensure value under a key in an input is in a set of possible values

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureNone

class datalad_next.constraints.**EnsureNone**

Bases: *EnsureValue*

Ensure an input is of value *None*

datalad_next.constraints.EnsurePath

class datalad_next.constraints.**EnsurePath**(*, *path_type: type = <class 'pathlib.Path'>*, *is_format: str | None = None*, *lexists: bool | None = None*, *is_mode: Callable | None = None*, *ref: Path | None = None*, *ref_is: str = 'parent-or-same-as'*, *dsarg: DatasetParameter | None = None*)

Bases: *Constraint*

Ensures input is convertible to a (platform) path and returns a *Path*

Optionally, the path can be tested for existence and whether it is absolute or relative.

for_dataset(*dataset:* DatasetParameter) → *Constraint*

Return an similarly parametrized variant that resolves paths against a given dataset (argument)

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureStr

class datalad_next.constraints.**EnsureStr**(*min_len: int = 0*, *match: str | None = None*)

Bases: *Constraint*

Ensure an input is a string of some min. length and matching a pattern

Pattern matching is optional and minimum length is zero (empty string is OK).

No type conversion is performed.

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureStrPrefix

class datalad_next.constraints.**EnsureStrPrefix**(*prefix*)

Bases: *EnsureStr*

Ensure an input is a string that starts with a given prefix.

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureRange

class datalad_next.constraints.**EnsureRange**(*min=None*, *max=None*)

Bases: *Constraint*

Ensure an input is within a particular range

No type checks are performed.

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureValue

class datalad_next.constraints.**EnsureValue**(*value*)

Bases: *Constraint*

Ensure an input is a particular value

long_description()

This method is deprecated. Use `input_description` instead

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureIterableOf

class datalad_next.constraints.**EnsureIterableOf**(*iter_type: type*, *item_constraint: Callable*, *min_len: int | None = None*, *max_len: int | None = None*)

Bases: *Constraint*

Ensure that an input is a list of a particular data type

property item_constraint

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureListOf

class datalad_next.constraints.**EnsureListOf**(*item_constraint: Callable*, *min_len: int | None = None*, *max_len: int | None = None*)

Bases: *EnsureIterableOf*

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureTupleOf

class datalad_next.constraints.**EnsureTupleOf**(*item_constraint: Callable*, *min_len: int | None = None*, *max_len: int | None = None*)

Bases: *EnsureIterableOf*

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureMapping

class datalad_next.constraints.**EnsureMapping**(*key:* Constraint, *value:* Constraint, *delimiter: str = ':'*, *allow_length2_sequence: bool = True*)

Bases: *Constraint*

Ensure a mapping of a key to a value of a specific nature

for_dataset(*dataset:* DatasetParameter) → *Constraint*

Return a constraint-variant for a specific dataset context

The default implementation returns the unmodified, identical constraint. However, subclasses can implement different behaviors.

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureGeneratorFromFileLike

class datalad_next.constraints.**EnsureGeneratorFromFileLike**(*item_constraint: Callable*, *exc_mode: str = 'raise'*)

Bases: *Constraint*

Ensure a constraint for each item read from a file-like.

A given value can either be a file-like (the outcome of *open()*, or *StringIO*), or - as an alias of STDIN, or a path to an existing file to be read from.

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureJSON

class datalad_next.constraints.**EnsureJSON**

Bases: *Constraint*

Ensures that string is JSON formatted and can be deserialized.

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureURL

class datalad_next.constraints.**EnsureURL**(*required: list | None = None*, *forbidden: list | None = None*, *match: str | None = None*)

Bases: *Constraint*

Ensures that a string is a valid URL with a select set of components

and/or:

- does not contain certain components
- matches a particular regular expression

Given that a large variety of strings are also a valid URL, a typical use of this constraint would involve using a *required=['scheme']* setting.

All URL attribute names supported by *urllib.parse.urlparse()* are also supported here: scheme, netloc, path, params, query, fragment, username, password, hostname, port.

See also:

https://docs.python.org/3/library/urllib.parse.html#urllib.parse.urlparse

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureParsedURL

class datalad_next.constraints.**EnsureParsedURL**(*required: list | None = None*, *forbidden: list | None = None*, *match: str | None = None*)

Bases: *EnsureURL*

Like *EnsureURL*, but returns a parsed URL

datalad_next.constraints.EnsureGitRefName

class datalad_next.constraints.**EnsureGitRefName**(*allow_onelevel: bool = True*, *normalize: bool = True*, *refspec_pattern: bool = False*)

Bases: *Constraint*

Ensures that a reference name is well formed

Validation is performed by calling *git check-ref-format*.

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureRemoteName

class datalad_next.constraints.**EnsureRemoteName**(*known: bool | None = None*, *dsarg:* DatasetParameter *| None = None*)

Bases: *Constraint*

Ensures a valid remote name, and optionally if such a remote is known

for_dataset(*dataset:* DatasetParameter) → *Constraint*

Return an similarly parametrized variant that checks remote names against a given dataset (argument)

short_description()

This method is deprecated. Use `input_synopsis` instead

datalad_next.constraints.EnsureSiblingName

class datalad_next.constraints.**EnsureSiblingName**(*known: bool | None = None*, *dsarg:* DatasetParameter *| None = None*)

Bases: *EnsureRemoteName*

Identical to `EnsureRemoteName`, but used the term "sibling"

Only error messages and documentation differ, with "remote" being replaced with "sibling".

datalad_next.constraints.EnsureCommandParameterization

class datalad_next.constraints.**EnsureCommandParameterization**(*param_constraints: Dict[str,* Constraint*]*, **, validate_defaults: Container[str] | None = None*, *joint_constraints: Dict[*ParameterConstraintContext*, Callable] | None = None*, *tailor_for_dataset: Dict[str, str] | None = None*)

Bases: *Constraint*

Base class for *ValidatedInterface* parameter validators

This class can be used as-is, by declaring individual constraints in the constructor, or it can be subclassed to consolidate all custom validation-related code for a command in a single place.

Commonly this constraint is used by declaring particular value constraints for individual parameters as a mapping. Declaring that the `path` parameter should receive something that is or can be coerced to a valid `Path` object looks like this:

```
EnsureCommandParameterization({'path': EnsurePath()})
```

This class differs from a standard `Constraint` implementation, because its `__call__()` method support additional arguments that are used by the internal `Interface` handling code to control how parameters are validated.

During validation, when no validator for a particular parameter is declared, any input value is passed on as-is, and otherwise an input is passed through the validator.

There is one exception to this rule: When a parameter value is identical to its default value (as declared in the command signature, and communicated via the `at_default` argument of `__call__()`), this default value is

also passed as-is, unless the respective parameter name is included in the `validate_defaults` constructor argument.

An important consequence of this behavior is that validators need not cover a default value. For example, a parameter constraint for `path=None`, where `None` is a special value used to indicate an optional and unset value, but actually only paths are acceptable input values. can simply use `EnsurePath()` and it is not necessary to do something like `EnsurePath() | EnsureNone()`.

However, *EnsureCommandParameterization* can also be specifically instructed to perform validation of defaults for individual parameters, as described above. A common use case is the auto-discovery of datasets, where often *None* is the default value of a *dataset* parameter (to make it optional), and an *EnsureDataset* constraint is used. This constraint can perform the auto-discovery (with the *None* value indicating that), but validation of defaults must be turned on for the *dataset* parameter in order to do that.

A second difference to a common `Constraint` implementation is the ability to perform an "exhaustive validation" on request (via `__call__(on_error=...)`). In this case, validation is not stopped at the first discovered violation, but all violations are collected and communicated by raising a `CommandParametrizationError` exception, which can be inspected by a caller for details on number and nature of all discovered violations.

Exhaustive validation and joint reporting are only supported for individual constraint implementations that raise *ConstraintError* exceptions. For legacy constraints, any raised exception of another type are not caught and reraised immediately.

__call__(*kwargs*, *at_default=None*, *required=None*, *on_error='raise-early'*) → Dict

Parameters

- **kwargs** (*dict*) -- Parameter name (`str`)) to value (any) mapping of the parameter set.
- **at_default** (*set or None*) -- Set of parameter names where the respective values in `kwargs` match their respective defaults. This is used for deciding whether or not to process them with an associated value constraint (see the `validate_defaults` constructor argument).
- **required** (*set or None*) -- Set of parameter names that are known to be required.
- **on_error** (*{'raise-early', 'raise-at-end'}*) -- Flag how to handle constraint violation. By default, validation is stopped at the first error and an exception is raised. When an exhaustive validation is performed, an eventual exception contains information on all constraint violations. Regardless of this mode more than one error can be reported (in case (future) implementation perform independent validations in parallel).

Raises

CommandParametrizationError -- Raised whenever one (or more) `ConstraintError` exceptions are caught during validation. Other exception types are not caught and pass through.

joint_validation(*params: Dict*, *on_error: str*) → Dict

Higher-order validation considering multiple parameters at a time

This method is called with all, individually validated, command parameters in keyword-argument form in the `params` dict argument.

Arbitrary additional validation steps can be performed on the full set of parameters that may involve raising exceptions on validation errors, but also value transformation or replacements of individual parameters based on the setting of others.

The parameter values returned by the method are passed on to the respective command implementation.

The default implementation iterates over the `joint_validators` specification given to the constructor, in order to perform any number of validations. This is a mapping of a `ParameterConstraintContext` instance to a callable implementing a validation for a particular parameter set.

Example:

```
_joint_validators_ = {
    ParameterConstraintContext(('p1', 'p2'), 'sum'):
        MyValidator._check_sum,
}

def _checksum(self, p1, p2):
    if (p1 + p2) < 3:
        self.raise_for(
           dict(p1=p1, p2=p2),
           'parameter sum is too large',
        )
```

The callable will be passed the arguments named in the `ParameterConstraintContext` as keyword arguments, using the same names as originally given to `EnsureCommandParameterization`.

Any raised `ConstraintError` is caught and reported together with the respective `ParameterConstraintContext`. The violating value reported in such a `ConstraintError` must be a mapping of parameter name to value, comprising the full parameter set (i.e., keys matching the `ParameterConstraintContext`). The use of `self.raise_for()` is encouraged.

If the callable anyhow modifies the passed arguments, it must return them as a kwargs-like mapping. If nothing is modified, it is OK to return `None`.

Returns

- *dict* -- The returned dict must have a value for each item passed in via `params`.
- **on_error** (*{'raise-early', 'raise-at-end'}*) -- Flag how to handle constraint violation. By default, validation is stopped at the first error and an exception is raised. When an exhaustive validation is performed, an eventual exception contains information on all constraint violations.

Raises

`ConstraintErrors` -- With *on_error='raise-at-end'* an implementation can choose to collect more than one higher-order violation and raise them as a *ConstraintErrors* exception.

2.3.5 datalad_next.consts

Common constants

COPY_BUFSIZE
`shutil` buffer size default, with Windows platform default changes backported from Python 3.10.

PRE_INIT_COMMIT_SHA
SHA value for `git hash-object -t tree /dev/null`, i.e. for nothing. This corresponds to the state of a Git repository before the first commit is made.

on_linux
`True` if executed on the Linux platform.

on_windows
`True` if executed on the Windows platform.

2.3.6 datalad_next.credman

Credential management

CredentialManager([cfg])	Facility to get, set, remove and query credentials.
verify_property_names(names)	Check credential property names for syntax-compliance.

datalad_next.credman.CredentialManager

class datalad_next.credman.**CredentialManager**(*cfg:* ConfigManager | *None* = *None*)

Bases: `object`

Facility to get, set, remove and query credentials.

A credential in this context is a set of properties (key-value pairs) associated with exactly one secret.

At present, the only backend for secret storage is the Python keyring package, as interfaced via a custom DataLad wrapper. Store for credential properties is implemented using DataLad's (i.e. Git's) configuration system. All properties are stored in the *global* (i.e., user) scope under configuration items following the pattern:

```
datalad.credential.<name>.<property>
```

where <name> is a credential name/identifier, and <property> is an arbitrarily named credential property, whose name must follow the git-config syntax for variable names (case-insensitive, only alphanumeric characters and `-`, and must start with an alphabetic character).

Create a `CredentialManager` instance is fast, virtually no initialization needs to be performed. All internal properties are lazily evaluated. This facilitates usage in code where it is difficult to incorporate a long-lived central instance.

API

With one exception, all parameter names of methods in the core API outside `**kwargs` must have a `_` prefix that distinguishes credential properties from method parameters. The one exception is the `name` parameter, which is used as a primary identifier (albeit being optional for some operations).

The `obtain()` method is provided as an additional convenience, and implements a standard workflow for obtaining a credential in a wide variety of scenarios (credential name, credential properties, secret either respectively already known or yet unknown).

get(*name=None*, *, *_prompt=None*, *_type_hint=None*, ***kwargs*)

Get properties and secret of a credential.

This is a read-only method that never modifies information stored on a credential in any backend.

Credential property lookup is supported via a number approaches. When providing `name`, all existing corresponding configuration items are found and reported, and an existing secret is retrieved from name-based secret backends (presently `keyring`). When providing a `type` property or a `_type_hint` the lookup of additional properties in the keyring-backend is enabled, using predefined property name lists for a number of known credential types.

For all given property keys that have no value assigned after the initial lookup, manual/interactive entry is attempted, whenever a custom `_prompt` was provided. This include requesting a secret. If manually entered information is contained in the return credential record, the record contains an additional `_edited` property with a value of `True`.

If no secret is known after lookup and a potential manual data entry, a plain `None` is returned instead of a full credential record.

Parameters

- **name** (*str, optional*) -- Name of the credential to be retrieved
- **_prompt** (*str or None*) -- Instructions for credential entry to be displayed when missing properties are encountered. If `None`, manual entry is disabled.
- **_type_hint** (*str or None*) -- In case no `type` property is included in `kwargs`, this parameter is used to determine a credential type, to possibly enable further lookup/entry of additional properties for a known credential type
- ****kwargs** -- Credential property name/value pairs to overwrite/amend potentially existing properties. For any property with a value of `None`, manual data entry will be performed, unless a value could be retrieved on lookup, or prompting was not enabled.

Returns

Return `None`, if no secret for the credential was found or entered. Otherwise returns the complete credential record, comprising all properties and the secret. An additional `_edited` key with a value of `True` is added whenever the returned record contains manually entered information.

Return type

dict or None

Raises

ValueError -- When the method is called without any information that could be used to identify a credential

obtain(*name: str | None = None*, ***, *prompt: str | None = None*, *type_hint: str | None = None*, *query_props: Dict | None = None*, *expected_props: List | Tuple | None = None*)

Obtain a credential by query or prompt (if needed)

This convenience method implements a standard workflow to obtain a credential. It supports credential selection by credential name/identifier, and falls back onto querying for a credential matching a set of specified properties (as key-value mappings). If no suitable credential is known, a user is prompted to enter one interactively (if possible in the current session).

If a credential was entered manually, any given `type_hint` will be included as a `type` property of the returned credential, and the returned credential has an `_edited=True` property. Likewise, any `realm` property included in the `query_props` is included in the returned credential in this case.

If desired, a credential workflow can be completed, after a credential was found to be valid/working, by storing or updating it in the credential store:

```
cm = CredentialManager()
cname, cprops = cm.obtain(...)
# verify credential is working
...
# set/update
cm.set(cname, _lastused=True, **cprops)
```

In the code sketch above, if `cname` is `None` (as it will be for a newly entered credential, `set()` will prompt for a name to store the credential under, and will offer a user the choice to skip storing a credential. For any previously known credential, the `last-used` property will be updated to enable preferred selection in future credential discovery attempts via `obtain()`.

Examples

Minimal call to get a credential entered (manually):

```
credman.obtain(type_hint='token', prompt='Credential please!')
```

Without a prompt text no interaction is attempted, and without a type hint it is unknown what (and how much) to enter.

Minimal call to retrieve a credential by its identifier:

```
credman.obtain('my-github-token')
```

Minimal call to retrieve the last-used credential for a particular authentication "realm". In this case "realm" is a property that was previously set to match a particular service/location, and is now used to match credentials against:

```
credman.obtain(query_props={'realm': 'mysecretlair'})
```

Parameters

- **name** (*str, optional*) -- Name of the credential to be retrieved
- **prompt** (*str, optional*) -- Passed to `CredentialManager.get()` if a credential name was provided, or no suitable credential could be found by querying.
- **type_hint** (*str, optional*) -- In case no `type` property is included in `query_props`, this parameter is passed to `CredentialManager.get()`.
- **query_props** (*dict, optional*) -- Credential property to be used for querying for a suitable credential. When multiple credentials match a query, the last-used credential is selected.
- **expected_props** (*list or tuple, optional*) -- When specified, a credential will be inspected to contain properties matching all listed property names, or a `ValueError` will be raised.

Returns

Credential name (possibly different from the input, when a credential was discovered based on properties), and credential properties.

Return type

(str, dict)

Raises

ValueError -- Raised when no matching credential could be found and none was entered. Also raised, when a credential selected from a query result or a manually entered one is missing any of the properties with a name given in `expected_props`.

query(*, *_sortby=None*, *_reverse=True*, **kwargs*)

Query for all (matching) credentials, sorted by a property

This method is a companion of `query_()`, and the same limitations regarding credential discovery apply.

In contrast to `query_()`, this method return a list instead of yielding credentials one by one. This returned list is optionally sorted.

Parameters

- **_sortby** (*str, optional*) -- Name of a credential property to provide a value to sort by. Credentials that do not carry the specified property always sort last, regardless of sort order.
- **_reverse** (*bool, optional*) -- Flag whether to sort ascending or descending when sorting. By default credentials are return in descending property value order. This flag does not impact the fact that credentials without the property to sort by always sort last.
- ****kwargs** -- Pass on as-is to `query_()`

Returns
Each item is a 2-tuple. The first element in each tuple is the credential name, the second element is the credential record as returned by `get()` for any matching credential.

Return type
list(str, dict)

query_(***kwargs*)

Query for all (matching) credentials.

Credentials are yielded in no particular order.

This method cannot find credentials for which only a secret was deposited in the keyring.

This method does support lookup of credentials defined in DataLad's "provider" configurations.

Parameters
****kwargs** -- If not given, any found credential is yielded. Otherwise, any credential must match all property name/value pairs

Yields
tuple(str, dict) -- The first element in the tuple is the credential name, the second element is the credential record as returned by `get()` for any matching credential.

remove(*name*, *, *type_hint=None*)

Remove a credential, including all properties and secret

Presently, all supported backends require the specification of a credential `name` for lookup. This may change in the future, when support for alternative backends is added, at which point the `name` parameter would become optional, and additional parameters would be added.

Returns
True if a credential was removed, and False if not (because no respective credential was found).

Return type
bool

Raises
RuntimeError -- This exception is raised whenever a property cannot be removed successfully. Likely cause is that it is defined in a configuration scope or backend for which write-access is not supported.

secret_names = {'user_password': 'password'}

set(*name*, *, *_lastused=False*, *_suggested_name=None*, *_context=None*, ***kwargs*)

Set credential properties and secret

Presently, all supported backends require the specification of a credential `name` for storage. This may change in the future, when support for alternative backends is added, at which point the `name` parameter would become optional.

All properties provided as *kwargs* with keys not starting with _ and with values that are not `None` will be stored. If `kwargs` do not contain a `secret` specification, manual entry will be attempted. The associated prompt with be either the name of the `secret` field of a known credential (as identified via a `type` property), or the label `'secret'`.

All properties with an associated value of `None` will be removed (unset).

Parameters

- **name** (*str or None*) -- Credential name. If None, the name will be prompted for and setting the credential is skipped if no name is provided.
- **_lastused** (*bool, optional*) -- If set, automatically add an additional credential property `'last-used'` with the current timestamp in ISO 8601 format.
- **_suggested_name** (*str, optional*) -- If *name* is None, this name (if given) is presented as a default suggestion that can be accepted without having to enter it manually. If this name suggestion conflicts with an existing credential, it is ignored and not presented as a suggestion.
- **_context** (*str, optional*) -- If given, will be included in the prompt for a missing credential name to provide context for a user. It should be written to fit into a parenthical statement after "Enter a name to save the credential (...)", e.g. "for download from <URL>".
- ****kwargs** -- Any number of credential property key/value pairs to set (update), or remove. With one exception, values of `None` indicate removal of a property from a credential. However, `secret=None` does not lead to the removal of a credential's secret, because it would result in an incomplete credential. Instead, it will cause a credential's effective `secret` property to be written to the secret store. The effective secret might come from other sources, such as particular configuration scopes or environment variables (i.e., matching the `datalad.credential.<name>.secret` configuration item. Properties whose names start with an underscore are automatically removed prior storage.

Returns

key/values of all modified credential properties with respect to their previously recorded values. None is returned in case a user did not enter a missing credential name. If a user entered a credential name, it is included in the returned dictionary under the 'name' key.

Return type

dict or None

Raises

- **RuntimeError** -- This exception is raised whenever a property cannot be removed successfully. Likely cause is that it is defined in a configuration scope or backend for which write-access is not supported.
- **ValueError** -- When property names in kwargs are not syntax-compliant.

```
valid_property_names_regex = re.compile('[a-z0-9]+[a-z0-9-]*$')
```

datalad_next.credman.verify_property_names

datalad_next.credman.**verify_property_names**(*names*)

Check credential property names for syntax-compliance.

Parameters
names (*iterable*)

Raises
ValueError -- When any non-compliant property names were found

2.3.7 datalad_next.datasets

Representations of DataLad datasets built on git/git-annex repositories

Two sets of repository abstractions are available *LeanGitRepo* and *LeanAnnexRepo* vs. *LegacyGitRepo* and *LegacyAnnexRepo*.

LeanGitRepo and *LeanAnnexRepo* provide a more modern, small-ish interface and represent the present standard API for low-level repository operations. They are geared towards interacting with Git and git-annex more directly, and are more suitable for generator-like implementations, promoting low response latencies, and a leaner processing footprint.

The `Legacy*Repo` classes provide a, now legacy, low-level API to repository operations. This functionality stems from the earliest days of DataLad and implements paradigms and behaviors that are no longer common to the rest of the DataLad API. *LegacyGitRepo* and *LegacyAnnexRepo* should no longer be used in new developments, and are not documented here.

Dataset(*args, **kwargs)	Representation of a DataLad dataset/repository
LeanGitRepo	alias of `GitRepo`
LeanAnnexRepo(*args, **kwargs)	git-annex repository representation with a minimized API
LegacyGitRepo	alias of `GitRepo`
LegacyAnnexRepo	alias of `AnnexRepo`

datalad_next.datasets.Dataset

class datalad_next.datasets.**Dataset**(**args*, ***kwargs*)

Bases: `object`

Representation of a DataLad dataset/repository

This is the core data type of DataLad: a representation of a dataset. At its core, datasets are (git-annex enabled) Git repositories. This class provides all operations that can be performed on a dataset.

Creating a dataset instance is cheap, all actual operations are delayed until they are actually needed. Creating multiple *Dataset* class instances for the same Dataset location will automatically yield references to the same object.

A dataset instance comprises of two major components: a *repo* attribute, and a *config* attribute. The former offers access to low-level functionality of the Git or git-annex repository. The latter gives access to a dataset's configuration manager.

Most functionality is available via methods of this class, but also as stand-alone functions with the same name in *datalad.api*.

add_archive_content(*, *dataset=None*, *annex=None*, *add_archive_leading_dir=False*, *strip_leading_dirs=False*, *leading_dirs_depth=None*, *leading_dirs_consider=None*, *use_current_dir=False*, *delete=False*, *key=False*, *exclude=None*, *rename=None*, *existing='fail'*, *annex_options=None*, *copy=False*, *commit=True*, *allow_dirty=False*, *stats=None*, *drop_after=False*, *delete_after=False*)

Add content of an archive under git annex control.

Given an already annex'ed archive, extract and add its files to the dataset, and reference the original archive as a custom special remote.

Examples

Add files from the archive 'big_tarball.tar.gz', but keep big_tarball.tar.gz in the index:

```
> add_archive_content(path='big_tarball.tar.gz')
```

Add files from the archive 'tarball.tar.gz', and remove big_tarball.tar.gz from the index:

```
> add_archive_content(path='big_tarball.tar.gz', delete=True)
```

Add files from the archive 's3.zip' but remove the leading directory:

```
> add_archive_content(path='s3.zip', strip_leading_dirs=True)
```

Parameters

- **archive** (*str*) -- archive file or a key (if *key=True* specified).
- **dataset** (*Dataset or None, optional*) -- "specify the dataset to save. [Default: None]
- **annex** -- DEPRECATED. Use the 'dataset' parameter instead. [Default: None]
- **add_archive_leading_dir** (*bool, optional*) -- place extracted content under a directory which would correspond to the archive name with all suffixes stripped. E.g. the content of *archive.tar.gz* will be extracted under *archive/*. [Default: False]
- **strip_leading_dirs** (*bool, optional*) -- remove one or more leading directories from the archive layout on extraction. [Default: False]
- **leading_dirs_depth** -- maximum depth of leading directories to strip. If not specified (None), no limit. [Default: None]
- **leading_dirs_consider** (*list of str or None, optional*) -- regular expression(s) for directories to consider to strip away. [Default: None]
- **use_current_dir** (*bool, optional*) -- extract the archive under the current directory, not the directory where the archive is located. This parameter is applied automatically if *key=True* was used. [Default: False]
- **delete** (*bool, optional*) -- delete original archive from the filesystem/Git in current tree. Note that it will be of no effect if *key=True* is given. [Default: False]
- **key** (*bool, optional*) -- signal if provided archive is not actually a filename on its own but an annex key. The archive will be extracted in the current directory. [Default: False]
- **exclude** (*list of str or None, optional*) -- regular expressions for filenames which to exclude from being added to annex. Applied after --rename if that one is specified. For exact matching, use anchoring. [Default: None]

- **rename** (*list of str or None, optional*) -- regular expressions to rename files before added them under to Git. The first defines how to split provided string into two parts: Python regular expression (with groups), and replacement string. [Default: None]
- **existing** -- what operation to perform if a file from an archive tries to overwrite an existing file with the same name. 'fail' (default) leads to an error result, 'overwrite' silently replaces existing file, 'archive-suffix' instructs to add a suffix (prefixed with a '-') matching archive name from which file gets extracted, and if that one is present as well, 'numeric-suffix' is in effect in addition, when incremental numeric suffix (prefixed with a '.') is added until no name collision is longer detected. [Default: 'fail']
- **annex_options** (*str or None, optional*) -- additional options to pass to git-annex. [Default: None]
- **copy** (*bool, optional*) -- copy the content of the archive instead of moving. [Default: False]
- **commit** (*bool, optional*) -- don't commit upon completion. [Default: True]
- **allow_dirty** (*bool, optional*) -- flag that operating on a dirty repository (uncommitted or untracked content) is ok. [Default: False]
- **stats** -- ActivityStats instance for global tracking. [Default: None]
- **drop_after** (*bool, optional*) -- drop extracted files after adding to annex. [Default: False]
- **delete_after** (*bool, optional*) -- extract under a temporary directory, git-annex add, and delete afterwards. To be used to "index" files within annex without actually creating corresponding files under git. Note that *annex dropunused* would later remove that load. [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']

add_readme(*, *dataset=None*, *existing='skip'*)

Add basic information about DataLad datasets to a README file

The README file is added to the dataset and the addition is saved in the dataset. Note: Make sure that no unsaved modifications to your dataset's .gitattributes file exist.

Parameters

- **filename** (*str, optional*) -- Path of the README file within the dataset. [Default: 'README.md']
- **dataset** (Dataset *or None, optional*) -- Dataset to add information to. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **existing** (*{'skip', 'append', 'replace'}, optional*) -- How to react if a file with the target name already exists: 'skip': do nothing; 'append': append information to the existing file; 'replace': replace the existing file with new content. [Default: 'skip']
- **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']

addurls(*urlformat*, *filenameformat*, *, *dataset=None*, *input_type='ext'*, *exclude_autometa=None*, *meta=None*, *key=None*, *message=None*, *dry_run=False*, *fast=False*, *ifexists=None*, *missing_value=None*, *save=True*, *version_urls=False*, *cfg_proc=None*, *jobs=None*, *drop_after=False*, *on_collision='error'*)

Create and update a dataset from a list of URLs.

Format specification

Several arguments take format strings. These are similar to normal Python format strings where the names from *URL-FILE* (column names for a comma- or tab-separated file or properties for JSON) are available as placeholders. If *URL-FILE* is a CSV or TSV file, a positional index can also be used (i.e., "{0}" for the first column). Note that a placeholder cannot contain a ':' or '!'.

In addition, the *FILENAME-FORMAT* arguments has a few special placeholders.

- _repindex

  The constructed file names must be unique across all fields rows. To avoid collisions, the special placeholder "_repindex" can be added to the formatter. Its value will start at 0 and increment every time a file name repeats.

- _url_hostname, _urlN, _url_basename*

  Various parts of the formatted URL are available. Take "http://datalad.org/asciicast/seamless_nested_repos.sh" as an example.

  "datalad.org" is stored as "_url_hostname". Components of the URL's path can be referenced as "_urlN". "_url0" and "_url1" would map to "asciicast" and "seamless_nested_repos.sh", respectively. The final part of the path is also available as "_url_basename".

  This name is broken down further. "_url_basename_root" and "_url_basename_ext" provide access to the root name and extension. These values are similar to the result of os.path.splitext, but, in the case of multiple periods, the extension is identified using the same length heuristic that git-annex uses. As a result, the extension of "file.tar.gz" would be ".tar.gz", not ".gz". In addition, the fields "_url_basename_root_py" and "_url_basename_ext_py" provide access to the result of os.path.splitext.

- _url_filename*

  These are similar to _url_basename* fields, but they are obtained with a server request. This is useful if the file name is set in the Content-Disposition header.

Examples

Consider a file "avatars.csv" that contains:

```
who,ext,link
neurodebian,png,https://avatars3.githubusercontent.com/u/260793
datalad,png,https://avatars1.githubusercontent.com/u/8927200
```

To download each link into a file name composed of the 'who' and 'ext' fields, we could run:

```
$ datalad addurls -d avatar_ds avatars.csv '{link}' '{who}.{ext}'
```

The *-d avatar_ds* is used to create a new dataset in "$PWD/avatar_ds".

If we were already in a dataset and wanted to create a new subdataset in an "avatars" subdirectory, we could use "//" in the *FILENAME-FORMAT* argument:

```
$ datalad addurls avatars.csv '{link}' 'avatars//{who}.{ext}'
```

If the information is represented as JSON lines instead of comma separated values or a JSON array, you can use a utility like jq to transform the JSON lines into an array that addurls accepts:

```
$ ... | jq --slurp . | datalad addurls - '{link}' '{who}.{ext}'
```

Note: For users familiar with 'git annex addurl': A large part of this plugin's functionality can be viewed as transforming data from *URL-FILE* into a "url filename" format that fed to 'git annex addurl --batch --with-files'.

Parameters

- **urlfile** -- A file that contains URLs or information that can be used to construct URLs. Depending on the value of --input-type, this should be a comma- or tab-separated file (with a header as the first row) or a JSON file (structured as a list of objects with string values). If '-', read from standard input, taking the content as JSON when --input-type is at its default value of 'ext'. Alternatively, an iterable of dicts can be given.
- **urlformat** -- A format string that specifies the URL for each entry. See the 'Format Specification' section above.
- **filenameformat** -- Like *URL-FORMAT*, but this format string specifies the file to which the URL's content will be downloaded. The name should be a relative path and will be taken as relative to the top-level dataset, regardless of whether it is specified via *dataset* or inferred. The file name may contain directories. The separator "//" can be used to indicate that the left-side directory should be created as a new subdataset. See the 'Format Specification' section above.
- **dataset** (*Dataset or None, optional*) -- Add the URLs to this dataset (or possibly subdatasets of this dataset). An empty or non-existent directory is passed to create a new dataset. New subdatasets can be specified with *FILENAME- FORMAT*. [Default: None]
- **input_type** (*{'ext', 'csv', 'tsv', 'json'}, optional*) -- Whether *URL-FILE* should be considered a CSV file, TSV file, or JSON file. The default value, "ext", means to consider *URL-FILE* as a JSON file if it ends with ".json" or a TSV file if it ends with ".tsv". Otherwise, treat it as a CSV file. [Default: 'ext']
- **exclude_autometa** -- By default, metadata field=value pairs are constructed with each column in *URL-FILE*, excluding any single column that is specified via *URL-FORMAT*. This argument can be used to exclude columns that match a regular expression. If set to '*' or an empty string, automatic metadata extraction is disabled completely. This argument does not affect metadata set explicitly with --meta. [Default: None]
- **meta** -- A format string that specifies metadata. It should be structured as "<field>=<value>". As an example, "location={3}" would mean that the value for the "location" metadata field should be set the value of the fourth column. This option can be given multiple times. [Default: None]

- **key** -- A format string that specifies an annex key for the file content. In this case, the file is not downloaded; instead the key is used to create the file without content. The value should be structured as "[et:]<input backend>[-s<bytes>]--<hash>". The optional "et:" prefix, which requires git-annex 8.20201116 or later, signals to toggle extension state of the input backend (i.e., MD5 vs MD5E). As an example, "et:MD5-s{size}--{md5sum}" would use the 'md5sum' and 'size' columns to construct the key, migrating the key from MD5 to MD5E, with an extension based on the file name. Note: If the *input* backend itself is an annex extension backend (i.e., a backend with a trailing "E"), the key's extension will not be updated to match the extension of the corresponding file name. Thus, unless the input keys and file names are generated from git-annex, it is recommended to avoid using extension backends as input. If an extension is desired, use the plain variant as input and prepend "et:" so that git-annex will migrate from the plain backend to the extension variant. [Default: None]
- **message** (*None or str, optional*) -- Use this message when committing the URL additions. [Default: None]
- **dry_run** (*bool, optional*) -- Report which URLs would be downloaded to which files and then exit. [Default: False]
- **fast** (*bool, optional*) -- If True, add the URLs, but don't download their content. WARNING: ONLY USE THIS OPTION IF YOU UNDERSTAND THE CONSEQUENCES. If the content of the URLs is not downloaded, then datalad will refuse to retrieve the contents with *datalad get <file>* by default because the content of the URLs is not verified. Add *annex.security.allow-unverified-downloads = ACKTHPPT* to your git config to bypass the safety check. Underneath, this passes the *--fast* flag to *git annex addurl*. [Default: False]
- **ifexists** (*{None, 'overwrite', 'skip'}, optional*) -- What to do if a constructed file name already exists. The default behavior is to proceed with the *git annex addurl*, which will fail if the file size has changed. If set to 'overwrite', remove the old file before adding the new one. If set to 'skip', do not add the new file. [Default: None]
- **missing_value** (*None or str, optional*) -- When an empty string is encountered, use this value instead. [Default: None]
- **save** (*bool, optional*) -- by default all modifications to a dataset are immediately saved. Giving this option will disable this behavior. [Default: True]
- **version_urls** (*bool, optional*) -- Try to add a version ID to the URL. This currently only has an effect on HTTP URLs for AWS S3 buckets. s3:// URL versioning is not yet supported, but any URL that already contains a "versionId=" parameter will be used as is. [Default: False]
- **cfg_proc** -- Pass this cfg_proc value when calling *create* to make datasets. [Default: None]
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item NOTE: This option can only parallelize input retrieval (get) and output recording (save). DataLad does NOT parallelize your scripts for you. [Default: None]
- **drop_after** (*bool, optional*) -- drop files after adding to annex. [Default: False]
- **on_collision** (*{'error', 'error-if-different', 'take-first', 'take-last'}, optional*) -- What to do when more than one row produces the same file name. By default an error is triggered. "error-if-different" suppresses that error if rows for a given file name collision have the same URL and metadata. "take-first" or "take-last" indicate to instead take the first row or last row from each set of colliding rows. [Default: 'error']

- **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']

clean(*, *what=None*, *dry_run=False*, *recursive=False*, *recursion_limit=None*)

Clean up after DataLad (possible temporary files etc.)

Removes temporary files and directories left behind by DataLad and git-annex in a dataset.

Examples

Clean all known temporary locations of a dataset:

```
> clean()
```

Report on all existing temporary locations of a dataset:

```
> clean(dry_run=True)
```

Clean all known temporary locations of a dataset and all its subdatasets:

```
> clean(recursive=True)
```

Clean only the archive extraction caches of a dataset and all its subdatasets:

```
> clean(what='cached-archives', recursive=True)
```

Report on existing annex transfer files of a dataset and all its subdatasets:

```
> clean(what='annex-transfer', recursive=True, dry_run=True)
```

Parameters

- **dataset** (*Dataset or None, optional*) -- specify the dataset to perform the clean operation on. If no dataset is given, an attempt is made to identify the dataset in current working directory. [Default: None]
- **what** (*sequence of {'cached-archives', 'annex-tmp', 'annex-transfer', 'search-index'} or None, optional*) -- What to clean. If none specified -- all known targets are considered. [Default: None]
- **dry_run** (*bool, optional*) -- Report on cleanable locations - not actually cleaning up anything. [Default: False]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [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']

clone(*path=None*, *git_clone_opts=None*, *, *dataset=None*, *description=None*, *reckless=None*)

Obtain a dataset (copy) from a URL or local directory

The purpose of this command is to obtain a new clone (copy) of a dataset and place it into a not-yet-existing or empty directory. As such *clone* provides a strict subset of the functionality offered by *install*. Only a single dataset can be obtained, and immediate recursive installation of subdatasets is not supported. However, once a (super)dataset is installed via *clone*, any content, including subdatasets can be obtained by a subsequent *get* command.

Primary differences over a direct *git clone* call are 1) the automatic initialization of a dataset annex (pure Git repositories are equally supported); 2) automatic registration of the newly obtained dataset as a subdataset (submodule), if a parent dataset is specified; 3) support for additional resource identifiers (DataLad resource identifiers as used on datasets.datalad.org, and RIA store URLs as used for store.datalad.org - optionally in specific versions as identified by a branch or a tag; see examples); and 4) automatic configurable generation of alternative access URL for common cases (such as appending '.git' to the URL in case the accessing the base URL failed).

In case the clone is registered as a subdataset, the original URL passed to *clone* is recorded in *.gitmodules* of the parent dataset in addition to the resolved URL used internally for git-clone. This allows to preserve datalad specific URLs like ria+ssh://... for subsequent calls to *get* if the subdataset was locally removed later on.

By default, the command returns a single Dataset instance for an installed dataset, regardless of whether it was newly installed ('ok' result), or found already installed from the specified source ('notneeded' result).

URL mapping configuration

'clone' supports the transformation of URLs via (multi-part) substitution specifications. A substitution specification is defined as a configuration setting 'datalad.clone.url-substition.<seriesID>' with a string containing a match and substitution expression, each following Python's regular expression syntax. Both expressions are concatenated to a single string with an arbitrary delimiter character. The delimiter is defined by prefixing the string with the delimiter. Prefix and delimiter are stripped from the expressions (Example: ",^http://(.*)$,https://1"). This setting can be defined multiple times, using the same '<seriesID>'. Substitutions in a series will be applied incrementally, in order of their definition. The first substitution in such a series must match, otherwise no further substitutions in a series will be considered. However, following the first match all further substitutions in a series are processed, regardless whether intermediate expressions match or not. Substitution series themselves have no particular order, each matching series will result in a candidate clone URL. Consequently, the initial match specification in a series should be as precise as possible to prevent inflation of candidate URLs.

See also:

handbook:3-001 (http://handbook.datalad.org/symbols)
More information on Remote Indexed Archive (RIA) stores

Examples

Install a dataset from GitHub into the current directory:

```
> clone(source='https://github.com/datalad-datasets/longnow-podcasts.git')
```

Install a dataset into a specific directory:

```
> clone(source='https://github.com/datalad-datasets/longnow-podcasts.git',
       path='myfavpodcasts')
```

Install a dataset as a subdataset into the current dataset:

```
> clone(dataset='.',
       source='https://github.com/datalad-datasets/longnow-podcasts.git')
```

Install the main superdataset from datasets.datalad.org:

```
> clone(source='///')
```

Install a dataset identified by a literal alias from store.datalad.org:

```
> clone(source='ria+http://store.datalad.org#~hcp-openaccess')
```

Install a dataset in a specific version as identified by a branch or tag name from store.datalad.org:

```
> clone(source='ria+http://store.datalad.org#76b6ca66-36b1-11ea-a2e6-
↪f0d5bf7b5561@myidentifier')
```

Install a dataset with group-write access permissions:

```
> clone(source='http://example.com/dataset', reckless='shared-group')
```

Parameters

- **source** (*str*) -- URL, DataLad resource identifier, local path or instance of dataset to be cloned.
- **path** -- path to clone into. If no *path* is provided a destination path will be derived from a source URL similar to git clone. [Default: None]
- **git_clone_opts** -- A list of command line arguments to pass to git clone. Note that not all options will lead to viable results. For example '--single- branch' will not result in a functional annex repository because both a regular branch and the git-annex branch are required. Note that a version in a RIA URL takes precedence over '--branch'. [Default: None]
- **dataset** (*Dataset or None, optional*) -- (parent) dataset to clone into. If given, the newly cloned dataset is registered as a subdataset of the parent. Also, if given, relative paths are interpreted as being relative to the parent dataset, and not relative to the working directory. [Default: None]
- **description** (*str or None, optional*) -- short description to use for a dataset location. Its primary purpose is to help humans to identify a dataset copy (e.g., "mike's dataset on lab server"). Note that when a dataset is published, this information becomes available on the remote side. [Default: None]

- **reckless** (*{None, True, False, 'auto', 'ephemeral'} or shared-..., optional*) -- Obtain a dataset or subdatset and set it up in a potentially unsafe way for performance, or access reasons. Use with care, any dataset is marked as 'untrusted'. The reckless mode is stored in a dataset's local configuration under 'datalad.clone.reckless', and will be inherited to any of its subdatasets. Supported modes are: ['auto']: hard-link files between local clones. In-place modification in any clone will alter original annex content. ['ephemeral']: symlink annex to origin's annex and discard local availability info via git- annex-dead 'here' and declares this annex private. Shares an annex between origin and clone w/o git-annex being aware of it. In case of a change in origin you need to update the clone before you're able to save new content on your end. Alternative to 'auto' when hardlinks are not an option, or number of consumed inodes needs to be minimized. Note that this mode can only be used with clones from non-bare repositories or a RIA store! Otherwise two different annex object tree structures (dirhashmixed vs dirhashlower) will be used simultaneously, and annex keys using the respective other structure will be inaccessible. ['shared-<mode>']: set up repository and annex permission to enable multi-user access. This disables the standard write protection of annex'ed files. <mode> can be any value support by 'git init --shared=', such as 'group', or 'all'. [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: constraint:action:{install}]
- **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: 'successdatasets-or- 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: 'item-or-list']

close()

Perform operations which would close any possible process using this Dataset

property config

Get a `ConfigManager` instance for a dataset's configuration

In case a dataset does not (yet) have an existing corresponding repository, the returned `ConfigManager` is the global instance that is also provided via `datalad.cfg`.

Note, that this property is evaluated every time it is used. If used multiple times within a function it's probably a good idea to store its value in a local variable and use this variable instead.

Return type
ConfigManager

configuration(*spec=None*, *, *scope=None*, *dataset=None*, *recursive=False*, *recursion_limit=None*)

Get and set dataset, dataset-clone-local, or global configuration

This command works similar to git-config, but some features are not supported (e.g., modifying system configuration), while other features are not available in git-config (e.g., multi-configuration queries).

Query and modification of three distinct configuration scopes is supported:

- 'branch': the persistent configuration in .datalad/config of a dataset branch
- 'local': a dataset clone's Git repository configuration in .git/config
- 'global': non-dataset-specific configuration (usually in $USER/.gitconfig)

Modifications of the persistent 'branch' configuration will not be saved by this command, but have to be committed with a subsequent *save* call.

Rules of precedence regarding different configuration scopes are the same as in Git, with two exceptions: 1) environment variables can be used to override any datalad configuration, and have precedence over any other configuration scope (see below). 2) the 'branch' scope is considered in addition to the standard git configuration scopes. Its content has lower precedence than Git configuration scopes, but it is committed to a branch, hence can be used to ship (default and branch-specific) configuration with a dataset.

Besides storing configuration settings statically via this command or `git config`, DataLad also reads any DATALAD_* environment on process startup or import, and maps it to a configuration item. Their values take precedence over any other specification. In variable names `_` encodes a `.` in the configuration name, and `__` encodes a `-`, such that `DATALAD_SOME__VAR` is mapped to `datalad.some-var`. Additionally, a DATALAD_CONFIG_OVERRIDES_JSON environment variable is queried, which may contain configuration key-value mappings as a JSON-formatted string of a JSON-object:

```
DATALAD_CONFIG_OVERRIDES_JSON='{"datalad.credential.example_com.user": "jane", .
↪..}'
```

This is useful when characters are part of the configuration key that cannot be encoded into an environment variable name. If both individual configuration variables *and* JSON-overrides are used, the former take precedent over the latter, overriding the respective *individual* settings from configurations declared in the JSON-overrides.

This command supports recursive operation for querying and modifying configuration across a hierarchy of datasets.

Examples

Dump the effective configuration, including an annotation for common items:

```
> configuration()
```

Query two configuration items:

```
> configuration('get', ['user.name', 'user.email'])
```

Recursively set configuration in all (sub)dataset repositories:

```
> configuration('set', [('my.config.name', 'value')], recursive=True)
```

Modify the persistent branch configuration (changes are not committed):

```
> configuration('set', [('my.config.name', 'value')], scope='branch')
```

Parameters

- **action** (*{'dump', 'get', 'set', 'unset'}, optional*) -- which action to perform. [Default: 'dump']
- **spec** -- configuration name (for actions 'get' and 'unset'), or name/value pair (for action 'set'). [Default: None]
- **scope** (*{'global', 'local', 'branch', None}, optional*) -- scope for getting or setting configuration. If no scope is declared for a query, all configuration sources (including overrides via environment variables) are considered according to the normal rules of precedence. A 'get' action can be constrained to scope 'branch', otherwise 'global' is used when not operating on a dataset, or 'local' (including 'global', when operating on a dataset. For action 'dump', a scope selection is ignored and all available scopes are considered. [Default: None]
- **dataset** (*Dataset or None, optional*) -- specify the dataset to query or to configure. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [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']

copy_file(*, *dataset=None*, *recursive=False*, *target_dir=None*, *specs_from=None*, *message=None*)

Copy files and their availability metadata from one dataset to another.

The difference to a system copy command is that here additional content availability information, such as registered URLs, is also copied to the target dataset. Moreover, potentially required git-annex special remote configurations are detected in a source dataset and are applied to a target dataset in an analogous fashion. It is possible to copy a file for which no content is available locally, by just copying the required metadata on content identity and availability.

Note: At the moment, only URLs for the special remotes 'web' (git-annex built-in) and 'datalad' are recognized and transferred.

The interface is modeled after the POSIX 'cp' command, but with one additional way to specify what to copy where: *specs_from* allows the caller to flexibly input source-destination path pairs.

This command can copy files out of and into a hierarchy of nested datasets. Unlike with other DataLad command, the *recursive* switch does not enable recursion into subdatasets, but is analogous to the POSIX 'cp' command switch and enables subdirectory recursion, regardless of dataset boundaries. It is not necessary to enable recursion in order to save changes made to nested target subdatasets.

Examples

Copy a file into a dataset 'myds' using a path and a target directory specification, and save its addition to 'myds':

```
> copy_file('path/to/myfile', dataset='path/to/myds')
```

Copy a file to a dataset 'myds' and save it under a new name by providing two paths:

```
> copy_file(path=['path/to/myfile', 'path/to/myds/newname'],
            dataset='path/to/myds')
```

Copy a file into a dataset without saving it:

```
> copy_file('path/to/myfile', target_dir='path/to/myds/')
```

Copy a directory and its subdirectories into a dataset 'myds' and save the addition in 'myds':

```
> copy_file('path/to/dir/', recursive=True, dataset='path/to/myds')
```

Copy files using a path and optionally target specification from a file:

```
> copy_file(dataset='path/to/myds', specs_from='path/to/specfile')
```

Parameters

- **path** (*sequence of str or None, optional*) -- paths to copy (and possibly a target path to copy to). [Default: None]
- **dataset** (Dataset *or None, optional*) -- root dataset to save after copy operations are completed. All destination paths must be within this dataset, or its subdatasets. If no dataset is given, dataset modifications will be left unsaved. [Default: None]
- **recursive** (*bool, optional*) -- copy directories recursively. [Default: False]
- **target_dir** (*str or None, optional*) -- copy all source files into this DIRECTORY. This value is overridden by any explicit destination path provided via 'specs_from'. When not given, this defaults to the path of the dataset specified via 'dataset'. [Default: None]
- **specs_from** -- read list of source (and destination) path names from a given file, or stdin (with '-'). Each line defines either a source path, or a source/destination path pair (separated by a null byte character). Alternatively, a list of 2-tuples with source/destination pairs can be given. [Default: None]
- **message** (*str or None, optional*) -- a description of the state or the changes made to a 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']

create(*initopts=None*, ***, *force=False*, *description=None*, *dataset=None*, *annex=True*, *fake_dates=False*, *cfg_proc=None*)

Create a new dataset from scratch.

This command initializes a new dataset at a given location, or the current directory. The new dataset can optionally be registered in an existing superdataset (the new dataset's path needs to be located within the superdataset for that, and the superdataset needs to be given explicitly via *dataset*). It is recommended to provide a brief description to label the dataset's nature *and* location, e.g. "Michael's music on black laptop". This helps humans to identify data locations in distributed scenarios. By default an identifier comprised of user and machine name, plus path will be generated.

This command only creates a new dataset, it does not add existing content to it, even if the target directory already contains additional files or directories.

Plain Git repositories can be created via *annex=False*. However, the result will not be a full dataset, and, consequently, not all features are supported (e.g. a description).

To create a local version of a remote dataset use the ~datalad.api.install command instead.

Note: Power-user info: This command uses git init and git annex init to prepare the new dataset. Registering to a superdataset is performed via a git submodule add operation in the discovered superdataset.

Examples

Create a dataset 'mydataset' in the current directory:

```
> create(path='mydataset')
```

Apply the text2git procedure upon creation of a dataset:

```
> create(path='mydataset', cfg_proc='text2git')
```

Create a subdataset in the root of an existing dataset:

```
> create(dataset='.', path='mysubdataset')
```

Create a dataset in an existing, non-empty directory:

```
> create(force=True)
```

Create a plain Git repository:

```
> create(path='mydataset', annex=False)
```

Parameters

- **path** (*str or Dataset or None, optional*) -- path where the dataset shall be created, directories will be created as necessary. If no location is provided, a dataset will be created in the location specified by *dataset* (if given) or the current working directory. Either way the command will error if the target directory is not empty. Use *force* to create a dataset in a non- empty directory. [Default: None]
- **initopts** -- options to pass to git init. Options can be given as a list of command line arguments or as a GitPython-style option dictionary. Note that not all options will lead to viable results. For example ' --bare' will not yield a repository where DataLad can adjust files in its working tree. [Default: None]
- **force** (*bool, optional*) -- enforce creation of a dataset in a non-empty directory. [Default: False]
- **description** (*str or None, optional*) -- short description to use for a dataset location. Its primary purpose is to help humans to identify a dataset copy (e.g., "mike's dataset on lab server"). Note that when a dataset is published, this information becomes available on the remote side. [Default: None]
- **dataset** (*Dataset or None, optional*) -- specify the dataset to perform the create operation on. If a dataset is given along with *path*, a new subdataset will be created in it at the *path* provided to the create command. If a dataset is given but *path* is unspecified, a new dataset will be created at the location specified by this option. [Default: None]
- **annex** (*bool, optional*) -- if disabled, a plain Git repository will be created without any annex. [Default: True]
- **fake_dates** (*bool, optional*) -- Configure the repository to use fake dates. The date for a new commit will be set to one second later than the latest commit in the repository. This can be used to anonymize dates. [Default: False]
- **cfg_proc** -- Run cfg_PROC procedure(s) (can be specified multiple times) on the created dataset. Use *run_procedure(discover=True)* to get a list of available procedures, such as cfg_text2git. [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: constraint:(action:{create} or status:{ok, notneeded})]

- **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: 'datasets']
- **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: 'item-or-list']

create_sibling(*, *name=None*, *target_dir=None*, *target_url=None*, *target_pushurl=None*, *dataset=None*, *recursive=False*, *recursion_limit=None*, *existing='error'*, *shared=None*, *group=None*, *ui=False*, *as_common_datasrc=None*, *publish_by_default=None*, *publish_depends=None*, *annex_wanted=None*, *annex_group=None*, *annex_groupwanted=None*, *inherit=False*, *since=None*)

Create a dataset sibling on a UNIX-like Shell (local or SSH)-accessible machine

Given a local dataset, and a path or SSH login information this command creates a remote dataset repository and configures it as a dataset sibling to be used as a publication target (see *publish* command).

Various properties of the remote sibling can be configured (e.g. name location on the server, read and write access URLs, and access permissions.

Optionally, a basic web-viewer for DataLad datasets can be installed at the remote location.

This command supports recursive processing of dataset hierarchies, creating a remote sibling for each dataset in the hierarchy. By default, remote siblings are created in hierarchical structure that reflects the organization on the local file system. However, a simple templating mechanism is provided to produce a flat list of datasets (see --target-dir).

Parameters

- **sshurl** (*str*) -- Login information for the target server. This can be given as a URL (ssh://host/path), SSH-style (user@host:path) or just a local path. Unless overridden, this also serves the future dataset's access URL and path on the server.
- **name** (*str or None, optional*) -- sibling name to create for this publication target. If *recursive* is set, the same name will be used to label all the subdatasets' siblings. When creating a target dataset fails, no sibling is added. [Default: None]

- **target_dir** (*str or None, optional*) -- path to the directory *on the server* where the dataset shall be created. By default this is set to the URL (or local path) specified via *sshurl*. If a relative path is provided here, it is interpreted as being relative to the user's home directory on the server (or relative to *sshurl*, when that is a local path). Additional features are relevant for recursive processing of datasets with subdatasets. By default, the local dataset structure is replicated on the server. However, it is possible to provide a template for generating different target directory names for all (sub)datasets. Templates can contain certain placeholder that are substituted for each (sub)dataset. For example: "/mydirectory/dataset%%RELNAME". Supported placeholders: %%RELNAME - the name of the datasets, with any slashes replaced by dashes. [Default: None]
- **target_url** (*str or None, optional*) -- "public" access URL of the to-be-created target dataset(s) (default: *sshurl*). Accessibility of this URL determines the access permissions of potential consumers of the dataset. As with *target_dir*, templates (same set of placeholders) are supported. Also, if specified, it is provided as the annex description. [Default: None]
- **target_pushurl** (*str or None, optional*) -- In case the *target_url* cannot be used to publish to the dataset, this option specifies an alternative URL for this purpose. As with *target_url*, templates (same set of placeholders) are supported. [Default: None]
- **dataset** (Dataset *or None, optional*) -- specify the dataset to create the publication target for. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **existing** (*{'skip', 'error', 'reconfigure', 'replace'}, optional*) -- action to perform, if a sibling is already configured under the given name and/or a target (non-empty) directory already exists. In this case, a dataset can be skipped ('skip'), the sibling configuration be updated ('reconfigure'), or process interrupts with error ('error'). DANGER ZONE: If 'replace' is used, an existing target directory will be forcefully removed, re-initialized, and the sibling (re-)configured (thus implies 'reconfigure'). *replace* could lead to data loss, so use with care. To minimize possibility of data loss, in interactive mode DataLad will ask for confirmation, but it would raise an exception in non-interactive mode. [Default: 'error']
- **shared** (*str or bool or None, optional*) -- if given, configures the access permissions on the server for multi- users (this could include access by a webserver!). Possible values for this option are identical to those of *git init --shared* and are described in its documentation. [Default: None]
- **group** (*str or None, optional*) -- Filesystem group for the repository. Specifying the group is particularly important when shared="group". [Default: None]
- **ui** (*bool or str, optional*) -- publish a web interface for the dataset with an optional user- specified name for the html at publication target. defaults to *index.html* at dataset root. [Default: False]
- **as_common_datasrc** -- configure the created sibling as a common data source of the dataset that can be automatically used by all consumers of the dataset (technical: git-annex auto-enabled special remote). [Default: None]
- **publish_by_default** (*list of str or None, optional*) -- add a refspec to be published to this sibling by default if nothing specified. [Default: None]

- **publish_depends** (*list of str or None, optional*) -- add a dependency such that the given existing sibling is always published prior to the new sibling. This equals setting a configuration item 'remote.SIBLINGNAME.datalad-publish-depends'. Multiple dependencies can be given as a list of sibling names. [Default: None]
- **annex_wanted** (*str or None, optional*) -- expression to specify 'wanted' content for the repository/sibling. See https://git-annex.branchable.com/git-annex-wanted/ for more information. [Default: None]
- **annex_group** (*str or None, optional*) -- expression to specify a group for the repository. See https://git- annex.branchable.com/git-annex-group/ for more information. [Default: None]
- **annex_groupwanted** (*str or None, optional*) -- expression for the groupwanted. Makes sense only if annex_wanted="groupwanted" and annex-group is given too. See https://git-annex.branchable.com/git-annex-groupwanted/ for more information. [Default: None]
- **inherit** (*bool, optional*) -- if sibling is missing, inherit settings (git config, git annex wanted/group/groupwanted) from its super-dataset. [Default: False]
- **since** (*str or None, optional*) -- limit processing to subdatasets that have been changed since a given state (by tag, branch, commit, etc). This can be used to create siblings for recently added subdatasets. If '^' is given, the last state of the current branch at the sibling is taken as a starting point. [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']

create_sibling_gin(*, *dataset=None*, *recursive=False*, *recursion_limit=None*, *name='gin'*, *existing='error'*, *api='https://gin.g-node.org'*, *credential=None*, *access_protocol='https-ssh'*, *publish_depends=None*, *private=False*, *description=None*, *dry_run=False*)

Create a dataset sibling on a GIN site (with content hosting)

GIN (G-Node infrastructure) is a free data management system. It is a GitHub-like, web-based repository store and provides fine-grained access control to shared data. GIN is built on Git and git-annex, and can natively host DataLad datasets, including their data content!

This command uses the main GIN instance at https://gin.g-node.org as the default target, but other deployments can be used via the 'api' parameter.

An SSH key, properly registered at the GIN instance, is required for data upload via DataLad. Data download from public projects is also possible via anonymous HTTP.

In order to be able to use this command, a personal access token has to be generated on the platform (Account->Your Settings->Applications->Generate New Token).

This command can be configured with "datalad.create-sibling-ghlike.extra-remote-settings.NETLOC.KEY=VALUE" in order to add any local KEY = VALUE configuration to the created sibling in the local *.git/config* file. NETLOC is the domain of the Gin instance to apply the configuration for. This leads to a behavior that is equivalent to calling datalad's `siblings('configure', ...)``||``siblings configure` command with the respective KEY-VALUE pair after creating the sibling. The configuration, like any other, could be set at user- or system level, so users do not need to add this configuration to every sibling created with the service at NETLOC themselves.

Added in version 0.16.

Examples

Create a repo 'myrepo' on GIN and register it as sibling 'mygin':

```
> create_sibling_gin('myrepo', name='mygin', dataset='.')
```

Create private repos with name(-prefix) 'myrepo' on GIN for a dataset and all its present subdatasets:

```
> create_sibling_gin('myrepo', dataset='.', recursive=True, private=True)
```

Create a sibling repo on GIN, and register it as a common data source in the dataset that is available regardless of whether the dataset was directly cloned from GIN:

```
> ds = Dataset('.')
> ds.create_sibling_gin('myrepo', name='gin')
# first push creates git-annex branch remotely and obtains annex UUID
> ds.push(to='gin')
> ds.siblings('configure', name='gin', as_common_datasrc='gin-storage')
# announce availability (redo for other siblings)
> ds.push(to='gin')
```

Parameters

- **reponame** (*str*) -- repository name, optionally including an '<organization>/' prefix if the repository shall not reside under a user's namespace. When operating recursively, a suffix will be appended to this name for each subdataset.
- **dataset** (*Dataset or None, optional*) -- dataset to create the publication target for. If not given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **name** (*str or None, optional*) -- name of the sibling in the local dataset installation (remote name). [Default: 'gin']
- **existing** (*{'skip', 'error', 'reconfigure', 'replace'}, optional*) -- behavior when already existing or configured siblings are discovered: skip the dataset ('skip'), update the configuration ('reconfigure'), or fail ('error'). DEPRECATED DANGER ZONE: With 'replace', an existing repository will be irreversibly removed, re-initialized, and the sibling (re-)configured (thus implies 'reconfigure'). *replace* could lead to data loss! In interactive sessions a confirmation prompt is shown, an exception is raised in non-interactive sessions. The 'replace' mode will be removed in a future release. [Default: 'error']
- **api** (*str or None, optional*) -- URL of the GIN instance without an 'api/<version>' suffix. [Default: 'https://gin.g-node.org']
- **credential** (*str or None, optional*) -- name of the credential providing a personal access token to be used for authorization. The token can be supplied via configuration setting 'datalad.credential.<name>.secret', or environment variable DATALAD_CREDENTIAL_<NAME>_SECRET, or will be queried from the active credential store using the provided name. If none is provided, the last-used token for the API URL realm will be used. If no matching credential exists, a credential named after the hostname part of the API URL is tried as a last fallback. [Default: None]
- **access_protocol** (*{'https', 'ssh', 'https-ssh'}, optional*) -- access protocol/URL to configure for the sibling. With 'https-ssh' SSH will be used for write access, whereas HTTPS is used for read access. [Default: 'https-ssh']
- **publish_depends** (*list of str or None, optional*) -- add a dependency such that the given existing sibling is always published prior to the new sibling. This equals setting a configuration item 'remote.SIBLINGNAME.datalad-publish-depends'. Multiple dependencies can be given as a list of sibling names. [Default: None]
- **private** (*bool, optional*) -- if set, create a private repository. [Default: False]
- **description** (*str or None, optional*) -- Brief description, displayed on the project's page. [Default: None]
- **dry_run** (*bool, optional*) -- if set, no repository will be created, only tests for sibling name collisions will be performed, and would-be repository names are reported for all relevant datasets. [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']

create_sibling_gitea(*, *dataset=None*, *recursive=False*, *recursion_limit=None*, *name='gitea'*, *existing='error'*, *api='https://gitea.com'*, *credential=None*, *access_protocol='https'*, *publish_depends=None*, *private=False*, *description=None*, *dry_run=False*)

Create a dataset sibling on a Gitea site

Gitea is a lightweight, free and open source code hosting solution with low resource demands that enable running it on inexpensive devices like a Raspberry Pi.

This command uses the main Gitea instance at https://gitea.com as the default target, but other deployments can be used via the 'api' parameter.

In order to be able to use this command, a personal access token has to be generated on the platform (Account->Settings->Applications->Generate Token).

This command can be configured with "datalad.create-sibling-ghlike.extra-remote-settings.NETLOC.KEY=VALUE" in order to add any local KEY = VALUE configuration to the created sibling in the local *.git/config* file. NETLOC is the domain of the Gitea instance to apply the configuration for. This leads to a behavior that is equivalent to calling datalad's `siblings('configure', ...)``||``siblings configure` command with the respective KEY-VALUE pair after creating the sibling. The configuration, like any other, could be set at user- or system level, so users do not need to add this configuration to every sibling created with the service at NETLOC themselves.

Added in version 0.16.

Parameters

- **reponame** (*str*) -- repository name, optionally including an '<organization>/' prefix if the repository shall not reside under a user's namespace. When operating recursively, a suffix will be appended to this name for each subdataset.
- **dataset** (Dataset *or None, optional*) -- dataset to create the publication target for. If not given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **name** (*str or None, optional*) -- name of the sibling in the local dataset installation (remote name). [Default: 'gitea']
- **existing** (*{'skip', 'error', 'reconfigure', 'replace'}, optional*) -- behavior when already existing or configured siblings are discovered: skip the dataset ('skip'), update the configuration ('reconfigure'), or fail ('error'). DEPRECATED DANGER ZONE: With 'replace', an existing repository will be irreversibly removed, re-initialized, and the sibling (re-)configured (thus implies 'reconfigure'). *replace* could lead to data loss! In interactive sessions a confirmation prompt is shown, an exception is raised in non-interactive sessions. The 'replace' mode will be removed in a future release. [Default: 'error']
- **api** (*str or None, optional*) -- URL of the Gitea instance without a 'api/<version>' suffix. [Default: 'https://gitea.com']
- **credential** (*str or None, optional*) -- name of the credential providing a personal access token to be used for authorization. The token can be supplied via configuration setting 'datalad.credential.<name>.secret', or environment variable DATALAD_CREDENTIAL_<NAME>_SECRET, or will be queried from the active credential store using the provided name. If none is provided, the last-used token for the API URL realm will be used. If no matching credential exists, a credential named after the hostname part of the API URL is tried as a last fallback. [Default: None]
- **access_protocol** (*{'https', 'ssh', 'https-ssh'}, optional*) -- access protocol/URL to configure for the sibling. With 'https-ssh' SSH will be used for write access, whereas HTTPS is used for read access. [Default: 'https']
- **publish_depends** (*list of str or None, optional*) -- add a dependency such that the given existing sibling is always published prior to the new sibling. This equals setting a configuration item 'remote.SIBLINGNAME.datalad-publish-depends'. Multiple dependencies can be given as a list of sibling names. [Default: None]
- **private** (*bool, optional*) -- if set, create a private repository. [Default: False]
- **description** (*str or None, optional*) -- Brief description, displayed on the project's page. [Default: None]
- **dry_run** (*bool, optional*) -- if set, no repository will be created, only tests for sibling name collisions will be performed, and would-be repository names are reported for all relevant datasets. [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']

create_sibling_github(*, *dataset=None*, *recursive=False*, *recursion_limit=None*, *name='github'*, *existing='error'*, *github_login=None*, *credential=None*, *github_organization=None*, *access_protocol='https'*, *publish_depends=None*, *private=False*, *description=None*, *dryrun=False*, *dry_run=False*, *api='https://api.github.com'*)

Create dataset sibling on GitHub.org (or an enterprise deployment).

GitHub is a popular commercial solution for code hosting and collaborative development. GitHub cannot host dataset content (but see LFS, http://handbook.datalad.org/r.html?LFS). However, in combination with other data sources and siblings, publishing a dataset to GitHub can facilitate distribution and exchange, while still allowing any dataset consumer to obtain actual data content from alternative sources.

In order to be able to use this command, a personal access token has to be generated on the platform (Account->Settings->Developer Settings->Personal access tokens->Generate new token).

This command can be configured with "datalad.create-sibling-ghlike.extra-remote-settings.NETLOC.KEY=VALUE" in order to add any local KEY = VALUE configuration to the created sibling in the local *.git/config* file. NETLOC is the domain of the Github instance to apply the configuration for. This leads to a behavior that is equivalent to calling datalad's `siblings('configure', ...)``||``siblings configure` command with the respective KEY-VALUE pair after creating the

sibling. The configuration, like any other, could be set at user- or system level, so users do not need to add this configuration to every sibling created with the service at NETLOC themselves.

Changed in version 0.16: The API has been aligned with the some `create_sibling_...` commands of other GitHub-like services, such as GOGS, GIN, GitTea.

Deprecated since version 0.16: The `dryrun` option will be removed in a future release, use the renamed `dry_run` option instead. The `github_login` option will be removed in a future release, use the `credential` option instead. The `github_organization` option will be removed in a future release, prefix the reposity name with `<org>/` instead.

Examples

Use a new sibling on GIN as a common data source that is auto- available when cloning from GitHub:

```
> ds = Dataset('.')

# the sibling on GIN will host data content
> ds.create_sibling_gin('myrepo', name='gin')

# the sibling on GitHub will be used for collaborative work
> ds.create_sibling_github('myrepo', name='github')

# register the storage of the public GIN repo as a data source
> ds.siblings('configure', name='gin', as_common_datasrc='gin-storage')

# announce its availability on github
> ds.push(to='github')
```

Parameters

- **reponame** (*str*) -- repository name, optionally including an '<organization>/' prefix if the repository shall not reside under a user's namespace. When operating recursively, a suffix will be appended to this name for each subdataset.
- **dataset** (Dataset *or None, optional*) -- dataset to create the publication target for. If not given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **name** (*str or None, optional*) -- name of the sibling in the local dataset installation (remote name). [Default: 'github']
- **existing** (*{'skip', 'error', 'reconfigure', 'replace'}, optional*) -- behavior when already existing or configured siblings are discovered: skip the dataset ('skip'), update the configuration ('reconfigure'), or fail ('error'). DEPRECATED DANGER ZONE: With 'replace', an existing repository will be irreversibly removed, re-initialized, and the sibling (re-)configured (thus implies 'reconfigure'). *replace* could lead to data loss! In interactive sessions a confirmation prompt is shown, an exception is raised in non-interactive sessions. The 'replace' mode will be removed in a future release. [Default: 'error']

- **github_login** (*str or None, optional*) -- Deprecated, use the credential parameter instead. If given must be a personal access token. [Default: None]
- **credential** (*str or None, optional*) -- name of the credential providing a personal access token to be used for authorization. The token can be supplied via configuration setting 'datalad.credential.<name>.secret', or environment variable DATALAD_CREDENTIAL_<NAME>_SECRET, or will be queried from the active credential store using the provided name. If none is provided, the last-used token for the API URL realm will be used. If no matching credential exists, a credential named after the hostname part of the API URL is tried as a last fallback. [Default: None]
- **github_organization** (*str or None, optional*) -- Deprecated, prepend a repo name with an '<orgname>/' prefix instead. [Default: None]
- **access_protocol** (*{'https', 'ssh', 'https-ssh'}, optional*) -- access protocol/URL to configure for the sibling. With 'https-ssh' SSH will be used for write access, whereas HTTPS is used for read access. [Default: 'https']
- **publish_depends** (*list of str or None, optional*) -- add a dependency such that the given existing sibling is always published prior to the new sibling. This equals setting a configuration item 'remote.SIBLINGNAME.datalad-publish-depends'. Multiple dependencies can be given as a list of sibling names. [Default: None]
- **private** (*bool, optional*) -- if set, create a private repository. [Default: False]
- **description** (*str or None, optional*) -- Brief description, displayed on the project's page. [Default: None]
- **dryrun** (*bool, optional*) -- Deprecated. Use the renamed `dry_run` parameter. [Default: False]
- **dry_run** (*bool, optional*) -- if set, no repository will be created, only tests for sibling name collisions will be performed, and would-be repository names are reported for all relevant datasets. [Default: False]
- **api** (*str or None, optional*) -- URL of the GitHub instance API. [Default: 'https://api.github.com']
- **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']

create_sibling_gitlab(*, *site=None*, *project=None*, *layout=None*, *dataset=None*, *recursive=False*, *recursion_limit=None*, *name=None*, *existing='error'*, *access=None*, *publish_depends=None*, *description=None*, *dryrun=False*, *dry_run=False*)

Create dataset sibling at a GitLab site

An existing GitLab project, or a project created via the GitLab web interface can be configured as a sibling with the siblings command. Alternatively, this command can create a GitLab project at any location/path a given user has appropriate permissions for. This is particularly helpful for recursive sibling creation for subdatasets. API access and authentication are implemented via python-gitlab, and all its features are supported. A particular GitLab site must be configured in a named section of a python-gitlab.cfg file (see https://python-gitlab.readthedocs.io/en/stable/cli.html#configuration for details), such as:

```
[mygit]
url = https://git.example.com
api_version = 4
private_token = abcdefghijklmnopqrst
```

Subsequently, this site is identified by its name ('mygit' in the example above).

(Recursive) sibling creation for all, or a selected subset of subdatasets is supported with two different project layouts (see --layout):

"flat"
: All datasets are placed as GitLab projects in the same group. The project name of the top-level dataset follows the configured datalad.gitlab-SITENAME-project configuration. The project names of contained subdatasets extend the configured name with the subdatasets' s relative path within the root dataset, with all path separator characters replaced by '-'. This path separator is configurable (see Configuration).

"collection"
: A new group is created for the dataset hierarchy, following the datalad.gitlab-SITENAME-project configuration. The root dataset is placed in a "project" project inside this group, and all nested subdatasets are represented inside the group using a "flat" layout. The root datasets project name is configurable (see Configuration). This command cannot create root-level groups! To use this layout for a collection located in the root of an account, create the target group via the GitLab web UI first.

GitLab cannot host dataset content. However, in combination with other data sources (and siblings), publishing a dataset to GitLab can facilitate distribution and exchange, while still allowing any dataset consumer to obtain actual data content from alternative sources.

Configuration

Many configuration switches and options for GitLab sibling creation can be provided arguments to the command. However, it is also possible to specify a particular setup in a dataset's configuration. This is particularly important when managing large collections of datasets. Configuration options are:

"datalad.gitlab-default-site"
Name of the default GitLab site (see --site)

"datalad.gitlab-SITENAME-siblingname"
Name of the sibling configured for the local dataset that points to the GitLab instance SITENAME (see --name)

"datalad.gitlab-SITENAME-layout"
Project layout used at the GitLab instance SITENAME (see --layout)

"datalad.gitlab-SITENAME-access"
Access method used for the GitLab instance SITENAME (see --access)

"datalad.gitlab-SITENAME-project"
Project "location/path" used for a datasets at GitLab instance SITENAME (see --project). Configuring this is useful for deriving project paths for subdatasets, relative to superdataset. The root-level group ("location") needs to be created beforehand via GitLab's web interface.

"datalad.gitlab-default-projectname"
The collection layout publishes (sub)datasets as projects with a custom name. The default name "project" can be overridden with this configuration.

"datalad.gitlab-default-pathseparator"
The flat and collection layout represent subdatasets with project names that correspond to the path, with the regular path separator replaced with a "-": superdataset-subdataset. This configuration can override this default separator.

This command can be configured with "datalad.create-sibling-ghlike.extra-remote-settings.NETLOC.KEY=VALUE" in order to add any local KEY = VALUE configuration to the created sibling in the local *.git/config* file. NETLOC is the domain of the Gitlab instance to apply the configuration for. This leads to a behavior that is equivalent to calling datalad's `siblings('configure', ...)``||``siblings configure` command with the respective KEY-VALUE pair after creating the sibling. The configuration, like any other, could be set at user- or system level, so users do not need to add this configuration to every sibling created with the service at NETLOC themselves.

Parameters

- **`path`** -- selectively create siblings for any datasets underneath a given path. By default only the root dataset is considered. [Default: None]
- **`site`** (*`None or str, optional`*) -- name of the GitLab site to create a sibling at. Must match an existing python-gitlab configuration section with location and authentication settings (see https://python- gitlab.readthedocs.io/en/stable/cli-usage.html#configuration). By default the dataset configuration is consulted. [Default: None]
- **`project`** (*`None or str, optional`*) -- project name/location at the GitLab site. If a subdataset of the reference dataset is processed, its project path is automatically determined by the *layout* configuration, by default. Users need to create the root-level GitLab group (NAME) via the webinterface before running the command. [Default: None]
- **`layout`** (*`{None, 'collection', 'flat'}, optional`*) -- layout of projects at the GitLab site, if a collection, or a hierarchy of datasets and subdatasets is to be created. By default the dataset configuration is consulted. [Default: None]

- **dataset** (*Dataset or None, optional*) -- reference or root dataset. If no path constraints are given, a sibling for this dataset will be created. In this and all other cases, the reference dataset is also consulted for the GitLab configuration, and desired project layout. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **name** (*str or None, optional*) -- name to represent the GitLab sibling remote in the local dataset installation. If not specified a name is looked up in the dataset configuration, or defaults to the *site* name. [Default: None]
- **existing** (*{'skip', 'error', 'reconfigure'}, optional*) -- desired behavior when already existing or configured siblings are discovered. 'skip': ignore; 'error': fail, if access URLs differ; 'reconfigure': use the existing repository and reconfigure the local dataset to use it as a sibling. [Default: 'error']
- **access** (*{None, 'http', 'ssh', 'ssh+http'}, optional*) -- access method used for data transfer to and from the sibling. 'ssh': read and write access used the SSH protocol; 'http': read and write access use HTTP requests; 'ssh+http': read access is done via HTTP and write access performed with SSH. Dataset configuration is consulted for a default, 'http' is used otherwise. [Default: None]
- **publish_depends** (*list of str or None, optional*) -- add a dependency such that the given existing sibling is always published prior to the new sibling. This equals setting a configuration item 'remote.SIBLINGNAME.datalad-publish-depends'. Multiple dependencies can be given as a list of sibling names. [Default: None]
- **description** (*str or None, optional*) -- brief description for the GitLab project (displayed on the site). [Default: None]
- **dryrun** (*bool, optional*) -- Deprecated. Use the renamed `dry_run` parameter. [Default: False]
- **dry_run** (*bool, optional*) -- if set, no repository will be created, only tests for name collisions will be performed, and would-be repository names are reported for all relevant datasets. [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']

create_sibling_gogs(*, *api=None*, *dataset=None*, *recursive=False*, *recursion_limit=None*, *name=None*, *existing='error'*, *credential=None*, *access_protocol='https'*, *publish_depends=None*, *private=False*, *description=None*, *dry_run=False*)

Create a dataset sibling on a GOGS site

GOGS is a self-hosted, free and open source code hosting solution with low resource demands that enable running it on inexpensive devices like a Raspberry Pi, or even directly on a NAS device.

In order to be able to use this command, a personal access token has to be generated on the platform (Account->Your Settings->Applications->Generate New Token).

This command can be configured with "datalad.create-sibling-ghlike.extra-remote-settings.NETLOC.KEY=VALUE" in order to add any local KEY = VALUE configuration to the created sibling in the local *.git/config* file. NETLOC is the domain of the Gogs instance to apply the configuration for. This leads to a behavior that is equivalent to calling datalad's `siblings('configure', ...)``||``siblings configure` command with the respective KEY-VALUE pair after creating the sibling. The configuration, like any other, could be set at user- or system level, so users do not need to add this configuration to every sibling created with the service at NETLOC themselves.

Added in version 0.16.

Parameters

- **reponame** (*str*) -- repository name, optionally including an '<organization>/' prefix if the repository shall not reside under a user's namespace. When operating recursively, a suffix will be appended to this name for each subdataset.
- **api** (*str or None, optional*) -- URL of the GOGS instance without a 'api/<version>' suffix. [Default: None]
- **dataset** (Dataset *or None, optional*) -- dataset to create the publication target for. If not given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]

- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **name** (*str or None, optional*) -- name of the sibling in the local dataset installation (remote name). [Default: None]
- **existing** (*{'skip', 'error', 'reconfigure', 'replace'}, optional*) -- behavior when already existing or configured siblings are discovered: skip the dataset ('skip'), update the configuration ('reconfigure'), or fail ('error'). DEPRECATED DANGER ZONE: With 'replace', an existing repository will be irreversibly removed, re-initialized, and the sibling (re-)configured (thus implies 'reconfigure'). *replace* could lead to data loss! In interactive sessions a confirmation prompt is shown, an exception is raised in non-interactive sessions. The 'replace' mode will be removed in a future release. [Default: 'error']
- **credential** (*str or None, optional*) -- name of the credential providing a personal access token to be used for authorization. The token can be supplied via configuration setting 'datalad.credential.<name>.secret', or environment variable DATALAD_CREDENTIAL_<NAME>_SECRET, or will be queried from the active credential store using the provided name. If none is provided, the last-used token for the API URL realm will be used. If no matching credential exists, a credential named after the hostname part of the API URL is tried as a last fallback. [Default: None]
- **access_protocol** (*{'https', 'ssh', 'https-ssh'}, optional*) -- access protocol/URL to configure for the sibling. With 'https-ssh' SSH will be used for write access, whereas HTTPS is used for read access. [Default: 'https']
- **publish_depends** (*list of str or None, optional*) -- add a dependency such that the given existing sibling is always published prior to the new sibling. This equals setting a configuration item 'remote.SIBLINGNAME.datalad-publish-depends'. Multiple dependencies can be given as a list of sibling names. [Default: None]
- **private** (*bool, optional*) -- if set, create a private repository. [Default: False]
- **description** (*str or None, optional*) -- Brief description, displayed on the project's page. [Default: None]
- **dry_run** (*bool, optional*) -- if set, no repository will be created, only tests for sibling name collisions will be performed, and would-be repository names are reported for all relevant datasets. [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']

create_sibling_ria(*name*, *, *dataset=None*, *storage_name=None*, *alias=None*, *post_update_hook=False*, *shared=None*, *group=None*, *storage_sibling=True*, *existing='error'*, *new_store_ok=False*, *trust_level=None*, *recursive=False*, *recursion_limit=None*, *disable_storage__=None*, *push_url=None*)

Creates a sibling to a dataset in a RIA store

Communication with a dataset in a RIA store is implemented via two siblings. A regular Git remote (repository sibling) and a git-annex special remote for data transfer (storage sibling) -- with the former having a publication dependency on the latter. By default, the name of the storage sibling is derived from the repository sibling's name by appending "-storage".

The store's base path is expected to not exist, be an empty directory, or a valid RIA store.

Notes

RIA URL format

Interactions with new or existing RIA stores require RIA URLs to identify the store or specific datasets inside of it.

The general structure of a RIA URL pointing to a store takes the form `ria+[scheme]://<storelocation>` (e.g., `ria+ssh://[user@]hostname:/absolute/path/to/ria-store`, or `ria+file:///absolute/path/to/ria-store`)

The general structure of a RIA URL pointing to a dataset in a store (for example for cloning) takes a similar form, but appends either the datasets UUID or a "~" symbol followed by the dataset's alias name: `ria+[scheme]://<storelocation>#<dataset-UUID>` or `ria+[scheme]://<storelocation>#~<aliasname>`. In addition, specific version identifiers can be appended to the URL with an additional "@" symbol: `ria+[scheme]://<storelocation>#<dataset-UUID>@<dataset-version>`, where `dataset-version` refers to a branch or tag.

RIA store layout

A RIA store is a directory tree with a dedicated subdirectory for each dataset in the store. The subdirectory name is constructed from the DataLad dataset ID, e.g. `124/68afe-59ec-11ea-93d7-f0d5bf7b5561`,

where the first three characters of the ID are used for an intermediate subdirectory in order to mitigate files system limitations for stores containing a large number of datasets.

By default, a dataset in a RIA store consists of two components: A Git repository (for all dataset contents stored in Git) and a storage sibling (for dataset content stored in git-annex).

It is possible to selectively disable either component using `storage-sibling 'off'` or `storage-sibling 'only'`, respectively. If neither component is disabled, a dataset's subdirectory layout in a RIA store contains a standard bare Git repository and an `annex/` subdirectory inside of it. The latter holds a Git-annex object store and comprises the storage sibling. Disabling the standard git-remote (`storage-sibling='only'`) will result in not having the bare git repository, disabling the storage sibling (`storage-sibling='off'`) will result in not having the `annex/` subdirectory.

Optionally, there can be a further subdirectory `archives` with (compressed) 7z archives of annex objects. The storage remote is able to pull annex objects from these archives, if it cannot find in the regular annex object store. This feature can be useful for storing large collections of rarely changing data on systems that limit the number of files that can be stored.

Each dataset directory also contains a `ria-layout-version` file that identifies the data organization (as, for example, described above).

Lastly, there is a global `ria-layout-version` file at the store's base path that identifies where dataset subdirectories themselves are located. At present, this file must contain a single line stating the version (currently "1"). This line MUST end with a newline character.

It is possible to define an alias for an individual dataset in a store by placing a symlink to the dataset location into an `alias/` directory in the root of the store. This enables dataset access via URLs of format: `ria+<protocol>://<storelocation>#~<aliasname>`.

Compared to standard git-annex object stores, the `annex/` subdirectories used as storage siblings follow a different layout naming scheme ('dirhashmixed' instead of 'dirhashlower'). This is mostly noted as a technical detail, but also serves to remind git-annex powerusers to refrain from running git-annex commands directly in-store as it can cause severe damage due to the layout difference. Interactions should be handled via the ORA special remote instead.

Error logging

To enable error logging at the remote end, append a pipe symbol and an "l" to the version number in ria-layout-version (like so: `1|l\n`).

Error logging will create files in an "error_log" directory whenever the git-annex special remote (storage sibling) raises an exception, storing the Python traceback of it. The logfiles are named according to the scheme `<dataset id>.<annex uuid of the remote>.log` showing "who" ran into this issue with which dataset. Because logging can potentially leak personal data (like local file paths for example), it can be disabled client-side by setting the configuration variable `annex.ora-remote.<storage-sibling-name>.ignore-remote-config`.

Parameters

- **url** (*str or None*) -- URL identifying the target RIA store and access protocol. If `push_url` is given in addition, this is used for read access only. Otherwise it will be used for write access too and to create the repository sibling in the RIA store. Note, that HTTP(S) currently is valid for consumption only thus requiring to provide `push_url`.
- **name** (*str or None*) -- Name of the sibling. With *recursive*, the same name will be used to label all the subdatasets' siblings.
- **dataset** (Dataset *or None, optional*) -- specify the dataset to process. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]

- **storage_name** (*str or None, optional*) -- Name of the storage sibling (git-annex special remote). Must not be identical to the sibling name. If not specified, defaults to the sibling name plus '-storage' suffix. If only a storage sibling is created, this setting is ignored, and the primary sibling name is used. [Default: None]
- **alias** (*str or None, optional*) -- Alias for the dataset in the RIA store. Add the necessary symlink so that this dataset can be cloned from the RIA store using the given ALIAS instead of its ID. With *recursive=True*, only the top dataset will be aliased. [Default: None]
- **post_update_hook** (*bool, optional*) -- Enable Git's default post-update-hook for the created sibling. This is useful when the sibling is made accessible via a "dumb server" that requires running 'git update-server-info' to let Git interact properly with it. [Default: False]
- **shared** (*str or bool or None, optional*) -- If given, configures the permissions in the RIA store for multi- users access. Possible values for this option are identical to those of *git init --shared* and are described in its documentation. [Default: None]
- **group** (*str or None, optional*) -- Filesystem group for the repository. Specifying the group is crucial when shared="group". [Default: None]
- **storage_sibling** (*{'only'} or bool or None, optional*) -- By default, an ORA storage sibling and a Git repository sibling are created (True|'on'). Alternatively, creation of the storage sibling can be disabled (False|'off'), or a storage sibling created only and no Git sibling ('only'). In the latter mode, no Git installation is required on the target host. [Default: True]
- **existing** (*{'skip', 'error', 'reconfigure'}, optional*) -- Action to perform, if a (storage) sibling is already configured under the given name and/or a target already exists. In this case, a dataset can be skipped ('skip'), an existing target repository be forcefully re-initialized, and the sibling (re-)configured ('reconfigure'), or the command be instructed to fail ('error'). [Default: 'error']
- **new_store_ok** (*bool, optional*) -- When set, a new store will be created, if necessary. Otherwise, a sibling will only be created if the url points to an existing RIA store. [Default: False]
- **trust_level** (*{'trust', 'semitrust', 'untrust', None}, optional*) -- specify a trust level for the storage sibling. If not specified, the default git-annex trust level is used. 'trust' should be used with care (see the git-annex-trust man page). [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **disable_storage** (*bool, optional*) -- This option is deprecated. Use '--storage-sibling off' instead. [Default: None]
- **push_url** (*str or None, optional*) -- URL identifying the target RIA store and access protocol for write access to the storage sibling. If given this will also be used for creation of the repository sibling in the RIA store. [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']

create_sibling_webdav(*, *dataset=None*, *name=None*, *storage_name=None*, *mode='annex'*, *credential=None*, *existing='error'*, *recursive=False*, *recursion_limit=None*)

Create a sibling(-tandem) on a WebDAV server

WebDAV is a standard HTTP protocol extension for placing files on a server that is supported by a number of commercial storage services (e.g. 4shared.com, box.com), but also instances of cloud-storage solutions like Nextcloud or ownCloud. These software packages are also the basis for some institutional or public cloud storage solutions, such as EUDAT B2DROP.

For basic usage, only the URL with the desired dataset location on a WebDAV server needs to be specified for creating a sibling. However, the sibling setup can be flexibly customized (no storage sibling, or only a storage sibling, multi-version storage, or human-browsable single-version storage).

This command does not check for conflicting content on the WebDAV server!

When creating siblings recursively for a dataset hierarchy, subdataset exports are placed at their corresponding relative paths underneath the root location on the WebDAV server.

Collaboration on WebDAV siblings

The primary use case for WebDAV siblings is dataset deposition, where only one site is uploading dataset and file content updates. For collaborative workflows with multiple contributors, please make sure to consult the documentation on the underlying `datalad-annex::` Git remote helper for advice on appropriate setups: http://docs.datalad.org/projects/next/

Git-annex implementation details

Storage siblings are presently configured to NOT be enabled automatically on cloning a dataset. Due to a limitation of git-annex, this would initially fail (missing credentials). Instead, an explicit `datalad siblings enable --name <storage-sibling-name>` command must be executed after cloning. If necessary, it will prompt for credentials.

This command does not (and likely will not) support embedding credentials in the repository (see `embedcreds` option of the git-annex `webdav` special remote; https://git-annex.branchable.com/special_remotes/webdav), because such credential copies would need to be updated, whenever they change or expire. Instead, credentials are retrieved from DataLad's credential system. In many cases, credentials are determined automatically, based on the HTTP authentication realm identified by a WebDAV server.

This command does not support setting up encrypted remotes (yet). Neither for the storage sibling, nor for the regular Git-remote. However, adding support for it is primarily a matter of extending the API of this command, and passing the respective options on to the underlying git-annex setup.

This command does not support setting up chunking for webdav storage siblings (https://git-annex.branchable.com/chunking).

Examples

Create a WebDAV sibling tandem for storage of a dataset's file content and revision history. A user will be prompted for any required credentials, if they are not yet known.:

```
> create_sibling_webdav(url='https://webdav.example.com/myds')
```

Such a dataset can be cloned by DataLad via a specially crafted URL. Again, credentials are automatically determined, or a user is prompted to enter them:

```
> clone('datalad-annex::?type=webdav&encryption=none&url=https://webdav.example.
↪com/myds')
```

A sibling can also be created with a human-readable file tree, suitable for data exchange with non-DataLad users, but only able to host a single version of each file:

```
> create_sibling_webdav(url='https://example.com/browseable', mode='filetree')
```

Cloning such dataset siblings is possible via a convenience URL:

```
> clone('webdavs://example.com/browseable')
```

In all cases, the storage sibling needs to explicitly enabled prior to file content retrieval:

```
> siblings('enable', name='example.com-storage')
```

Parameters

- **`url`** -- URL identifying the sibling root on the target WebDAV server.
- **`dataset`** -- specify the dataset to process. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **`name`** -- name of the sibling. If none is given, the hostname-part of the WebDAV URL will be used. With *recursive*, the same name will be used to label all the subdatasets' siblings. [Default: None]
- **`storage_name`** -- name of the storage sibling (git-annex special remote). Must not be identical to the sibling name. If not specified, defaults to the sibling name plus '-storage'

suffix. If only a storage sibling is created, this setting is ignored, and the primary sibling name is used. [Default: None]

- **mode** -- Siblings can be created in various modes: full-featured sibling tandem, one for a dataset's Git history and one storage sibling to host any number of file versions ('annex'). A single sibling for the Git history only ('git-only'). A single annex sibling for multi-version file storage only ('annex-only'). As an alternative to the standard (annex) storage sibling setup that is capable of storing any number of historical file versions using a content hash layout ('annex'|'annex-only'), the 'filetree' mode can used. This mode offers a human-readable data organization on the WebDAV remote that matches the file tree of a dataset (branch). However, it can, consequently, only store a single version of each file in the file tree. This mode is useful for depositing a single dataset snapshot for consumption without DataLad. The 'filetree' mode nevertheless allows for cloning such a single-version dataset, because the full dataset history can still be pushed to the WebDAV server. Git history hosting can also be turned off for this setup ('filetree- only'). When both a storage sibling and a regular sibling are created together, a publication dependency on the storage sibling is configured for the regular sibling in the local dataset clone. [Default: 'annex']
- **credential** -- name of the credential providing a user/password credential to be used for authorization. The credential can be supplied via configuration setting 'datalad.credential.<name>.user|secret', or environment variable DATALAD_CREDENTIAL_<NAME>_USER|SECRET, or will be queried from the active credential store using the provided name. If none is provided, the last-used credential for the authentication realm associated with the WebDAV URL will be used. Only if a credential name was given, it will be encoded in the URL of the created WebDAV Git remote, credential auto-discovery will be performed on each remote access. [Default: None]
- **existing** -- action to perform, if a (storage) sibling is already configured under the given name. In this case, sibling creation can be skipped ('skip') or the sibling (re-)configured ('reconfigure') in the dataset, or the command be instructed to fail ('error'). [Default: 'error']
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [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']

credentials(*spec=None*, *, *name=None*, *prompt=None*, *dataset=None*)

Credential management and query

This command enables inspection and manipulation of credentials used throughout DataLad.

The command provides four basic actions:

QUERY

When executed without any property specification, all known credentials with all their properties will be yielded. Please note that this may not include credentials that only comprise of a secret and no other properties, or legacy credentials for which no trace in the configuration can be found. Therefore, the query results are not guaranteed to contain all credentials ever configured by DataLad.

When additional property/value pairs are specified, only credentials that have matching values for all given properties will be reported. This can be used, for example, to discover all suitable credentials for a specific "realm", if credentials were annotated with such information.

SET

This is the companion to 'get', and can be used to store properties and secret of a credential. Importantly, and in contrast to a 'get' operation, given properties with no values indicate a removal request. Any matching properties on record will be removed. If a credential is to be stored for which no secret is on record yet, an interactive session will prompt a user for a manual secret entry.

Only changed properties will be contained in the result record.

The appearance of the interactive secret entry can be configured with the two settings *datalad.credentials.repeat-secret-entry* and *datalad.credentials.hidden-secret-entry*.

REMOVE

This action will remove any secret and properties associated with a credential identified by its name.

GET (plumbing operation)

This is a *read-only* action that will never store (updates of) credential properties or secrets. Given properties will amend/overwrite those already on record. When properties with no value are given, and also no value for the respective properties is on record yet, their value will be requested interactively, if a `prompt` text was provided too. This can be used to ensure a complete credential record, comprising any number of properties.

Details on credentials

A credential comprises any number of properties, plus exactly one secret. There are no constraints on the format or property values or the secret, as long as they are encoded as a string.

Credential properties are normally stored as configuration settings in a user's configuration ('global' scope) using the naming scheme:

datalad.credential.<name>.<property>

Therefore both credential name and credential property name must be syntax-compliant with Git configuration items. For property names this means only alphanumeric characters and dashes. For credential names virtually no naming restrictions exist (only null-byte and newline are forbidden). However, when naming credentials it is recommended to use simple names in order to enable convenient one-off credential overrides by specifying DataLad configuration items via their environment variable counterparts (see the documentation of the `configuration` command for details. In short, avoid underscores and special characters other than '.' and '-'.

While there are no constraints on the number and nature of credential properties, a few particular properties are recognized on used for particular purposes:

- 'secret': always refers to the single secret of a credential
- 'type': identifies the type of a credential. With each standard type, a list of mandatory properties is associated (see below)
- 'last-used': is an ISO 8601 format time stamp that indicated the last (successful) usage of a credential

Standard credential types and properties

The following standard credential types are recognized, and their mandatory field with their standard names will be automatically included in a 'get' report.

- 'user_password': with properties 'user', and the password as secret
- 'token': only comprising the token as secret
- 'aws-s3': with properties 'key-id', 'session', 'expiration', and the secret_id as the credential secret

Legacy support

DataLad credentials not configured via this command may not be fully discoverable (i.e., including all their properties). Discovery of such legacy credentials can be assisted by specifying a dedicated 'type' property.

Examples

Report all discoverable credentials:

```
> credentials()
```

Set a new credential mycred & input its secret interactively:

```
> credentials('set', name='mycred')
```

Remove a credential's type property:

```
> credentials('set', name='mycred', spec={'type': None})
```

Get all information on a specific credential in a structured record:

```
> credentials('get', name='mycred')
```

Upgrade a legacy credential by annotating it with a 'type' property:

```
> credentials('set', name='legacycred', spec={'type': 'user_password')
```

Set a new credential of type user_password, with a given user property, and input its secret interactively:

```
> credentials('set', name='mycred', spec={'type': 'user_password', 'user': '
↪<username>'})
```

Obtain a (possibly yet undefined) credential with a minimum set of properties. All missing properties and secret will be prompted for, no information will be stored! This is mostly useful for ensuring availability of an appropriate credential in an application context:

```
> credentials('get', prompt='Can I haz info plz?', name='newcred', spec={
↪'newproperty': None})
```

Parameters

- **action** -- which action to perform. [Default: 'query']
- **spec** -- specification of credential properties. Properties are given as name/value pairs. Properties with a *None* value indicate a property to be deleted (action 'set'), or a property to be entered interactively, when no value is set yet, and a prompt text is given (action 'get'). All property names are case-insensitive, must start with a letter or a digit, and may only contain '-' apart from these characters. Property specifications should be given a as dictionary, e.g., spec={'type': 'user_password'}. However, a CLI-like list of string arguments is also supported, e.g., spec=['type=user_password']. [Default: None]
- **name** -- name of a credential to set, get, or remove. [Default: None]
- **prompt** -- message to display when entry of missing credential properties is required for action 'get'. This can be used to present information on the nature of a credential and for instructions on how to obtain a credential. [Default: None]
- **dataset** -- specify a dataset whose configuration to inspect rather than the global (user) settings. [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']

diff(*, *fr='HEAD'*, *to=None*, *dataset=None*, *annex=None*, *untracked='normal'*, *recursive=False*, *recursion_limit=None*)

Report differences between two states of a dataset (hierarchy)

The two to-be-compared states are given via the --from and --to options. These state identifiers are evaluated in the context of the (specified or detected) dataset. In the case of a recursive report on a dataset hierarchy, corresponding state pairs for any subdataset are determined from the subdataset record in the respective superdataset. Only changes recorded in a subdataset between these two states are reported, and so on.

Any paths given as additional arguments will be used to constrain the difference report. As with Git's diff, it will not result in an error when a path is specified that does not exist on the filesystem.

Reports are very similar to those of the *status* command, with the distinguished content types and states being identical.

Examples

Show unsaved changes in a dataset:

```
> diff()
```

Compare a previous dataset state identified by shasum against current worktree:

```
> diff(fr='SHASUM')
```

Compare two branches against each other:

```
> diff(fr='branch1', to='branch2')
```

Show unsaved changes in the dataset and potential subdatasets:

```
> diff(recursive=True)
```

Show unsaved changes made to a particular file:

```
> diff(path='path/to/file')
```

Parameters

- **path** (*sequence of str or None, optional*) -- path to constrain the report to. [Default: None]
- **fr** (*str, optional*) -- original state to compare to, as given by any identifier that Git understands. [Default: 'HEAD']
- **to** (*str or None, optional*) -- state to compare against the original state, as given by any identifier that Git understands. If none is specified, the state of the working tree will be compared. [Default: None]
- **dataset** (*Dataset or None, optional*) -- specify the dataset to query. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **annex** (*{None, 'basic', 'availability', 'all'}, optional*) -- Switch whether to include information on the annex content of individual files in the status report, such as recorded file size. By default no annex information is reported (faster). Three report modes are available: basic information like file size and key name ('basic'); additionally test whether file content is present in the local annex ('availability'; requires one or two additional file system stat calls, but does not call git-annex), this will add the result properties 'has_content' (boolean flag) and 'objloc' (absolute path to an existing annex object file); or 'all' which will report all available information (presently identical to 'availability'). [Default: None]
- **untracked** (*{'no', 'normal', 'all'}, optional*) -- If and how untracked content is reported when comparing a revision to the state of the working tree. 'no': no untracked content is reported; 'normal': untracked files and entire untracked directories are reported as such; 'all': report individual files even in fully untracked directories. [Default: 'normal']
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [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']

download(*, *dataset=None*, *force=None*, *credential=None*, *hash=None*)

Download from URLs

This command is the front-end to an extensible framework for performing downloads from a variety of URL schemes. Built-in support for the schemes 'http', 'https', 'file', and 'ssh' is provided. Extension packages may add additional support.

In contrast to other downloader tools, this command integrates with the DataLad credential management and is able to auto-discover credentials. If no credential is available, it automatically prompts for them, and offers to store them for reuse after a successful authentication.

Simultaneous hashing (checksumming) of downloaded content is supported with user-specified algorithms.

The command can process any number of downloads (serially). it can read download specifications from (command line) arguments, files, or STDIN. It can deposit downloads to individual files, or stream to STDOUT.

Implementation and extensibility

Each URL scheme is processed by a dedicated handler. Additional schemes can be supported by sub-classing datalad_next.url_operations.UrlOperations and implementing the *download()* method. Extension packages can register new handlers, by patching them into the *datalad_next.download._urlscheme_handlers* registry dict.

Examples

Download webpage to "myfile.txt":

```
> download({"http://example.com": "myfile.txt"})
```

Read download specification from STDIN (e.g. JSON-lines):

```
> download("-")
```

Simultaneously hash download, hexdigest reported in result record:

```
> download("http://example.com/data.xml", hash=["sha256"])
```

Download from SSH server:

```
> download("ssh://example.com/home/user/data.xml")
```

Parameters

- **spec** -- Download sources and targets can be given in a variety of formats: as a URL, or as a URL-path-pair that is mapping a source URL to a dedicated download target path. Any number of URLs or URL-path-pairs can be provided, either as an argument list, or read from a file (one item per line). Such a specification input file can be given as a path to an existing file (as a single value, not as part of a URL- path-pair). When the special path identifier '-' is used, the download is written to STDOUT. A specification can also be read in JSON-lines encoding (each line being a string with a URL or an object mapping a URL-string to a path-string). In addition, specifications can also be given as a list or URLs, or as a list of dicts with a URL to path mapping. Paths are supported in string form, or as *Path* objects.
- **dataset** -- Dataset to be used as a configuration source. Beyond reading configuration items, this command does not interact with the dataset. [Default: None]
- **force** -- By default, a target path for a download must not exist yet. 'force- overwrite' disabled this check. [Default: None]
- **credential** -- name of a credential to be used for authorization. If no credential is identified, the last-used credential for the authentication realm associated with the download target will be used. If there is no credential available yet, it will be prompted for. Once used successfully, a prompt for entering to save such a new credential will be presented. [Default: None]
- **hash** -- Name of a hashing algorithm supported by the Python 'hashlib' module, e.g. 'md5' or 'sha256'. [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']

download_url(*, *dataset=None*, *path=None*, *overwrite=False*, *archive=False*, *save=True*, *message=None*)

Download content

It allows for a uniform download interface to various supported URL schemes (see command help for details), re-using or asking for authentication details maintained by datalad.

Examples

Download files from an http and S3 URL:

```
> download_url(urls=['http://example.com/file.dat', 's3://bucket/file2.dat'])
```

Download a file to a path and provide a commit message:

```
> download_url(urls='s3://bucket/file2.dat', message='added a file', path=
↪'myfile.dat')
```

Append a trailing slash to the target path to download into a specified directory:

```
> download_url(['http://example.com/file.dat'], path='data/')
```

Leave off the trailing slash to download into a regular file:

```
> download_url(['http://example.com/file.dat'], path='data')
```

Parameters

- **urls** (*non-empty sequence of str*) -- URL(s) to be downloaded. Supported protocols: 'ftp', 'http', 'https', 's3', 'shub'.
- **dataset** (Dataset *or None, optional*) -- specify the dataset to add files to. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. Use save=False to prevent adding files to the dataset. [Default: None]
- **path** (*str or None, optional*) -- target for download. If the path has a trailing separator, it is treated as a directory, and each specified URL is downloaded under that directory to a base name taken from the URL. Without a trailing separator, the value specifies the name of the downloaded file (file name extensions inferred from the URL may be added to

it, if they are not yet present) and only a single URL should be given. In both cases, leading directories will be created if needed. This argument defaults to the current directory. [Default: None]

- **overwrite** (*bool, optional*) -- flag to overwrite it if target file exists. [Default: False]
- **archive** (*bool, optional*) -- pass the downloaded files to add_archive_content(..., delete=True). [Default: False]
- **save** (*bool, optional*) -- by default all modifications to a dataset are immediately saved. Giving this option will disable this behavior. [Default: True]
- **message** (*str or None, optional*) -- a description of the state or the changes made to a 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']

drop(*, *what='filecontent'*, *reckless=None*, *dataset=None*, *recursive=False*, *recursion_limit=None*, *jobs=None*, *check=None*, *if_dirty=None*)

Drop content of individual files or entire (sub)datasets

This command is the antagonist of 'get'. It can undo the retrieval of file content, and the installation of subdatasets.

Dropping is a safe-by-default operation. Before dropping any information, the command confirms the continued availability of file-content (see e.g., configuration 'annex.numcopies'), and the state of all dataset branches from at least one known dataset sibling. Moreover, prior removal of an entire dataset annex, that it is confirmed that it is no longer marked as existing in the network of dataset siblings.

Importantly, all checks regarding version history availability and local annex availability are performed using the current state of remote siblings as known to the local dataset. This is done for performance reasons and for resilience in case of absent network connectivity. To ensure decision making based on up-to-date information, it is advised to execute a dataset update before dropping dataset components.

Examples

Drop single file content:

```
> drop('path/to/file')
```

Drop all file content in the current dataset:

```
> drop('.')
```

Drop all file content in a dataset and all its subdatasets:

```
> drop(dataset='.', recursive=True)
```

Disable check to ensure the configured minimum number of remote sources for dropped data:

```
> drop(path='path/to/content', reckless='availability')
```

Drop (uninstall) an entire dataset (will fail with subdatasets present):

```
> drop(what='all')
```

Kill a dataset recklessly with any existing subdatasets too(this will be fast, but will disable any and all safety checks):

```
> drop(what='all', reckless='kill', recursive=True)
```

Parameters

- **path** (*sequence of str or None, optional*) -- path of a dataset or dataset component to be dropped. [Default: None]
- **what** (*{'filecontent', 'allkeys', 'datasets', 'all'}, optional*) -- select what type of items shall be dropped. With 'filecontent', only the file content (git-annex keys) of files in a dataset's worktree will be dropped. With 'allkeys', content of any version of any file in any branch (including, but not limited to the worktree) will be dropped. This effectively empties the annex of a local dataset. With 'datasets', only complete datasets will be dropped (implies 'allkeys' mode for each such dataset), but no filecontent will be dropped for any files in datasets that are not dropped entirely. With 'all', content for any matching file or dataset will be dropped entirely. [Default: 'filecontent']
- **reckless** (*{'modification', 'availability', 'undead', 'kill', None}, optional*) -- disable individual or all data safety measures that would normally

prevent potentially irreversible data-loss. With 'modification', unsaved modifications in a dataset will not be detected. This improves performance at the cost of permitting potential loss of unsaved or untracked dataset components. With 'availability', detection of dataset/branch-states that are only available in the local dataset, and detection of an insufficient number of file- content copies will be disabled. Especially the latter is a potentially expensive check which might involve numerous network transactions. With 'undead', detection of whether a to-be-removed local annex is still known to exist in the network of dataset-clones is disabled. This could cause zombie-records of invalid file availability. With 'kill', all safety-checks are disabled. [Default: None]

- **dataset** (*Dataset or None, optional*) -- specify the dataset to perform drop from. If no dataset is given, the current working directory is used as operation context. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item NOTE: This option can only parallelize input retrieval (get) and output recording (save). DataLad does NOT parallelize your scripts for you. [Default: None]
- **check** (*bool, optional*) -- DEPRECATED: use '--reckless availability'. [Default: None]
- **if_dirty** -- DEPRECATED and IGNORED: use --reckless instead. [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']

export_archive(*, *dataset=None*, *archivetype='tar'*, *compression='gz'*, *missing_content='error'*)

Export the content of a dataset as a TAR/ZIP archive.

Parameters

- **filename** (*str or None, optional*) -- File name of the generated TAR archive. If no file name is given the archive will be generated in the current directory and will be named: datalad_<dataset_uuid>.(tar.*|zip). To generate that file in a different directory, provide an existing directory as the file name. [Default: None]
- **dataset** (Dataset *or None, optional*) -- "specify the dataset to export. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **archivetype** (*{'tar', 'zip'}, optional*) -- Type of archive to generate. [Default: 'tar']
- **compression** (*{'gz', 'bz2', ''}, optional*) -- Compression method to use. 'bz2' is not supported for ZIP archives. No compression is used when an empty string is given. [Default: 'gz']
- **missing_content** (*{'error', 'continue', 'ignore'}, optional*) -- By default, any discovered file with missing content will result in an error and the export is aborted. Setting this to 'continue' will issue warnings instead of failing on error. The value 'ignore' will only inform about problem at the 'debug' log level. The latter two can be helpful when generating a TAR archive from a dataset where some file content is not available locally. [Default: 'error']
- **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']

export_archive_ora(*opts=None*, *, *dataset=None*, *remote=None*, *annex_wanted=None*, *froms=None*, *missing_content='error'*)

Export an archive of a local annex object store for the ORA remote.

Keys in the local annex object store are reorganized in a temporary directory (using links to avoid storage duplication) to use the 'hashdirlower' setup used by git-annex for bare repositories and the directory-type special remote. This alternative object store is then moved into a 7zip archive that is suitable for use in a ORA remote dataset store. Placing such an archive into:

```
<dataset location>/archives/archive.7z
```

Enables the ORA special remote to locate and retrieve all keys contained in the archive.

Parameters

- **target** (*str or None*) -- if an existing directory, an 'archive.7z' is placed into it, otherwise this is the path to the target archive.
- **opts** -- list of options for 7z to replace the default '-mx0' to generate an uncompressed archive. [Default: None]
- **dataset** (Dataset *or None, optional*) -- specify the dataset to process. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **remote** (*str or None, optional*) -- name of the target sibling, wanted/preferred settings will be used to filter the files added to the archives. [Default: None]
- **annex_wanted** -- git-annex-preferred-content expression for git-annex find to filter files. Should start with 'or' or 'and' when used in combination with *--for*. [Default: None]
- **froms** -- one or multiple tree-ish from which to select files. [Default: None]
- **missing_content** (*{'error', 'continue', 'ignore'}, optional*) -- By default, any discovered file with missing content will result in an error and the export is aborted. Setting this to 'continue' will issue warnings instead of failing on error. The value 'ignore' will only inform about problem at the 'debug' log level. The latter two can be helpful when generating a TAR archive from a dataset where some file content is not available locally. [Default: 'error']

- **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']

export_to_figshare(*, *dataset=None*, *missing_content='error'*, *no_annex=False*, *article_id=None*)

Export the content of a dataset as a ZIP archive to figshare

Very quick and dirty approach. Ideally figshare should be supported as a proper git annex special remote. Unfortunately, figshare does not support having directories, and can store only a flat list of files. That makes it impossible for any sensible publishing of complete datasets.

The only workaround is to publish dataset as a zip-ball, where the entire content is wrapped into a .zip archive for which figshare would provide a navigator.

Parameters

- **filename** (*str or None, optional*) -- File name of the generated ZIP archive. If no file name is given the archive will be generated in the top directory of the dataset and will be named: datalad_<dataset_uuid>.zip. [Default: None]
- **dataset** (Dataset *or None, optional*) -- "specify the dataset to export. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]

- **missing_content** (*{'error', 'continue', 'ignore'}, optional*) -- By default, any discovered file with missing content will result in an error and the plugin is aborted. Setting this to 'continue' will issue warnings instead of failing on error. The value 'ignore' will only inform about problem at the 'debug' log level. The latter two can be helpful when generating a TAR archive from a dataset where some file content is not available locally. [Default: 'error']
- **no_annex** (*bool, optional*) -- By default the generated .zip file would be added to annex, and all files would get registered in git-annex to be available from such a tarball. Also upon upload we will register for that archive to be a possible source for it in annex. Setting this flag disables this behavior. [Default: False]
- **article_id** (*int or None, optional*) -- Which article to publish to. [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']

foreach_dataset(*, *cmd_type='auto'*, *dataset=None*, *state='present'*, *recursive=False*, *recursion_limit=None*, *contains=None*, *bottomup=False*, *subdatasets_only=False*, *output_streams='pass-through'*, *chpwd='ds'*, *safe_to_consume='auto'*, *jobs=None*)

Run a command or Python code on the dataset and/or each of its sub-datasets.

This command provides a convenience for the cases were no dedicated DataLad command is provided to operate across the hierarchy of datasets. It is very similar to *git submodule foreach* command with the following major differences

- by default (unless *subdatasets_only=True*) it would include operation on the original dataset as well,
- subdatasets could be traversed in bottom-up order,
- can execute commands in parallel (see *jobs* option), but would account for the order, e.g. in bottom-up order command is executed in super-dataset only after it is executed in all subdatasets.

Additional notes:

- for execution of "external" commands we use the environment used to execute external git and git-annex commands.

Command format

cmd_type='external': A few placeholders are supported in the command via Python format specification:

- "{pwd}" will be replaced with the full path of the current working directory.
- "{ds}" and "{refds}" will provide instances of the dataset currently operated on and the reference "context" dataset which was provided via `dataset` argument.
- "{tmpdir}" will be replaced with the full path of a temporary directory.

Examples

Aggressively git clean all datasets, running 5 parallel jobs:

```
> foreach_dataset(['git', 'clean', '-dfx'], recursive=True, jobs=5)
```

Parameters

- **cmd** -- command for execution. For *cmd_type='exec'* or *cmd_type='eval'* (Python code) should be either a string or a list with only a single item. If 'eval', the actual function can be passed, which will be provided all placeholders as keyword arguments.
- **cmd_type** (*{'auto', 'external', 'exec', 'eval'}, optional*) -- type of the command. *external*: to be run in a child process using dataset's runner; 'exec': Python source code to execute using 'exec(), no value returned; 'eval': Python source code to evaluate using 'eval()', return value is placed into 'result' field. 'auto': If used via Python API, and *cmd* is a Python function, it will use 'eval', and otherwise would assume 'external'. [Default: 'auto']
- **dataset** (Dataset *or None, optional*) -- specify the dataset to operate on. If no dataset is given, an attempt is made to identify the dataset based on the input and/or the current working directory. [Default: None]
- **state** (*{'present', 'absent', 'any'}, optional*) -- indicate which (sub)datasets to consider: either only locally present, absent, or any of those two kinds. [Default: 'present']
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **contains** (*list of str or None, optional*) -- limit to the subdatasets containing the given path. If a root path of a subdataset is given, the last considered dataset will be the

subdataset itself. Can be a list with multiple paths, in which case datasets that contain any of the given paths will be considered. [Default: None]

- **bottomup** (*bool, optional*) -- whether to report subdatasets in bottom-up order along each branch in the dataset tree, and not top-down. [Default: False]
- **subdatasets_only** (*bool, optional*) -- whether to exclude top level dataset. It is implied if a non-empty *contains* is used. [Default: False]
- **output_streams** (*{'capture', 'pass-through', 'relpath'}, optional*) -- ways to handle outputs. 'capture' and return outputs from 'cmd' in the record ('stdout', 'stderr'); 'pass-through' to the screen (and thus absent from returned record); prefix with 'relpath' captured output (similar to like grep does) and write to stdout and stderr. In 'relpath', relative path is relative to the top of the dataset if *dataset* is specified, and if not - relative to current directory. [Default: 'pass-through']
- **chpwd** (*{'ds', 'pwd'}, optional*) -- 'ds' will change working directory to the top of the corresponding dataset. With 'pwd' no change of working directory will happen. Note that for Python commands, due to use of threads, we do not allow chdir=ds to be used with jobs > 1. Hint: use 'ds' and 'refds' objects' methods to execute commands in the context of those datasets. [Default: 'ds']
- **safe_to_consume** (*{'auto', 'all-subds-done', 'superds-done', 'always'}, optional*) -- Important only in the case of parallel (jobs greater than 1) execution. 'all-subds-done' instructs to not consider superdataset until command finished execution in all subdatasets (it is the value in case of 'auto' if traversal is bottomup). 'superds-done' instructs to not process subdatasets until command finished in the super-dataset (it is the value in case of 'auto' in traversal is not bottom up, which is the default). With 'always' there is no constraint on either to execute in sub or super dataset. [Default: 'auto']
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item NOTE: This option can only parallelize input retrieval (get) and output recording (save). DataLad does NOT parallelize your scripts for you. [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']

get(*, *source=None*, *dataset=None*, *recursive=False*, *recursion_limit=None*, *get_data=True*, *description=None*, *reckless=None*, *jobs='auto'*)

Get any dataset content (files/directories/subdatasets).

This command only operates on dataset content. To obtain a new independent dataset from some source use the *clone* command.

By default this command operates recursively within a dataset, but not across potential subdatasets, i.e. if a directory is provided, all files in the directory are obtained. Recursion into subdatasets is supported too. If enabled, relevant subdatasets are detected and installed in order to fulfill a request.

Known data locations for each requested file are evaluated and data are obtained from some available location (according to git-annex configuration and possibly assigned remote priorities), unless a specific source is specified.

Getting subdatasets

Just as DataLad supports getting file content from more than one location, the same is supported for subdatasets, including a ranking of individual sources for prioritization.

The following location candidates are considered. For each candidate a cost is given in parenthesis, higher values indicate higher cost, and thus lower priority:

- A datalad URL recorded in *.gitmodules* (cost 590). This allows for datalad URLs that require additional handling/resolution by datalad, like ria-schemes (ria+http, ria+ssh, etc.)
- A URL or absolute path recorded for git in *.gitmodules* (cost 600).
- URL of any configured superdataset remote that is known to have the desired submodule commit, with the submodule path appended to it. There can be more than one candidate (cost 650).
- In case *.gitmodules* contains a relative path instead of a URL, the URL of any configured superdataset remote that is known to have the desired submodule commit, with this relative path appended to it. There can be more than one candidate (cost 650).
- In case *.gitmodules* contains a relative path as a URL, the absolute path of the superdataset, appended with this relative path (cost 900).

Additional candidate URLs can be generated based on templates specified as configuration variables with the pattern

datalad.get.subdataset-source-candidate-<name>

where *name* is an arbitrary identifier. If *name* starts with three digits (e.g. '400myserver') these will be interpreted as a cost, and the respective candidate will be sorted into the generated candidate list according to this cost. If no cost is given, a default of 700 is used.

A template string assigned to such a variable can utilize the Python format mini language and may reference a number of properties that are inferred from the parent dataset's knowledge about the target subdataset. Properties include any submodule property specified in the respective *.gitmodules* record. For convenience, an existing *datalad-id* record is made available under the shortened name *id*.

Additionally, the URL of any configured remote that contains the respective submodule commit is available as *remoteurl-<name>* property, where *name* is the configured remote name.

Hence, such a template could be *http://example.org/datasets/{id}* or *http://example.org/datasets/{path}*, where *{id}* and *{path}* would be replaced by the *datalad-id* or *path* entry in the *.gitmodules* record.

If this config is committed in *.datalad/config*, a clone of a dataset can look up any subdataset's URL according to such scheme(s) irrespective of what URL is recorded in *.gitmodules*.

Lastly, all candidates are sorted according to their cost (lower values first), and duplicate URLs are stripped, while preserving the first item in the candidate list.

Note: Power-user info: This command uses git annex get to fulfill file handles.

Examples

Get a single file:

```
> get('path/to/file')
```

Get contents of a directory:

```
> get('path/to/dir/')
```

Get all contents of the current dataset and its subdatasets:

```
> get(dataset='.', recursive=True)
```

Get (clone) a registered subdataset, but don't retrieve data:

```
> get('path/to/subds', get_data=False)
```

Parameters

- **path** (*sequence of str or None, optional*) -- path/name of the requested dataset component. The component must already be known to a dataset. To add new components to a dataset use the *add* command. [Default: None]
- **source** (*str or None, optional*) -- label of the data source to be used to fulfill requests. This can be the name of a dataset sibling or another known source. [Default: None]
- **dataset** (Dataset *or None, optional*) -- specify the dataset to perform the add operation on, in which case *path* arguments are interpreted as being relative to this dataset. If no dataset is given, an attempt is made to identify a dataset for each input *path*. [Default: None]

- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or {'existing'} or None, optional*) -- limit recursion into subdataset to the given number of levels. Alternatively, 'existing' will limit recursion to subdatasets that already existed on the filesystem at the start of processing, and prevent new subdatasets from being obtained recursively. [Default: None]
- **get_data** (*bool, optional*) -- whether to obtain data for all file handles. If disabled, *get* operations are limited to dataset handles. [Default: True]
- **description** (*str or None, optional*) -- short description to use for a dataset location. Its primary purpose is to help humans to identify a dataset copy (e.g., "mike's dataset on lab server"). Note that when a dataset is published, this information becomes available on the remote side. [Default: None]
- **reckless** (*{None, True, False, 'auto', 'ephemeral'} or shared-..., optional*) -- Obtain a dataset or subdatset and set it up in a potentially unsafe way for performance, or access reasons. Use with care, any dataset is marked as 'untrusted'. The reckless mode is stored in a dataset's local configuration under 'datalad.clone.reckless', and will be inherited to any of its subdatasets. Supported modes are: ['auto']: hard-link files between local clones. In-place modification in any clone will alter original annex content. ['ephemeral']: symlink annex to origin's annex and discard local availability info via git- annex-dead 'here' and declares this annex private. Shares an annex between origin and clone w/o git-annex being aware of it. In case of a change in origin you need to update the clone before you're able to save new content on your end. Alternative to 'auto' when hardlinks are not an option, or number of consumed inodes needs to be minimized. Note that this mode can only be used with clones from non-bare repositories or a RIA store! Otherwise two different annex object tree structures (dirhashmixed vs dirhashlower) will be used simultaneously, and annex keys using the respective other structure will be inaccessible. ['shared-<mode>']: set up repository and annex permission to enable multi-user access. This disables the standard write protection of annex'ed files. <mode> can be any value support by 'git init --shared=', such as 'group', or 'all'. [Default: None]
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item. [Default: 'auto']
- **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']

get_superdataset(*datalad_only=False*, *topmost=False*, *registered_only=True*)

Get the dataset's superdataset

Parameters

- **datalad_only** (*bool, optional*) -- Whether to consider only "datalad datasets" (with non-None id), or (if False, which is default) - any git repository
- **topmost** (*bool, optional*) -- Return the topmost super-dataset. Might then be the current one.
- **registered_only** (*bool, optional*) -- Test whether any discovered superdataset actually contains the dataset in question as a registered subdataset (as opposed to just being located in a subdirectory without a formal relationship).

Return type
Dataset or None

property id

Identifier of the dataset.

This identifier is supposed to be unique across datasets, but identical for different versions of the same dataset (that have all been derived from the same original dataset repository).

Note, that a plain git/git-annex repository doesn't necessarily have a dataset id yet. It is created by *Dataset.create()* and stored in .datalad/config. If None is returned while there is a valid repository, there may have never been a call to *create* in this branch before current commit.

Note, that this property is evaluated every time it is used. If used multiple times within a function it's probably a good idea to store its value in a local variable and use this variable instead.

Returns
This is either a stored UUID, or *None*.

Return type
str

install(*, *source=None*, *dataset=None*, *get_data=False*, *description=None*, *recursive=False*, *recursion_limit=None*, *reckless=None*, *jobs='auto'*, *branch=None*)

Install one or many datasets from remote URL(s) or local PATH source(s).

This command creates local sibling(s) of existing dataset(s) from (remote) locations specified as URL(s) or path(s). Optional recursion into potential subdatasets, and download of all referenced data is supported. The new dataset(s) can be optionally registered in an existing superdataset by identifying it via the *dataset* argument (the new dataset's path needs to be located within the superdataset for that).

If no explicit *source* option is specified, then all positional URL- OR-PATH arguments are considered to be "sources" if they are URLs or target locations if they are paths. If a target location path corresponds to a submodule, the source location for it is figured out from its record in the *.gitmodules*. If *source* is specified, then a single optional positional PATH would be taken as the destination path for that dataset.

It is possible to provide a brief description to label the dataset's nature *and* location, e.g. "Michael's music on black laptop". This helps humans to identify data locations in distributed scenarios. By default an identifier comprised of user and machine name, plus path will be generated.

When only partial dataset content shall be obtained, it is recommended to use this command without the *get-data* flag, followed by a ~datalad.api.get operation to obtain the desired data.

Note: Power-user info: This command uses git clone, and git annex init to prepare the dataset. Registering to a superdataset is performed via a git submodule add operation in the discovered superdataset.

Examples

Install a dataset from GitHub into the current directory:

```
> install(source='https://github.com/datalad-datasets/longnow-podcasts.git')
```

Install a dataset as a subdataset into the current dataset:

```
> install(dataset='.',
          source='https://github.com/datalad-datasets/longnow-podcasts.git')
```

Install a dataset into 'podcasts' (not 'longnow-podcasts') directory, and get all content right away:

```
> install(path='podcasts',
          source='https://github.com/datalad-datasets/longnow-podcasts.git',
          get_data=True)
```

Install a dataset with all its subdatasets:

```
> install(source='https://github.com/datalad-datasets/longnow-podcasts.git',
          recursive=True)
```

Parameters

- **path** -- path/name of the installation target. If no *path* is provided a destination path will be derived from a source URL similar to git clone. [Default: None]
- **source** (*str or None, optional*) -- URL or local path of the installation source. [Default: None]
- **dataset** (Dataset *or None, optional*) -- specify the dataset to perform the install operation on. If no dataset is given, an attempt is made to identify the dataset in a parent directory of the current working directory and/or the *path* given. [Default: None]
- **get_data** (*bool, optional*) -- if given, obtain all data content too. [Default: False]

- **description** (*str or None, optional*) -- short description to use for a dataset location. Its primary purpose is to help humans to identify a dataset copy (e.g., "mike's dataset on lab server"). Note that when a dataset is published, this information becomes available on the remote side. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **reckless** (*{None, True, False, 'auto', 'ephemeral'} or shared-..., optional*) -- Obtain a dataset or subdatset and set it up in a potentially unsafe way for performance, or access reasons. Use with care, any dataset is marked as 'untrusted'. The reckless mode is stored in a dataset's local configuration under 'datalad.clone.reckless', and will be inherited to any of its subdatasets. Supported modes are: ['auto']: hard-link files between local clones. In-place modification in any clone will alter original annex content. ['ephemeral']: symlink annex to origin's annex and discard local availability info via git- annex-dead 'here' and declares this annex private. Shares an annex between origin and clone w/o git-annex being aware of it. In case of a change in origin you need to update the clone before you're able to save new content on your end. Alternative to 'auto' when hardlinks are not an option, or number of consumed inodes needs to be minimized. Note that this mode can only be used with clones from non-bare repositories or a RIA store! Otherwise two different annex object tree structures (dirhashmixed vs dirhashlower) will be used simultaneously, and annex keys using the respective other structure will be inaccessible. ['shared-<mode>']: set up repository and annex permission to enable multi-user access. This disables the standard write protection of annex'ed files. <mode> can be any value support by 'git init --shared=', such as 'group', or 'all'. [Default: None]
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item. [Default: 'auto']
- **branch** (*str or None, optional*) -- Clone source at this branch or tag. This option applies only to the top-level dataset not any subdatasets that may be cloned when installing recursively. Note that if the source is a RIA URL with a version, it takes precedence over this option. [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: <function is_result_matching_pathsource_argument at 0x7f8d8f62a940>]
- **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: 'successdatasets-or- 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: 'item-or-list']

is_installed()

Returns whether a dataset is installed.

A dataset is installed when a repository for it exists on the filesystem.

Return type
bool

next_status(*, *untracked='normal'*, *recursive='repository'*, *eval_subdataset_state='full'*) → Generator[*StatusResult*, None, None] | list[*StatusResult*]

Report on the (modification) status of a dataset

Note: This is a preview of an command implementation aiming to replace the DataLad `status` command.

For now, expect anything here to change again.

This command provides a report that is roughly identical to that of `git status`. Running with default parameters yields a report that should look familiar to Git and DataLad users alike, and contain the same information as offered by `git status`.

The main difference to `git status` are:

- Support for recursion into submodule. `git status` does that too, but the report is limited to the global state of an entire submodule, whereas this command can issue detailed reports in changes inside a submodule (any nesting depth).
- Support for directory-constrained reporting. Much like `git status` limits its report to a single repository, this command can optionally limit its report to a single directory and its direct children. In this report subdirectories are considered containers (much like) submodules, and a change summary is provided for them.
- Support for a "mono" (monolithic repository) report. Unlike a standard recursion into submodules, and checking each of them for changes with respect to the HEAD commit of the worktree, this report compares a submodule with respect to the state recorded in its parent repository. This provides an equally comprehensive status report from the point of view of a queried repository, but does not include a dedicated item on the global state of a submodule. This makes nested hierarchy of repositories appear like a single (mono) repository.

- Support for "adjusted mode" git-annex repositories. These utilize a managed branch that is repeatedly rewritten, hence is not suitable for tracking within a parent repository. Instead, the underlying "corresponding branch" is used, which contains the equivalent content in an un-adjusted form, persistently. This command detects this condition and automatically check a repositories state against the corresponding branch state.

Presently missing/planned features

- There is no support for specifying paths (or pathspecs) for constraining the operation to specific dataset parts. This will be added in the future.
- There is no reporting of git-annex properties, such as tracked file size. It is undetermined whether this will be added in the future. However, even without a dedicated switch, this command has support for datasets (and their submodules) in git-annex's "adjusted mode".

Differences to the ``status`` command implementation prior DataLad v2

- Like `git status` this implementation reports on dataset modification, whereas the previous `status` also provided a listing of unchanged dataset content. This is no longer done. Equivalent functionality for listing dataset content is provided by the `ls_file_collection` command.
- The implementation is substantially faster. Depending on the context the speed-up is typically somewhere between 2x and 100x.
- The implementation does not suffer from the limitation re type change detection.
- Python and CLI API of the command use uniform parameter validation.

Parameters

- **dataset** -- Dataset to be used as a configuration source. Beyond reading configuration items, this command does not interact with the dataset. [Default: None]
- **untracked** -- Determine how untracked content is considered and reported when comparing a revision to the state of the working tree. 'no': no untracked content is considered as a change; 'normal': untracked files and entire untracked directories are reported as such; 'all': report individual files even in fully untracked directories. In addition to these git-status modes, 'whole-dir' (like normal, but include empty directories), and 'no-empty-dir' (alias for 'normal') are understood. [Default: 'normal']
- **recursive** -- Mode of recursion for status reporting. With 'no' the report is restricted to a single directory and its direct children. With 'repository', the report comprises all repository content underneath current working directory or root of a given dataset, but is limited to items directly contained in that repository. With 'datasets', the report also comprises any content in any subdatasets. Each subdataset is evaluated against its respective HEAD commit. With 'mono', a report similar to 'datasets' is generated, but any subdataset is evaluate with respect to the state recorded in its parent repository. In contrast to the 'datasets' mode, no report items on a joint submodule are generated. [Default: 'repository']
- **eval_subdataset_state** -- Evaluation of subdataset state (modified or untracked content) can be expensive for deep dataset hierarchies as subdataset have to be tested recursively for uncommitted modifications. Setting this option to 'no' or 'commit' can substantially boost performance by limiting what is being tested. With 'no' no state is evaluated and subdataset are not investigated for modifications. With 'commit' only a discrepancy of the HEAD commit gitsha of a subdataset and the gitsha recorded in the superdataset's record is evaluated. With 'full' any other modifications are considered too. [Default: 'full']
- **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']

no_annex(*pattern*, *ref_dir='.'*, *makedirs=False*)

Configure a dataset to never put some content into the dataset's annex

This can be useful in mixed datasets that also contain textual data, such as source code, which can be efficiently and more conveniently managed directly in Git.

Patterns generally look like this:

```
code/*
```

which would match all file in the code directory. In order to match all files under `code/`, including all its subdirectories use such a pattern:

```
code/**
```

Note that this command works incrementally, hence any existing configuration (e.g. from a previous plugin run) is amended, not replaced.

Parameters

- **dataset** (*Dataset or None*) -- "specify the dataset to configure. If no dataset is given, an attempt is made to identify the dataset based on the current working directory.
- **pattern** -- list of path patterns. Any content whose path is matching any pattern will not be annexed when added to a dataset, but instead will be tracked directly in Git. Path pattern have to be relative to the directory given by the *ref_dir* option. By default, patterns should be relative to the root of the dataset.
- **ref_dir** -- Relative path (within the dataset) to the directory that is to be configured. All patterns are interpreted relative to this path, and configuration is written to a `.gitattributes` file in this directory. [Default: '.']
- **makedirs** (*bool, optional*) -- If set, any missing directories will be created in order to be able to place a file into `--ref-dir`. [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']

property path

path to the dataset

property pathobj

pathobj for the dataset

push(*, *dataset=None*, *to=None*, *since=None*, *data='auto-if-wanted'*, *force=None*, *recursive=False*, *recursion_limit=None*, *jobs=None*)

Push a dataset to a known sibling.

This makes a saved state of a dataset available to a sibling or special remote data store of a dataset. Any target sibling must already exist and be known to the dataset.

By default, all files tracked in the last saved state (of the current branch) will be copied to the target location. Optionally, it is possible to limit a push to changes relative to a particular point in the version history of a dataset (e.g. a release tag) using the since option in conjunction with the specification of a reference dataset. In recursive mode subdatasets will also be evaluated, and only those subdatasets are pushed where a change was recorded that is reflected in the current state of the top-level reference dataset.

Note: Power-user info: This command uses git push, and git annex copy to push a dataset. Publication targets are either configured remote Git repositories, or git-annex special remotes (if they support data upload).

The following feature is added by the datalad-next extension:

If a target is a git-annex special remote that has "exporttree" set to "yes", push will call 'git-annex export' to export the current HEAD to the remote target. This will usually result in a copy of the file tree, to which HEAD refers, on the remote target. A git-annex special remote with "exporttree" set to "yes" can, for example, be created with the datalad command "create-sibling-webdav" with the option "--mode=filetree" or "--mode=filetree-only".

Parameters

- **path** (*sequence of str or None, optional*) -- path to constrain a push to. If given, only data or changes for those paths are considered for a push. [Default: None]
- **dataset** (Dataset *or None, optional*) -- specify the dataset to push. [Default: None]
- **to** (*str or None, optional*) -- name of the target sibling. If no name is given an attempt is made to identify the target based on the dataset's configuration (i.e. a configured tracking branch, or a single sibling that is configured for push). [Default: None]
- **since** (*str or None, optional*) -- specifies commit-ish (tag, shasum, etc.) from which to look for changes to decide whether pushing is necessary. If '^' is given, the last state of the current branch at the sibling is taken as a starting point. [Default: None]
- **data** (*{'anything', 'nothing', 'auto', 'auto-if-wanted'}, optional*) -- what to do with (annex'ed) data. 'anything' would cause transfer of all annexed content, 'nothing' would avoid call to *git annex copy* altogether. 'auto' would use 'git annex copy' with '--auto' thus transferring only data which would satisfy "wanted" or "numcopies" settings for the remote (thus "nothing" otherwise). 'auto-if-wanted' would enable '--auto' mode only if there is a "wanted" setting for the remote, and transfer 'anything' otherwise. [Default: 'auto-if- wanted']
- **force** (*{'all', 'gitpush', 'checkdatapresent', 'export', None}, optional*) -- force particular operations, possibly overruling safety protections or optimizations: use --force with git-push ('gitpush'); do not use --fast with git-annex copy ('checkdatapresent'); force an annex export (to git annex remotes with "exporttree" set to "yes"); combine all force modes ('all'). [Default: None]

- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item. [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']

recall_state(*whereto*)

Something that can be used to checkout a particular state (tag, commit) to "undo" a change or switch to a otherwise desired previous state.

Parameters

whereto (*str*)

remove(*, *dataset=None*, *drop='datasets'*, *reckless=None*, *message=None*, *jobs=None*, *recursive=None*, *check=None*, *save=None*, *if_dirty=None*)

Remove components from datasets

Removing "unlinks" a dataset component, such as a file or subdataset, from a dataset. Such a removal advances the state of a dataset, just like adding new content. A remove operation can be undone, by restoring a previous dataset state, but might require re-obtaining file content and subdatasets from remote locations.

This command relies on the 'drop' command for safe operation. By default, only file content from datasets which will be uninstalled as part of a removal will be dropped. Otherwise file content is retained, such that restoring a previous version also immediately restores file content access, just as it is the case for files directly committed to Git. This default behavior can be changed to always drop content prior removal, for cases where a minimal storage footprint for local datasets installations is desirable.

Removing a dataset component is always a recursive operation. Removing a directory, removes all content underneath the directory too. If subdatasets are located under a to-be-removed path, they will be uninstalled entirely, and all their content dropped. If any subdataset can not be uninstalled safely, the remove operation will fail and halt.

Changed in version 0.16: More in-depth and comprehensive safety-checks are now performed by default. The `if_dirty` argument is ignored, will be removed in a future release, and can be removed for a safe-by-default behavior. For other cases consider the `reckless` argument. The `save` argument is ignored and will be removed in a future release, a dataset modification is now always saved. Consider save's `amend` argument for post-remove fix-ups. The `recursive` argument is ignored, and will be removed in a future release. Removal operations are always recursive, and the parameter can be stripped from calls for a safe-by-default behavior.

Deprecated since version 0.16: The `check` argument will be removed in a future release. It needs to be replaced with `reckless`.

Examples

Permanently remove a subdataset (and all further subdatasets contained in it) from a dataset:

```
> remove(dataset='path/to/dataset', path='path/to/subds')
```

Permanently remove a superdataset (with all subdatasets) from the filesystem:

```
> remove(dataset='path/to/dataset')
```

DANGER-ZONE: Fast wipe-out a dataset and all its subdataset, bypassing all safety checks:

```
> remove(dataset='path/to/dataset', reckless='kill')
```

Parameters

- **path** (*sequence of str or None, optional*) -- path of a dataset or dataset component to be removed. [Default: None]
- **dataset** (*Dataset or None, optional*) -- specify the dataset to perform remove from. If no dataset is given, the current working directory is used as operation context. [Default: None]
- **drop** (*{'datasets', 'all'}, optional*) -- which dataset components to drop prior removal. This parameter is passed on to the underlying drop operation as its 'what' argument. [Default: 'datasets']
- **reckless** (*{'modification', 'availability', 'undead', 'kill', None}, optional*) -- disable individual or all data safety measures that would normally

prevent potentially irreversible data-loss. With 'modification', unsaved modifications in a dataset will not be detected. This improves performance at the cost of permitting potential loss of unsaved or untracked dataset components. With 'availability', detection of dataset/branch-states that are only available in the local dataset, and detection of an insufficient number of file- content copies will be disabled. Especially the latter is a potentially expensive check which might involve numerous network transactions. With 'undead', detection of whether a to-be-removed local annex is still known to exist in the network of dataset-clones is disabled. This could cause zombie-records of invalid file availability. With 'kill', all safety-checks are disabled. [Default: None]

- **message** (*str or None, optional*) -- a description of the state or the changes made to a dataset. [Default: None]
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item NOTE: This option can only parallelize input retrieval (get) and output recording (save). DataLad does NOT parallelize your scripts for you. [Default: None]
- **recursive** -- DEPRECATED and IGNORED: removal is always a recursive operation. [Default: None]
- **check** (*bool, optional*) -- DEPRECATED: use '--reckless availability'. [Default: None]
- **save** (*bool, optional*) -- DEPRECATED and IGNORED; use *save --amend* instead. [Default: None]
- **if_dirty** -- DEPRECATED and IGNORED: use --reckless instead. [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']

property repo

Get an instance of the version control system/repo for this dataset, or None if there is none yet (or none anymore).

If testing the validity of an instance of GitRepo is guaranteed to be really cheap this could also serve as a test whether a repo is present.

Note, that this property is evaluated every time it is used. If used multiple times within a function it's probably a good idea to store its value in a local variable and use this variable instead.

Return type
GitRepo or AnnexRepo

rerun(*, *since=None*, *dataset=None*, *branch=None*, *message=None*, *onto=None*, *script=None*, *report=False*, *assume_ready=None*, *explicit=False*, *jobs=None*)

Re-execute previous *datalad run* commands.

This will unlock any dataset content that is on record to have been modified by the command in the specified revision. It will then re-execute the command in the recorded path (if it was inside the dataset). Afterwards, all modifications will be saved.

Report mode

When called with report=True, this command reports information about what would be re-executed as a series of records. There will be a record for each revision in the specified revision range. Each of these will have one of the following "rerun_action" values:

- run: the revision has a recorded command that would be re-executed
- skip-or-pick: the revision does not have a recorded command and would be either skipped or cherry picked
- merge: the revision is a merge commit and a corresponding merge would be made

The decision to skip rather than cherry pick a revision is based on whether the revision would be reachable from HEAD at the time of execution.

In addition, when a starting point other than HEAD is specified, there is a rerun_action value "checkout", in which case the record includes information about the revision the would be checked out before rerunning any commands.

Note: Currently the "onto" feature only sets the working tree of the current dataset to a previous state. The working trees of any subdatasets remain unchanged.

Examples

Re-execute the command from the previous commit:

```
> rerun()
```

Re-execute any commands in the last five commits:

```
> rerun(since='HEAD~5')
```

Do the same as above, but re-execute the commands on top of HEAD~5 in a detached state:

```
> rerun(onto='', since='HEAD~5')
```

Parameters

- **revision** (*str or None, optional*) -- rerun command(s) in *revision*. By default, the command from this commit will be executed, but *since* can be used to construct a revision range. The default value is like "HEAD" but resolves to the main branch when on an adjusted branch. [Default: None]
- **since** (*str or None, optional*) -- If *since* is a commit-ish, the commands from all commits that are reachable from *revision* but not *since* will be re-executed (in other words, the commands in git log SINCE..REVISION). If SINCE is an empty string, it is set to the parent of the first commit that contains a recorded command (i.e., all commands in git log REVISION will be re-executed). [Default: None]
- **dataset** (Dataset *or None, optional*) -- specify the dataset from which to rerun a recorded command. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. If a dataset is given, the command will be executed in the root directory of this dataset. [Default: None]
- **branch** (*str or None, optional*) -- create and checkout this branch before rerunning the commands. [Default: None]
- **message** (*str or None, optional*) -- use MESSAGE for the reran commit rather than the recorded commit message. In the case of a multi-commit rerun, all the reran commits will have this message. [Default: None]
- **onto** (*str or None, optional*) -- start point for rerunning the commands. If not specified, commands are executed at HEAD. This option can be used to specify an alternative start point, which will be checked out with the branch name specified by *branch* or in a detached state otherwise. As a special case, an empty value for this option means the parent of the first run commit in the specified revision list. [Default: None]
- **script** (*str or None, optional*) -- extract the commands into this file rather than rerunning. Use - to write to stdout instead. [Default: None]
- **report** (*bool, optional*) -- Don't actually re-execute anything, just display what would be done. [Default: False]
- **assume_ready** (*{None, 'inputs', 'outputs', 'both'}, optional*) -- Assume that inputs do not need to be retrieved and/or outputs do not need to unlocked or removed before running the command. This option allows you to avoid the expense of these preparation steps if you know that they are unnecessary. Note that this option also affects any additional outputs that are automatically inferred based on inspecting changed files in the run commit. [Default: None]

- **explicit** (*bool, optional*) -- Consider the specification of inputs and outputs in the run record to be explicit. Don't warn if the repository is dirty, and only save modifications to the outputs from the original record. Note that when several run commits are specified, this applies to every one. Care should also be taken when using *onto* because checking out a new HEAD can easily fail when the working tree has modifications. [Default: False]
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item NOTE: This option can only parallelize input retrieval (get) and output recording (save). DataLad does NOT parallelize your scripts for you. [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']

run(*, *dataset=None*, *inputs=None*, *outputs=None*, *expand=None*, *assume_ready=None*, *explicit=False*, *message=None*, *sidecar=None*, *dry_run=None*, *jobs=None*)

Run an arbitrary shell command and record its impact on a dataset.

It is recommended to craft the command such that it can run in the root directory of the dataset that the command will be recorded in. However, as long as the command is executed somewhere underneath the

dataset root, the exact location will be recorded relative to the dataset root.

If the executed command did not alter the dataset in any way, no record of the command execution is made.

If the given command errors, a *CommandError* exception with the same exit code will be raised, and no modifications will be saved. A command execution will not be attempted, by default, when an error occurred during input or output preparation. This default `stop` behavior can be overridden via *on_failure=...*.

In the presence of subdatasets, the full dataset hierarchy will be checked for unsaved changes prior command execution, and changes in any dataset will be saved after execution. Any modification of subdatasets is also saved in their respective superdatasets to capture a comprehensive record of the entire dataset hierarchy state. The associated provenance record is duplicated in each modified (sub)dataset, although only being fully interpretable and re-executable in the actual top-level superdataset. For this reason the provenance record contains the dataset ID of that superdataset.

Command format

A few placeholders are supported in the command via Python format specification. "{pwd}" will be replaced with the full path of the current working directory. "{dspath}" will be replaced with the full path of the dataset that run is invoked on. "{tmpdir}" will be replaced with the full path of a temporary directory. "{inputs}" and "{outputs}" represent the values specified by *inputs* and *outputs*. If multiple values are specified, the values will be joined by a space. The order of the values will match that order from the command line, with any globs expanded in alphabetical order (like bash). Individual values can be accessed with an integer index (e.g., "{inputs[0]}").

Note that the representation of the inputs or outputs in the formatted command string depends on whether the command is given as a list of arguments or as a string. The concatenated list of inputs or outputs will be surrounded by quotes when the command is given as a list but not when it is given as a string. This means that the string form is required if you need to pass each input as a separate argument to a preceding script (i.e., write the command as "./script {inputs}", quotes included). The string form should also be used if the input or output paths contain spaces or other characters that need to be escaped.

To escape a brace character, double it (i.e., "{{" or "}}").

Custom placeholders can be added as configuration variables under "datalad.run.substitutions". As an example:

Add a placeholder "name" with the value "joe":

```
% datalad configuration --scope branch set datalad.run.substitutions.
↪name=joe
% datalad save -m "Configure name placeholder" .datalad/config
```

Access the new placeholder in a command:

```
% datalad run "echo my name is {name} >me"
```

Examples

Run an executable script and record the impact on a dataset:

```
> run(message='run my script', cmd='code/script.sh')
```

Run a command and specify a directory as a dependency for the run. The contents of the dependency will be retrieved prior to running the script:

```
> run(cmd='code/script.sh', message='run my script',
      inputs=['data/*'])
```

Run an executable script and specify output files of the script to be unlocked prior to running the script:

```
> run(cmd='code/script.sh', message='run my script',
      inputs=['data/*'], outputs=['output_dir'])
```

Specify multiple inputs and outputs:

```
> run(cmd='code/script.sh',
      message='run my script',
      inputs=['data/*', 'datafile.txt'],
      outputs=['output_dir', 'outfile.txt'])
```

Use ** to match any file at any directory depth recursively. Single * does not check files within matched directories.:

```
> run(cmd='code/script.sh',
      message='run my script',
      inputs=['data/**/*.dat'],
      outputs=['output_dir/**'])
```

Parameters

- **cmd** -- command for execution. A leading '--' can be used to disambiguate this command from the preceding options to DataLad. [Default: None]
- **dataset** (*Dataset or None, optional*) -- specify the dataset to record the command results in. An attempt is made to identify the dataset based on the current working directory. If a dataset is given, the command will be executed in the root directory of this dataset. [Default: None]
- **inputs** -- A dependency for the run. Before running the command, the content for this relative path will be retrieved. A value of "." means "run datalad get .". The value can also be a glob. [Default: None]
- **outputs** -- Prepare this relative path to be an output file of the command. A value of "." means "run datalad unlock ." (and will fail if some content isn't present). For any other value, if the content of this file is present, unlock the file. Otherwise, remove it. The value can also be a glob. [Default: None]
- **expand** (*{None, 'inputs', 'outputs', 'both'}, optional*) -- Expand globs when storing inputs and/or outputs in the commit message. [Default: None]
- **assume_ready** (*{None, 'inputs', 'outputs', 'both'}, optional*) -- Assume that inputs do not need to be retrieved and/or outputs do not need to unlocked or removed before running the command. This option allows you to avoid the expense of these preparation steps if you know that they are unnecessary. [Default: None]
- **explicit** (*bool, optional*) -- Consider the specification of inputs and outputs to be explicit. Don't warn if the repository is dirty, and only save modifications to the listed outputs. [Default: False]
- **message** (*str or None, optional*) -- a description of the state or the changes made to a dataset. [Default: None]
- **sidecar** (*None or bool, optional*) -- By default, the configuration variable 'datalad.run.record-sidecar' determines whether a record with information on a command's execution is placed into a separate record file instead of the commit message (default:

off). This option can be used to override the configured behavior on a case-by-case basis. Sidecar files are placed into the dataset's '.datalad/runinfo' directory (customizable via the 'datalad.run.record-directory' configuration variable). [Default: None]

- **dry_run** (*{None, 'basic', 'command'}, optional*) -- Do not run the command; just display details about the command execution. A value of "basic" reports a few important details about the execution, including the expanded command and expanded inputs and outputs. "command" displays the expanded command only. Note that input and output globs underneath an uninstalled dataset will be left unexpanded because no subdatasets will be installed for a dry run. [Default: None]
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item NOTE: This option can only parallelize input retrieval (get) and output recording (save). DataLad does NOT parallelize your scripts for you. [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: 'stop']
- **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']

run_procedure(*, *dataset=None*, *discover=False*, *help_proc=False*)

Run prepared procedures (DataLad scripts) on a dataset

Concept

A "procedure" is an algorithm with the purpose to process a dataset in a particular way. Procedures can be useful in a wide range of scenarios, like adjusting dataset configuration in a uniform fashion, populating a dataset with particular content, or automating other routine tasks, such as synchronizing dataset content with certain siblings.

Implementations of some procedures are shipped together with DataLad, but additional procedures can be provided by 1) any DataLad extension, 2) any (sub-)dataset, 3) a local user, or 4) a local system administrator. DataLad will look for procedures in the following locations and order:

Directories identified by the configuration settings

- 'datalad.locations.user-procedures' (determined by platformdirs.user_config_dir; defaults to '$HOME/.config/datalad/procedures' on GNU/Linux systems)
- 'datalad.locations.system-procedures' (determined by platformdirs.site_config_dir; defaults to '/etc/xdg/datalad/procedures' on GNU/Linux systems)
- 'datalad.locations.dataset-procedures'

and subsequently in the 'resources/procedures/' directories of any installed extension, and, lastly, of the DataLad installation itself.

Please note that a dataset that defines 'datalad.locations.dataset-procedures' provides its procedures to any dataset it is a subdataset of. That way you can have a collection of such procedures in a dedicated dataset and install it as a subdataset into any dataset you want to use those procedures with. In case of a naming conflict with such a dataset hierarchy, the dataset you're calling run-procedures on will take precedence over its subdatasets and so on.

Each configuration setting can occur multiple times to indicate multiple directories to be searched. If a procedure matching a given name is found (filename without a possible extension), the search is aborted and this implementation will be executed. This makes it possible for individual datasets, users, or machines to override externally provided procedures (enabling the implementation of customizable processing "hooks").

Procedure implementation

A procedure can be any executable. Executables must have the appropriate permissions and, in the case of a script, must contain an appropriate "shebang" line. If a procedure is not executable, but its filename ends with '.py', it is automatically executed by the 'python' interpreter (whichever version is available in the present environment). Likewise, procedure implementations ending on '.sh' are executed via 'bash'.

Procedures can implement any argument handling, but must be capable of taking at least one positional argument (the absolute path to the dataset they shall operate on).

For further customization there are two configuration settings per procedure available:

- 'datalad.procedures.<NAME>.call-format' fully customizable format string to determine how to execute procedure NAME (see also datalad-run). It currently requires to include the following placeholders:
  - '{script}': will be replaced by the path to the procedure
  - '{ds}': will be replaced by the absolute path to the dataset the procedure shall operate on
  - '{args}': (not actually required) will be replaced by

    all but the first element of *spec* if *spec* is a list or tuple As an example the default format string for a call to a python script is: "python {script} {ds} {args}"

- 'datalad.procedures.<NAME>.help' will be shown on *datalad run-procedure --help-proc NAME* to provide a description and/or usage info for procedure NAME

Examples

Find out which procedures are available on the current system:

```
> run_procedure(discover=True)
```

Run the 'yoda' procedure in the current dataset:

```
> run_procedure(spec='cfg_yoda', recursive=True)
```

Parameters

- **spec** -- Name and possibly additional arguments of the to-be-executed procedure. [PY: Can also be a dictionary coming from run- procedure(discover=True).]. [Default: None]
- **dataset** (*Dataset or None, optional*) -- specify the dataset to run the procedure on. An attempt is made to identify the dataset based on the current working directory. [Default: None]
- **discover** (*bool, optional*) -- if given, all configured paths are searched for procedures and one result record per discovered procedure is yielded, but no procedure is executed. [Default: False]
- **help_proc** (*bool, optional*) -- if given, get a help message for procedure NAME from config setting datalad.procedures.NAME.help. [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']

save(*, *message=None*, *dataset=None*, *version_tag=None*, *recursive=False*, *recursion_limit=None*, *updated=False*, *message_file=None*, *to_git=None*, *jobs=None*, *amend=False*)

Save the current state of a dataset

Saving the state of a dataset records changes that have been made to it. This change record is annotated with a user-provided description. Optionally, an additional tag, such as a version, can be assigned to the saved state. Such tag enables straightforward retrieval of past versions at a later point in time.

Note: Before Git v2.22, any Git repository without an initial commit located inside a Dataset is ignored, and content underneath it will be saved to the respective superdataset. DataLad datasets always have an initial commit, hence are not affected by this behavior.

Examples

Save any content underneath the current directory, without altering any potential subdataset:

```
> save(path='.')
```

Save specific content in the dataset:

```
> save(path='myfile.txt')
```

Attach a commit message to save:

```
> save(path='myfile.txt', message='add file')
```

Save any content underneath the current directory, and recurse into any potential subdatasets:

```
> save(path='.', recursive=True)
```

Save any modification of known dataset content in the current directory, but leave untracked files (e.g. temporary files) untouched:

```
> save(path='.', updated=True)
```

Tag the most recent saved state of a dataset:

```
> save(version_tag='bestyet')
```

Save a specific change but integrate into last commit keeping the already recorded commit message:

```
> save(path='myfile.txt', amend=True)
```

Parameters

- **path** (*sequence of str or None, optional*) -- path/name of the dataset component to save. If given, only changes made to those components are recorded in the new state. [Default: None]
- **message** (*str or None, optional*) -- a description of the state or the changes made to a dataset. [Default: None]
- **dataset** (Dataset *or None, optional*) -- "specify the dataset to save. [Default: None]
- **version_tag** (*str or None, optional*) -- an additional marker for that state. Every dataset that is touched will receive the tag. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **updated** (*bool, optional*) -- if given, only saves previously tracked paths. [Default: False]
- **message_file** (*str or None, optional*) -- take the commit message from this file. This flag is mutually exclusive with -m. [Default: None]
- **to_git** (*bool, optional*) -- flag whether to add data directly to Git, instead of tracking data identity only. Use with caution, there is no guarantee that a file put directly into Git like this will not be annexed in a subsequent save operation. If not specified, it will be up to git-annex to decide how a file is tracked, based on a dataset's configuration to track particular paths, file types, or file sizes with either Git or git-annex. (see https://git-annex.branchable.com/tips/largefiles). [Default: None]
- **jobs** (*int or None or {'auto'}, optional*) -- how many parallel jobs (where possible) to use. "auto" corresponds to the number defined by 'datalad.runtime.max-annex-jobs' configuration item. [Default: None]
- **amend** (*bool, optional*) -- if set, changes are not recorded in a new, separate commit, but are integrated with the changeset of the previous commit, and both together are recorded by replacing that previous commit. This is mutually exclusive with recursive operation. [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']

siblings(*, *dataset=None*, *name=None*, *url=None*, *pushurl=None*, *description=None*, *fetch=False*, *as_common_datasrc=None*, *publish_depends=None*, *publish_by_default=None*, *annex_wanted=None*, *annex_required=None*, *annex_group=None*, *annex_groupwanted=None*, *inherit=False*, *get_annex_info=True*, *recursive=False*, *recursion_limit=None*)

Manage sibling configuration

This command offers four different actions: 'query', 'add', 'remove', 'configure', 'enable'. 'query' is the default action and can be used to obtain information about (all) known siblings. 'add' and 'configure' are highly similar actions, the only difference being that adding a sibling with a name that is already registered will fail, whereas re-configuring a (different) sibling under a known name will not be considered an error. 'enable' can be used to complete access configuration for non-Git sibling (aka git-annex special remotes). Lastly, the 'remove' action allows for the removal (or de-configuration) of a registered sibling.

For each sibling (added, configured, or queried) all known sibling properties are reported. This includes:

"name"
Name of the sibling

"path"
Absolute path of the dataset

"url"
For regular siblings at minimum a "fetch" URL, possibly also a "pushurl"

Additionally, any further configuration will also be reported using a key that matches that in the Git configuration.

By default, sibling information is rendered as one line per sibling following this scheme:

```
<dataset_path>: <sibling_name>(<+|->) [<access_specification]
```

where the + and - labels indicate the presence or absence of a remote data annex at a particular remote, and *access_specification* contains either a URL and/or a type label for the sibling.

Parameters

- **action** (*{'query', 'add', 'remove', 'configure', 'enable'}, optional*) -- command action selection (see general documentation). [Default: 'query']
- **dataset** (*Dataset or None, optional*) -- specify the dataset to configure. If no dataset is given, an attempt is made to identify the dataset based on the input and/or the current working directory. [Default: None]
- **name** (*str or None, optional*) -- name of the sibling. For addition with path "URLs" and sibling removal this option is mandatory, otherwise the hostname part of a given URL is used as a default. This option can be used to limit 'query' to a specific sibling. [Default: None]
- **url** (*str or None, optional*) -- the URL of or path to the dataset sibling named by *name*. For recursive operation it is required that a template string for building subdataset sibling URLs is given. List of currently available placeholders: %%NAME the name of the dataset, where slashes are replaced by dashes. [Default: None]
- **pushurl** (*str or None, optional*) -- in case the *url* cannot be used to publish to the dataset sibling, this option specifies a URL to be used instead. If no *url* is given, *pushurl* serves as *url* as well. [Default: None]
- **description** (*str or None, optional*) -- short description to use for a dataset location. Its primary purpose is to help humans to identify a dataset copy (e.g., "mike's dataset on lab server"). Note that when a dataset is published, this information becomes available on the remote side. [Default: None]
- **fetch** (*bool, optional*) -- fetch the sibling after configuration. [Default: False]
- **as_common_datasrc** -- configure a sibling as a common data source of the dataset that can be automatically used by all consumers of the dataset. The sibling must be a regular Git remote with a configured HTTP(S) URL. [Default: None]
- **publish_depends** (*list of str or None, optional*) -- add a dependency such that the given existing sibling is always published prior to the new sibling. This equals setting a configuration item 'remote.SIBLINGNAME.datalad-publish-depends'. Multiple dependencies can be given as a list of sibling names. [Default: None]
- **publish_by_default** (*list of str or None, optional*) -- add a refspec to be published to this sibling by default if nothing specified. [Default: None]
- **annex_wanted** (*str or None, optional*) -- expression to specify 'wanted' content for the repository/sibling. See https://git-annex.branchable.com/git-annex-wanted/ for more information. [Default: None]
- **annex_required** (*str or None, optional*) -- expression to specify 'required' content for the repository/sibling. See https://git-annex.branchable.com/git-annex-required/ for more information. [Default: None]
- **annex_group** (*str or None, optional*) -- expression to specify a group for the repository. See https://git- annex.branchable.com/git-annex-group/ for more information. [Default: None]
- **annex_groupwanted** (*str or None, optional*) -- expression for the groupwanted. Makes sense only if annex_wanted="groupwanted" and annex-group is given too. See https://git-annex.branchable.com/git-annex-groupwanted/ for more information. [Default: None]

- **inherit** (*bool, optional*) -- if sibling is missing, inherit settings (git config, git annex wanted/group/groupwanted) from its super-dataset. [Default: False]
- **get_annex_info** (*bool, optional*) -- Whether to query all information about the annex configurations of siblings. Can be disabled if speed is a concern. [Default: True]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [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']

status(*, *dataset=None*, *annex=None*, *untracked='normal'*, *recursive=False*, *recursion_limit=None*, *eval_subdataset_state='full'*, *report_filetype=None*)

Report on the state of dataset content.

This is an analog to *git status* that is simultaneously crippled and more powerful. It is crippled, because it only supports a fraction of the functionality of its counter part and only distinguishes a subset of the states that Git knows about. But it is also more powerful as it can handle status reports for a whole hierarchy

of datasets, with the ability to report on a subset of the content (selection of paths) across any number of datasets in the hierarchy.

Path conventions

All reports are guaranteed to use absolute paths that are underneath the given or detected reference dataset, regardless of whether query paths are given as absolute or relative paths (with respect to the working directory, or to the reference dataset, when such a dataset is given explicitly). Moreover, so-called "explicit relative paths" (i.e. paths that start with '.' or '..') are also supported, and are interpreted as relative paths with respect to the current working directory regardless of whether a reference dataset with specified.

When it is necessary to address a subdataset record in a superdataset without causing a status query for the state _within_ the subdataset itself, this can be achieved by explicitly providing a reference dataset and the path to the root of the subdataset like so:

```
datalad status --dataset . subdspath
```

In contrast, when the state of the subdataset within the superdataset is not relevant, a status query for the content of the subdataset can be obtained by adding a trailing path separator to the query path (rsync-like syntax):

```
datalad status --dataset . subdspath/
```

When both aspects are relevant (the state of the subdataset content and the state of the subdataset within the superdataset), both queries can be combined:

```
datalad status --dataset . subdspath subdspath/
```

When performing a recursive status query, both status aspects of subdataset are always included in the report.

Content types

The following content types are distinguished:

- 'dataset' -- any top-level dataset, or any subdataset that is properly registered in superdataset
- 'directory' -- any directory that does not qualify for type 'dataset'
- 'file' -- any file, or any symlink that is placeholder to an annexed file when annex-status reporting is enabled
- 'symlink' -- any symlink that is not used as a placeholder for an annexed file

Content states

The following content states are distinguished:

- 'clean'
- 'added'
- 'modified'
- 'deleted'
- 'untracked'

Examples

Report on the state of a dataset:

```
> status()
```

Report on the state of a dataset and all subdatasets:

```
> status(recursive=True)
```

Address a subdataset record in a superdataset without causing a status query for the state _within_ the subdataset itself:

```
> status(dataset='.', path='mysubdataset')
```

Get a status query for the state within the subdataset without causing a status query for the superdataset (using trailing path separator in the query path)::

```
> status(dataset='.', path='mysubdataset/')
```

Report on the state of a subdataset in a superdataset and on the state within the subdataset:

```
> status(dataset='.', path=['mysubdataset', 'mysubdataset/'])
```

Report the file size of annexed content in a dataset:

```
> status(annex=True)
```

Parameters

- **path** (*sequence of str or None, optional*) -- path to be evaluated. [Default: None]
- **dataset** (Dataset *or None, optional*) -- specify the dataset to query. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **annex** (*{None, 'basic', 'availability', 'all'}, optional*) -- Switch whether to include information on the annex content of individual files in the status report, such as recorded file size. By default no annex information is reported (faster). Three report modes are available: basic information like file size and key name ('basic'); additionally test whether file content is present in the local annex ('availability'; requires one or two additional file system stat calls, but does not call git-annex), this will add the result properties 'has_content' (boolean flag) and 'objloc' (absolute path to an existing annex object file); or 'all' which will report all available information (presently identical to 'availability'). [Default: None]
- **untracked** (*{'no', 'normal', 'all'}, optional*) -- If and how untracked content is reported when comparing a revision to the state of the working tree. 'no': no untracked content is reported; 'normal': untracked files and entire untracked directories are reported as such; 'all': report individual files even in fully untracked directories. [Default: 'normal']
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]

- **eval_subdataset_state** (*{'no', 'commit', 'full'}, optional*) -- Evaluation of subdataset state (clean vs. modified) can be expensive for deep dataset hierarchies as subdataset have to be tested recursively for uncommitted modifications. Setting this option to 'no' or 'commit' can substantially boost performance by limiting what is being tested. With 'no' no state is evaluated and subdataset result records typically do not contain a 'state' property. With 'commit' only a discrepancy of the HEAD commit shasum of a subdataset and the shasum recorded in the superdataset's record is evaluated, and the 'state' result property only reflects this aspect. With 'full' any other modification is considered too (see the 'untracked' option for further tailoring modification testing). [Default: 'full']
- **report_filetype** (*{'raw', 'eval', None}, optional*) -- THIS OPTION IS IGNORED. It will be removed in a future release. Dataset component types are always reported as-is (previous 'raw' mode), unless annex-reporting is enabled with the *annex* option, in which case symlinks that represent annexed files will be reported as type='file'. [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']

subdatasets(*, *dataset=None*, *state='any'*, *fulfilled=None(DEPRECATED)*, *recursive=False*, *recursion_limit=None*, *contains=None*, *bottomup=False*, *set_property=None*, *delete_property=None*)

Report subdatasets and their properties.

The following properties are reported (if possible) for each matching subdataset record.

"name"
Name of the subdataset in the parent (often identical with the relative path in the parent dataset)

"path"
Absolute path to the subdataset

"parentds"
Absolute path to the parent dataset

"gitshasum"
SHA1 of the subdataset commit recorded in the parent dataset

"state"
Condition of the subdataset: 'absent', 'present'

"gitmodule_url"
URL of the subdataset recorded in the parent

"gitmodule_name"
Name of the subdataset recorded in the parent

"gitmodule_<label>"
Any additional configuration property on record.

Performance note: Property modification, requesting *bottomup* reporting order, or a particular numerical *recursion_limit* implies an internal switch to an alternative query implementation for recursive query that is more flexible, but also notably slower (performs one call to Git per dataset versus a single call for all combined).

The following properties for subdatasets are recognized by DataLad (without the 'gitmodule_' prefix that is used in the query results):

"datalad-recursiveinstall"
If set to 'skip', the respective subdataset is skipped when DataLad is recursively installing its superdataset. However, the subdataset remains installable when explicitly requested, and no other features are impaired.

"datalad-url"
If a subdataset was originally established by cloning, 'datalad-url' records the URL that was used to do so. This might be different from 'url' if the URL contains datalad specific pieces like any URL of the form "ria+<some protocol>...".

Parameters

- **path** (*sequence of str or None, optional*) -- path/name to query for subdatasets. Defaults to the current directory, or the entire dataset if called as a dataset method. [Default: None]
- **dataset** (Dataset *or None, optional*) -- specify the dataset to query. If no dataset is given, an attempt is made to identify the dataset based on the input and/or the current working directory. [Default: None]
- **state** (*{'present', 'absent', 'any'}, optional*) -- indicate which (sub)datasets to consider: either only locally present, absent, or any of those two kinds. [Default: 'any']

- **fulfilled** (*bool or None, optional*) -- DEPRECATED: use *state* instead. If given, must be a boolean flag indicating whether to consider either only locally present or absent datasets. By default all subdatasets are considered regardless of their status. [Default: None(DEPRECATED)]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]
- **contains** (*list of str or None, optional*) -- limit to the subdatasets containing the given path. If a root path of a subdataset is given, the last considered dataset will be the subdataset itself. Can be a list with multiple paths, in which case datasets that contain any of the given paths will be considered. [Default: None]
- **bottomup** (*bool, optional*) -- whether to report subdatasets in bottom-up order along each branch in the dataset tree, and not top-down. [Default: False]
- **set_property** (*list of 2-item sequence of str or None, optional*) -- Name and value of one or more subdataset properties to be set in the parent dataset's .gitmodules file. The property name is case- insensitive, must start with a letter, and consist only of alphanumeric characters. The value can be a Python format() template string wrapped in '<>' (e.g. '<{gitmodule_name}>'). Supported keywords are any item reported in the result properties of this command, plus 'refds_relpath' and 'refds_relname': the relative path of a subdataset with respect to the base dataset of the command call, and, in the latter case, the same string with all directory separators replaced by dashes. [Default: None]
- **delete_property** (*list of str or None, optional*) -- Name of one or more subdataset properties to be removed from the parent dataset's .gitmodules file. [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']

tree(*, *depth=None*, *recursive=False*, *recursion_limit=None*, *include_files=False*, *include_hidden=False*)

Visualize directory and dataset hierarchies

This command mimics the UNIX/MS-DOS 'tree' utility to generate and display a directory tree, with DataLad-specific enhancements.

It can serve the following purposes:

1. Glorified 'tree' command
2. Dataset discovery
3. Programmatic directory traversal

Glorified 'tree' command

The rendered command output uses 'tree'-style visualization:

```
/tmp/mydir
├── [DS~0] ds_A/
│   └── [DS~1] subds_A/
└── [DS~0] ds_B/
    ├── dir_B/
    │   ├── file.txt
    │   ├── subdir_B/
    │   └── [DS~1] subds_B0/
    └── [DS~1] (not installed) subds_B1/

5 datasets, 2 directories, 1 file
```

Dataset paths are prefixed by a marker indicating subdataset hierarchy level, like `[DS~1]`. This is the absolute subdataset level, meaning it may also take into account superdatasets located above the tree root and thus not included in the output. If a subdataset is registered but not installed (such as after a non-recursive `datalad clone`), it will be prefixed by `(not installed)`. Only DataLad datasets are considered, not pure git/git-annex repositories.

The 'report line' at the bottom of the output shows the count of displayed datasets, in addition to the count of directories and files. In this context, datasets and directories are mutually exclusive categories.

By default, only directories (no files) are included in the tree, and hidden directories are skipped. Both behaviours can be changed using command options.

Symbolic links are always followed. This means that a symlink pointing to a directory is traversed and counted as a directory (unless it potentially creates a loop in the tree).

Dataset discovery

Using the `recursive` or `recursion_limit` option, this command generates the layout of dataset hierarchies based on subdataset nesting level, regardless of their location in the filesystem.

In this case, tree depth is determined by subdataset depth. This mode is thus suited for discovering available datasets when their location is not known in advance.

By default, only datasets are listed, without their contents. If `depth` is specified additionally, the contents of each dataset will be included up to `depth` directory levels (excluding subdirectories that are themselves datasets).

Tree filtering options such as `include_hidden` only affect which directories are reported as dataset contents, not which directories are traversed to find datasets.

Performance note: since no assumption is made on the location of datasets, running this command with the `recursive` or `recursion_limit` option does a full scan of the whole directory tree. As such, it can be significantly slower than a call with an equivalent output that uses `depth` to limit the tree instead.

Programmatic directory traversal

The command yields a result record for each tree node (dataset, directory or file). The following properties are reported, where available:

"path"
Absolute path of the tree node

"type"
Type of tree node: "dataset", "directory" or "file"

"depth"
Directory depth of node relative to the tree root

"exhausted_levels"
Depth levels for which no nodes are left to be generated (the respective subtrees have been 'exhausted')

"count"
Dict with cumulative counts of datasets, directories and files in the tree up until the current node. File count is only included if the command is run with the `include_files` option.

"dataset_depth"
Subdataset depth level relative to the tree root. Only included for node type "dataset".

"dataset_abs_depth"
Absolute subdataset depth level. Only included for node type "dataset".

"dataset_is_installed"
Whether the registered subdataset is installed. Only included for node type "dataset".

"symlink_target"
If the tree node is a symlink, the path to the link target

"is_broken_symlink"
If the tree node is a symlink, whether it is a broken symlink

Examples

Show up to 3 levels of subdirectories below the current directory, including files and hidden contents:

```
> tree(depth=3, include_files=True, include_hidden=True)
```

Find all top-level datasets located anywhere under `/tmp`:

```
> tree('/tmp', recursion_limit=0)
```

Report all subdatasets recursively and their directory contents, up to 1 subdirectory deep within each dataset:

```
> tree(recursive=True, depth=1)
```

Parameters

- **path** -- path to directory from which to generate the tree. Defaults to the current directory. [Default: '.']
- **depth** -- limit the tree to maximum level of subdirectories. If not specified, will generate the full tree with no depth constraint. If paired with `recursive` or `recursion_limit`, refers to the maximum directory level to output below each dataset. [Default: None]
- **recursive** (*bool, optional*) -- produce a dataset tree of the full hierarchy of nested subdatasets. *Note*: may have slow performance on large directory trees. [Default: False]
- **recursion_limit** -- limit the dataset tree to maximum level of nested subdatasets. 0 means include only top-level datasets, 1 means top-level datasets and their immediate subdatasets, etc. *Note*: may have slow performance on large directory trees. [Default: None]
- **include_files** (*bool, optional*) -- include files in the tree. [Default: False]
- **include_hidden** (*bool, optional*) -- include hidden files/directories in the tree. This option does not affect which directories will be searched for datasets when specifying `recursive` or `recursion_limit`. For example, datasets located underneath the hidden folder *.datalad* will be reported even if `include_hidden` is omitted. [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']

uninstall(*, *dataset=None*, *recursive=False*, *check=True*, *if_dirty='save-before'*)

DEPRECATED: use the *drop* command

Parameters

- **path** (*sequence of str or None, optional*) -- path/name of the component to be uninstalled. [Default: None]
- **dataset** (Dataset *or None, optional*) -- specify the dataset to perform the operation on. If no dataset is given, an attempt is made to identify a dataset based on the *path* given. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **check** (*bool, optional*) -- whether to perform checks to assure the configured minimum number (remote) source for data. [Default: True]
- **if_dirty** -- desired behavior if a dataset with unsaved changes is discovered: 'fail' will trigger an error and further processing is aborted; 'save-before' will save all changes prior any further action; 'ignore' let's datalad proceed as if the dataset would not have unsaved changes. [Default: 'save-before']
- **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']

unlock(*, *dataset=None*, *recursive=False*, *recursion_limit=None*)

Unlock file(s) of a dataset

Unlock files of a dataset in order to be able to edit the actual content

Examples

Unlock a single file:

```
> unlock(path='path/to/file')
```

Unlock all contents in the dataset:

```
> unlock('.')
```

Parameters

- **path** (*sequence of str or None, optional*) -- file(s) to unlock. [Default: None]
- **dataset** (*Dataset or None, optional*) -- "specify the dataset to unlock files in. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [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']

update(*, *sibling=None*, *merge=False*, *how=None*, *how_subds=None*, *follow='sibling'*, *dataset=None*, *recursive=False*, *recursion_limit=None*, *fetch_all=None*, *reobtain_data=False*)

Update a dataset from a sibling.

Examples

Update from a particular sibling:

```
> update(sibling='siblingname')
```

Update from a particular sibling and merge the changes from a configured or matching branch from the sibling (see *follow* for details):

```
> update(sibling='siblingname', how='merge')
```

Update from the sibling 'origin', traversing into subdatasets. For subdatasets, merge the revision registered in the parent dataset into the current branch:

```
> update(sibling='origin', how='merge', follow='parentds', recursive=True)
```

Fetch and merge the remote tracking branch into the current dataset. Then update each subdataset by resetting its current branch to the revision registered in the parent dataset, fetching only if the revision isn't already present:

```
> update(how='merge', how_subds='reset', follow='parentds-lazy', recursive=True)
```

Parameters

- **path** (*sequence of str or None, optional*) -- constrain to-be-updated subdatasets to the given path for recursive operation. [Default: None]
- **sibling** (*str or None, optional*) -- name of the sibling to update from. When unspecified, updates from all siblings are fetched. If there is more than one sibling and changes will be brought into the working tree (as requested via *merge*, *how*, or *how_subds*), a sibling will be chosen based on the configured remote for the current branch. [Default: None]
- **merge** (*bool or {'any', 'ff-only'}, optional*) -- merge obtained changes from the sibling. This is a subset of the functionality that can be achieved via the newer *how*. merge=True or merge="any" is equivalent to how="merge". merge="ff-only" is equivalent to how="ff-only". [Default: False]
- **how** (*{'fetch', 'merge', 'ff-only', 'reset', 'checkout', None}, optional*) -- how to update the dataset. The default ("fetch") simply fetches the changes from the sibling but doesn't incorporate them into the working tree. A value of "merge" or "ff-only" merges in changes, with the latter restricting the allowed merges to fast-forwards. "reset" incorporates the changes with 'git reset --hard <target>', staying on the current branch but discarding any changes that aren't shared with the target. "checkout", on the other hand, runs 'git checkout <target>', switching from the current branch to a detached state. When recursive=True is specified, this action will also apply to subdatasets unless overridden by *how_subds*. [Default: None]
- **how_subds** (*{'fetch', 'merge', 'ff-only', 'reset', 'checkout', None}, optional*) -- Override the behavior of *how* in subdatasets. [Default: None]
- **follow** (*{'sibling', 'parentds', 'parentds-lazy'}, optional*) -- source of updates for subdatasets. For 'sibling', the update will be done by merging in a branch from the (specified or inferred) sibling. The branch brought in will either be the current branch's configured branch, if it points to a branch that belongs to the sibling, or a sibling branch with a name that matches the current branch. For 'parentds', the revision registered in the parent dataset of the subdataset is merged in. 'parentds-lazy' is like 'parentds', but prevents fetching from a subdataset's sibling if the registered revision is present in the subdataset. Note that the current dataset is always updated according to 'sibling'. This option has no effect unless a merge is requested and recursive=True is specified. [Default: 'sibling']
- **dataset** (*Dataset or None, optional*) -- specify the dataset to update. If no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [Default: None]

- **fetch_all** (*bool, optional*) -- this option has no effect and will be removed in a future version. When no siblings are given, an all-sibling update will be performed. [Default: None]
- **reobtain_data** (*bool, optional*) -- if enabled, file content that was present before an update will be re-obtained in case a file was changed by the update. [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']

wtf(*, *sensitive=None*, *sections=None*, *flavor='full'*, *decor=None*, *clipboard=None*)

Generate a report about the DataLad installation and configuration

IMPORTANT: Sharing this report with untrusted parties (e.g. on the web) should be done with care, as it may include identifying information, and/or credentials or access tokens.

Parameters

- **dataset** (Dataset *or None, optional*) -- "specify the dataset to report on. no dataset is given, an attempt is made to identify the dataset based on the current working directory. [Default: None]

- **sensitive** (*{None, 'some', 'all'}, optional*) -- if set to 'some' or 'all', it will display sections such as config and metadata which could potentially contain sensitive information (credentials, names, etc.). If 'some', the fields which are known to be sensitive will still be masked out. [Default: None]
- **sections** (*list of {None, 'configuration', 'credentials', 'datalad', 'dataset', 'dependencies', 'environment', 'extensions', 'git-annex', 'location', 'metadata', 'metadata.extractors', 'metadata.filters', 'metadata.indexers', 'python', 'system', '*'}, optional*) -- section to include. If not set - depends on flavor. '*' could be used to force all sections. If there are subsections like section.subsection available, then specifying just 'section' would select all subsections for that section. [Default: None]
- **flavor** (*{'full', 'short'}, optional*) -- Flavor of WTF. 'full' would produce markdown with exhaustive list of sections. 'short' will provide a condensed summary only of datalad and dependencies by default. Use *section* to list other sections. [Default: 'full']
- **decor** (*{'html_details', None}, optional*) -- decoration around the rendering to facilitate embedding into issues etc, e.g. use 'html_details' for posting collapsible entry to GitHub issues. [Default: None]
- **clipboard** (*bool, optional*) -- if set, do not print but copy to clipboard (requires pyperclip module). [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_next.datasets.LeanGitRepo

datalad_next.datasets.**LeanGitRepo**

alias of GitRepo

datalad_next.datasets.LeanAnnexRepo

class datalad_next.datasets.**LeanAnnexRepo**(**args*, ***kwargs*)

Bases: AnnexRepo

git-annex repository representation with a minimized API

This is a companion of *LeanGitRepo*. In the same spirit, it restricts its API to a limited set of method that extend *LeanGitRepo*.

datalad_next.datasets.LegacyGitRepo

datalad_next.datasets.**LegacyGitRepo**

alias of GitRepo

datalad_next.datasets.LegacyAnnexRepo

datalad_next.datasets.**LegacyAnnexRepo**

alias of AnnexRepo

2.3.8 datalad_next.exceptions

Special purpose exceptions

CapturedException(exc[, limit, ...])	This class represents information about an occurred exception (including its traceback), while not holding any references to the actual exception object or its traceback, frame references, etc.
IncompleteResultsError([results, failed, msg])	Exception to be raised whenever results are incomplete.
NoDatasetFound	Raised whenever a dataset is required, but none could be determined

datalad_next.exceptions.CapturedException

class datalad_next.exceptions.**CapturedException**(*exc*, *limit=None*, *capture_locals=False*, *level=8*, *logger=None*)

Bases: object

This class represents information about an occurred exception (including its traceback), while not holding any references to the actual exception object or its traceback, frame references, etc.

Just keep the textual information for logging or whatever other kind of reporting.

format_oneline_tb(*limit=None*, *include_str=True*)

Format an exception traceback as a one-line summary

Returns a string of the form [filename:contextname:linenumber, ...]. If include_str is True (default), this is prepended with the string representation of the exception.

format_short()

Returns a short representation of the original exception

Form: ExceptionName(exception message)

Return type
str

format_standard()

Returns python's standard formatted traceback output

Return type
str

format_with_cause()

Returns a representation of the original exception including the underlying causes

property message

Returns only the message of the original exception

Return type
str

property name

Returns the class name of the original exception

Return type
str

datalad_next.exceptions.IncompleteResultsError

exception datalad_next.exceptions.**IncompleteResultsError**(*results=None*, *failed=None*, *msg=None*)

Exception to be raised whenever results are incomplete.

Any results produced nevertheless are to be passed as *results*, and become available via the *results* attribute.

datalad_next.exceptions.NoDatasetFound

exception datalad_next.exceptions.**NoDatasetFound**

Raised whenever a dataset is required, but none could be determined

2.3.9 datalad_next.iterable_subprocess

Context manager to communicate with a subprocess using iterables

This offers a higher level interface to subprocesses than Python's built-in subprocess module, and is particularly helpful when data won't fit in memory and has to be streamed.

This also allows an external subprocess to be naturally placed in a chain of iterables as part of a data processing pipeline.

This code has been taken from https://pypi.org/project/iterable-subprocess/ and was subsequently adjusted for cross-platform compatibility and performance, as well as tighter integration with DataLad.

The original code was made available under the terms of the MIT License, and was written by Michal Charemza.

iterable_subprocess(program, input_chunks[, ...])	

datalad_next.iterable_subprocess.iterable_subprocess

datalad_next.iterable_subprocess.**iterable_subprocess**(*program*, *input_chunks*, *chunk_size=65536*, *cwd=None*, *bufsize=-1*)

2.3.10 datalad_next.itertools

Various iterators, e.g., for subprocess pipelining and output processing

align_pattern(iterable, pattern)	Yield data chunks that contain a complete pattern, if it is present
decode_bytes(iterable[, encoding, ...])	Decode bytes in an `iterable` into strings
itemize(iterable, sep, *[, keep_ends])	Yields complete items (only), assembled from an iterable
load_json(iterable)	Convert items yielded by `iterable` into JSON objects and yield them
load_json_with_flag(iterable)	Convert items from `iterable` into JSON objects and a success flag
route_out(iterable, data_store, splitter)	Route data around the consumer of this iterable
route_in(iterable, data_store, joiner)	Yield previously rerouted data to the consumer

datalad_next.itertools.align_pattern

datalad_next.itertools.**align_pattern**(*iterable: Iterable[str | bytes | bytearray]*, *pattern: str | bytes | bytearray*) → Generator[str | bytes | bytearray, None, None]

Yield data chunks that contain a complete pattern, if it is present

`align_pattern` makes it easy to find a pattern (`str`, `bytes`, or `bytearray`) in data chunks. It joins data-chunks in such a way, that a simple containment-check (e.g. `pattern in chunk`) on the chunks that `align_pattern` yields will suffice to determine whether the pattern is present in the stream yielded by the underlying iterable or not.

To achieve this, `align_pattern` will join consecutive chunks to ensures that the following two assertions hold:

1. Each chunk that is yielded by `align_pattern` has at least the length of the pattern (unless the underlying iterable is exhausted before the length of the pattern is reached).
2. The pattern is not split between two chunks, i.e. no chunk that is yielded by `align_pattern` ends with a prefix of the pattern (unless it is the last chunk that the underlying iterable yield).

The pattern might be present multiple times in a yielded data chunk.

Note: the `pattern` is compared verbatim to the content in the data chunks, i.e. no parsing of the `pattern` is performed and no regular expressions or wildcards are supported.

```
>>> from datalad_next.itertools import align_pattern
>>> tuple(align_pattern([b'abcd', b'e', b'fghi'], pattern=b'def'))
(b'abcdefghi',)
>>> # The pattern can be present multiple times in a yielded chunk
>>> tuple(align_pattern([b'abcd', b'e', b'fdefghi'], pattern=b'def'))
(b'abcdefdefghi',)
```

Use this function if you want to locate a pattern in an input stream. It allows to use a simple `in`-check to determine whether the pattern is present in the yielded result chunks.

The function always yields everything it has fetched from the underlying iterable. So after a yield it does not cache any data from the underlying iterable. That means, if the functionality of `align_pattern` is no longer required, the underlying iterator can be used, when `align_pattern` has yielded a data chunk. This allows more efficient processing of the data that remains in the underlying iterable.

Parameters

- **iterable** (*Iterable*) -- An iterable that yields data chunks.
- **pattern** (*str | bytes | bytearray*) -- The pattern that should be contained in the chunks. Its type must be compatible to the type of the elements in `iterable`.

Yields

str | bytes | bytearray -- data chunks that have at least the size of the pattern and do not end with a prefix of the pattern. Note that a data chunk might contain the pattern multiple times.

datalad_next.itertools.decode_bytes

datalad_next.itertools.**decode_bytes**(*iterable: Iterable[bytes]*, *encoding: str = 'utf-8'*, *backslash_replace: bool = True*) → Generator[str, None, None]

Decode bytes in an `iterable` into strings

This function decodes `bytes` or `bytearray` into `str` objects, using the specified encoding. Importantly, the decoding input can be spread across multiple chunks of heterogeneous sizes, for example output read from a process or pieces of a download.

Multi-byte encodings that are spread over multiple byte chunks are supported, and chunks are joined as necessary. For example, the utf-8 encoding for ö is `b'\xc3\xb6'`. If the encoding is split in the middle because a chunk ends with `b'\xc3'` and the next chunk starts with `b'\xb6'`, a naive decoding approach like the following would fail:

```
>>> [chunk.decode() for chunk in [b'\xc3', b'\xb6']]
 Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
   File "<stdin>", line 1, in <listcomp>
 UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 0: unexpected␣
↪end of data
```

Compared to:

```
>>> from datalad_next.itertools import decode_bytes
>>> tuple(decode_bytes([b'\xc3', b'\xb6']))
('ö',)
```

Input chunks are only joined, if it is necessary to properly decode bytes:

```
>>> from datalad_next.itertools import decode_bytes
>>> tuple(decode_bytes([b'\xc3', b'\xb6', b'a']))
('ö', 'a')
```

If `backslash_replace` is `True`, undecodable bytes will be replaced with a backslash-substitution. Otherwise, undecodable bytes will raise a `UnicodeDecodeError`:

```
>>> tuple(decode_bytes([b'\xc3']))
('\\xc3',)
>>> tuple(decode_bytes([b'\xc3'], backslash_replace=False))
Traceback (most recent call last):
    ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 1: invalid␣
↪continuation byte
```

Backslash-replacement of undecodable bytes is an ambiguous mapping, because, for example, `b'\xc3'` can already be present in the input.

Parameters

- **iterable** (*Iterable[bytes]*) -- Iterable that yields bytes that should be decoded
- **encoding** (str (default: `'utf-8'`)) -- Encoding to be used for decoding.
- **backslash_replace** (bool (default: `True`)) -- If `True`, backslash-escapes are used for undecodable bytes. If `False`, a `UnicodeDecodeError` is raised if a byte sequence cannot be decoded.

Yields
str -- Decoded strings that are generated by decoding the data yielded by `iterable` with the specified `encoding`

Raises
`UnicodeDecodeError` -- If `backslash_replace` is `False` and the data yielded by `iterable` cannot be decoded with the specified `encoding`

datalad_next.itertools.itemize

datalad_next.itertools.**itemize**(*iterable: Iterable[T]*, *sep: T | None*, *, *keep_ends: bool = False*) → Generator[T, None, None]

Yields complete items (only), assembled from an iterable

This function consumes chunks from an iterable and yields items defined by a separator. An item might span multiple input chunks. Input (chunks) can be `bytes`, `bytearray`, or `str` objects. The result type is determined by the type of the first input chunk. During its runtime, the type of the elements in `iterable` must not change.

Items are defined by a separator given via `sep`. If `sep` is `None`, the line-separators built into `str.splitlines()` are used, and each yielded item will be a line. If `sep` is not *None*, its type must be compatible to the type of the elements in `iterable`.

A separator could, for example, be `b'\n'`, in which case the items would be terminated by Unix line-endings, i.e. each yielded item is a single line. The separator could also be, `b'\x00'` (or `'\x00'`), to split zero-byte delimited content, like the output of `git ls-files -z`.

Separators can be longer than one byte or character, e.g. `b'\r\n'`, or `b'\n----------\n'`.

Content after the last separator, possibly merged across input chunks, is always yielded as the last item, even if it is not terminated by the separator.

Performance notes:

- Using `None` as a separator (splitlines-mode) is slower than providing a specific separator.
- If another separator than `None` is used, the runtime with `keep_end=False` is faster than with `keep_end=True`.

Parameters
- **`iterable`** (*`Iterable[str | bytes | bytearray]`*) -- The iterable that yields the input data
- **`sep`** (*`str | bytes | bytearray | None`*) -- The separator that defines items. If `None`, the items are determined by the line-separators that are built into `str.splitlines()`.
- **`keep_ends`** (*`bool`*) -- If *True*, the item-separator will remain at the end of a yielded item. If *False*, items will not contain the separator. Preserving separators implies a runtime cost, unless the separator is `None`.

Yields
str | bytes | bytearray -- The items determined from the input iterable. The type of the yielded items depends on the type of the first element in `iterable`.

Examples

```
>>> from datalad_next.itertools import itemize
>>> with open('/etc/passwd', 'rt') as f:
...     print(tuple(itemize(iter(f.read, ''), sep=None))[0:2])
('root:x:0:0:root:/root:/bin/bash',
 'systemd-timesync:x:497:497:systemd Time Synchronization:/:/usr/sbin/nologin')
>>> with open('/etc/passwd', 'rt') as f:
...     print(tuple(itemize(iter(f.read, ''), sep=':'))[0:10])
('root', 'x', '0', '0', 'root', '/root',
 '/bin/bash\nsystemd-timesync', 'x', '497', '497')
>>> with open('/etc/passwd', 'rt') as f:
...     print(tuple(itemize(iter(f.read, ''), sep=':', keep_ends=True))[0:10])
('root:', 'x:', '0:', '0:', 'root:', '/root:',
 '/bin/bash\nsystemd-timesync:', 'x:', '497:', '497:')
```

datalad_next.itertools.load_json

datalad_next.itertools.**load_json**(*iterable: Iterable[bytes | str]*) → Generator[Any, None, None]

Convert items yielded by `iterable` into JSON objects and yield them

This function fetches items from the underlying iterable. The items are expected to be `bytes`, `str`, or `bytearry`, and contain one JSON-encoded object. Items are converted into a JSON-object, by feeding them into `json.loads`.

On successful conversion to a JSON-object, `load_json` will yield the resulting JSON-object. If the conversion to a JSON-object fails, `load_json` will raise a `json.decoder.JSONDecodeError`:

```
>>> from datalad_next.itertools import load_json, load_json_with_flag
>>> tuple(load_json(['{"a": 1}']))
({'a': 1},)
>>> tuple(load_json(['{"c": 3']))   # Faulty JSON-encoding, doctest: +SKIP
Traceback (most recent call last):
    ...
json.decoder.JSONDecodeError: Expecting ',' delimiter: line 1 column 8 (char 7)
```

Using `load_json` together with `itemize` allows the processing of JSON-lines data. `itemize` will yield a single item for each line and `load_json` will convert it into a JSON-object.

Note: JSON-decoding is slightly faster if the items of type `str`. Items of type `bytes` or `bytearray` will work as well, but processing might be slower.

Parameters

iterable (*Iterable[bytes | str]*) -- The iterable that yields the JSON-strings or -bytestrings that should be parsed and converted into JSON-objects

Yields

Any -- The JSON-object that are generated from the data yielded by `iterable`

Raises

json.decoder.JSONDecodeError -- If the data yielded by `iterable` is not a valid JSON-string

datalad_next.itertools.load_json_with_flag

datalad_next.itertools.**load_json_with_flag**(*iterable: Iterable[bytes | str]*) → Generator[tuple[Any | json.decoder.JSONDecodeError, bool], None, None]

Convert items from `iterable` into JSON objects and a success flag

`load_json_with_flag` works analogous to `load_json`, but reports success and failure differently.

On successful conversion to a JSON-object, `load_json_with_flag` will yield a tuple of two elements. The first element contains the JSON-object, the second element is `True`.

If the conversion to a JSON-object fails, `load_json_with_flag` will yield a tuple of two elements, where the first element contains the `json.decoder.JSONDecodeError` that was raised during conversion, and the second element is `False`:

```
>>> from datalad_next.itertools import load_json, load_json_with_flag
>>> tuple(load_json_with_flag(['{"b": 2}']))
(({'b': 2}, True),)
>>> tuple(load_json_with_flag(['{"d": 4']))   # Faulty JSON-encoding
((JSONDecodeError("Expecting ',' delimiter: line 1 column 8 (char 7)"), False),)
```

Parameters

iterable (*Iterable[bytes | str]*) -- The iterable that yields the JSON-strings or -bytestrings that should be parsed and converted into JSON-objects

Yields

tuple[Any | json.decoder.JSONDecodeError, bool] -- A tuple containing of a decoded JSON-object and `True`, if the JSON string could be decoded correctly. If the JSON string could not be decoded correctly, the tuple will contain the `json.decoder.JSONDecodeError` that was raised during JSON-decoding and `False`.

datalad_next.itertools.route_out

datalad_next.itertools.**route_out**(*iterable: Iterable*, *data_store: list*, *splitter: Callable[[Any], tuple[Any, Any]]*) → Generator

Route data around the consumer of this iterable

route_out() allows its user to:

1. store data that is received from an iterable,
2. determine whether this data should be yielded to a consumer of `route_out`, by calling `splitter()`.

To determine which data is to be yielded to the consumer and which data should only be stored but not yielded, *route_out()* calls `splitter()`. `splitter()` is called for each item of the input iterable, with the item as sole argument. The function should return a tuple of two elements. The first element is the data that is to be yielded to the consumer. The second element is the data that is to be stored in the list `data_store`. If the first element of the tuple is `datalad_next.itertools.StoreOnly`, no data is yielded to the consumer.

route_in() can be used to combine data that was previously stored by *route_out()* with the data that is yielded by *route_out()* and with the data the was not processed, i.e. not yielded by *route_out()*.

The items yielded by *route_in()* will be in the same order in which they were passed into *route_out()*, including the items that were not yielded by *route_out()* because `splitter()` returned `StoreOnly` in the first element of the result-tuple.

The combination of the two functions *route_out()* and *route_in()* can be used to "carry" additional data along with data that is processed by iterators. And it can be used to route data around iterators that cannot process certain data.

For example, a user has an iterator to divide the number 2 by all numbers in a list. The user wants the iterator to process all numbers in a divisor list, except from zeros, In this case *route_out()* and *route_in()* can be used as follows:

```
from math import nan
from datalad_next.itertools import route_out, route_in, StoreOnly

def splitter(divisor):
    # if divisor == 0, return `StoreOnly` in the first element of the
    # result tuple to indicate that route_out should not yield this
    # element to its consumer
    return (StoreOnly, divisor) if divisor == 0 else (divisor, divisor)

def joiner(processed_data, stored_data):
    #
    return nan if processed_data is StoreOnly else processed_data

divisors = [0, 1, 0, 2, 0, 3, 0, 4]
store = list()
r = route_in(
    map(
        lambda x: 2.0 / x,
        route_out(
            divisors,
            store,
            splitter
        )
    ),
    store,
    joiner
)
print(list(r))
```

The example about will print `[nan, 2.0, nan, 1.0, nan, 0.6666666666666666, nan, 0.5]`.

Parameters

- **iterable** (*Iterable*) -- The iterable that yields the input data
- **data_store** (*list*) -- The list that is used to store the data that is routed out
- **splitter** (*Callable[[Any], tuple[Any, Any | None]]*) -- The function that is used to determine which part of the input data, if any, is to be yielded to the consumer and which data is to be stored in the list `data_store`. The function is called for each item of the input iterable with the item as sole argument. It should return a tuple of two elements. If the first element is not `datalad_next.itertools.StoreOnly`, it is yielded to the consumer. If the first element is `datalad_next.itertools.StoreOnly`, nothing is yielded to the consumer. The second element is stored in the list `data_store`. The cardinality of `data_store` will be the same as the cardinality of the input iterable.

datalad_next.itertools.route_in

datalad_next.itertools.**route_in**(*iterable: Iterable*, *data_store: list*, *joiner: Callable[[Any, Any], Any]*) → Generator

Yield previously rerouted data to the consumer

This function is the counter-part to *route_out()*. It takes the iterable `iterable` and a data store given in `data_store` and yields items in the same order in which *route_out()* received them from its underlying iterable (using the same data store). This includes items that were not yielded by *route_out()*, but only stored.

route_in() uses `joiner()`-function to determine how stored and optionally processed data should be joined into a single item, which is then yielded by *route_in()*. *route_in()* calls `joiner()` with a 2-tuple. The first element of the tuple is either `datalad_next.itertools.StoreOnly` or the next item from the underlying iterator. The second element is the data that was stored in the data store. The result of `joiner()` which will be yielded by *route_in()*.

This module provides a standard joiner-function: `join_with_list()` that works with splitter-functions that return a list as second element of the result tuple.

The cardinality of `iterable` must match the number of processed data elements in the data store. The output cardinality of *route_in()* will be the cardinality of the input iterable of the corresponding *route_out()*-call. Given the following code:

```
store_1 = list()
route_in(
    some_generator(
        route_out(input_iterable, store_1, splitter_1)
    ),
    store_1,
    joiner_1
)
```

route_in() will yield the same number of elements as `input_iterable`. But, the number of elements processed by `some_generator` is determined by the `splitter_1()` in *route_out()*, i.e. by the number of `splitter_1()`-results that have don't have `datalad_next.itertools.don_process` as first element.

Parameters

- **iterable** (*Iterable*) -- The iterable that yields the input data.
- **data_store** (*list*) -- The list from which the data that is to be "routed in" is read.
- **joiner** (*Callable[[Any, Any], Any]*) -- A function that determines how the items that are yielded by `iterable` should be combined with the corresponding data from `data_store`, in order to yield the final result. The first argument to `joiner` is the item that is yielded by `iterable`, or `datalad_next.itertools.StoreOnly` if no data was processed in the corresponding step. The second argument is the data that was stored in `data_store` in the corresponding step.

2.3.11 datalad_next.iter_collections

Iterators for particular types of collections

Most importantly this includes different collections (or containers) for files, such as a file system directory, or an archive (also see the `ls_file_collection` command). However, this module is not per-se limited to file collections.

Most, if not all, implementation come in the form of a function that takes a collection identifier or a collection location (e.g., a file system path), and possibly some additional options. When called, an iterator is returned that produces collection items in the form of data class instances of a given type. The particular type can be different across different collections.

iter_annexworktree(path, *[, untracked, ...])	Companion to `iter_gitworktree()` for git-annex repositories
iter_dir(path, *[, fp])	Uses `Path.iterdir()` to iterate over a directory and reports content
iter_gitdiff(path, from_treeish, to_treeish, *)	Report differences between Git tree-ishes or tracked worktree content
iter_gitstatus(path, *[, untracked, ...])	Recursion mode 'no'
iter_gittree(path, treeish, *[, recursive])	Uses `git ls-tree` to report on a tree in a Git repository
iter_gitworktree(path, *[, untracked, ...])	Uses `git ls-files` to report on a work tree of a Git repository
iter_submodules(path)	Given a path, report all submodules of a repository work-tree underneath
iter_tar(path, *[, fp])	Uses the standard library `tarfile` module to report on TAR archives
iter_zip(path, *[, fp])	Uses the standard library `zipfile` module to report on ZIP-files
TarfileItem(type, name, size[, mtime, mode, ...])	
ZipfileItem(type, name, size[, mtime, mode, ...])	
FileSystemItem(type, name, size[, mtime, ...])	
FileSystemItemType(value)	Enumeration of file system path types
GitTreeItemType(value)	Enumeration of item types of Git trees
GitWorktreeItem(name[, gitsha, gittype])	
GitWorktreeFileSystemItem(type, name, size)	
GitDiffItem(name[, gitsha, gittype, ...])	`GitTreeItem` with "previous" property values given a state comparison
GitDiffStatus(value)	Enumeration of statuses for diff items
GitContainerModificationType(value)	An enumeration.

datalad_next.iter_collections.iter_annexworktree

datalad_next.iter_collections.**iter_annexworktree**(*path: Path*, *, *untracked: str | None = 'all'*, *link_target: bool = False*, *fp: bool = False*, *recursive: str = 'repository'*) → Generator[AnnexWorktreeItem | AnnexWorktreeFileSystemItem, None, None]

Companion to `iter_gitworktree()` for git-annex repositories

This iterator wraps `iter_gitworktree()`. For each item, it determines whether it is an annexed file. If so, it amends the yielded item with information on the respective annex key, the byte size of the key, and its (would-be) location in the repository's annex.

The basic semantics of all arguments are identical to `iter_gitworktree()`. Importantly, with `fp=True`, an annex object is opened directly, if available. If not available, no attempt is made to open the associated symlink or pointer file.

With `link_target` and `fp` disabled items of type `AnnexWorktreeItem` are yielded, otherwise `AnnexWorktreeFileSystemItem` instances are yielded. In both cases, `annexkey`, `annexsize`, and `annnexobjpath` properties are provided.

Note: Although `annexobjpath` is always set for annexed content, that does not imply that an object at this path actually exists. The latter will only be the case if the annexed content is present in the work tree, typically as a result of a *datalad get*- or *git annex get*-call.

Parameters

- **path** (*Path*) -- Path of a directory in a git-annex repository to report on. This directory need not be the root directory of the repository, but must be part of the repository's work tree.
- **untracked** (*{'all', 'whole-dir', 'no-empty-dir'} or None, optional*) -- If not `None`, also reports on untracked work tree content. `all` reports on any untracked file; `whole-dir` yields a single report for a directory that is entirely untracked, and not individual untracked files in it; `no-empty-dir` skips any reports on untracked empty directories.
- **link_target** (*bool, optional*) -- If `True`, information matching a *FileSystemItem* will be included for each yielded item, and the targets of any symlinks will be reported, too.
- **fp** (*bool, optional*) -- If `True`, information matching a *FileSystemItem* will be included for each yielded item, but without a link target detection, unless `link_target` is given. Moreover, each file-type item includes a file-like object to access the file's content. This file handle will be closed automatically when the next item is yielded.
- **recursive** (*{'repository', 'no'}, optional*) -- Pass on to `iter_gitworktree()`, thereby determining which items this iterator will yield.

Yields

`AnnexWorktreeItem` or `AnnexWorktreeFileSystemItem` -- The `name` attribute of an item is a `PurePath` instance with the corresponding (relative) path, in platform conventions.

datalad_next.iter_collections.iter_dir

datalad_next.iter_collections.**iter_dir**(*path: Path*, *, *fp: bool = False*) → Generator[DirectoryItem, None, None]

Uses `Path.iterdir()` to iterate over a directory and reports content

The iterator produces an `DirectoryItem` instance with standard information on file system elements, such as `size`, or `mtime`.

In addition to a plain `Path.iterdir()` the report includes a path-type label (distinguished are `file`, `directory`, `symlink`).

Parameters

- **path** (*Path*) -- Path of the directory to report content for (iterate over).
- **fp** (*bool, optional*) -- If `True`, each file-type item includes a file-like object to access the file's content. This file handle will be closed automatically when the next item is yielded.

Yields

`DirectoryItem` -- The `name` attribute of an item is a `Path` instance, with the format matching the main `path` argument. When an absolute `path` is given, item names are absolute paths too. When a relative path is given, it is relative to CWD, and items names are relative paths (relative to CWD) too.

datalad_next.iter_collections.iter_gitdiff

datalad_next.iter_collections.**iter_gitdiff**(*path: Path*, *from_treeish: str | None*, *to_treeish: str | None*, *, *recursive: str = 'repository'*, *find_renames: int | None = None*, *find_copies: int | None = None*, *yield_tree_items: str | None = None*, *eval_submodule_state: str = 'full'*) → Generator[*GitDiffItem*, None, None]

Report differences between Git tree-ishes or tracked worktree content

This function is a wrapper around the Git command `diff-tree` and `diff-index`. Therefore most semantics also apply here.

The main difference with respect to the Git commands are: 1) uniform support for non-recursive, single tree reporting (no subtrees); and 2) support for submodule recursion.

Notes on 'no' recursion mode

When comparing to the worktree, `git diff-index` always reports on subdirectories. For homogeneity with the report on a committed tree, a non-recursive mode emulation is implemented. It compresses all reports from a direct subdirectory into a single report on that subdirectory. The `gitsha` of that directory item will always be `None`. Moreover, no type or typechange inspection, or further filesystem queries are performed. Therefore, `prev_gittype` will always be `None`, and any change other than the addition of the directory will be labeled as a `GitDiffStatus.modification`.

Parameters

- **path** (*Path*) -- Path of a directory in a Git repository to report on. This directory need not be the root directory of the repository, but must be part of the repository. If the directory is not the root directory of a non-bare repository, the iterator is constrained to items underneath that directory.
- **from_treeish** (*str or None*) -- Git "tree-ish" that defines the comparison reference. If `None`, `to_treeeish` must not be `None` (see its documentation for details).

- **to_treeish** -- Git "tree-ish" that defines the comparison target. If `None`, `from_treeish` must not be `None`, and that tree-ish will be compared against the worktree. (see its documentation for details). If `from_treeish` is `None`, the given tree-ish is compared to its immediate parents (see `git diff-tree` documentation for details).
- **recursive** (*{'repository', 'submodules', 'no'}, optional*) -- Behavior for recursion into subtrees. By default (`repository`), all trees within the repository underneath `path`) are reported, but no tree within submodules. With `submodules`, recursion includes any submodule that is present. If `no`, only direct children are reported on.
- **find_renames** (*int, optional*) -- If given, this defines the similarity threshold for detecting renames (see `git diff-{index,tree} --find-renames`). By default, no rename detection is done and reported items never have the `rename` status. Instead, a renames would be reported as a deletion and an addition.
- **find_copied** (*int, optional*) -- If given, this defines the similarity threshold for detecting copies (see `git diff-{index,tree} --find-copies`). By default, no copy detection is done and reported items never have the `copy` status. Instead, a copy would be reported as addition. This option always implies the use of the `--find-copies-harder` Git option that enables reporting of copy sources, even when they have not been modified in the same change. This is a very expensive operation for large projects, so use it with caution.
- **yield_tree_items** (*{'submodules', 'directories', 'all', None}, optional*) -- Whether to yield an item on type of subtree that will also be recursed into. For example, a submodule item, when submodule recursion is enabled. When disabled, subtree items (directories, submodules) will still be reported whenever there is no recursion into them. For example, submodule items are reported when `recursive='repository'`, even when `yield_tree_items=None`.

Yields

GitDiffItem -- The `name` and `prev_name` attributes of an item are a `str` with the corresponding (relative) path, as reported by Git (in POSIX conventions).

datalad_next.iter_collections.iter_gitstatus

datalad_next.iter_collections.**iter_gitstatus**(*path: Path*, *, *untracked: str | None = 'all'*, *recursive: str = 'repository'*, *eval_submodule_state: str = 'full'*) → Generator[*GitDiffItem*, None, None]

Recursion mode 'no'

This mode limits the reporting to immediate directory items of a given path. This mode is not necessarily faster than a 'repository' recursion. Its primary purpose is the ability to deliver a collapsed report in that subdirectories are treated similar to submodules -- as containers that maybe have modified or untracked content.

Parameters

- **path** (*Path*) -- Path of a directory in a Git repository to report on. This directory need not be the root directory of the repository, but must be part of the repository. If the directory is not the root directory of a non-bare repository, the iterator is constrained to items underneath that directory.
- **untracked** (*{'all', 'whole-dir', 'no-empty-dir'} or None, optional*) -- If not `None`, also reports on untracked work tree content. `all` reports on any untracked file; `whole-dir` yields a single report for a directory that is entirely untracked, and not individual untracked files in it; `no-empty-dir` skips any reports on untracked empty directories. Also see `eval_submodule_state` for how this parameter is applied in submodule recursion.

- **recursive** (*{'no', 'repository', 'submodules', 'monolithic'}, optional*) -- Behavior for recursion into subtrees. By default (`repository`), all trees within the repository underneath `path`) are reported, but no tree within submodules. With `submodules`, recursion includes any submodule that is present. If `no`, only direct children are reported on.
- **eval_submodule_state** (*{"no", "commit", "full"}, optional*) -- If 'full' (default), the state of a submodule is evaluated by considering all modifications, with the treatment of untracked files determined by *untracked*. If 'commit', the modification check is restricted to comparing the submodule's "HEAD" commit to the one recorded in the superdataset. If 'no', the state of the subdataset is not evaluated. When a git-annex repository in adjusted mode is detected, the reference commit that the worktree is being compared to is the basis of the adjusted branch (i.e., the corresponding branch).

Yields

GitDiffItem -- The `name` and `prev_name` attributes of an item are a `str` with the corresponding (relative) path, as reported by Git (in POSIX conventions).

datalad_next.iter_collections.iter_gittree

datalad_next.iter_collections.**iter_gittree**(*path: Path*, *treeish: str*, ***, *recursive: str = 'repository'*) → Generator[GitTreeItem, None, None]

Uses `git ls-tree` to report on a tree in a Git repository

Parameters

- **path** (*Path*) -- Path of a directory in a Git repository to report on. This directory need not be the root directory of the repository, but must be part of the repository. If the directory is not the root directory of a non-bare repository, the iterator is constrained to items underneath that directory.
- **recursive** (*{'repository', 'no'}, optional*) -- Behavior for recursion into subtrees. By default (`repository`), all tree within the repository underneath `path`) are reported, but not tree within submodules. If `no`, only direct children are reported on.

Yields

`GitTreeItem` -- The `name` attribute of an item is a `str` with the corresponding (relative) path, as reported by Git (in POSIX conventions).

datalad_next.iter_collections.iter_gitworktree

datalad_next.iter_collections.**iter_gitworktree**(*path: Path*, ***, *untracked: str | None = 'all'*, *link_target: bool = False*, *fp: bool = False*, *recursive: str = 'repository'*) → Generator[*GitWorktreeItem* | *GitWorktreeFileSystemItem*, None, None]

Uses `git ls-files` to report on a work tree of a Git repository

This iterator can be used to report on all tracked, and untracked content of a Git repository's work tree. This includes files that have been removed from the work tree (deleted), unless their removal has already been staged.

For any tracked content, yielded items include type information and gitsha as last known to Git. This means that such reports reflect the last committed or staged content, not the state of a potential unstaged modification in the work tree.

When no reporting of link targets or file-objects are requested, items of type *GitWorktreeItem* are yielded, otherwise *GitWorktreeFileSystemItem* instances. In both cases, `gitsha` and `gittype` properties are provided. Either of them being `None` indicates untracked work tree content.

Note: The `gitsha` is not equivalent to a SHA1 hash of a file's content, but is the SHA-type blob identifier as reported and used by Git.

Parameters

- **path** (*Path*) -- Path of a directory in a Git repository to report on. This directory need not be the root directory of the repository, but must be part of the repository's work tree.
- **untracked** (*{'all', 'whole-dir', 'no-empty-dir'} or None, optional*) -- If not `None`, also reports on untracked work tree content. `all` reports on any untracked file; `whole-dir` yields a single report for a directory that is entirely untracked, and not individual untracked files in it; `no-empty-dir` skips any reports on untracked empty directories.
- **link_target** (*bool, optional*) -- If `True`, information matching a *FileSystemItem* will be included for each yielded item, and the targets of any symlinks will be reported, too.
- **fp** (*bool, optional*) -- If `True`, information matching a *FileSystemItem* will be included for each yielded item, but without a link target detection, unless `link_target` is given. Moreover, each file-type item includes a file-like object to access the file's content. This file handle will be closed automatically when the next item is yielded.
- **recursive** (*{'repository', 'no'}, optional*) -- Behavior for recursion into subdirectories of `path`. By default (`repository`), all directories within the repository are reported. This possibly includes untracked ones (see `untracked`), but not directories within submodules. If `no`, only direct children of `path` are reported on. For any worktree items in subdirectories of `path` only a single record for the containing immediate subdirectory `path` is yielded. For example, with 'path/subdir/file1' and 'path/subdir/file2' there will only be a single item with `name='subdir'` and `type='directory'`.

Yields

GitWorktreeItem or *GitWorktreeFileSystemItem* -- The `name` attribute of an item is a `PurePath` instance with the corresponding (relative) path, in platform conventions.

datalad_next.iter_collections.iter_submodules

datalad_next.iter_collections.**iter_submodules**(*path: Path*) → Generator[GitTreeItem, None, None]

Given a path, report all submodules of a repository worktree underneath

This is a thin convenience wrapper around `iter_gitworktree()`.

datalad_next.iter_collections.iter_tar

datalad_next.iter_collections.**iter_tar**(*path: Path*, *, *fp: bool = False*) → Generator[*TarfileItem*, None, None]

Uses the standard library `tarfile` module to report on TAR archives

A TAR archive can represent more or less the full bandwidth of file system properties, therefore reporting on archive members is implemented similar to `iter_dir()`. The iterator produces an *TarfileItem* instance with standard information on file system elements, such as `size`, or `mtime`.

Parameters

- **path** (*Path*) -- Path of the TAR archive to report content for (iterate over).

- **fp** (*bool, optional*) -- If `True`, each file-type item includes a file-like object to access the file's content. This file handle will be closed automatically when the next item is yielded or the function returns.

Yields

TarfileItem -- The `name` attribute of an item is a `str` with the corresponding archive member name (in POSIX conventions).

datalad_next.iter_collections.iter_zip

datalad_next.iter_collections.**iter_zip**(*path: Path*, *, *fp: bool = False*) → Generator[*ZipfileItem*, None, None]

Uses the standard library `zipfile` module to report on ZIP-files

A ZIP archive can represent more or less the full bandwidth of file system properties, therefore reporting on archive members is implemented similar to `iter_dir()`. The iterator produces an *ZipfileItem* instance with standard information on file system elements, such as `size`, or `mtime`.

Parameters

- **path** (*Path*) -- Path of the ZIP archive to report content for (iterate over).
- **fp** (*bool, optional*) -- If `True`, each file-type item includes a file-like object to access the file's content. This file handle will be closed automatically when the next item is yielded or the function returns.

Yields

ZipfileItem -- The `name` attribute of an item is a `str` with the corresponding archive member name (in POSIX conventions).

datalad_next.iter_collections.TarfileItem

class datalad_next.iter_collections.**TarfileItem**(*type: 'FileSystemItemType'*, *name: 'str'*, *size: 'int'*, *mtime: 'float | None' = None*, *mode: 'int | None' = None*, *uid: 'int | None' = None*, *gid: 'int | None' = None*, *link_target: 'str | None' = None*, *fp: 'IO | None' = None*)

Bases: *FileSystemItem*

link_target: str | None = None

Just as for `name`, a link target is also reported in POSIX format.

property link_target_path: PurePosixPath

Returns the link_target as a `PurePosixPath` instance

name: str

TAR uses POSIX paths as item identifiers. Not all POSIX paths can be represented on all (non-POSIX) file systems, therefore the item name is represented in POSIX form, instead of in platform conventions.

property path: PurePosixPath

Returns the item name as a `PurePosixPath` instance

datalad_next.iter_collections.ZipfileItem

class datalad_next.iter_collections.**ZipfileItem**(*type: 'FileSystemItemType'*, *name: 'str'*, *size: 'int'*, *mtime: 'float | None' = None*, *mode: 'int | None' = None*, *uid: 'int | None' = None*, *gid: 'int | None' = None*, *link_target: 'Any | None' = None*, *fp: 'IO | None' = None*)

Bases: *FileSystemItem*

name: str

property path: PurePosixPath

Returns the item name as a `PurePosixPath` instance

ZIP uses POSIX paths as item identifiers from version 6.3.3 onwards. Not all POSIX paths are legal paths on non-POSIX file systems or platforms. Therefore we cannot use a platform-dependent `PurePath`-instance to address ZIP-file items, anq we use `PurePosixPath`-instances instead.

datalad_next.iter_collections.FileSystemItem

class datalad_next.iter_collections.**FileSystemItem**(*type: 'FileSystemItemType'*, *name: 'Any'*, *size: 'int'*, *mtime: 'float | None' = None*, *mode: 'int | None' = None*, *uid: 'int | None' = None*, *gid: 'int | None' = None*, *link_target: 'Any | None' = None*, *fp: 'IO | None' = None*)

Bases: `PathBasedItem`, `TypedItem`

fp: IO | None = None

classmethod from_path(*path: Path*, ***, *link_target: bool = True*)

Populate item properties from a single *stat* and *readlink* call

The given `path` must exist. The `link_target` flag indicates whether to report the result of `readlink` for a symlink-type path.

gid: int | None = None

link_target: Any | None = None

link_target_path() → PurePath

Returns the link_target as a `PurePath` instance

mode: int | None = None

mtime: float | None = None

size: int

type: *FileSystemItemType*

uid: int | None = None

datalad_next.iter_collections.FileSystemItemType

class datalad_next.iter_collections.**FileSystemItemType**(*value*)

Bases: Enum

Enumeration of file system path types

The associated `str` values are chosen to be appropriate for downstream use (e.g, as type labels in DataLad result records).

directory = 'directory'

file = 'file'

hardlink = 'hardlink'

specialfile = 'specialfile'

symlink = 'symlink'

datalad_next.iter_collections.GitTreeItemType

class datalad_next.iter_collections.**GitTreeItemType**(*value*)

Bases: Enum

Enumeration of item types of Git trees

directory = 'directory'

executablefile = 'executablefile'

file = 'file'

submodule = 'submodule'

symlink = 'symlink'

datalad_next.iter_collections.GitWorktreeItem

class datalad_next.iter_collections.**GitWorktreeItem**(*name: 'PurePath', gitsha: 'str | None' = None, gittype: 'GitTreeItemType | None' = None*)

Bases: GitTreeItem

name: PurePath

datalad_next.iter_collections.GitWorktreeFileSystemItem

class datalad_next.iter_collections.**GitWorktreeFileSystemItem**(*type: 'FileSystemItemType', name: 'PurePath', size: 'int', mtime: 'float | None' = None, mode: 'int | None' = None, uid: 'int | None' = None, gid: 'int | None' = None, link_target: 'Any | None' = None, fp: 'IO | None' = None, gitsha: 'str | None' = None, gittype: 'GitTreeItemType | None' = None*)

Bases: *FileSystemItem*

gitsha: str | None = None

gittype: *GitTreeItemType* | None = None

name: PurePath

datalad_next.iter_collections.GitDiffItem

class datalad_next.iter_collections.**GitDiffItem**(*name: str*, *gitsha: str | None = None*, *gittype:* GitTreeItemType | *None = None*, *prev_name: str | None = None*, *prev_gitsha: str | None = None*, *prev_gittype:* GitTreeItemType | *None = None*, *status:* GitDiffStatus | *None = None*, *percentage: int | None = None*, *modification_types:* *tuple[*GitContainerModificationType*, ...] | None = None*)

Bases: GitTreeItem

GitTreeItem with "previous" property values given a state comparison

add_modification_type(*value:* GitContainerModificationType)

modification_types: tuple[*GitContainerModificationType*, ...] | None = None

Qualifiers for modification types of container-type items (directories, submodules).

percentage: int | None = None

This is the percentage of similarity for copy-status and rename-status diff items, and the percentage of dissimilarity for modifications.

prev_gitsha: str | None = None

prev_gittype: *GitTreeItemType* | None = None

prev_name: str | None = None

property prev_path: PurePosixPath | None

Returns the item prev_name as a PurePosixPath instance

status: *GitDiffStatus* | None = None

datalad_next.iter_collections.GitDiffStatus

class datalad_next.iter_collections.**GitDiffStatus**(*value*)

Bases: Enum

Enumeration of statuses for diff items

addition = 'addition'

copy = 'copy'

deletion = 'deletion'

modification = 'modification'

other = 'other'

rename = 'rename'

typechange = 'typechange'

unknown = 'unknown'

unmerged = 'unmerged'

datalad_next.iter_collections.GitContainerModificationType

class datalad_next.iter_collections.**GitContainerModificationType**(*value*)

Bases: Enum

An enumeration.

modified_content = 'modified content'

new_commits = 'new commits'

untracked_content = 'untracked content'

2.3.12 datalad_next.repo_utils

Common repository operations

get_worktree_head(path)	

datalad_next.repo_utils.get_worktree_head

datalad_next.repo_utils.**get_worktree_head**(*path: Path*) → tuple[str | None, str | None]

2.3.13 datalad_next.runners

Execution of subprocesses

This module provides all relevant components for subprocess execution. The main work horse is *iter_subproc()*, a context manager that enables interaction with a subprocess in the form of an iterable for input/output processing. Execution errors are communicated with the *CommandError* exception. In addition, a few convenience functions are provided to execute Git commands (including git-annex).

iter_subproc(args, *[, input, chunk_size, ...])	Context manager to communicate with a subprocess using iterables
call_git(args, *[, cwd, force_c_locale])	Call Git with no output capture, raises on non-zero exit.
call_git_lines(args, *[, cwd, input, ...])	Call Git for any (small) number of lines of output
call_git_oneline(args, *[, cwd, input, ...])	Call Git for a single line of output
call_git_success(args, *[, cwd, capture_output])	Call Git and report success or failure of the command
iter_git_subproc(args, **kwargs)	`iter_subproc()` wrapper for calling Git commands
CommandError([cmd, msg, code, stdout, ...])	Thrown if a command call fails.

datalad_next.runners.iter_subproc

datalad_next.runners.**iter_subproc**(*args: List[str]*, *, *input: Iterable[bytes] | None = None*, *chunk_size: int = 65536*, *cwd: Path | None = None*, *bufsize: int = -1*)

Context manager to communicate with a subprocess using iterables

This offers a higher level interface to subprocesses than Python's built-in `subprocess` module. It allows a subprocess to be naturally placed in a chain of iterables as part of a data processing pipeline. It is also helpful when data won't fit in memory and has to be streamed.

This is a convenience wrapper around `datalad_next.iterable_subprocess`, which itself is a slightly modified (for use on Windows) fork of https://github.com/uktrade/iterable-subprocess, written by Michal Charemza.

This function provides a context manager. On entering the context, the subprocess is started, the thread to read from standard error is started, the thread to populate subprocess input is started. When running, the standard input thread iterates over the input, passing chunks to the process, while the standard error thread fetches the error output, and while the main thread iterates over the process's output from client code in the context.

On context exit, the main thread closes the process's standard output, waits for the standard input thread to exit, waits for the standard error thread to exit, and wait for the process to exit. If the process exited with a non-zero return code, a `CommandError` is raised, containing the process's return code.

If the context is exited due to an exception that was raised in the context, the main thread terminates the process via `Popen.terminate()`, closes the process's standard output, waits for the standard input thread to exit, waits for the standard error thread to exit, waits for the process to exit, and re-raises the exception.

Note, if an exception is raised in the context, this exception will bubble up to the main thread. That means no `CommandError` will be raised if the subprocess exited with a non-zero return code. To access the return code in case of an exception inside the context, use the `code`-attribute of the `as`-variable. This object will always contain the return code of the subprocess. For example, the following code will raise a `StopIteration`-exception in the context (by repeatedly using `next()`). The subprocess will exit with `2` due to the illegal option `-@`, and no `CommandError` is raised. The return code is read from the variable `ls_stdout`

```
>>> from datalad_next.runners import iter_subproc
>>> try:
...     with iter_subproc(['ls', '-@']) as ls_stdout:
...         while True:
...             next(ls_stdout)
... except Exception as e:
...     print(repr(e), ls_stdout.returncode)
StopIteration() 2
```

Parameters

- **args** (*list*) -- Sequence of program arguments to be passed to `subprocess.Popen`.
- **input** (*iterable, optional*) -- If given, chunks of `bytes` to be written, iteratively, to the subprocess's `stdin`.
- **chunk_size** (*int, optional*) -- Size of chunks to read from the subprocess's stdout/stderr in bytes.
- **cwd** (*Path*) -- Working directory for the subprocess, passed to `subprocess.Popen`.
- **bufsize** (*int, optional*) -- Buffer size to use for the subprocess's `stdin`, `stdout`, and `stderr`. See `subprocess.Popen` for details.

Return type

contextmanager

datalad_next.runners.call_git

datalad_next.runners.**call_git**(*args: list[str]*, *, *cwd: Path | None = None*, *force_c_locale: bool = False*) → None

Call Git with no output capture, raises on non-zero exit.

If `cwd` is not None, the function changes the working directory to `cwd` before executing the command.

If `force_c_locale` is `True` the environment of the Git process is altered to ensure output according to the C locale. This is useful when output has to be processed in a locale invariant fashion.

datalad_next.runners.call_git_lines

datalad_next.runners.**call_git_lines**(*args: list[str]*, *, *cwd: Path | None = None*, *input: str | None = None*, *force_c_locale: bool = False*) → list[str]

Call Git for any (small) number of lines of output

`args` is a list of arguments for the Git command. This list must not contain the Git executable itself. It will be prepended (unconditionally) to the arguments before passing them on.

If `cwd` is not None, the function changes the working directory to `cwd` before executing the command.

If `input` is not None, the argument becomes the subprocess's stdin. This is intended for small-scale inputs. For call that require processing large inputs, `iter_git_subproc()` is to be preferred.

If `force_c_locale` is `True` the environment of the Git process is altered to ensure output according to the C locale. This is useful when output has to be processed in a locale invariant fashion.

Raises

CommandError if the call exits with a non-zero status. --

datalad_next.runners.call_git_oneline

datalad_next.runners.**call_git_oneline**(*args: list[str]*, *, *cwd: Path | None = None*, *input: str | None = None*, *force_c_locale: bool = False*) → str

Call Git for a single line of output

If `cwd` is not None, the function changes the working directory to `cwd` before executing the command.

If `input` is not None, the argument becomes the subprocess's stdin. This is intended for small-scale inputs. For call that require processing large inputs, `iter_git_subproc()` is to be preferred.

If `force_c_locale` is `True` the environment of the Git process is altered to ensure output according to the C locale. This is useful when output has to be processed in a locale invariant fashion.

Raises

- **CommandError if the call exits with a non-zero status.** --
- **AssertionError if there is more than one line of output.** --

datalad_next.runners.call_git_success

datalad_next.runners.**call_git_success**(*args: list[str]*, *, *cwd: Path | None = None*, *capture_output: bool = False*) → bool

Call Git and report success or failure of the command

`args` is a list of arguments for the Git command. This list must not contain the Git executable itself. It will be prepended (unconditionally) to the arguments before passing them on.

If `cwd` is not None, the function changes the working directory to `cwd` before executing the command.

If `capture_output` is `True`, process output is captured, but not returned. By default process output is not captured.

datalad_next.runners.iter_git_subproc

datalad_next.runners.**iter_git_subproc**(*args: list[str]*, ***kwargs*)

`iter_subproc()` wrapper for calling Git commands

All argument semantics are identical to those of `iter_subproc()`, except that `args` must not contain the Git binary, but need to be exclusively arguments to it. The respective *git* command/binary is automatically added internally.

datalad_next.runners.CommandError

exception datalad_next.runners.**CommandError**(*cmd: str | list[str] = ''*, *msg: str = ''*, *code: int | None = None*, *stdout: str | bytes = ''*, *stderr: str | bytes = ''*, *cwd: str | os.PathLike | None = None*, ***kwargs: Any*)

Thrown if a command call fails.

Note: Subclasses should override *to_str* rather than *__str__* because *to_str* is called directly in datalad.cli.main.

Low-level tooling from datalad-core

Deprecated since version 1.4: The functionality described here has been deprecated, and the associated imports from datalad-core are scheduled for removal with version 2.0. Use the implementations listed above instead.

Few process execution/management utilities are provided, for generic command execution, and for execution command in the context of a Git repository.

GitRunner	alias of `GitWitlessRunner`
Runner	alias of `WitlessRunner`

datalad_next.runners.GitRunner

datalad_next.runners.**GitRunner**

alias of GitWitlessRunner

datalad_next.runners.Runner

datalad_next.runners.**Runner**

alias of WitlessRunner

Additional information on the design of the subprocess execution tooling is available from https://docs.datalad.org/design/threaded_runner.html

A standard exception type is used to communicate any process termination with a non-zero exit code

CommandError([cmd, msg, code, stdout, ...])	Thrown if a command call fails.

Command output can be processed via "protocol" implementations that are inspired by asyncio.SubprocessProtocol.

KillOutput([done_future, encoding])	WitlessProtocol that swallows stdout/stderr of a subprocess
NoCapture([done_future, encoding])	WitlessProtocol that captures no subprocess output
StdOutCapture([done_future, encoding])	WitlessProtocol that only captures and returns stdout of a subprocess
StdErrCapture([done_future, encoding])	WitlessProtocol that only captures and returns stderr of a subprocess
StdOutErrCapture([done_future, encoding])	WitlessProtocol that captures and returns stdout/stderr of a subprocess

datalad_next.runners.KillOutput

class datalad_next.runners.**KillOutput**(*done_future: Any | None = None*, *encoding: str | None = None*)

Bases: WitlessProtocol

WitlessProtocol that swallows stdout/stderr of a subprocess

pipe_data_received(*fd: int*, *data: bytes*) → None

proc_err = True

proc_out = True

datalad_next.runners.NoCapture

class datalad_next.runners.**NoCapture**(*done_future: Any | None = None*, *encoding: str | None = None*)

Bases: `WitlessProtocol`

WitlessProtocol that captures no subprocess output

As this is identical with the behavior of the WitlessProtocol base class, this class is merely a more readable convenience alias.

datalad_next.runners.StdOutCapture

class datalad_next.runners.**StdOutCapture**(*done_future: Any | None = None*, *encoding: str | None = None*)

Bases: `WitlessProtocol`

WitlessProtocol that only captures and returns stdout of a subprocess

proc_out = True

datalad_next.runners.StdErrCapture

class datalad_next.runners.**StdErrCapture**(*done_future: Any | None = None*, *encoding: str | None = None*)

Bases: `WitlessProtocol`

WitlessProtocol that only captures and returns stderr of a subprocess

proc_err = True

datalad_next.runners.StdOutErrCapture

class datalad_next.runners.**StdOutErrCapture**(*done_future: Any | None = None*, *encoding: str | None = None*)

Bases: `WitlessProtocol`

WitlessProtocol that captures and returns stdout/stderr of a subprocess

proc_err = True

proc_out = True

2.3.14 datalad_next.shell

A persistent shell connection

This module provides a context manager that establishes a connection to a shell and can be used to execute multiple commands in that shell. Shells are usually remote shells, e.g. connected via an `ssh`-client, but local shells like `zsh`, `bash` or `PowerShell` can also be used.

The context manager returns an instance of *ShellCommandExecutor* that can be used to execute commands in the shell via the method *ShellCommandExecutor.__call__()*. The method will return an instance of a subclass of *ShellCommandResponseGenerator* that can be used to retrieve the output of the command, the result code of the command, and the stderr-output of the command.

Every response generator expects a certain output structure. It is responsible for ensuring that the output structure is generated. To this end every response generator provides a method `ShellCommandResponseGenerator.get_command_list()`. The method *ShellCommandExecutor.__call__* will pass the user-provided command to `ShellCommandResponseGenerator.get_command_list()` and receive a list of final commands that should be executed in the connected shell and that will generate the expected output structure. Instances of *ShellCommandResponseGenerator* have therefore four tasks:

1. Create a final command list that is used to execute the user provided command. This could, for example, execute the command, print an end marker, and print the return code of the command.
2. Parse the output of the command, yield it to the user.
3. Read the return code and provide it to the user.
4. Provide stderr-output to the user.

A very versatile example of a response generator is the class *VariableLengthResponseGenerator*. It can be used to execute a command that will result in an output of unknown length, e.g. `ls`, and will yield the output of the command to the user. It does that by using a random *end marker* to detect the end of the output and read the trailing return code. This is suitable for almost all commands.

If *VariableLengthResponseGenerator* is so versatile, why not just implement its functionality in *ShellCommandExecutor*? There are two major reasons for that:

1. Although the *VariableLengthResponseGenerator* is very versatile, it is not the most efficient implementation for commands that produce large amounts of output. In addition, there is also a minimal risk that the end marker is part of the output of the command, which would trip up the response generator. Putting response generation into a separate class allows to implement specific operations more efficiently and more safely. For example, *DownloadResponseGenerator* implements the download of files. It takes a remote file name as user "command" and creates a final command list that emits the length of the file, a newline, the file content, a return code, and a newline. This allows *DownloadResponseGenerator* to parse the output without relying on an end marker, thus increasing efficiency and safety
2. Factoring out the response generation creates an interface that can be used to support the syntax of different shells and the difference in command names and options in different operating systems. For example, the response generator class *VariableLengthResponseGeneratorPowerShell* supports the invocation of commands with variable length output in a `PowerShell`.

In short, parser generator classes encapsulate details of shell-syntax and operation implementation. That allows support of different shell syntax, and the efficient implementation of specific higher level operations, e.g. `download`. It also allows users to extend the functionality of *ShellCommandExecutor* by providing their own response generator classes.

The module `datalad_next.shell.response_generators` provides two generally applicable abstract response generator classes:

- *VariableLengthResponseGenerator*
- *FixedLengthResponseGenerator*

The functionality of the former is described above. The latter can be used to execute a command that will result in output of known length, e.g. `echo -n 012345`. It reads the specified number of bytes and a trailing return code. This is more performant than the variable length response generator (because it does not have to search for the end marker). In addition, it does not rely on the uniqueness of the end marker. It is most useful for operation like `download`, where the length of the output can be known in advance.

As mentioned above, the classes *VariableLengthResponseGenerator* and *FixedLengthResponseGenerator* are abstract. The module `datalad_next.shell.response_generators` provides the following concrete implementations for them:

- *VariableLengthResponseGeneratorPosix*
- *VariableLengthResponseGeneratorPowerShell*

- *FixedLengthResponseGeneratorPosix*
- *FixedLengthResponseGeneratorPowerShell*

When *shell()* is executed it will use a `VariableLengthResponseClass` to skip the login message of the shell. This is done by executing a *zero command* (a command that will possibly generate some output, and successfully return) in the shell. The zero command is provided by the concrete implementation of class *VariableLengthResponseGenerator*. For example, the zero command for POSIX shells is `test 0 -eq 0`, for PowerShell it is `Write-Host hello`.

Because there is no way for func:*shell* to determine the kind of shell it connects to, the user can provide an alternative response generator class, in the `zero_command_rg_class`-parameter. Instance of that class will then be used to execute the zero command. Currently, the following two response generator classes are available:

- *VariableLengthResponseGeneratorPosix*: works with POSIX-compliant shells, e.g. `sh` or `bash`. This is the default.
- *VariableLengthResponseGeneratorPowerShell*: works with PowerShell.

Whenever a command is executed via *ShellCommandExecutor.__call__()*, the class identified by `zero_command_rg_class` will be used by default to create the final command list and to parse the result. Users can override this on a per-call basis by providing a different response generator class in the `response_generator`-parameter of *ShellCommandExecutor.__call__()*.

ShellCommandExecutor(process_inputs, stdout, ...)	Execute a command in a shell and return a generator that yields output
ShellCommandResponseGenerator(stdout_gen, ...)	An abstract class the specifies the minimal functionality of a response generator
VariableLengthResponseGenerator(stdout)	Response generator that handles outputs of unknown length
VariableLengthResponseGeneratorPosix(stdout)	A variable length response generator for POSIX shells
VariableLengthResponseGeneratorPowerShell(std	A variable length response generator for PowerShell shells
FixedLengthResponseGenerator(stdout, length)	Response generator for efficient handling of outputs of known length
FixedLengthResponseGeneratorPosix(stdout, length)	
FixedLengthResponseGeneratorPowerShell(...)	
DownloadResponseGenerator(stdout)	Response generator interface for efficient download
DownloadResponseGeneratorPosix(stdout)	A response generator for efficient download commands from Linux systems
operations.posix.upload(shell, local_path, ...)	Upload a local file to a named file in the connected shell
operations.posix.download(shell, ...[, ...])	Download a file from the connected shell
operations.posix.delete(shell, files, *[, ...])	Delete files on the connected shell

datalad_next.shell.ShellCommandExecutor

class datalad_next.shell.**ShellCommandExecutor**(*process_inputs: Queue*, *stdout: OutputFrom*, *shell_cmd: list[str]*, *default_rg_class: type[*VariableLengthResponseGenerator*]*)

Bases: `object`

Execute a command in a shell and return a generator that yields output

Instances of *ShellCommandExecutor* allow to execute commands that are provided as byte-strings via its *__call__()*-method.

To execute the command and collect its output, return code, and stderr-output, *ShellCommandExecutor* uses instances of subclasses of *ShellCommandResponseGenerator*, e.g. *VariableLengthResponseGeneratorPosix*.

__call__(*command: bytes | str*, ***, *stdin: Iterable[bytes] | None = None*, *response_generator:* ShellCommandResponseGenerator *| None = None*, *encoding: str = 'utf-8'*, *check: bool = False*) → ExecutionResult

Execute a command in the connected shell and return the result

This method executes the given command in the connected shell. It assembles all output on stdout, all output on stderr that was written during the execution of the command, and the return code of the command. (The response generator defines when the command output is considered complete. Usually that is done by checking for a random end-of-output marker.)

Parameters

- **command** (*bytes | str*) -- The command to execute. If the command is given as a string, it will be encoded to bytes using the encoding given in *encoding*.
- **stdin** (*Iterable[byte] | None, optional, default: None*) -- If given, the bytes are sent to stdin of the command.

  Note: If the command reads its `stdin` until EOF, you have to use `self.close()` to close `stdin` of the command. Otherwise, the command will usually not terminate. Once `self.close()` is called, no more commands can be executed with this *ShellCommandExecutor*-instance. If you want to execute further commands in the same *ShellCommandExecutor*-instance, you must ensure that commands consume a fixed amount of input, for example, by using *head -c <byte-count> | <command>*.
- **response_generator** (ShellCommandResponseGenerator *| None, optional, default: None*) -- If given, the responder generator (usually an instance of a subclass of `ShellCommandResponseGenerator`), that is used to generate the command line and to parse the output of the command. This can be used to implement, for example, fixed length output processing.
- **encoding** (*str, optional, default: 'utf-8'*) -- The encoding that is used to encode the command if it is given as a string. Note: the encoding should match the decoding the is used in the connected shell.
- **check** (*bool, optional, default: False*) -- If True, a `CommandError`-exception is raised if the return code of the command is not zero.

Returns

An instance of `ExecutionResult` that contains the `stdout`-output, the `stderr`-output, and the return code of the command.

Return type

`ExecutionResult`

Raises
CommandError -- If the return code of the command is not zero and *check* is True.

`close()`

stop input to the shell

This method closes stdin of the shell. This will in turn terminate the shell, no further commands can be executed in the shell.

`command_zero`(*response_generator:* VariableLengthResponseGenerator) → None

Execute the zero command

This method is only used by *shell()* to skip any login messages

`start`(*command: bytes | str*, *, *stdin: Iterable[bytes] | None = None*, *response_generator:* ShellCommandResponseGenerator | *None = None*, *encoding: str = 'utf-8'*) → *ShellCommandResponseGenerator*

Execute a command in the connected shell

Execute a command in the connected shell and return a generator that provides the content written to stdout of the command. After the generator is exhausted, the return code of the command is available in the `returncode`-attribute of the generator.

Parameters

- **command** (*bytes | str*) -- The command to execute. If the command is given as a string, it will be encoded to bytes using the encoding given in *encoding*.
- **stdin** (*Iterable[byte] | None, optional, default: None*) -- If given, the bytes are sent to stdin of the command.

  Note: If the command reads its `stdin` until EOF, you have to use `self.close()` to close `stdin` of the command. Otherwise, the command will usually not terminate. Once `self.close()` is called, no more commands can be executed with this *ShellCommandExecutor*-instance. If you want to execute further commands in the same *ShellCommandExecutor*-instance, you must ensure that commands consume a fixed amount of input, for example, by using *head -c <byte-count> | <command>*.
- **response_generator** (ShellCommandResponseGenerator | *None, optional, default: None*) -- If given, the responder generator (usually an instance of a subclass of `ShellCommandResponseGenerator`), that is used to generate the command line and to parse the output of the command. This can be used to implement, for example, fixed length output processing.
- **encoding** (*str, optional, default: 'utf-8'*) -- The encoding that is used to encode the command if it is given as a string. Note: the encoding should match the decoding the is used in the connected shell.

Returns
A generator that yields the output of `stdout` of the command. The generator is exhausted when all output is read. After that, the return code of the command execution is available in the `returncode`-attribute of the generator, and the stderr-output is available in the `stderr_deque`-attribute of the response generator. If a response generator was passed in via the `response_generator`-parameter, the same instance will be returned.

Return type
ShellCommandResponseGenerator

datalad_next.shell.ShellCommandResponseGenerator

class datalad_next.shell.**ShellCommandResponseGenerator**(*stdout_gen: Generator*, *stderr_deque: deque*)

Bases: Generator

An abstract class the specifies the minimal functionality of a response generator

Subclasses of this class can be used to implement operation-specific, shell-specific or OS-specific details of the command execution and the command output parsing.

The return code is available in the returncode-attribute, the stderr-output is available in the stderr_deque-attribute (a deque-instance), of instances of this class.

abstract get_final_command(*command: bytes*) → bytes

Return a final command list that executes command

This method should return a "final" command-pipeline that executes command and generates the output structure that the response generator expects. This structure will typically be parsed in the implementation of *send()*.

This method is usually only called by *ShellCommandExecutor.__call__()*.

abstract send(_) → bytes

Deliver the next part of generated output

Whenever the response generator is iterated over, this method is called and should deliver the next part of the command output or raise StopIteration if the command has finished.

throw(*typ*, *val=Ellipsis*, *tb=Ellipsis*)

Raise an exception in the generator. Return next yielded value or raise StopIteration.

datalad_next.shell.VariableLengthResponseGenerator

class datalad_next.shell.**VariableLengthResponseGenerator**(*stdout: OutputFrom*)

Bases: *ShellCommandResponseGenerator*

Response generator that handles outputs of unknown length

This response generator is used to execute a command that will result in an output of unknown length, e.g. ls. The final command list it creates will execute the command and print a random end-marker and the return code after the output of the command. The *send()*-method of this class uses the end-marker to determine then end of the command output.

send(_) → bytes

Deliver the next part of generated output

Whenever the response generator is iterated over, this method is called and should deliver the next part of the command output or raise StopIteration if the command has finished.

abstract property zero_command: bytes

Return a command that functions as "zero command"

datalad_next.shell.VariableLengthResponseGeneratorPosix

class datalad_next.shell.**VariableLengthResponseGeneratorPosix**(*stdout*)

Bases: *VariableLengthResponseGenerator*

A variable length response generator for POSIX shells

get_final_command(*command: bytes*) → bytes

Return a command list that executes `command` and prints the end-marker

The POSIX version for variable length response generators.

This method is usually only called by *ShellCommandExecutor.__call__()*.

property zero_command: bytes

Return a command that functions as "zero command"

datalad_next.shell.VariableLengthResponseGeneratorPowerShell

class datalad_next.shell.**VariableLengthResponseGeneratorPowerShell**(*stdout*)

Bases: *VariableLengthResponseGenerator*

A variable length response generator for PowerShell shells

get_final_command(*command: bytes*) → bytes

Return a command list that executes `command` and prints the end-marker

The PowerShell version for variable length response generators.

This method is usually only called by *ShellCommandExecutor.__call__()*.

property zero_command: bytes

Return a command that functions as "zero command"

datalad_next.shell.FixedLengthResponseGenerator

class datalad_next.shell.**FixedLengthResponseGenerator**(*stdout: OutputFrom*, *length: int*)

Bases: *ShellCommandResponseGenerator*

Response generator for efficient handling of outputs of known length

This response generator is used to execute commands that have an output of known length. The final command list it creates will execute the command and print the return code followed by a newline.

The *send()*-method of this response generator will read the specified number of bytes and a trailing return code. This is more performant than scanning the output for an end-marker.

send(*_*) → bytes

Deliver the next part of generated output

Whenever the response generator is iterated over, this method is called and should deliver the next part of the command output or raise `StopIteration` if the command has finished.

datalad_next.shell.FixedLengthResponseGeneratorPosix

class datalad_next.shell.**FixedLengthResponseGeneratorPosix**(*stdout: OutputFrom*, *length: int*)

Bases: *FixedLengthResponseGenerator*

get_final_command(*command: bytes*) → bytes

Return a final command list for a command with a fixed length output

The POSIX version for fixed length response generators.

This method is usually only called by *ShellCommandExecutor.__call__()*.

datalad_next.shell.FixedLengthResponseGeneratorPowerShell

class datalad_next.shell.**FixedLengthResponseGeneratorPowerShell**(*stdout: OutputFrom*, *length: int*)

Bases: *FixedLengthResponseGenerator*

get_final_command(*command: bytes*) → bytes

Return a final command list for a command with a fixed length output

The PowerShell version for fixed length response generators.

This method is usually only called by *ShellCommandExecutor.__call__()*.

datalad_next.shell.DownloadResponseGenerator

class datalad_next.shell.**DownloadResponseGenerator**(*stdout: OutputFrom*)

Bases: *ShellCommandResponseGenerator*

Response generator interface for efficient download

This response generator is used to implement download in a single command call (instead of using one command to determine the length of a file and a subsequent fixed-length command to download the file). It assumes that the shell sends `<length>\n`, the content of the file, and `<return code>\n`. The response generator delegates the creation of the appropriate final command list to its subclasses.

send(*_*) → bytes

Deliver the next part of generated output

Whenever the response generator is iterated over, this method is called and should deliver the next part of the command output or raise `StopIteration` if the command has finished.

datalad_next.shell.DownloadResponseGeneratorPosix

class datalad_next.shell.**DownloadResponseGeneratorPosix**(*stdout: OutputFrom*)

Bases: *DownloadResponseGenerator*

A response generator for efficient download commands from Linux systems

get_final_command(*remote_file_name: bytes*) → bytes

Return a final command list for the download of `remote_file_name`

The POSIX version for download response generators.

This method is usually only called by *ShellCommandExecutor.__call__()*.

datalad_next.shell.operations.posix.upload

datalad_next.shell.operations.posix.**upload**(*shell: ShellCommandExecutor*, *local_path: Path*, *remote_path: PurePosixPath*, *progress_callback: Callable[[int, int], None] | None = None*, *check: bool = False*) → ExecutionResult

Upload a local file to a named file in the connected shell

This function uploads a file to the connected shell `shell`. It uses `head` to limit the number of bytes that the remote shell will read. This ensures that the upload is terminated.

The requirements for upload are: - The connected shell must be a POSIX shell. - `head` must be installed in the remote shell.

Parameters

- **shell** (*ShellCommandExecutor*) -- The shell that should be used to upload the file.
- **local_path** (*Path*) -- The file that should be uploaded.
- **remote_path** (*PurePosixPath*) -- The name of the file on the connected shell that will contain the uploaded content.
- **progress_callback** (*callable[[int, int], None], optional, default: None*) -- If given, the callback is called with the number of bytes that have been sent and the total number of bytes that should be sent.
- **check** (*bool, optional, default: False*) -- If `True`, raise a `CommandError` if the remote operation does not exit with a `0` as return code.

Returns
The result of the upload operation.

Return type
ExecutionResult

Raises
CommandError: -- If the remote operation does not exit with a `0` as return code, and `check` is `True`, a `CommandError` is raised. It will contain the exit code and the last (up to `chunk_size` (defined by the `chunk_size` keyword argument to `shell()`)) bytes of stderr output.

datalad_next.shell.operations.posix.download

datalad_next.shell.operations.posix.**download**(*shell: ShellCommandExecutor, remote_path: PurePosixPath, local_path: Path, progress_callback: Callable[[int, int], None] | None = None, *, response_generator_class: type[DownloadResponseGenerator] = <class 'datalad_next.shell.operations.posix.DownloadResponseGeneratorPosix'>, check: bool = False*) → ExecutionResult

Download a file from the connected shell

This method downloads a file from the connected shell.

The requirements for download via instances of class *DownloadResponseGeneratorPosix* are: - The connected shell must support *ls -dln*. - The connected shell must support *echo -e*. - The connected shell must support *awk*. - The connected shell must support *cat*.

Parameters

- **shell** (ShellCommandExecutor) -- The shell from which a file should be downloaded.
- **remote_path** (*PurePosixPath*) -- The name of the file on the connected shell that should be downloaded.
- **local_path** (*Path*) -- The name of the local file that will contain the downloaded content.
- **progress_callback** (*callable[[int, int], None], optional, default: None*) -- If given, the callback is called with the number of bytes that have been received and the total number of bytes that should be received.
- **response_generator_class** (*type[DownloadResponseGenerator], optional, default: DownloadResponseGeneratorPosix*) -- The response generator that should be used to handle the download output. It must be a subclass of DownloadResponseGenerator. The default works if the connected shell runs on a Unix-like system that provides *ls -dln* and *awk*, e.g. Linux or OSX.
- **check** (*bool, optional, default: False*) -- If True, raise a CommandError if the remote operation does not exit with a 0 as return code.

Returns
The result of the download operation.

Return type
ExecutionResult

Raises
CommandError: -- If the remote operation does not exit with a 0 as return code, and check is True, a CommandError is raised. It will contain the exit code and the last (up to chunk_size (defined by the chunk_size keyword argument to shell())) bytes of stderr output.

datalad_next.shell.operations.posix.delete

datalad_next.shell.operations.posix.**delete**(*shell: ShellCommandExecutor*, *files: list[PurePosixPath]*, *, *force: bool = False*, *check: bool = False*) → ExecutionResult

Delete files on the connected shell

The requirements for delete are: - The connected shell must be a POSIX shell. - rm must be installed in the remote shell.

Parameters

- **shell** (ShellCommandExecutor) -- The shell from which a file should be downloaded.
- **files** (*list[PurePosixPath]*) -- The "paths" of the files that should be deleted.
- **force** (*bool*) -- If True, enforce removal, if possible. For example, the command could change the permissions of the files to be deleted to ensure their removal.
- **check** (*bool, optional, default: False*) -- If True, raise a CommandError if the remote operation does not exit with a 0 as return code.

Raises
CommandError: -- If the remote operation does not exit with a 0 as return code, and check is True, a CommandError is raised. It will contain the exit code and the last (up to chunk_size (defined by the chunk_size keyword argument to shell())) bytes of stderr output.

datalad_next.shell.**shell**(*shell_cmd: list[str], *, credential: str | None = None, chunk_size: int = 65536, zero_command_rg_class: type[VariableLengthResponseGenerator] = <class 'datalad_next.shell.response_generators.VariableLengthResponseGeneratorPosix'>*) → Generator[*ShellCommandExecutor*, None, None]

Context manager that provides an interactive connection to a shell

This context manager uses the provided argument `shell_cmd` to start a shell-subprocess. Usually the commands provided in `shell_cmd` will start a client for a remote shell, e.g. `ssh`.

shell() returns an instance of *ShellCommandExecutor* in the `as`-variable. This instance can be used to interact with the shell. That means, it can be used to execute commands in the shell, receive the data that the commands write to their `stdout` and `stderr`, and retrieve the return code of the executed commands. All commands that are executed via the returned instance of *ShellCommandExecutor* are executed in the same shell instance.

Simple example that invokes a single command:

```
>>> from datalad_next.shell import shell
>>> with shell(['ssh', 'localhost']) as ssh:
...     result = ssh(b'ls -l /etc/passwd')
...     print(result.stdout)
...     print(result.returncode)
...
b'-rw-r--r-- 1 root root 2773 Nov 14 10:05 /etc/passwd\n'
0
```

Example that invokes two commands, the second of which exits with a non-zero return code. The error output is retrieved from `result.stderr`, which contains all `stderr` data that was written since the last command was executed:

```
>>> from datalad_next.shell import shell
>>> with shell(['ssh', 'localhost']) as ssh:
...     print(ssh(b'head -1 /etc/passwd').stdout)
...     result = ssh(b'ls /no-such-file')
...     print(result.stdout)
...     print(result.returncode)
...     print(result.stderr)
...
b'root:x:0:0:root:/root:/bin/bash\n'
b''
2
b"Pseudo-terminal will not be allocated because stdin is not a terminal.\r\nls:␣
↪cannot access '/no-such-file': No such file or directory\n"
```

The following example demonstrates how to use the `check`-parameter to raise a `CommandError`-exception if the return code of the command is not zero. This delegates error handling to the calling code and help to keep the code clean:

```
>>> from datalad_next.shell import shell
>>> with shell(['ssh', 'localhost']) as ssh:
...     print(ssh(b'ls /no-such-file', check=True).stdout)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "/home/cristian/Develop/datalad-next/datalad_next/shell/shell.py", line 279,␣
↪in __call__
    return create_result(
  File "/home/cristian/Develop/datalad-next/datalad_next/shell/shell.py", line 349,␣
```

(continues on next page)

(continued from previous page)

```
↪in create_result
    result.to_exception(command, error_message)
  File "/home/cristian/Develop/datalad-next/datalad_next/shell/shell.py", line 52,␣
↪in to_exception
    raise CommandError(
datalad.runner.exception.CommandError: CommandError: 'ls /no-such-file' failed with␣
↪exitcode 2 [err: 'cannot access '/no-such-file': No such file or directory']
```

Manual checking of the return code:

```
>>> from datalad_next.shell import shell
>>> def file_exists(file_name):
...     with shell(['ssh', 'localhost']) as ssh:
...         result = ssh(f'ls {file_name}')
...         return result.returncode == 0
... print(file_exists('/etc/passwd'))
True
... print(file_exists('/no-such-file'))
False
```

An example for result content checking:

```
>>> from datalad_next.shell import shell
>>> with shell(['ssh', 'localhost']) as ssh:
...     result = ssh(f'grep root /etc/passwd', check=True).stdout
...     if len(result.splitlines()) != 1:
...         raise ValueError('Expected exactly one line')
```

For long running commands a generator-based result fetching can be used. To use generator-based output the command has to be executed with the method *ShellCommandExecutor.start()*. This method returns a generator that provides command output as soon as it is available:

```
>>> import time
>>> from datalad_next.shell import shell
>>> with shell(['ssh', 'localhost']) as ssh:
...     result_generator = ssh.start(b'c=0; while [ $c -lt 6 ]; do head -2 /etc/
↪passwd; sleep 2; c=$(( $c + 1 )); done')
...     for result in result_generator:
...         print(time.time(), result)
...     assert result_generator.returncode == 0
1713358098.82588 b'root:x:0:0:root:/root:/bin/bash\nsystemd-
↪timesync:x:497:497:systemd Time Synchronization:/:/usr/sbin/nologin\n'
1713358100.8315682 b'root:x:0:0:root:/root:/bin/bash\nsystemd-
↪timesync:x:497:497:systemd Time Synchronization:/:/usr/sbin/nologin\n'
1713358102.8402972 b'root:x:0:0:root:/root:/bin/bash\nsystemd-
↪timesync:x:497:497:systemd Time Synchronization:/:/usr/sbin/nologin\n'
1713358104.8490314 b'root:x:0:0:root:/root:/bin/bash\nsystemd-
↪timesync:x:497:497:systemd Time Synchronization:/:/usr/sbin/nologin\n'
1713358106.8577306 b'root:x:0:0:root:/root:/bin/bash\nsystemd-
↪timesync:x:497:497:systemd Time Synchronization:/:/usr/sbin/nologin\n'
1713358108.866439 b'root:x:0:0:root:/root:/bin/bash\nsystemd-
↪timesync:x:497:497:systemd Time Synchronization:/:/usr/sbin/nologin\n'
```

(The exact output of the above example might differ, depending on the length of the first two entries in the `/etc/passwd`-file.)

Parameters

- **shell_cmd** (*list[str]*) -- The command to execute the shell. It should be a list of strings that is given to `iter_subproc()` as *args*-parameter. For example: `['ssh', '-p', '2222', 'localhost']`.
- **chunk_size** (*int, optional*) -- The size of the chunks that are read from the shell's `stdout` and `stderr`. This also defines the size of stored `stderr`-content.
- **zero_command_rg_class** (*type[VariableLengthResponseGenerator], optional, default: 'VariableLengthResponseGeneratorPosix'*) -- Shell uses an instance of the specified response generator class to execute the *zero command* ("zero command" is the command used to skip the login messages of the shell). This class will also be used as the default response generator for all further commands executed in the *ShellCommandExecutor*-instances that is returned by *shell()*. Currently, the following concrete subclasses of *VariableLengthResponseGenerator* exist:
  - *VariableLengthResponseGeneratorPosix*: compatible with POSIX-compliant shells, e.g. `sh` or `bash`.
  - *VariableLengthResponseGeneratorPowerShell*: compatible with PowerShell.

Yields

ShellCommandExecutor

2.3.15 datalad_next.tests

Tooling for test implementations

BasicGitTestRepo([path, puke_if_exists])	Creates a basic test git repository.
DEFAULT_BRANCH	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
DEFAULT_REMOTE	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
assert_in(first, second[, msg])	
assert_in_results(results, **kwargs)	Verify that the particular combination of keys and values is found in one of the results
assert_result_count(results, n, **kwargs)	Verify specific number of results (matching criteria, if any)
assert_status(label, results)	Verify that each status dict in the results has a given status label
create_tree(path, tree[, ...])	Given a list of tuples (name, load) create such a tree
*eq_*(first, second[, msg])	
get_deeply_nested_structure(path)	Here is what this does (assuming UNIX, locked): | .
*ok_*(expr[, msg])	
ok_good_symlink(path)	
ok_broken_symlink(path)	
run_main(args[, exit_code, expect_stderr])	Run main() of the datalad, do basic checks and provide outputs
skip_if_on_windows([func])	Skip test completely under Windows
skip_if_root([func])	Skip test if uid == 0.
skip_wo_symlink_capability(func)	Skip test when environment does not support symlinks
swallow_logs([new_level, file_, name])	Context manager to consume all logs.
skipif_no_network	A decorator for applying a mark on test functions and classes.

datalad_next.tests.BasicGitTestRepo

class datalad_next.tests.**BasicGitTestRepo**(*path=None*, *puke_if_exists=True*)

Bases: TestRepo

Creates a basic test git repository.

REPO_CLASS

alias of GitRepo

create_info_file()

populate()

datalad_next.tests.DEFAULT_BRANCH

datalad_next.tests.**DEFAULT_BRANCH = 'master'**

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

datalad_next.tests.DEFAULT_REMOTE

datalad_next.tests.**DEFAULT_REMOTE = 'origin'**

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

datalad_next.tests.assert_in

datalad_next.tests.**assert_in**(*first*, *second*, *msg=None*)

datalad_next.tests.assert_in_results

datalad_next.tests.**assert_in_results**(*results*, ***kwargs*)

Verify that the particular combination of keys and values is found in one of the results

datalad_next.tests.assert_result_count

datalad_next.tests.**assert_result_count**(*results*, *n*, ***kwargs*)

Verify specific number of results (matching criteria, if any)

datalad_next.tests.assert_status

datalad_next.tests.**assert_status**(*label*, *results*)

Verify that each status dict in the results has a given status label

label can be a sequence, in which case status must be one of the items in this sequence.

datalad_next.tests.create_tree

datalad_next.tests.**create_tree**(*path: str*, *tree: Tuple[Tuple[str | File, str | bytes | TreeSpec], ...] | List[Tuple[str | File, str | bytes | TreeSpec]] | Dict[str | File, str | bytes | TreeSpec]*, *archives_leading_dir: bool = True*, *remove_existing: bool = False*) → None

Given a list of tuples (name, load) create such a tree

if load is a tuple itself -- that would create either a subtree or an archive with that content and place it into the tree if name ends with .tar.gz

datalad_next.tests.eq_

datalad_next.tests.**eq_**(*first*, *second*, *msg=None*)

datalad_next.tests.get_deeply_nested_structure

datalad_next.tests.**get_deeply_nested_structure**(*path*)

Here is what this does (assuming UNIX, locked): | . | ├── directory_untracked | │ └── link2dir -> ../subdir | ├── OBSCURE_FILENAME_file_modified | ├── link2dir -> subdir | ├── link2subdsdir -> subds_modified/subdir | ├── link2subdsroot -> subds_modified | ├── subdir | │ ├── annexed_file.txt -> ../.git/annex/objects/... | │ ├── file_modified | │ ├── git_file.txt | │ └── link2annex_files.txt -> annexed_file.txt | └── subds_modified | ├── link2superdsdir -> ../subdir | ├── subdir | │ └── annexed_file.txt -> ../.git/annex/objects/... | └── subds_lvl1_modified | └── OBSCURE_FILENAME_directory_untracked | └── untracked_file

When a system has no symlink support, the link2... components are not included.

datalad_next.tests.ok_

datalad_next.tests.**ok_**(*expr*, *msg=None*)

datalad_next.tests.ok_good_symlink

datalad_next.tests.**ok_good_symlink**(*path*)

datalad_next.tests.ok_broken_symlink

datalad_next.tests.**ok_broken_symlink**(*path*)

datalad_next.tests.run_main

datalad_next.tests.**run_main**(*args*, *exit_code=0*, *expect_stderr=False*)

Run main() of the datalad, do basic checks and provide outputs

Parameters

- **args** (*list*) -- List of string cmdline arguments to pass
- **exit_code** (*int*) -- Expected exit code. Would raise AssertionError if differs
- **expect_stderr** (*bool or string*) -- Whether to expect stderr output. If string -- match

Returns
Output produced

Return type
stdout, stderr strings

datalad_next.tests.skip_if_on_windows

datalad_next.tests.**skip_if_on_windows**(*func=None*)

Skip test completely under Windows

datalad_next.tests.skip_if_root

datalad_next.tests.**skip_if_root**(*func=None*)

Skip test if uid == 0.

Note that on Windows (or anywhere else *os.geteuid* is not available) the test is _not_ skipped.

datalad_next.tests.skip_wo_symlink_capability

datalad_next.tests.**skip_wo_symlink_capability**(*func*)

Skip test when environment does not support symlinks

Perform a behavioral test instead of top-down logic, as on windows this could be on or off on a case-by-case basis.

datalad_next.tests.swallow_logs

datalad_next.tests.**swallow_logs**(*new_level: str | int | None = None*, *file_: str | Path | None = None*, *name: str = 'datalad'*) → Iterator[SwallowLogsAdapter]

Context manager to consume all logs.

datalad_next.tests.skipif_no_network

datalad_next.tests.**skipif_no_network = MarkDecorator(mark=Mark(name='skipif', args=(False,), kwargs={'reason': 'DATALAD_TESTS_NONETWORK is set'}))**

A decorator for applying a mark on test functions and classes.

`MarkDecorators` are created with `pytest.mark`:

```
mark1 = pytest.mark.NAME  # Simple MarkDecorator
mark2 = pytest.mark.NAME(name1=value)  # Parametrized MarkDecorator
```

and can then be applied as decorators to test functions:

```
@mark2
def test_function():
    pass
```

When a `MarkDecorator` is called, it does the following:

1. If called with a single class as its only positional argument and no additional keyword arguments, it attaches the mark to the class so it gets applied automatically to all test cases found in that class.
2. If called with a single function as its only positional argument and no additional keyword arguments, it attaches the mark to the function, containing all the arguments already stored internally in the `MarkDecorator`.
3. When called in any other case, it returns a new `MarkDecorator` instance with the original `MarkDecorator`'s content updated with the arguments passed to this call.

Note: The rules above prevent a `MarkDecorator` from storing only a single function or class reference as its positional argument with no additional keyword or positional arguments. You can work around this by using *with_args()*.

2.3.16 datalad_next.tests.fixtures

Collection of fixtures for facilitation test implementations

`datalad_next.tests.fixtures.`**check_gitconfig_global**`()`

No test must modify a user's global Git config.

If such modifications are needed, a custom configuration setup limited to the scope of the test requiring it must be arranged.

`datalad_next.tests.fixtures.`**check_plaintext_keyring**`()`

No test must modify a user's keyring.

If such modifications are needed, a custom keyring setup limited to the scope of the test requiring it must be arranged. The `tmp_keyring` fixture can be employed in such cases.

`datalad_next.tests.fixtures.`**credman**`(`*datalad_cfg*, *tmp_keyring*`)`

Provides a temporary credential manager

It comes with a temporary global datalad config and a temporary keyring as well.

This manager can be used to deploy or manipulate credentials within the scope of a single test.

`datalad_next.tests.fixtures.`**datalad_cfg**`()`

Temporarily alter configuration to use a plain "global" configuration

The global configuration manager at *datalad.cfg* is reloaded after adjusting *GIT_CONFIG_GLOBAL* to point to a new temporary *.gitconfig* file.

After test execution the file is removed, and the global *ConfigManager* is reloaded once more.

Any test using this fixture will be skipped for Git versions earlier than 2.32, because the *GIT_CONFIG_GLOBAL* environment variable used here was only introduced with that version.

`datalad_next.tests.fixtures.`**datalad_interactive_ui**`(`*monkeypatch*`)`

Yields a UI replacement to query for operations and stage responses

No output will be written to STDOUT/ERR by this UI.

A standard usage pattern is to stage one or more responses, run the to-be-tested code, and verify that the desired user interaction took place:

```
> datalad_interactive_ui.staged_responses.append('skip')
> ...
> assert ... datalad_interactive_ui.log
```

datalad_next.tests.fixtures.**datalad_noninteractive_ui**(*monkeypatch*)

Yields a UI replacement to query for operations

No output will be written to STDOUT/ERR by this UI.

A standard usage pattern is to run the to-be-tested code, and verify that the desired user messaging took place:

```
> ...
> assert ... datalad_interactive_ui.log
```

datalad_next.tests.fixtures.**dataset**(*datalad_cfg*, *tmp_path_factory*)

Provides a `Dataset` instance for a not-yet-existing repository

The instance points to an existing temporary path, but `create()` has not been called on it yet.

datalad_next.tests.fixtures.**existing_dataset**(*dataset*)

Provides a `Dataset` instance pointing to an existing dataset/repo

This fixture uses an instance provided by the `dataset` fixture and calls `create()` on it, before it yields the `Dataset` instance.

datalad_next.tests.fixtures.**existing_noannex_dataset**(*dataset*)

just like `existing_dataset`, but created with `annex=False`

datalad_next.tests.fixtures.**http_credential**()

Provides the HTTP Basic authentication credential necessary to access the HTTP server provided by the `http_server_with_basicauth` fixture.

datalad_next.tests.fixtures.**http_server**(*tmp_path_factory*)

Provides an HTTP server, serving a temporary directory

The fixtures yields an instance of `HTTPPath`, providing the following essential attributes:

- `path`: `Path` instance of the served temporary directory
- `url`: HTTP URL to access the HTTP server

datalad_next.tests.fixtures.**http_server_with_basicauth**(*tmp_path_factory*, *http_credential*)

Like `http_server` but requiring authentication via `http_credential`

datalad_next.tests.fixtures.**httpbin**(*httpbin_service*)

Does the same thing as `httpbin_service`, but skips on function-scope

`httpbin_service` always returns access URLs for HTTPBIN. However, in some cases it is simply not desirable to run a test. For example, the appveyor workers are more or less constantly unable to access the public service. This fixture is evaluated at function-scope and skips the test whenever any of these undesired conditions is detected. Otherwise it just relays `httpbin_service`.

datalad_next.tests.fixtures.**httpbin_service**()

Return canonical access URLs for the HTTPBIN service

This fixture tries to spin up a httpbin Docker container at localhost:8765; if successful, it returns this URL as the 'standard' URL. If the attempt fails, a URL pointing to the canonical instance is returned.

For tests that need to have the service served via a specific protocol (https vs http), the corresponding URLs are returned too. They always point to the canonical deployment, as some tests require both protocols simultaneously and a local deployment generally won't have https.

datalad_next.tests.fixtures.**modified_dataset**(*tmp_path_factory*)

Produces a dataset with various modifications

The fixture is module-scope, aiming to be reused by many tests focused on reporting. It does not support any further modification. The fixture will fail, if any such modification is detected.

`git status` will report:

```
 git status -uall
On branch dl-test-branch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   dir_m/file_a
        new file:   file_a

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
  (commit or discard the untracked or modified content in submodules)
        deleted:    dir_d/file_d
        deleted:    dir_m/file_d
        modified:   dir_m/file_m
        deleted:    dir_sm/sm_d
        modified:   dir_sm/sm_m (modified content)
        modified:   dir_sm/sm_mu (modified content, untracked content)
        modified:   dir_sm/sm_n (new commits)
        modified:   dir_sm/sm_nm (new commits, modified content)
        modified:   dir_sm/sm_nmu (new commits, modified content, untracked content)
        modified:   dir_sm/sm_u (untracked content)
        deleted:    file_d
        modified:   file_m

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        dir_m/dir_u/file_u
        dir_m/file_u
        dir_u/file_u
        file_u
```

Suffix indicates the ought-to state (multiple possible):

a - added c - clean d - deleted n - new commits m - modified u - untracked content

Prefix indicated the item type:

file - file sm - submodule dir - directory

datalad_next.tests.fixtures.**no_result_rendering**(*monkeypatch*)

Disable datalad command result rendering for all command calls

This is achieved by forcefully supplying *result_renderer='disabled'* to any command call via a patch to internal argument normalizer `get_allargs_as_kwargs()`.

datalad_next.tests.fixtures.**reduce_logging**()

Reduce the logging output during test runs

DataLad emits a large amount of repetitive INFO log messages that only clutter the test output, and hardly ever help to identify an issue. This fixture modifies the standard logger to throw away all INFO level log messages.

With this approach, such messages are still fed to and processes by the logger (in contrast to an apriori level setting).

datalad_next.tests.fixtures.**sshserver**(*sshserver_setup*, *datalad_cfg*, *monkeypatch*)

datalad_next.tests.fixtures.**sshserver_setup**(*tmp_path_factory*)

datalad_next.tests.fixtures.**tmp_keyring**()

Patch plaintext keyring to temporarily use a different storage

No credential read or write actions will impact any existing credential store of any configured backend.

The patched backend is yielded by the fixture.

datalad_next.tests.fixtures.**webdav_credential**()

Provides HTTP Basic authentication credential necessary to access the server provided by the `webdav_server` fixture.

datalad_next.tests.fixtures.**webdav_server**(*tmp_path_factory*, *webdav_credential*)

Provides a WebDAV server, serving a temporary directory

The fixtures yields an instance of `WebDAVPath`, providing the following essential attributes:

- `path`: `Path` instance of the served temporary directory
- `url`: HTTP URL to access the WebDAV server

Server access requires HTTP Basic authentication with the credential provided by the `webdav_credential` fixture.

2.3.17 datalad_next.types

Custom types and dataclasses

AnnexKey(name, backend[, size, mtime, ...])	Representation of a git-annex key
ArchivistLocator(akey, member[, size, atype])	Representation of a `dl+archive:` archive member locator
ArchiveType(value)	Enumeration of archive types

datalad_next.types.AnnexKey

class datalad_next.types.**AnnexKey**(*name: str*, *backend: str*, *size: int | None = None*, *mtime: int | None = None*, *chunksize: int | None = None*, *chunknumber: int | None = None*)

Bases: `object`

Representation of a git-annex key

https://git-annex.branchable.com/internals/key_format/

backend: str

chunknumber: int | None = None

chunksize: int | None = None

classmethod from_str(*key: str*)

Return an `AnnexKey` instance from a key string

mtime: int | None = None

name: str

size: int | None = None

datalad_next.types.ArchivistLocator

class datalad_next.types.**ArchivistLocator**(*akey:* AnnexKey, *member:* PurePosixPath, *size:* int | None = None, *atype:* ArchiveType | None = None)

Bases: object

Representation of a dl+archive: archive member locator

These locators are used by the datalad-archives and archivist git-annex special remotes. They identify a member of a archive that is itself identified by an annex key.

Each member is annotated with its size (in bytes). Optionally, the file format type of the archive can be annotated too.

Syntax of dl+archives: locators

The locators the following minimal form:

```
dl+archive:<archive-key>#path=<path-in-archive>
```

where <archive-key> is a regular git-annex key of an archive file, and <path-in-archive> is a POSIX-style relative path pointing to a member within the archive.

Two optional, additional attributes size and atype are recognized (only size is also understood by the datalad-archives special remote).

size declares the size of the (extracted) archive member in bytes:

```
dl+archive:<archive-key>#path=<path-in-archive>&size=<size-in-bytes>
```

atype declares the type of the containing archive using a label. Currently recognized labels are tar (a TAR archive, compressed or not), and zip (a ZIP archive). See *ArchiveType* for all recognized labels.

If no type information is given, *ArchivistLocator.from_str()* will try to determine the archive type from the archive key (via *E-type git-annex backends, such as DataLad's default MD5E).

The order in the fragment part of the URL (after #) is significant. path must come first, followed by size or atype. If both size and atype are present, size must be declared first. A complete example of a URL is:

```
dl+archive:MD5-s389--e9f624eb778e6f945771c543b6e9c7b2#path=dir/file.csv&size=234&
↪atype=tar
```

akey: *AnnexKey*

atype: *ArchiveType* | None = None

classmethod from_str(*url: str*)

Return ArchivistLocator from str form

member: PurePosixPath

size: int | None = None

datalad_next.types.ArchiveType

class datalad_next.types.**ArchiveType**(*value*)

Bases: Enum

Enumeration of archive types

Each one should have an associated ArchiveOperations handler.

tar = 'tar'

zip = 'zip'

2.3.18 datalad_next.uis

UI abstractions for user communication

ansi_colors	Definitions for ansi colors etc
ui_switcher	Poor man helper to switch between different backends at run-time.

datalad_next.uis.ansi_colors

Definitions for ansi colors etc

datalad_next.uis.ui_switcher

datalad_next.uis.**ui_switcher = <datalad.ui._UI_Switcher object>**

Poor man helper to switch between different backends at run-time.

2.3.19 datalad_next.url_operations

Handlers for operations on various URL types and protocols

Available handlers:

UrlOperations(*[, cfg])	Abstraction for operations on URLs
AnyUrlOperations([cfg])	Handler for operations on any supported URLs
FileUrlOperations(*[, cfg])	Handler for operations on *file://* URLs
HttpUrlOperations([cfg, headers])	Handler for operations on *http(s)://* URLs
SshUrlOperations(*[, cfg])	Handler for operations on `ssh://` URLs
UrlOperationsRemoteError(url[, message, ...])	
UrlOperationsResourceUnknown(url[, message, ...])	A connection request succeeded in principle, but target was not found
UrlOperationsInteractionError(url[, ...])	
UrlOperationsAuthenticationError(url[, ...])	
UrlOperationsAuthorizationError(url[, ...])	

datalad_next.url_operations.UrlOperations

class datalad_next.url_operations.**UrlOperations**(*, *cfg:* ConfigManager | *None* = *None*)

Bases: object

Abstraction for operations on URLs

Support for specific URL schemes can be implemented via sub-classes. Such classes must comply with the following conditions:

- Any configuration look-up must be performed with the *self.cfg* property, which is guaranteed to be a *ConfigManager* instance.
- When downloads are to be supported, implement the *download()* method and comply with the behavior described in its documentation.

This class provides a range of helper methods to aid computation of hashes and progress reporting.

property cfg: *ConfigManager*

delete(*url: str*, *, *credential: str | None = None*, *timeout: float | None = None*) → Dict

Delete a resource identified by a URL

Parameters

- **url** (*str*) -- Valid URL with any scheme supported by a particular implementation.
- **credential** (*str, optional*) -- The name of a dedicated credential to be used for authentication in order to perform the deletion. Particular implementations may or may not require or support authentication. They also may or may not support automatic credential lookup.
- **timeout** (*float, optional*) -- If given, specifies a timeout in seconds. If the operation is not completed within this time, it will raise a *TimeoutError*-exception. If timeout is None, the operation will never timeout.

Returns
A mapping of property names to values for the deletion.

Return type
dict

Raises

- *UrlOperationsRemoteError* -- This exception is raised on any deletion-related error on the remote side, with a summary of the underlying issues as its message. It may carry a status code (e.g. HTTP status code) as its `status_code` property. Any underlying exception must be linked via the *__cause__* property (e.g. *raise UrlOperationsRemoteError(...) from ...*).
- *UrlOperationsInteractionError* --
- *UrlOperationsAuthenticationError* --
- *UrlOperationsAuthorizationError* --
- *UrlOperationsResourceUnknown* -- Implementations that can distinguish several remote error types beyond indication a general `UrlOperationsRemoteError`: `UrlOperationsInteractionError` general issues in communicating with the remote side; `UrlOperationsAuthenticationError` for errors related to (failed) authentication at the remote; `UrlOperationsAuthorizationError` for (lack of) authorizating to access a particular resource of perform a particular operation; `UrlOperationsResourceUnknown` if the target of an operation does not exist.

- **TimeoutError** -- If *timeout* is given and the operation does not complete within the number of seconds that a specified by *timeout*.

download(*from_url: str*, *to_path: Path | None*, *, *credential: str | None = None*, *hash: list[str] | None = None*, *timeout: float | None = None*) → Dict

Download from a URL to a local file or stream to stdout

Parameters

- **from_url** (*str*) -- Valid URL with any scheme supported by a particular implementation.
- **to_path** (*Path or None*) -- A local platform-native path or *None*. If *None* the downloaded data is written to *stdout*, otherwise it is written to a file at the given path. The path is assumed to not exist. Any existing file will be overwritten.
- **credential** (*str, optional*) -- The name of a dedicated credential to be used for authentication in order to perform the download. Particular implementations may or may not require or support authentication. They also may or may not support automatic credential lookup.
- **hash** (*list(algorithm_names), optional*) -- If given, must be a list of hash algorithm names supported by the *hashlib* module. A corresponding hash will be computed simultaenous to the download (without reading the data twice), and included in the return value.
- **timeout** (*float, optional*) -- If given, specifies a timeout in seconds. If the operation is not completed within this time, it will raise a *TimeoutError*-exception. If timeout is None, the operation will never timeout.

Returns

A mapping of property names to values for the completed download. If *hash* algorithm names are provided, a corresponding key for each algorithm is included in this mapping, with the hexdigest of the corresponding checksum as the value.

Return type

dict

Raises

- *UrlOperationsRemoteError* -- This exception is raised on any deletion-related error on the remote side, with a summary of the underlying issues as its message. It may carry a status code (e.g. HTTP status code) as its `status_code` property. Any underlying exception must be linked via the *__cause__* property (e.g. *raise UrlOperationsRemoteError(...) from ...*).
- *UrlOperationsInteractionError* --
- *UrlOperationsAuthenticationError* --
- *UrlOperationsAuthorizationError* --
- *UrlOperationsResourceUnknown* -- Implementations that can distinguish several remote error types beyond indication a general `UrlOperationsRemoteError`: `UrlOperationsInteractionError` general issues in communicating with the remote side; `UrlOperationsAuthenticationError` for errors related to (failed) authentication at the remote; `UrlOperationsAuthorizationError` for (lack of) authorizating to access a particular resource of perform a particular operation; `UrlOperationsResourceUnknown` if the target of an operation does not exist.
- **TimeoutError** -- If *timeout* is given and the operation does not complete within the number of seconds that a specified by *timeout*.

stat(*url: str*, *, *credential: str | None = None*, *timeout: float | None = None*) → Dict

Gather information on a URL target, without downloading it

Returns

A mapping of property names to values of the URL target. The particular composition of properties depends on the specific URL. A standard property is 'content-length', indicating the size of a download.

Return type

dict

Raises

- *UrlOperationsRemoteError* -- This exception is raised on any access-related error on the remote side, with a summary of the underlying issues as its message. It may carry a status code (e.g. HTTP status code) as its `status_code` property. Any underlying exception must be linked via the *__cause__* property (e.g. *raise UrlOperationsRemoteError(...) from ...*).
- *UrlOperationsInteractionError* --
- *UrlOperationsAuthenticationError* --
- *UrlOperationsAuthorizationError* --
- *UrlOperationsResourceUnknown* -- Implementations that can distinguish several remote error types beyond indication a general `UrlOperationsRemoteError`: `UrlOperationsInteractionError` general issues in communicating with the remote side; `UrlOperationsAuthenticationError` for errors related to (failed) authentication at the remote; `UrlOperationsAuthorizationError` for (lack of) authorizating to access a particular resource of perform a particular operation; `UrlOperationsResourceUnknown` if the target of an operation does not exist.
- **TimeoutError** -- If *timeout* is given and the operation does not complete within the number of seconds that a specified by *timeout*.

upload(*from_path: Path | None*, *to_url: str*, *, *credential: str | None = None*, *hash: list[str] | None = None*, *timeout: float | None = None*) → Dict

Upload from a local file or stream to a URL

Parameters

- **from_path** (*Path or None*) -- A local platform-native path or *None*. If *None* the upload data is read from *stdin*, otherwise it is read from a file at the given path.
- **to_url** (*str*) -- Valid URL with any scheme supported by a particular implementation. The target is assumed to not conflict with existing content, and may be overwritten.
- **credential** (*str, optional*) -- The name of a dedicated credential to be used for authentication in order to perform the upload. Particular implementations may or may not require or support authentication. They also may or may not support automatic credential lookup.
- **hash** (*list(algorithm_names), optional*) -- If given, must be a list of hash algorithm names supported by the *hashlib* module. A corresponding hash will be computed simultaenous to the upload (without reading the data twice), and included in the return value.
- **timeout** (*float, optional*) -- If given, specifies a timeout in seconds. If the operation is not completed within this time, it will raise a *TimeoutError*-exception. If timeout is None, the operation will never timeout.

Returns
A mapping of property names to values for the completed upload. If *hash* algorithm names are provided, a corresponding key for each algorithm is included in this mapping, with the hexdigest of the corresponding checksum as the value.

Return type
dict

Raises

- **FileNotFoundError** -- If the source file cannot be found.
- *UrlOperationsRemoteError* -- This exception is raised on any deletion-related error on the remote side, with a summary of the underlying issues as its message. It may carry a status code (e.g. HTTP status code) as its `status_code` property. Any underlying exception must be linked via the *__cause__* property (e.g. *raise UrlOperationsRemoteError(...) from ...*).
- *UrlOperationsInteractionError* --
- *UrlOperationsAuthenticationError* --
- *UrlOperationsAuthorizationError* --
- *UrlOperationsResourceUnknown* -- Implementations that can distinguish several remote error types beyond indication a general `UrlOperationsRemoteError`: `UrlOperationsInteractionError` general issues in communicating with the remote side; `UrlOperationsAuthenticationError` for errors related to (failed) authentication at the remote; `UrlOperationsAuthorizationError` for (lack of) authorizating to access a particular resource of perform a particular operation; `UrlOperationsResourceUnknown` if the target of an operation does not exist.
- **TimeoutError** -- If *timeout* is given and the operation does not complete within the number of seconds that a specified by *timeout*.

datalad_next.url_operations.AnyUrlOperations

class datalad_next.url_operations.**AnyUrlOperations**(*cfg:* ConfigManager | *None* = *None*)

Bases: *UrlOperations*

Handler for operations on any supported URLs

The methods inspect a given URL and call the corresponding methods for the *UrlOperations* implementation that matches the URL best. The "best match" is the match expression of a registered URL handler that yields the longest match against the given URL.

Parameter identity and semantics are unchanged with respect to the underlying implementations. See their documentation for details.

An instance retains and reuses URL scheme handler instances for subsequent operations, such that held connections or cached credentials can be reused efficiently.

delete(*url: str*, *, *credential: str* | *None* = *None*, *timeout: float* | *None* = *None*) → Dict

Call **UrlOperations.delete()* for the respective URL scheme

download(*from_url: str*, *to_path: Path* | *None*, *, *credential: str* | *None* = *None*, *hash: list[str]* | *None* = *None*, *timeout: float* | *None* = *None*) → Dict

Call **UrlOperations.download()* for the respective URL scheme

is_supported_url(*url*) → bool

stat(*url: str*, *, *credential: str | None = None*, *timeout: float | None = None*) → Dict

Call **UrlOperations.stat()* for the respective URL scheme

upload(*from_path: Path | None*, *to_url: str*, *, *credential: str | None = None*, *hash: list[str] | None = None*, *timeout: float | None = None*) → Dict

Call **UrlOperations.upload()* for the respective URL scheme

datalad_next.url_operations.FileUrlOperations

class datalad_next.url_operations.**FileUrlOperations**(*, *cfg:* ConfigManager | *None = None*)

Bases: *UrlOperations*

Handler for operations on *file://* URLs

Access to local data via file-scheme URLs is supported with the same API and feature set as other URL-schemes (simultaneous content hashing and progress reporting.

delete(*url: str*, *, *credential: str | None = None*, *timeout: float | None = None*) → Dict

Delete the target of a file:// URL

The target can be a file or a directory. If it is a directory, it has to be empty.

See *datalad_next.url_operations.UrlOperations.delete()* for parameter documentation and exception behavior.

Raises

UrlOperationsResourceUnknown -- For deletion targets found absent.

download(*from_url: str*, *to_path: Path | None*, *, *credential: str | None = None*, *hash: list[str] | None = None*, *timeout: float | None = None*) → Dict

Copy a file:// URL target to a local path

See *datalad_next.url_operations.UrlOperations.download()* for parameter documentation and exception behavior.

Raises

UrlOperationsResourceUnknown -- For download targets found absent.

stat(*url: str*, *, *credential: str | None = None*, *timeout: float | None = None*) → Dict

Gather information on a URL target, without downloading it

See *datalad_next.url_operations.UrlOperations.stat()* for parameter documentation and exception behavior.

Raises

UrlOperationsResourceUnknown -- For access targets found absent.

upload(*from_path: Path | None*, *to_url: str*, *, *credential: str | None = None*, *hash: list[str] | None = None*, *timeout: float | None = None*) → Dict

Copy a local file to a file:// URL target

Any missing parent directories of the URL target are created as necessary.

See *datalad_next.url_operations.UrlOperations.upload()* for parameter documentation and exception behavior.

Raises

FileNotFoundError -- If the source file cannot be found.

datalad_next.url_operations.HttpUrlOperations

class datalad_next.url_operations.**HttpUrlOperations**(*cfg=None*, *headers: Dict | None = None*)

Bases: *UrlOperations*

Handler for operations on *http(s)://* URLs

This handler is built on the *requests* package. For authentication, it employes *datalad_next.utils.requests_auth.DataladAuth*, an adaptor that consults the DataLad credential system in order to fulfill HTTP authentication challenges.

download(*from_url: str*, *to_path: Path | None*, ***, *credential: str | None = None*, *hash: list[str] | None = None*, *timeout: float | None = None*) → Dict

Download via HTTP GET request

See *datalad_next.url_operations.UrlOperations.download()* for parameter documentation and exception behavior.

Raises
UrlOperationsResourceUnknown -- For download targets found absent.

get_headers(*headers: Dict | None = None*) → Dict

probe_url(*url*, *timeout=10.0*, *headers=None*)

Probe a HTTP(S) URL for redirects and authentication needs

This functions performs a HEAD request against the given URL, while waiting at most for the given timeout duration for a server response.

Parameters

- **url** (*str*) -- URL to probe
- **timeout** (*float, optional*) -- Maximum time to wait for a server response to the probe
- **headers** (*dict, optional*) -- Any custom headers to use for the probe request. If none are provided, or the provided headers contain no 'user-agent' field, the default DataLad user agent is added automatically.

Returns

The first value is the URL against the final request was performed, after following any redirects and applying normalizations.

The second value is a mapping with a particular set of properties inferred from probing the webserver. The following key-value pairs are supported:

- 'is_redirect' (bool), True if any redirection occurred. This boolean property is a more accurate test than comparing input and output URL
- 'status_code' (int), HTTP response code (of the final request in case of redirection).
- 'auth' (dict), present if the final server response contained any 'www-authenticate' headers, typically the case for 401 responses. The dict contains a mapping of server-reported authentication scheme names (e.g., 'basic', 'bearer') to their respective properties (dict). These can be any nature and number, depending on the respective authentication scheme. Most notably, they may contain a 'realm' property that can be used to determine suitable credentials for authentication.

Return type
str or None, dict

Raises

requests.RequestException -- May raise any exception of the *requests* package, most notably *ConnectionError*, *Timeout*, *TooManyRedirects*, etc.

stat(*url: str*, *, *credential: str | None = None*, *timeout: float | None = None*) → Dict

Gather information on a URL target, without downloading it

See *datalad_next.url_operations.UrlOperations.stat()* for parameter documentation and exception behavior.

Raises

UrlOperationsResourceUnknown -- For access targets found absent.

datalad_next.url_operations.SshUrlOperations

class datalad_next.url_operations.**SshUrlOperations**(*, *cfg:* ConfigManager | *None = None*)

Bases: *UrlOperations*

Handler for operations on `ssh://` URLs

For downloading files, only servers that support execution of the commands 'printf', 'ls -nl', 'awk', and 'cat' are supported. This includes a wide range of operating systems, including devices that provide these commands via the 'busybox' software.

Note: The present implementation does not support SSH connection multiplexing, (re-)authentication is performed for each request. This limitation is likely to be removed in the future, and connection multiplexing supported where possible (non-Windows platforms).

---

download(*from_url: str*, *to_path: Path | None*, *, *credential: str | None = None*, *hash: list[str] | None = None*, *timeout: float | None = None*) → Dict

Download a file by streaming it through an SSH connection.

On the server-side, the file size is determined and sent. Afterwards the file content is sent via *cat* to the SSH client.

See *datalad_next.url_operations.UrlOperations.download()* for parameter documentation and exception behavior.

stat(*url: str*, *, *credential: str | None = None*, *timeout: float | None = None*) → Dict

Gather information on a URL target, without downloading it

See *datalad_next.url_operations.UrlOperations.stat()* for parameter documentation and exception behavior.

upload(*from_path: Path | None*, *to_url: str*, *, *credential: str | None = None*, *hash: list[str] | None = None*, *timeout: float | None = None*) → Dict

Upload a file by streaming it through an SSH connection.

It, more or less, runs *ssh <host> 'cat > <path>'*.

See *datalad_next.url_operations.UrlOperations.upload()* for parameter documentation and exception behavior.

datalad_next.url_operations.UrlOperationsRemoteError

exception datalad_next.url_operations.**UrlOperationsRemoteError**(*url*, *message=None*, *status_code: Any | None = None*)

datalad_next.url_operations.UrlOperationsResourceUnknown

exception datalad_next.url_operations.**UrlOperationsResourceUnknown**(*url*, *message=None*, *status_code: Any | None = None*)

A connection request succeeded in principle, but target was not found

Equivalent of an HTTP404 response.

datalad_next.url_operations.UrlOperationsInteractionError

exception datalad_next.url_operations.**UrlOperationsInteractionError**(*url*, *message=None*, *status_code: Any | None = None*)

datalad_next.url_operations.UrlOperationsAuthenticationError

exception datalad_next.url_operations.**UrlOperationsAuthenticationError**(*url: str*, *credential: dict | None = None*, *message: str | None = None*, *status_code: Any = None*)

datalad_next.url_operations.UrlOperationsAuthorizationError

exception datalad_next.url_operations.**UrlOperationsAuthorizationError**(*url: str*, *credential: dict | None = None*, *message: str | None = None*, *status_code: Any | None = None*)

2.3.20 datalad_next.utils

Assorted utility functions

DataladAuth(cfg[, credential])	Requests-style authentication handler using DataLad credentials
MultiHash(algorithms)	Compute any number of hashes as if computing just one
check_symlink_capability(path, target)	helper similar to datalad.tests.utils_pytest.has_symlink_capability
chpwd(path[, mkdir, logsuffix])	Wrapper around os.chdir which also adjusts environ['PWD']
ensure_list(s[, copy, iterate])	Given not a list, would place it into a list.
external_versions	Helper to figure out/use versions of the externals (modules, cmdline tools, etc).
log_progress(lgrcall, pid, *args, **kwargs)	Emit progress log messages
parse_www_authenticate(hdr)	Parse HTTP www-authenticate header
patched_env(**env)	Context manager for patching the process environment
rmtree(path[, chmod_files, children_only])	To remove git-annex .git it is needed to make all files and directories writable again first
get_specialremote_param_dict(params)	**param params**
get_specialremote_credential_properties(parar	Determine properties of credentials special remote configuration
update_specialremote_credential(srtype, ...)	**param srtype**
needs_specialremote_credential_envpatch(...)	Returns whether the environment needs to be patched with credentials
get_specialremote_credential_envpatch(...)	Create an environment path for a particular remote type and credential

datalad_next.utils.DataladAuth

class datalad_next.utils.**DataladAuth**(*cfg:* ConfigManager, *credential: str | None = None*)

Bases: AuthBase

Requests-style authentication handler using DataLad credentials

Similar to request_toolbelt's *AuthHandler*, this is a meta implementation that can be used with different actual authentication schemes. In contrast to *AuthHandler*, a credential can not only be specified directly, but credentials can be looked up based on the target URL and the server-supported authentication schemes.

In addition to programmatic specification and automated lookup, manual credential entry using interactive prompts is also supported.

At present, this implementation is not thread-safe.

handle_401(*r*, ***kwargs*)

Callback that received any response to a request

Any non-4xx response or a response lacking a 'www-authenticate' header is ignored.

Server-provided 'www-authenticated' challenges are inspected, and corresponding credentials are looked-up (if needed) and subsequently tried in a re-request to the original URL after performing any necessary actions to meet a given challenge. Such a re-request is then using the same connection as the original request.

Particular challenges are implemented in dedicated classes, e.g. requests.auth.HTTPBasicAuth.

Credential look-up or entry is performed by `datalad_next.requests_auth.DataladAuth._get_credential()`.

handle_redirect(*r*, ***kwargs*)

Callback that received any response to a request

Any non-redirect response is ignore.

This callback drops an explicitly set credential whenever the redirect causes a non-encrypted connection to be used after the original request was encrypted, or when the *netloc* of the redirect differs from the original target.

save_entered_credential(*suggested_name: str | None = None*, *context: str | None = None*) → Dict | None

Utility method to save a pending credential in the store

Pending credentials have been entered manually, and were subsequently used successfully for authentication.

Saving a credential will prompt for entering a name to identify the credentials.

datalad_next.utils.MultiHash

class datalad_next.utils.**MultiHash**(*algorithms: list[str]*)

Bases: `object`

Compute any number of hashes as if computing just one

Supports any hash algorithm supported by the `hashlib` module of the standard library.

get_hexdigest() → Dict[str, str]

Returns a mapping of algorithm name to hexdigest for all algorithms

update(*data: ByteString*) → None

Updates all configured digests

datalad_next.utils.check_symlink_capability

datalad_next.utils.**check_symlink_capability**(*path: Path*, *target: Path*) → bool

helper similar to datalad.tests.utils_pytest.has_symlink_capability

However, for use in a datalad command context, we shouldn't assume to be able to write to tmpfile and also not import a whole lot from datalad's test machinery. Finally, we want to know, whether we can create a symlink at a specific location, not just somewhere. Therefore use arbitrary path to test-build a symlink and delete afterwards. Suitable location can therefore be determined by high lever code.

Parameters

- **path** (*Path*)
- **target** (*Path*)

Return type
bool

datalad_next.utils.chpwd

class datalad_next.utils.**chpwd**(*path: str | Path | None*, *mkdir: bool = False*, *logsuffix: str = ''*)

Bases: object

Wrapper around os.chdir which also adjusts environ['PWD']

The reason is that otherwise PWD is simply inherited from the shell and we have no ability to assess directory path without dereferencing symlinks.

If used as a context manager it allows to temporarily change directory to the given path

datalad_next.utils.ensure_list

datalad_next.utils.**ensure_list**(*s: Any*, *copy: bool = False*, *iterate: bool = True*) → list

Given not a list, would place it into a list. If None - empty list is returned

Parameters

- **s** (*list or anything*)
- **copy** (*bool, optional*) -- If list is passed, it would generate a shallow copy of the list
- **iterate** (*bool, optional*) -- If it is not a list, but something iterable (but not a str) iterate over it.

datalad_next.utils.external_versions

datalad_next.utils.**external_versions** = <datalad.support.external_versions.ExternalVersions object>

Helper to figure out/use versions of the externals (modules, cmdline tools, etc).

To avoid collision between names of python modules and command line tools, prepend names for command line tools with *cmd:*.

It maintains a dictionary of *distuil.version.LooseVersion`s to make comparisons easy. Note that even if version string conform the StrictVersion "standard", LooseVersion will be used. If version can't be deduced for the external, `UnknownVersion()* is assigned. If external is not present (can't be imported, or custom check throws exception), None is returned without storing it, so later call will re-evaluate fully.

datalad_next.utils.log_progress

datalad_next.utils.**log_progress**(*lgrcall*, *pid*, **args*, ***kwargs*)

Emit progress log messages

This helper can be used to handle progress reporting without having to maintain display mode specific code.

Typical progress reporting via this function involves three types of calls:

1. Start reporting progress about a process
2. Update progress information about a process
3. Report completion of a process

In order to be able to associate all three steps with a particular process, the *pid* identifier is used. This is an arbitrary string that must be chosen to be unique across all different, but simultaneously running progress reporting activities within a Python session. For many practical purposes this can be achieved by, for example, including path information in the identifier.

To initialize a progress report this function is called without an *update* parameter. To report a progress update, this function is called with an *update* parameter. To finish a reporting on a particular activity a final call without an *update* parameter is required.

Parameters

- **lgrcall** (*callable*) -- Something like lgr.debug or lgr.info
- **pid** (*str*) -- Some kind of ID for the process the progress is reported on.
- ***args** (*str*) -- Log message, and potential arguments
- **total** (*int*) -- Max progress quantity of the process.
- **label** (*str*) -- Process description. Should be very brief, goes in front of progress bar on the same line.
- **unit** (*str*) -- Progress report unit. Should be very brief, goes after the progress bar on the same line.
- **update** (*int*) -- To (or by) which quantity to advance the progress. Also see *increment*.
- **increment** (*bool*) -- If set, *update* is interpreted as an incremental value, not absolute.
- **initial** (*int*) -- If set, start value for progress bar
- **noninteractive_level** (*int, optional*) -- In a non-interactive session where progress bars are not displayed, only log a progress report, if a logger's effective level includes the specified level. This can be useful logging all progress is inappropriate or too noisy for a log.
- **maint** (*{'clear', 'refresh'}*) -- This is a special attribute that can be used by callers that are not actually reporting progress, but need to ensure that their (console) output does not interfere with any possibly ongoing progress reporting. Setting this attribute to 'clear' will cause the central ProgressHandler to temporarily stop the display of any active progress bars. With 'refresh', all active progress bars will be redisplayed. After a 'clear' individual progress bars would be reactivated upon the next update log message, even without an explicit 'refresh'.

datalad_next.utils.parse_www_authenticate

datalad_next.utils.**parse_www_authenticate**(*hdr: str*) → dict

Parse HTTP www-authenticate header

This helper uses `requests` utilities to parse the `www-authenticate` header as represented in a `requests.Response` instance. The header may contain any number of challenge specifications.

The implementation follows RFC7235, where a challenge parameters set is specified as: either a comma-separated list of parameters, or a single sequence of characters capable of holding base64-encoded information, and parameters are name=value pairs, where the name token is matched case-insensitively, and each parameter name MUST only occur once per challenge.

Returns

Keys are casefolded challenge labels (e.g., 'basic', 'digest'). Values are: `None` (no parameter), `str` (a token68), or `dict` (name/value mapping of challenge parameters)

Return type

dict

datalad_next.utils.patched_env

datalad_next.utils.**patched_env**(***env*)

Context manager for patching the process environment

Any number of kwargs can be given. Keys represent environment variable names, and values their values. A value of `None` indicates that the respective variable should be unset, i.e., removed from the environment.

datalad_next.utils.rmtree

datalad_next.utils.**rmtree**(*path: str | Path*, *chmod_files: bool | Literal['auto'] = 'auto'*, *children_only: bool = False*, **args: Any*, ***kwargs: Any*) → None

To remove git-annex .git it is needed to make all files and directories writable again first

Parameters

- **path** (*Path or str*) -- Path to remove
- **chmod_files** (*string or bool, optional*) -- Whether to make files writable also before removal. Usually it is just a matter of directories to have write permissions. If 'auto' it would chmod files on windows by default
- **children_only** (*bool, optional*) -- If set, all files and subdirectories would be removed while the path itself (must be a directory) would be preserved
- ***args**
- ****kwargs** -- Passed into shutil.rmtree call

datalad_next.utils.get_specialremote_param_dict

datalad_next.utils.**get_specialremote_param_dict**(*params*)

Parameters
params (*list*)

Return type
dict

datalad_next.utils.get_specialremote_credential_properties

datalad_next.utils.**get_specialremote_credential_properties**(*params*)

Determine properties of credentials special remote configuration

The input is a parameterization as it would be given to *git annex initremote|enableremote <name> ...*, or as stored in *remote.log*. These parameters are inspected and a dictionary of credential properties, suitable for *CredentialManager.query()* is returned. This inspection may involve network activity, e.g. HTTP requests.

Parameters
params (*list or dict*) -- Either a list of strings of the format 'param=value', or a dictionary with parameter names as keys.

Returns
Credential property name-value mapping. This mapping can be passed to *CredentialManager.query()*. If no credential properties could be inferred, for example, because the special remote type is not recognized *None* is returned.

Return type
dict or None

datalad_next.utils.update_specialremote_credential

datalad_next.utils.**update_specialremote_credential**(*srtype*, *credman*, *credname*, *credprops*, *credtype_hint=None*, *duplicate_hint=None*)

Parameters

- **srtype** (*str*)
- **credman** (CredentialManager)
- **credname** (*str or Name*)
- **credprops** (*dict*)

datalad_next.utils.needs_specialremote_credential_envpatch

datalad_next.utils.**needs_specialremote_credential_envpatch**(*remote_type*)

Returns whether the environment needs to be patched with credentials

Returns
False, if the special remote type is not recognized as one needing credentials, or if there are credentials already present. True, otherwise.

Return type
bool

datalad_next.utils.get_specialremote_credential_envpatch

datalad_next.utils.**get_specialremote_credential_envpatch**(*remote_type*, *cred*)

Create an environment path for a particular remote type and credential

Returns
A dict with all required items to patch the environment, or None if not enough information is available, or nothing needs to be patched.

Return type
dict or None

class datalad_next.utils.**ParamDictator**(*params: Dict*)

Bases: object

Parameter dict access helper

This class can be used to wrap a dict containing function parameter name-value mapping, and get/set values by parameter name attribute rather than via the __getitem__ dict API.

2.4 Git-remote helpers

datalad_annex	git-remote-datalad-annex to fetch/push via any git-annex special remote

2.4.1 datalad_next.gitremotes.datalad_annex

git-remote-datalad-annex to fetch/push via any git-annex special remote

In essence, this Git remote helper bootstraps a utility repository in order to push/fetch the state of a repository to any location accessible by any git-annex special remote implementation. All information necessary for this bootstrapping is taken from the remote URL specification. The internal utility repository is removed again after every invocation. Therefore changes to the remote access configuration can be made any time by simply modifying the configured remote URL.

When installed, this remote helper is invoked for any "URLs" that start with the prefix `datalad-annex::`. Following this prefix, two types of specifications are support.

1. Plain parameters list:

   ```
   datalad-annex::?type=<special-remote-type>&[...][exporttree=yes]
   ```

   In this case the prefix is followed by a URL query string that comprises all necessary (and optional) parameters that would be normally given to the `git annex initremote` command. It is required to specify the special remote `type`, and it is possible to request "export" mode for any special remote that supports it. Depending on the chosen special remote additional parameters may be required or supported. Please consult the git-annex documentation at https://git-annex.branchable.com/special_remotes/

2. URL:

   ```
   datalad-annex::<url>[?...]
   ```

   Alternatively, an actual URL can be given after the prefix. In this case, the, now optional, URL query string can still be used to specify arbitrary parameters for special remote initialization. In addition, the query string specification can use Python-format-style placeholder to reference particular URL components as parameters values, in order to avoid double-specification.

   The list of supported placeholders is `scheme`, `netloc`, `path`, `fragment`, `username`, `password`, `hostname`, `port`, corresponding to the respective URL components. In addition, a `noquery` placeholder is supported, which resolves to the entire URL except any query string. An example of such a URL specification is:

   ```
   datalad-annex::file:///tmp/example?type=directory&directory={path}&encryption=none'
   ```

   which would initialize a `type=directory` special remote pointing at `/tmp/example`.

Caution with collaborative workflows

There is no protection against simultaneous, conflicting repository state uploads from two different locations! Similar to git-annex's "export" feature, this feature is most appropriately used as a dataset deposition mechanism, where uploads are conducted from a single site only -- deposited for consumption by any number of parties.

If this Git remote helper is to be used for multi-way collaboration, with two or more parties contributing updates, it is advisable to employ a separate `datalad-annex::` target site for each contributor, such that only one site is pushing to any given location. Updates are exchanged by the remaining contributors adding the respective other `datalad-annex::` sites as additional Git remotes, analog to forks of a repository.

Special remote type support

In addition to the regular list of special remotes, plain http(s) access via URLs is also supported via the 'web' special remote. For such cases, only the base URL and the 'type=web' parameter needs to be given, e.g:

```
git clone 'datalad-annex::https://example.com?type=web&url={noquery}'
```

When a plain URL is given, with no parameter specification in a query string, the parameters `type=web` and `exporttree=yes` are added automatically by default. This means that this remote helper can clone from any remote deposit accessible via `http(s)` that matches the layout depicted in the next section.

Remote layout

The representation of a repository at a remote depends on the chosen type of special remote. In general, two files will be deposited. One text file containing a list of Git `refs` contained in the deposit, and one ZIP file with a (compressed) archive of a bare Git repository. Beside the idiosyncrasies of particular special remotes, to major modes determine the layout of a remote deposit. In "normal" mode, two annex keys (`XDLRA--refs`, `XDLRA--repo-export`) will be deposited. In "export" mode, a directory tree is created that is designed to blend with arbitrary repository content, such that a git remote and a git-annex export can be pushed to the same location without conflicting with each other. The aforementioned files will be represented like this:

```
.datalad
└── dotgit  # named to not be confused with an actual Git repository
    ├── refs
    └── repo.zip
```

The default LZMA-compression of the ZIP file (in both export and normal mode) can be turned off with the `dladotgit=uncompressed` URL parameter.

Credential handling

Some git-annex special remotes require the specification of credentials via environment variables. With the URL parameter `dlacredential=<name>` it is possible to query DataLad for a user/password credential to be used for this purpose. This convenience functionality is supported for the special remotes `glacier`, `s3`, and `webdav`.

When a credential of the given name does not exist, or no credential name was specified, an attempt is made to determine a suitable credential based on, for example, a detected HTTP authentication realm. If no matching credential could be found, the user will be prompted to enter a credential. After having successfully established access, the entered credential will be saved in the local credential store.

DataLad-based credentials are only utilized, when the native git-annex credential setup via environment variables is not in use (see the documentation of a particular special remote implementation for more information).

Implementation details

This Git remote implementation uses *two* extra repositories, besides the repository (R) it is used with, to do its work:

(A) A tiny repository that is entirely bootstrapped from the remote URL, and is used to retrieve/deposit a complete state of the actual repo an a remote site, via a git-annex special remote setup.

(B) A local, fully functional mirror repo of the remotely stored repository state.

On fetch/push the existence of both additional repositories is ensured. The remote state of retrieved via repo (A), and unpacked to repo (B). The actual fetch/push Git operations are performed locally between the repo (R) and repo (B). On push, repo (B) is then packed up again, and deposited on the remote site via git-annex transfer in repo (A).

Due to a limitation of this implementation, it is possible that when the last upload step fails, Git nevertheless advances the pushed refs, making it appear as if the push was completely successful. That being said, Git will still issue a message (`error: failed to push some refs to..`) and the git-push process will also exit with a non-zero status.

In addition, all of the remote's refs will be annotated with an additional ref named `refs/dlra-upload-failed/<remote-name>/<ref-name>` to indicate the upload failure. These markers will be automatically removed after the next successful upload.

Note: Confirmed to work with git-annex version 8.20211123 onwards.

Todo:

- At the moment, only one format for repository deposition is supported (a ZIP archive of a working bare repository). However this is not a good format for the purpose of long-term archiving, because it require a functional Git installation to work with. It would be fairly doable to make the deposited format configurable, and support additional formats. An interesting one would be a fast-export stream, basically a plain text serialization of an entire repository.
- recognize that a different repo is being pushed over an existing one at the remote
- think about adding additional information into the header of *refs* maybe give it some kind of stamp that also makes it easier to validate by the XDLRA backend
- think about preventing duplication between the repo and its local mirror could they safely share git objects? If so, in which direction?

---

class datalad_next.gitremotes.datalad_annex.**RepoAnnexGitRemote**(*gitdir: str*, *remote: str*, *url: str*, *instream: ~typing.IO = <_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>*, *outstream: ~typing.IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *errstream: ~typing.IO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*)

Bases: `object`

git-remote-helper implementation

`communicate()` is the entrypoint.

communicate() → None

Implement the necessary pieces of the git-remote-helper protocol

Uses the input, output and error streams configured for the class instance.

get_mirror_refs() → str

Return the refs of the current mirror repo

Return type
str

get_remote_refs(*raise_on_error: bool = False*) → str | None

Report remote refs

The underlying special remote is asked whether it has the key containing the refs list for the remote. If it does, it is retrieved and reported.

Returns
If the remote has a refs record, it is returned as a string, formatted like a refs file in a Git directory. Otherwise, *None* is returned.

Return type
str or None

`internal_parameters = ('dladotgit=uncompressed', 'dlacredential=')`

`log`(**args*, *level: int = 2*) → None
Send log messages to the errstream

`property mirrorrepo: GitRepo`
Local remote mirror repository

If accessed when there is no local mirror repo, as new one is created automatically, either from the remote state (if there is any), or an empty one.

Returns
This is always only a plain Git repository (bare).

Return type
GitRepo

`refs_key = 'XDLRA--refs'`

`replace_mirrorrepo_from_remote_deposit`() → None
Replaces the local mirror repo with one obtained from the remote

This method assumes that the remote does have one. This should be checked by inspecting *get_remote_refs()* before calling this method.

`replace_mirrorrepo_from_remote_deposit_if_needed`() → tuple[str | None, str]
Replace the mirror if the remote has refs and they differ

`replace_remote_deposit_from_mirrorrepo`() → None
Package the local mirrorrepo up, and copy to the special remote

The mirror is assumed to be ready/complete. It will be cleaned with *gc* to minimize the upload size. The mirrorrepo is then compressed into an LZMA ZIP archive, and a separate refs list for it is created in addition. Both are then copied to the special remote.

`repo_export_key = 'XDLRA--repo-export'`

`property repoannex: AnnexRepo`
Repo annex repository

If accessed when there is no repo annex, as new one is created automatically. It is bootstrapped entirely from the parameters encoded in the remote URL.

Returns
This is always an annex repository. It is configured with a single special remote, parameterized from the Git repo URL.

Return type
AnnexRepo

Raises

- *CommandError* --
- `ValueError` --

```
safe_content = ['branches', 'hooks', 'info', 'objects', 'refs', 'config',
'packed-refs', 'description', 'HEAD']
```

send(*msg: str*) → None

Communicate with Git

```
support_githelper_options = {'verbosity': EnsureInt()}
```

```
xdlra_key_locations = {'XDLRA--refs': {'loc': '.datalad/dotgit/refs', 'prefix':
'3f7/4a3'}, 'XDLRA--repo-export': {'loc': '.datalad/dotgit/repo.zip', 'prefix':
'eb3/ca0'}}
```

2.5 Git-annex backends

base	Interface and essential utilities to implement external git-annex backends
xdlra	git-annex external backend XDLRA for git-remote-datalad-annex

2.5.1 datalad_next.annexbackends.base

Interface and essential utilities to implement external git-annex backends

exception datalad_next.annexbackends.base.**AnnexError**

Bases: Exception

Common base class for all annexbackend exceptions.

class datalad_next.annexbackends.base.**Backend**(*annex*)

Bases: object

Metaclass for backends.

It implements the communication with git-annex via the external backend protocol. More information on the protocol is available at https://git-annex.branchable.com/design/external_backend_protocol/

External backends can be built by implementing the abstract methods defined in this class.

annex

The Master object to which this backend is linked. Master acts as an abstraction layer for git-annex.

Type
Master

abstract can_verify()

Returns whether the backend can verify the content of files match a key it generated. The verification does not need to be cryptographically secure, but should catch data corruption.

Return type
bool

error(*error_msg*)

Communicate a generic error.

Can be sent at any time if things get too messed up to continue. If the program receives an error() from git-annex, it can exit with its own error(). Eg.: self.annex.error("Error received. Exiting.") raise SystemExit

Parameters
error_msg (*str*) -- The error message received from git-annex

abstract gen_key(*local_file*)

Examine the content of *local_file* and from it generate a key.

While it is doing this, it can send any number of PROGRESS messages indication the position in the file that it's gotten to.

Parameters
local_file (*str*) -- Path for which to generate a key. Note that in some cases, local_file may contain whitespace.

Returns
The generated key.

Return type
str

Raises
BackendError -- If the file could not be received from the backend.

abstract is_cryptographically_secure()

Returns whether keys it generates are verified using a cryptographically secure hash.

Note that sha1 is not a cryptographically secure hash any longer. A program can change its answer to this question as the state of the art advances, and should aim to stay ahead of the state of the art by a reasonable amount of time.

Return type
bool

abstract is_stable()

Returns whether a key it has generated will always have the same content. The answer to this is almost always yes; URL keys are an example of a type of key that may have different content at different times.

Return type
bool

abstract verify_content(*key*, *content_file*)

Examine a file and verify it has the content expected given a key

While it is doing this, it can send any number of PROGRESS messages indicating the position in the file that it's gotten to.

If *can_verify() == False*, git-annex not ask to do this.

Return type
bool

exception datalad_next.annexbackends.base.**BackendError**

Bases: *AnnexError*

Must be raised by the backend when a request did not succeed.

class datalad_next.annexbackends.base.**Master**(*output=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: object

Metaclass for backends.

input

Where to listen for git-annex request messages. Default: sys.stdin

Type
io.TextIOBase

output

Where to send replies and backend messages Default: sys.stdout

Type
io.TextIOBase

backend

A class implementing the Backend interface to which this master is linked.

Type
Backend

LinkBackend(*backend*)

Link the Master to a backend. This must be done before calling Listen()

Parameters
backend (Backend) -- A class implementing Backend interface to which this master will be linked.

Listen(*input=<_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>*)

Listen on *input* for messages from git annex.

Parameters
input (*io.TextIOBase*) -- Where to listen for git-annex request messages. Default: sys.stdin

Raises
NotLinkedError -- If there is no backend linked to this master.

debug(**args*)

Tells git-annex to display the message if --debug is enabled.

Parameters
message (*str*) -- The message to be displayed to the user

error(**args*)

Generic error. Can be sent at any time if things get too messed up to continue. When possible, raise a BackendError inside the respective functions. The backend program should exit after sending this, as git-annex will not talk to it any further.

Parameters
error_msg (*str*) -- The error message to be sent to git-annex

progress(*progress*)

Indicates the current progress of the transfer (in bytes). May be repeated any number of times during the transfer process, but it's wasteful to update the progress until at least another 1% of the file has been sent. This is highly recommended for `*_store()`. (It is optional but good for `*_retrieve()`.)

Parameters
progress (*int*) -- The current progress of the transfer in bytes.

exception datalad_next.annexbackends.base.**NotLinkedError**

Bases: *AnnexError*

Will be raised when a Master instance is accessed without being linked to a Backend instance

class datalad_next.annexbackends.base.**Protocol**(*backend*)

Bases: object

Helper class handling the receiving part of the protocol (git-annex to backend) It parses the requests coming from git-annex and calls the respective method of the backend object.

command(*line*)

do_CANVERIFY()

do_ERROR(*message*)

do_GENKEY(**arg*)

do_GETVERSION()

do_ISCRYPTOGRAPHICALLYSECURE()

do_ISSTABLE()

do_VERIFYKEYCONTENT(**arg*)

lookupMethod(*command*)

exception datalad_next.annexbackends.base.**ProtocolError**

Bases: *AnnexError*

Base class for protocol errors

exception datalad_next.annexbackends.base.**UnexpectedMessage**

Bases: *ProtocolError*

Raised when git-annex sends a message which is not expected at the moment

exception datalad_next.annexbackends.base.**UnsupportedRequest**

Bases: *ProtocolError*

Must be raised when an optional request is not supported by the backend.

2.5.2 datalad_next.annexbackends.xdlra

git-annex external backend XDLRA for git-remote-datalad-annex

class datalad_next.annexbackends.xdlra.**DataladRepoAnnexBackend**(*annex*)

Bases: *Backend*

Implementation of an external git-annex backend

This backend is tightly coupled to the *git-remote-datalad-annex* and hardly of any general utility. It is essentially aiming to be the leanest possible implementation to get git-annex to transport the content of two distinct files to and from a special remote. This backend is unlike most backends, because there is no fixed association of a particular file content to a particular key. In other words, the key content is expected to change without any change in the key name.

Only two keys are supported:

- `XDLRA--refs`
- `XDLRA--repo-export`

`XDLRA--refs` contains a "refs" list of a Git repository, similar to the output of `git for-each-ref`. `XDLRA--repo-export` hold a ZIP archive of a bare Git repository.

can_verify()

Returns whether the backend can verify the content of files match a key it generated. The verification does not need to be cryptographically secure, but should catch data corruption.

Return type
bool

gen_key(*local_file*)

Examine the content of *local_file* and from it generate a key.

While it is doing this, it can send any number of PROGRESS messages indication the position in the file that it's gotten to.

Parameters
local_file (*str*) -- Path for which to generate a key. Note that in some cases, local_file may contain whitespace.

Returns
The generated key.

Return type
str

Raises
BackendError -- If the file could not be received from the backend.

is_cryptographically_secure()

Returns whether keys it generates are verified using a cryptographically secure hash.

Note that sha1 is not a cryptographically secure hash any longer. A program can change its answer to this question as the state of the art advances, and should aim to stay ahead of the state of the art by a reasonable amount of time.

Return type
bool

is_stable()

Returns whether a key it has generated will always have the same content. The answer to this is almost always yes; URL keys are an example of a type of key that may have different content at different times.

Return type
bool

verify_content(*key*, *content_file*)

Examine a file and verify it has the content expected given a key

While it is doing this, it can send any number of PROGRESS messages indicating the position in the file that it's gotten to.

If *can_verify()* == *False*, git-annex not ask to do this.

Return type
bool

datalad_next.annexbackends.xdlra.**main**()

Entry point for the backend utility

2.6 Git-annex special remotes

SpecialRemote(annex)	Base class of all datalad-next git-annex special remotes
archivist	git-annex special remote *archivist* for obtaining files from archives
uncurl	uncurl git-annex external special remote

2.6.1 datalad_next.annexremotes.SpecialRemote

class datalad_next.annexremotes.**SpecialRemote**(*annex*)

Bases: SpecialRemote

Base class of all datalad-next git-annex special remotes

get_remote_gitcfg(*remotetypename: str*, *name: str*, *default: Any | None = None*, ***kwargs*)

Get a particular Git configuration item for the special remote

This target configuration here is *not* the git-annex native special remote configuration that is provided or altered with `initremote` and `enableremote`, and is committed to the `git-annex` branch. Instead this is a clone and remote specific configuration, declared in Git's configuration system.

The configuration items queried have the naming scheme:

```
remote.<remotename>.<remotetypename>-<name>
datalad.<remotetypename>.<name>
```

where <remotename> is the name of the Git remote, the special remote is operating under, <remotetypename> is the name of the special remote implementation (e.g., `uncurl`), and <name> is the name of a particular configuration flavor.

Parameters

- **remotetypename** (*str*) -- Name of the special remote implementation configuration is requested for.
- **name** (*str*) -- The name of the "naked" configuration item, without any sub/sections. Must be a valid git-config variable name, i.e., case-insensitive, only alphanumeric characters and -, and must start with an alphabetic character.
- **default** -- A default value to be returned if there is no configuration.
- ****kwargs** -- Passed on to *datalad_next.config.ConfigManager.get()*

Returns

If a remote-specific configuration exists, it is reported. Otherwise a remote-type specific configuration is reported, or the default provided with the method call, if no configuration is found at all.

Return type

Any

property remotename: str

Name of the (git) remote the special remote is operating under

property repo: *LeanAnnexRepo*

Returns a representation of the underlying git-annex repository

An instance of *LeanAnnexRepo* is returned, which intentionally provides a restricted API only. In order to limit further proliferation of the AnnexRepo API.

2.6.2 datalad_next.annexremotes.archivist

git-annex special remote *archivist* for obtaining files from archives

class datalad_next.annexremotes.archivist.**ArchivistRemote**(*annex*)

Bases: *SpecialRemote*

git-annex special remote *archivist* for obtaining files from archives

Successor of the *datalad-archive* special remote. It claims and acts on particular archive locator "URLs", registered for individual annex keys (see *datalad_next.types.archivist.ArchivistLocator*). These locators identify another annex key that represents an archive (e.g., a tarball or a zip files) that contains the respective annex key as a member. This special remote trigger the extraction of such members from any candidate archive when retrieval of a key is requested.

This special remote cannot store or remove content. The desired usage is to register a locator "URL" for any relevant key via `git annex addurl|registerurl` or `datalad addurls`.

Configuration

The behavior of this special remote can be tuned via a number of configuration settings.

datalad.archivist.legacy-mode=yes|[no]

If enabled, all special remote operations fall back onto the legacy `datalad-archives` special remote implementation. This mode is only provided for backward-compatibility. This legacy implementation unconditionally downloads archive files completely, and keeps an internal cache of the full extracted archive around. The implied 200% (or more) storage cost overhead for obtaining a complete dataset can be prohibitive for datasets tracking large amount of data (in archive files).

Implementation details

CHECKPRESENT

When performing a non-download test for the (continued) presence of an annex key (as triggered via `git annex fsck --fast` or `git annex checkpresentkey`), the underlying archive containing a key will NOT be inspected. Instead, only the continued availability of the annex key for the containing archive will be tested. In other words: this implementation trust the archive member annotation to be correct/valid, and it also trusts the archive content to be unchanged. The latter will be generally the case, but may no with URL-style keys.

Not implementing such a trust-approach *would* have a number of consequences. Depending on where the archive is located (local/remote) and what format it is (fsspec-inspectable or not), we would need to download it completely in order to verify a matching archive member. Moreover, an archive might also reference another archive as a source, leading to a multiplication of transfer demands.

__getattribute__(*name: str*)

Redirect top-level API calls to legacy implementation, if needed

checkpresent(*key: str*) → bool

Verifies continued availability of the archive referenced by the key

No content verification of the archive, or of the particular archive member is performed. See "Implementation details" of this class for a rational.

Returns
True if the referenced archive key is present on any remote. False if not.

Return type
bool

checkurl(*url: str*) → bool

Parses *ArchivistLocator*-style URLs

Returns `True` for any syntactically correct URL with all required properties.

The implementation is identical to `claimurl()`.

claimurl(*url: str*) → bool

Returns True for *ArchivistLocator*-style URLs

Only a lexical check is performed. Any other URL will result in `False` to be returned.

initremote()

This method does nothing, because the special remote requires no particular setup.

prepare()

Prepare the special remote for requests by git-annex

If the special remote is instructed to run in "legacy mode", all subsequent operations will be processed by the `datalad-archives` special remote implementation!

remove(*key: str*)

Raises `UnsupportedRequest`. This operation is not supported.

transfer_retrieve(*key: str*, *localfilename: str*)

Retrieve an archive member from a (remote) archive

All registered locators for a requested key will be sorted by availability and size of the references archives. For each archive the most suitable handler will be initialized, and extraction of the identified member will be attempted. If that fails, the next handler is tried until all candidate handlers are exhausted. Depending on the archive availability and type, archives may need to be retrieved from remote sources.

transfer_store(*key: str*, *filename: str*)

Raises `UnsupportedRequest`. This operation is not supported.

datalad_next.annexremotes.archivist.**main**()

CLI entry point installed as `git-annex-remote-archivist`

2.6.3 datalad_next.annexremotes.uncurl

uncurl git-annex external special remote

This implementation is a git-annex accessible interface to datalad-next's URL operations framework. It serves two main purposes:

1. Combine git-annex's capabilities of registering and accessing file content via URLs with DataLad's access credential management and (additional or alternative) transport protocol implementations.
2. Minimize the maintenance effort for datasets (primarily) composed from content that is remotely accessible via URLs from systems other than Datalad or git-annex in the event of an infrastructure transition (e.g. moving to a different technical system or a different data organization on a storage system).

Requirements

This special remote implementation requires git-annex version 8.20210127 (or later) to be available.

Download helper

The simplest way to use this remote is to initialize it without any particular configuration:

```
$ git annex initremote uncurl type=external externaltype=uncurl encryption=none
initremote uncurl ok
(recording state in git...)
```

Once initialized, or later enabled in a clone, `git-annex addurl` will check with the *uncurl* remote whether it can handle a particular URL, and will let the remote perform the download in case of positive response. By default, the remote will claim any URLs with a scheme that the local datalad-next installation supports. This always includes `file://`, `http://`, and `https://`, but is extensible, and a particular installation may also support `ssh://` (by default when openssh is installed), or other schemes.

This additional URL support is also available for other commands. Here is an example how `datalad addurls` can be given any uncurl-supported URLs (here an SSH-URL) directly, provided that the `uncurl` remote was initialized for a dataset (as shown above):

```
$ echo '[{"url":"ssh://my.server.org/home/me/file", "file":"dummy"}]' \
    | datalad addurls - '{url}' '{file}'
```

This makes legacy commands (e.g., `datalad download-url`), unnecessary, and facilitates the use of more advanced `datalad addurls` features (e.g., automatic creation of subdatasets) that are not provided by lower-level commands like `git annex addurl`.

Download helper with credential management support

With this setup, download requests now also use DataLad's credential system for authentication. DataLad will automatically lookup matching credentials, prompt for manual entry if none are found, and offer to store them securely for later use after having used them successfully:

```
$ git annex addurl http://httpbin.org/basic-auth/myuser/mypassword
Credential needed for access to http://httpbin.org/basic-auth/myuser/mypassword
user: myuser
```

(continues on next page)

(continued from previous page)

```
password:
password (repeat):
Enter a name to save the credential
(for accessing http://httpbin.org/basic-auth/myuser/mypassword) securely for future
reuse, or 'skip' to not save the credential
name: httpbin-dummy

addurl http://httpbin.org/basic-auth/myuser/mypassword (from uncurl) (to ...)
ok
(recording state in git...)
```

By adding files via downloads from URLs in this fashion, datasets can be built that track information across a range of locations/services, using a possibly heterogeneous set of access methods.

This feature is very similar to the `datalad` special remote implementation included in the core DataLad package. The difference here is that alternative implementations of downloaders are employed and the `datalad-next` credential system is used instead of the "providers" mechanism from DataLad's core package.

Transforming recorded URLs

The main benefit of using *uncurl* is, however, only revealed when the original snapshot of where data used to be accessible becomes invalid, maybe because data were moved to a different storage system, or simply a different host.

This would typically require an update of each, now broken, access URL. For datasets with thousands or even millions of files this can be an expensive operation. For data portal operators providing a large number of datasets it is even more tedious.

uncurl enables programmatic, on-access URL rewriting. This is similar, in spirit, to Git's `url.<base>.insteadOf` URL modification feature. However, modification possibilities reach substantially beyond replacing a base URL.

This feature is based on two customizable settings: 1) a *URL template*; and 2) a *set of match expressions* that extract additional identifiers from any recorded access URL for an annex key.

Here is an example: Let's say a file in a dataset has a recorded access URL of:

```
https://data.example.org/c542/s7612_figure1.pdf
```

We can let *uncurl* know that `c542` is actually an identifier for a particular collection of items in this data store. Likewise `s7612` is an identifier of a particular item in that collection, and `figure1.pdf` is the name of a component in that collection item. The following Python regular expression can be used to "decompose" the above URL into these semantic components:

```
(?P<site>https://[^/]+)/(?P<collection>c[^/]+)/(?P<item>s[^/]+)_(?P<component>.*)$
```

This expression is not the most readable, but it basically chunks the URL into segments of `(?P<name>...)`, so-called named groups (see a live demo of this expression).

This expression, and additional ones like it, can set as a configuration parameter of an *uncurl* remote setup. Extending the configuration established by the `initremote` call above:

```
$ git annex enableremote uncurl \
    'match=(?P<site>https://[^/]+)/(?P<collection>c[^/]+)/(?P<item>s[^/]+)_(?P<component>
↪.*)$'
```

The last argument is quoted to prevent it from being processed by the shell.

With the match expression configured, URL rewriting can be enabled by declaring a URL template as another configuration item. The URL template uses the Python Format String Syntax. If the new URL for the file above is now `http://newsite.net/ex-archive/c542_s7612_figure1.pdf`, we can declare the following URL template to have *uncurl* go to the new site:

```
http://newsite.net/ex-archive/{collection}_{item}_{component}
```

This template references the identifiers of the named groups we defined in the match expression. Again, the URL template can be set via `git annex enableremote`:

```
$ git annex enableremote uncurl \
    'url=http://newsite.net/ex-archive/{collection}_{item}_{component}'
```

There is no need to separate the `enableremote` calls. Both configuration can be given at the same time. In fact, they can also be given to `initremote` immediately.

The three identifiers `site`, `collection`, `item`, and `component` are actually a custom addition to a standard set of identifiers that are available for composing URLs via a template.

- `datalad_dsid` - the DataLad dataset ID (UUID)
- `annex_dirhash` - "mixed" variant of the two level hash for a particular key (uses POSIX directory separators, and included a trailing separator)
- `annex_dirhash_lower` - "lower case" variant of the two level hash for a particular key (uses POSIX directory separators, and included a trailing separator)
- `annex_key` - git-annex key name for a request
- `annex_remoteuuid` - UUID of the special remote (location) used by git-annex
- `git_remotename` - Name of the Git remote for the uncurl special remote

---

Note: The URL template must "resolve" to a complete and valid URL. This cannot be verified at configuration time, because even the URL scheme could be a dynamic setting.

Uploading content

The *uncurl* special remote can upload file content or store annex keys via supported URL schemes whenever a URL template is defined. At minimum, storing at `file://` and `ssh://` URLs are supported. But other URL scheme handlers with upload support may be available at a local DataLad installation.

Deleting content

As for uploading, deleting content is only permitted with a configured URL template. Moreover, it also depends on the delete operation being supported for a particular URL scheme.

Configuration overrides

Both match expressions and the URL template can also be configured in a dataset's configuration (committed branch configuration, or any Git configuration scope (local, global, system) using the following configuration item names:

- `remote.<remotename>.uncurl-url`
- `remote.<remotename>.uncurl-match`

where `<remotename>` is the name of the special remote in the dataset.

A URL template provided via configuration *overrides* one defined in the special remote setup via `init/enableremote`.

Match expressions defined as configuration items *extend* the set of match expressions that may be included in the special remote setup via `init/enableremote`. The `remote.<remotename>.uncurl-match` configuration item can be set as often as necessary (which one match expression each).

Tips

When multiple match expressions are defined, it is recommended to use unique names for each match-group to avoid collisions.

class datalad_next.annexremotes.uncurl.**UncurlRemote**(*annex: Master*)

Bases: *SpecialRemote*

checkpresent(*key: str*) → bool

Requests the remote to check if a key is present in it.

Parameters
key (*str*)

Returns
True if the key is present in the remote. False if the key is not present.

Return type
bool

Raises
RemoteError -- If the presence of the key couldn't be determined, eg. in case of connection error.

checkurl(*url: str*) → bool

When running *git-annex addurl*, this is called after CLAIMURL indicated that we could handle a URL. It can return information on the URL target (e.g., size of the download, a target filename, or a sequence thereof with additional URLs pointing to individual components that would jointly make up the full download from the given URL. However, all of that is optional, and a simple *True* returned is sufficient to make git-annex call *TRANSFER RETRIEVE*.

claimurl(*url: str*) → bool

Needs to check if want to handle a given URL

If match expressions are configured, matches the URL against all known URL expressions, and returns *True* if there is any match, or *False* otherwise.

If no match expressions are configured, return *True* of the URL scheme is supported, or *False* otherwise.

extract_tmpl_props(*tmpl: str*, *, *urls: list[str] | None = None*, *key: str | None = None*) → dict[str, str]

get_key_urls(*key: str*) → list[str]

get_mangled_url(*fallback_url: str | None*, *tmpl: str*, *tmpl_props: dict[str, str]*) → str | None

initremote() → None

Gets called when *git annex initremote* or *git annex enableremote* are run. This is where any one-time setup tasks can be done, for example creating the remote folder. Note: This may be run repeatedly over time, as a remote is initialized in different repositories, or as the configuration of a remote is changed. So any one-time setup tasks should be done idempotently.

Raises
RemoteError -- If the remote could not be initialized.

is_recognized_url(*url: str*) → bool

prepare() → None

Tells the remote that it's time to prepare itself to be used. Gets called whenever git annex is about to access any of the below methods, so it shouldn't be too expensive. Otherwise it will slow down operations like *git annex whereis* or *git annex info*.

Internet connection *can* be established here, though it's recommended to defer this until it's actually needed.

Raises
RemoteError -- If the remote could not be prepared.

remove(*key: str*) → None

Requests the remote to remove a key's contents.

Parameters
key (*str*)

Raises
RemoteError -- If the key couldn't be deleted from the remote.

transfer_retrieve(*key: str*, *filename: str*) → None

Get the file identified by *key* from the remote and store it in *local_file*.

While the transfer is running, the remote can repeatedly call annex.progress(size) to indicate the number of bytes already stored. This will influence the progress shown to the user.

Parameters

- **key** (*str*) -- The Key to get from the remote.
- **local_file** (*str*) -- Path where to store the file. Note that in some cases, local_file may contain whitespace.

Raises
RemoteError -- If the file could not be received from the remote.

transfer_store(*key: str*, *filename: str*) → None

Store the file in *local_file* to a unique location derived from *key*.

It's important that, while a Key is being stored, checkpresent(key) not indicate it's present until all the data has been transferred. While the transfer is running, the remote can repeatedly call annex.progress(size) to indicate the number of bytes already stored. This will influence the progress shown to the user.

Parameters

- **key** (*str*) -- The Key to be stored in the remote. In most cases, this is going to be the remote file name. It should be at least be unambiguously derived from it.
- **local_file** (*str*) -- Path to the file to upload. Note that in some cases, local_file may contain whitespace. Note that local_file should not influence the filename used on the remote.

Raises
RemoteError -- If the file could not be stored to the remote.

datalad_next.annexremotes.uncurl.**main**()
cmdline entry point

2.7 DataLad patches

Patches that are automatically applied to DataLad when loading the `datalad-next` extension package.

annexrepo	Credential support for `AnnexRepo.enable_remote()` and `siblings enable`
cli_configoverrides	Post DataLad config overrides CLI/ENV as GIT_CONFIG items in process ENV
commanderror	Improve `CommandError` rendering and add `returncode` alias for `code`
common_cfg	Change the default of `datalad.annex.retry` to 1
configuration	Enable `configuration()` to query `global` scope without a dataset
create_sibling_ghlike	Improved credential handling for `create_sibling_<github-like>()`
create_sibling_gitlab	Streamline user experience
customremotes_main	Connect `log_progress`-style progress reporting to git-annex, add *close()*
distribution_dataset	`DatasetParameter` support for `resolve_path()`
interface_utils	Uniform pre-execution parameter validation for commands
push_optimize	Make push avoid refspec handling for special remote push targets
push_to_export_remote	Add support for export to WebDAV remotes to `push()`
run	Enhance `run()` placeholder substitutions to honor configuration defaults
siblings	Auto-deploy credentials when enabling special remotes
test_keyring	Recognize DATALAD_TESTS_TMP_KEYRING_PATH to set alternative secret storage
update	Robustify `update()` target detection for adjusted mode datasets

2.7.1 datalad_next.patches.annexrepo

Credential support for `AnnexRepo.enable_remote()` and `siblings enable`

Supported targets for automatic credential deployments are determined by `needs_specialremote_credential_envpatch()`. At the time of this writing this includes the git-annex built-in remote types `webdav`, `s3`, and `glacier`.

This patch also changes the function to raise its custom exception with the context of an original underlying exception for better error reporting.

datalad_next.patches.annexrepo.**annexRepo__enable_remote**(*self*, *name*, *options=None*, *env=None*)
Enables use of an existing special remote

Parameters

- **name** (*str*) -- name, the special remote was created with
- **options** (*list, optional*)

2.7.2 datalad_next.patches.cli_configoverrides

Post DataLad config overrides CLI/ENV as GIT_CONFIG items in process ENV

This enables their propagation to any subprocess. This includes the specification of overrides via the `datalad -c ...` option of the main CLI entrypoint.

datalad_next.patches.cli_configoverrides.**parse_overrides_from_cmdline**(*cmdlineargs*)

2.7.3 datalad_next.patches.commanderror

Improve `CommandError` rendering and add `returncode` alias for `code`

This patch does two things:

It overwrites `__repr__`, otherwise `CommandError` would use ``RuntimeError`'s variant and ignore all additional structured information except for `.msg` -- which is frequently empty and confuses with a *CommandError('')* display.

It adds a `returncode` alias for `code`. This unifies return code access between `CommandError` and *Popen* `-like objects, which usually have a `returncode` attribute.

datalad_next.patches.commanderror.**commanderror_getattr**(*self*, *item*)

datalad_next.patches.commanderror.**commanderror_repr**(*self*) → str

datalad_next.patches.commanderror.**commanderror_setattr**(*self*, *key*, *value*)

2.7.4 datalad_next.patches.common_cfg

Change the default of `datalad.annex.retry` to 1

This prevents unconditional retries, and thereby improves the legibility of errors (now only one error instead of three identical errors).

This change does not override user-settings, only the default.

2.7.5 datalad_next.patches.configuration

Enable `configuration()` to query `global` scope without a dataset

class datalad_next.patches.configuration.**Configuration**

Bases: `Configuration`

static __call__(*action='dump'*, *spec=None*, ***, *scope=None*, *dataset=None*, *recursive=False*, *recursion_limit=None*)

Get and set dataset, dataset-clone-local, or global configuration

This command works similar to git-config, but some features are not supported (e.g., modifying system configuration), while other features are not available in git-config (e.g., multi-configuration queries).

Query and modification of three distinct configuration scopes is supported:

- 'branch': the persistent configuration in .datalad/config of a dataset branch
- 'local': a dataset clone's Git repository configuration in .git/config
- 'global': non-dataset-specific configuration (usually in $USER/.gitconfig)

Modifications of the persistent 'branch' configuration will not be saved by this command, but have to be committed with a subsequent *save* call.

Rules of precedence regarding different configuration scopes are the same as in Git, with two exceptions: 1) environment variables can be used to override any datalad configuration, and have precedence over any other configuration scope (see below). 2) the 'branch' scope is considered in addition to the standard git configuration scopes. Its content has lower precedence than Git configuration scopes, but it is committed to a branch, hence can be used to ship (default and branch-specific) configuration with a dataset.

Besides storing configuration settings statically via this command or `git config`, DataLad also reads any DATALAD_* environment on process startup or import, and maps it to a configuration item. Their values take precedence over any other specification. In variable names _ encodes a `.` in the configuration name, and __ encodes a `-`, such that `DATALAD_SOME__VAR` is mapped to `datalad.some-var`. Additionally, a DATALAD_CONFIG_OVERRIDES_JSON environment variable is queried, which may contain configuration key-value mappings as a JSON-formatted string of a JSON-object:

```
DATALAD_CONFIG_OVERRIDES_JSON='{"datalad.credential.example_com.user": "jane", .
↪..}'
```

This is useful when characters are part of the configuration key that cannot be encoded into an environment variable name. If both individual configuration variables *and* JSON-overrides are used, the former take precedent over the latter, overriding the respective *individual* settings from configurations declared in the JSON-overrides.

This command supports recursive operation for querying and modifying configuration across a hierarchy of datasets.

Examples

Dump the effective configuration, including an annotation for common items:

```
> configuration()
```

Query two configuration items:

```
> configuration('get', ['user.name', 'user.email'])
```

Recursively set configuration in all (sub)dataset repositories:

```
> configuration('set', [('my.config.name', 'value')], recursive=True)
```

Modify the persistent branch configuration (changes are not committed):

```
> configuration('set', [('my.config.name', 'value')], scope='branch')
```

Parameters

- **action** (*{'dump', 'get', 'set', 'unset'}, optional*) -- which action to perform. [Default: 'dump']
- **spec** -- configuration name (for actions 'get' and 'unset'), or name/value pair (for action 'set'). [Default: None]

- **scope** (*{'global', 'local', 'branch', None}, optional*) -- scope for getting or setting configuration. If no scope is declared for a query, all configuration sources (including overrides via environment variables) are considered according to the normal rules of precedence. A 'get' action can be constrained to scope 'branch', otherwise 'global' is used when not operating on a dataset, or 'local' (including 'global', when operating on a dataset. For action 'dump', a scope selection is ignored and all available scopes are considered. [Default: None]
- **dataset** (*Dataset or None, optional*) -- specify the dataset to query or to configure. [Default: None]
- **recursive** (*bool, optional*) -- if set, recurse into potential subdatasets. [Default: False]
- **recursion_limit** (*int or None, optional*) -- limit recursion into subdatasets to the given number of levels. [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_next.patches.configuration.**configuration**(*action*, *scope*, *specs*, *res_kwargs*, *ds=None*)

2.7.6 datalad_next.patches.create_sibling_ghlike

Improved credential handling for `create_sibling_<github-like>()`

This patch makes the storage of a newly entered credential conditional on a successful authorization, in the spirit of datalad/datalad#3126.

Moreover, stored credentials now contain a `realm` property that identified the API endpoint. This makes it possible to identify candidates of suitable credentials without having to specific their name, similar to a request context url used by the old providers setup.

This automatic realm-based credential lookup is now also implemented. When no credential name is specified, the most recently used credential matching the API realm will be used automatically. If determined like this, it will be tested for successful authorization, and will then be stored again with an updated `last-used` timestamp.

2.7.7 datalad_next.patches.create_sibling_gitlab

Streamline user experience

Discontinue advertizing the `hierarchy` layout, and better explain limitations of the command.

2.7.8 datalad_next.patches.customremotes_main

Connect `log_progress`-style progress reporting to git-annex, add *close()*

This patch introduces a dedicated progress log handler as a proxy between standard datalad progress logging and a git-annex special remote as an approach to report (data transfer) progress to a git-annex parent process.

This functionality is only (to be) used in dedicated special remote processes.

This patch also adds a standard *close()* handler to special remotes, and calls that handler in a context manager to ensure releasing any resources. This replaces the custom *stop()* method, which is undocumented and only used by the *datalad-archive* special remote.

class datalad_next.patches.customremotes_main.**AnnexProgressLogHandler**(*annexremote: SpecialRemote*)

Bases: `Handler`

Log handler to funnel progress logs to git-annex

For this purpose the handler wraps *datalad_next.annexremotes.SpecialRemote* instance. When it receives progress log messages, it converts any increment reports to absolute values, and then calls the special remote's `send_progress()` method, which will cause the respective progress update protocol message to be issued.

Note: Git-annex only supports "context-free" progress reporting. When a progress report is send, it is assumed to be on a currently running transfer. Only a single integer value can be reported, and it corresponds to the number of bytes transferred.

This approach implemented here cannot distinguish progress reports that corresponding to git-annex triggered data transfers and other (potentially co-occurring) operations. The likelihood of unrelated operations reporting progress is relatively low, because this handler is only supposed to be used in dedicated special remote processes, but remains possible.

This implementation is set up to support tracking multiple processes, and could report one of them selectively. However, at present any progress update is relayed to git-annex directly. This could lead to confusing and non-linear progress reporting.

---

emit(*record: LogRecord*)

Process a log record

Any incoming log record, compliant with http://docs.datalad.org/design/progress_reporting.html is processed. Increment reports are converted to absolute values, and each update is eventually passed on to special remote, which issues a progress report to git-annex.

datalad_next.patches.customremotes_main.**only_progress_logrecords**(*record: LogRecord*) → bool

Log filter to ignore any non-progress log message

datalad_next.patches.customremotes_main.**patched_underscore_main**(*args: list*, *cls: Type[*SpecialRemote*]*)

Full replacement for datalad.customremotes.main._main()

Its only purpose is to create a running instance of a SpecialRemote. The only difference to the original in datalad-core is that once this instance exists, it is linked to a log handler that converts incoming progress log messages to the equivalent annex protocol progress reports.

This additional log handler is a strict addition to the log handling setup established at this point. There should be no interference with any other log message processing.

See also:

AnnexProgressLogHandler

datalad_next.patches.customremotes_main.**specialremote_defaultclose_noop**(*self*)

2.7.9 datalad_next.patches.distribution_dataset

`DatasetParameter` support for `resolve_path()`

This is the standard result of `EnsureDataset`, which unlike the datalad-core version actually carries a `Dataset` instance.

This patch ensure the traditional handling of "dataset instance from a string-type parameter in this context.

datalad_next.patches.distribution_dataset.**resolve_path**(*path*, *ds=None*, *ds_resolved=None*)

Resolve a path specification (against a Dataset location)

Any path is returned as an absolute path. If, and only if, a dataset object instance is given as *ds*, relative paths are interpreted as relative to the given dataset. In all other cases, relative paths are treated as relative to the current working directory.

Note however, that this function is not able to resolve arbitrarily obfuscated path specifications. All operations are purely lexical, and no actual path resolution against the filesystem content is performed. Consequently, common relative path arguments like '../something' (relative to PWD) can be handled properly, but things like 'down/../under' cannot, as resolving this path properly depends on the actual target of any (potential) symlink leading up to '..'.

Parameters

- **path** (*str or PathLike or list*) -- Platform-specific path specific path specification. Multiple path specifications can be given as a list
- **ds** (*Dataset or PathLike or None*) -- Dataset instance to resolve relative paths against.

- **ds_resolved** (*Dataset or None*) -- A dataset instance that was created from *ds* outside can be provided to avoid multiple instantiation on repeated calls.

Returns
When a list was given as input a list is returned, a Path instance otherwise.

Return type
pathlib.Path object or list(Path)

2.7.10 datalad_next.patches.interface_utils

Uniform pre-execution parameter validation for commands

With this patch commands can now opt-in to receive fully validated parameters. This can substantially simplify the implementation complexity of a command at the expense of a more elaborate specification of the structural and semantic properties of the parameters.

For details on implementing validation for individual commands see *datalad_next.commands.ValidatedInterface*.

datalad_next.patches.interface_utils.**get_allargs_as_kwargs**(*call*, *args*, *kwargs*)

Generate a kwargs dict from a call signature and *args, **kwargs

Basically resolving the argnames for all positional arguments, and resolving the defaults for all kwargs that are not given in a kwargs dict

Returns
The first return value is a mapping of argument names to their respective values. The second return value in the tuple is a set of argument names for which the effective value is identical to the default declared in the signature of the callable. The third value is a set with names of all mandatory arguments, whether or not they are included in the returned mapping.

Return type
(dict, set, set)

2.7.11 datalad_next.patches.push_optimize

Make push avoid refspec handling for special remote push targets

This change introduces a replacement for core's `push.py:_push()` with a more intelligible flow. It replaces the stalled https://github.com/datalad/datalad/pull/6666

Importantly, it makes one behavior change, which is desirable IMHO. Instead of rejecting to git-push any refspec for a repo with a detached HEAD, it will attempt to push a git-annex branch for an AnnexRepo. The respective test that ensured this behavior beyond the particular conditions the original problem occurred in was adjusted accordingly.

All `push` tests from core are imported and executed to ensure proper functioning.

Summary of the original commits patching the core implementation.

- Consolidate publication dependency handling in one place
- Consolidate tracking of git-push-dryrun exec Make a failed attempt discriminable from no prior attempt.
- Factor out helper to determine refspecs-to-push for a target
- Consolidate more handling of git-pushed and make conditional on an actual git-remote target This change is breaking behavior, because previously a source repository without an active branch would have been rejected for a push attempt. However, this is a bit questionable, because the git-annex branch might well need a push.

- Simplify push-logic: no need for a fetch, if there is no git-push
- Factor out helper to sync a remote annex-branch
- Adjust test to constrain the evaluated conditions (replacement tests is included here) As per the reasoning recorded in datalad#1811 (comment) the test ensuring the continue fix of datalad#1811 is actually verifying a situation that is not fully desirable. It prevents pushing of thew 'git-annex' branch whenever a repo is on a detached HEAD. This change let's the test run on a plain Git repo, where there is indeed nothing to push in this case.

2.7.12 datalad_next.patches.push_to_export_remote

Add support for export to WebDAV remotes to `push()`

This approach generally works for any special remote configured with `exporttree=yes`, but is only tested for `type=webdav`. A smooth operation requires automatic deployment of credentials. Support for that is provide and limited by the capabilities of `needs_specialremote_credential_envpatch()`.

`datalad_next.patches.push_to_export_remote.`**`get_export_records`**`(`*repo: AnnexRepo*`)` → Generator

Read exports that git-annex recorded in its 'export.log'-file

Interpret the lines in export.log. Each line has the following structure:

time-stamp " " source-annex-uuid ":" destination-annex-uuid " " treeish

Parameters

repo (*AnnexRepo*) -- The annex repo from which exports should be determined

Returns

Generator yielding one dictionary for each export entry in git-annex. Each dictionary contains the keys: "timestamp", "source-annex-uuid", "destination-annex-uuid", "treeish". The timestamp-value is a float, all other values are strings.

Return type

Generator

2.7.13 datalad_next.patches.run

Enhance `run()` placeholder substitutions to honor configuration defaults

Previously, `run()` would not recognize configuration defaults for placeholder substitution. This means that any placeholders globally declared in `datalad.interface.common_cfg`, or via `register_config()` in DataLad extensions would not be effective.

This patch makes run's `format_command()` helper include such defaults explicitly, and thereby enable the global declaration of substitution defaults.

Moreoever a `{python}` placeholder is now defined via this mechanism, and points to the value of `sys.executable` by default. This particular placeholder was found to be valuable for improving the portability of run-recording across (specific) Python versions, or across different (virtual) environments. See https://github.com/datalad/datalad-container/issues/224 for an example use case.

https://github.com/datalad/datalad/pull/7509

`datalad_next.patches.run.`**`format_command`**`(`*dset*, *command*, ***kwds*`)`

Plug in placeholders in *command*.

Parameters

- **dset** (Dataset)

- **command** (*str or list*)
- **converted** (*kwds is passed to the format call. inputs and outputs are*)
- **necessary.** (*to GlobbedPaths if*)

Return type
formatted command (str)

2.7.14 datalad_next.patches.siblings

Auto-deploy credentials when enabling special remotes

This is the companion of the `annexRepo__enable_remote` patch, and simply removes the webdav-specific credential handling in `siblings()`. It is no longer needed, because credential deployment moved to a lower layer, covering more special remote types.

Manual credential entry on `enableremote` is not implemented here, but easily possible following the patterns from *datalad-annex::* and `create_sibling_webdav()`

2.7.15 datalad_next.patches.test_keyring

Recognize DATALAD_TESTS_TMP_KEYRING_PATH to set alternative secret storage

Within *pytest* DataLad uses the plaintext keyring backend. This backend has no built-in way to configure a custom file location for secret storage from the outside. This patch looks for a DATALAD_TESTS_TMP_KEYRING_PATH environment variable, and uses its value as a file path for the storage.

This makes it possible to (temporarily) switch storage. This feature is used by the `tmp_keyring` pytest fixture. This patch is needed in addition to the test fixture in order to apply such changes also to child processes, such as special remotes and git remotes.

2.7.16 datalad_next.patches.update

Robustify `update()` target detection for adjusted mode datasets

The true cause of the problem is not well understood. https://github.com/datalad/datalad/issues/7507 documents that it is not easy to capture the breakage in a test.

CHAPTER

THREE

DEVELOPING WITH DATALAD NEXT

This extension package moves fast in comparison to the DataLad core package. Nevertheless, attention is paid to API stability, adequate semantic versioning, and informative changelogs.

Besides the DataLad commands shipped with this extension package, a number of Python utilities are provided that facilitate the implementation of workflows and additional functionality. An overview is available in the *reference manual*.

3.1 Public vs internal Python API

Anything that can be imported directly from any of the top-level sub-packages in *datalad_next* is considered to be part of the public API. Changes to this API determine the versioning, and development is done with the aim to keep this API as stable as possible. This includes signatures and return value behavior.

As an example:

```
from datalad_next.runners import iter_git_subproc
```

imports a part of the public API, but:

```
from datalad_next.runners.git import iter_git_subproc
```

does not.

3.2 Use of the internal API

Developers can obviously use parts of the non-public API. However, this should only be done with the understanding that these components may change from one release to another, with no guarantee of transition periods, deprecation warnings, etc.

Developers are advised to never reuse any components with names starting with _ (underscore). Their use should be limited to their individual sub-package.

CHAPTER

FOUR

CONTRIBUTOR INFORMATION

4.1 Developer Guide

This guide sheds light on new and reusable subsystems developed in `datalad-next`. The target audience are developers that intend to build up on or use functionality provided by this extension.

4.1.1 `datalad-next`'s Constraint System

`datalad_next.constraints` implements a system to perform data validation, coercion, and parameter documentation for commands via a flexible set of "Constraints". You can find an overview of available Constraints in the respective module overview of the *Python tooling*.

Adding parameter validation to a command

In order to equip an existing or new command with the constraint system, the following steps are required:

- Set the commands base class to `ValidatedInterface`:

```
from datalad_next.commands import ValidatedInterface

@build_doc
class MyCommand(ValidatedInterface):
    """Download from URLs"""
```

- Declare a `_validator_` class member:

```
from datalad_next.commands import (
    EnsureCommandParameterization,
    ValidatedInterface,
)

@build_doc
class MyCommand(ValidatedInterface):
    """Download from URLs"""

_validator_ = EnsureCommandParameterization(dict(
     [...]
 ))
```

- Determine for each parameter of the command whether it has constraints, and what those constraints are. If you're transitioning an existing command, remove any `constraints=` declaration in the `_parameter_` class member.
- Add a fitting Constraint declaration for each parameter into the `_validator_` as a key-value pair where the key is the parameter and its value is a Constraint. There does not need to be a Constraint per parameter; only add entries for parameters that need validation.

```
from datalad_next.commands import (
    EnsureCommandParameterization,
    ValidatedInterface,
)
from datalad_next.constraints import EnsureChoice
from datalad_next.constraints import EnsureDataset

@build_doc
class Download(ValidatedInterface):
    """Download from URLs"""

_validator_ = EnsureCommandParameterization(dict(
     dataset=EnsureDataset(installed=True),
     force=EnsureChoice('yes','no','maybe'),
 ))
```

Combining constraints

Constraints can be combined in different ways. The `|`, `&`, and `()` operators allow `AND`, `OR`, and grouping of Constraints. The following example from the `download` command defines a chain of possible Constraints:

```
spec_item_constraint = url2path_constraint | (
     (
         EnsureJSON() | EnsureURLFilenamePairFromURL()
     ) & url2path_constraint)
```

Constrains can also be combined using `AnyOf` or `AllOf` MultiConstraints, which correspond almost entirely to `|` and `&`. Here's another example from the `download` command:

```
spec_constraint = AnyOf(
    spec_item_constraint,
    EnsureListOf(spec_item_constraint),
    EnsureGeneratorFromFileLike(
        spec_item_constraint,
        exc_mode='yield',
    ),
```

One can combine an arbitrary number of Constraints. They are evaluated in the order in which they were specified. Logical OR constraints will return the value from the first constraint that does not raise an exception, and logical AND constraints pass the return values of each constraint into the next.

Implementing additional constraints

TODO

Parameter Documentation

TODO

4.1.2 Contributing to `datalad-next`

We're happy about contributions of any kind to this project - thanks for considering making one!

Please take a look at CONTRIBUTING.md for an overview of development principles and common questions, and get in touch in case of questions or to discuss features, bugs, or other issues.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

datalad_next.annexbackends.base, 230
datalad_next.annexbackends.xdlra, 233
datalad_next.annexremotes.archivist, 236
datalad_next.annexremotes.uncurl, 238
datalad_next.archive_operations, 37
datalad_next.commands, 38
datalad_next.config, 41
datalad_next.constraints, 46
datalad_next.consts, 61
datalad_next.credman, 62
datalad_next.datasets, 67
datalad_next.exceptions, 164
datalad_next.gitremotes.datalad_annex, 226
datalad_next.iter_collections, 174
datalad_next.iterable_subprocess, 166
datalad_next.itertools, 166
datalad_next.patches.annexrepo, 243
datalad_next.patches.cli_configoverrides, 244
datalad_next.patches.commanderror, 244
datalad_next.patches.common_cfg, 244
datalad_next.patches.configuration, 244
datalad_next.patches.create_sibling_ghlike, 247
datalad_next.patches.create_sibling_gitlab, 247
datalad_next.patches.customremotes_main, 247
datalad_next.patches.distribution_dataset, 248
datalad_next.patches.interface_utils, 249
datalad_next.patches.push_optimize, 249
datalad_next.patches.push_to_export_remote, 250
datalad_next.patches.run, 250
datalad_next.patches.siblings, 251
datalad_next.patches.test_keyring, 251
datalad_next.patches.update, 251
datalad_next.repo_utils, 184
datalad_next.runners, 184
datalad_next.shell, 189
datalad_next.tests, 201
datalad_next.tests.fixtures, 206
datalad_next.types, 209
datalad_next.uis, 211
datalad_next.uis.ansi_colors, 211
datalad_next.url_operations, 211
datalad_next.utils, 219

INDEX

Symbols

__call__() (*datalad_next.constraints.EnsureCommandParameterization method*), 60
__call__() (*datalad_next.patches.configuration.Configuration static method*), 244
__call__() (*datalad_next.shell.ShellCommandExecutor method*), 192
__getattribute__() (*datalad_next.annexremotes.archivist.ArchivistRemote method*), 236
__repr__() (*datalad_next.constraints.Constraint method*), 48
__str__() (*datalad_next.constraints.Constraint method*), 48

A

action (*datalad_next.commands.CommandResult attribute*), 39
add() (*datalad_next.config.ConfigManager method*), 42
add_archive_content() (*datalad_next.datasets.Dataset method*), 67
add_modification_type() (*datalad_next.iter_collections.GitDiffItem method*), 183
add_readme() (*datalad_next.datasets.Dataset method*), 70
addition (*datalad_next.iter_collections.GitDiffStatus attribute*), 183
addurls() (*datalad_next.datasets.Dataset method*), 71
akey (*datalad_next.types.ArchivistLocator attribute*), 210
align_pattern() (*in module datalad_next.itertools*), 167
AllOf (*class in datalad_next.constraints*), 49
annex (*datalad_next.annexbackends.base.Backend attribute*), 230
AnnexError, 230
AnnexKey (*class in datalad_next.types*), 209
AnnexProgressLogHandler (*class in datalad_next.patches.customremotes_main*), 247
annexRepo__enable_remote() (*in module datalad_next.patches.annexrepo*), 243
AnyOf (*class in datalad_next.constraints*), 49
AnyUrlOperations (*class in datalad_next.url_operations*), 215
ArchiveType (*class in datalad_next.types*), 211
ArchivistLocator (*class in datalad_next.types*), 210
ArchivistRemote (*class in datalad_next.annexremotes.archivist*), 236
assert_in() (*in module datalad_next.tests*), 203
assert_in_results() (*in module datalad_next.tests*), 203
assert_result_count() (*in module datalad_next.tests*), 203
assert_status() (*in module datalad_next.tests*), 203
atype (*datalad_next.types.ArchivistLocator attribute*), 210

B

Backend (*class in datalad_next.annexbackends.base*), 230
backend (*datalad_next.annexbackends.base.Master attribute*), 232
backend (*datalad_next.types.AnnexKey attribute*), 209
BackendError, 231
BasicGitTestRepo (*class in datalad_next.tests*), 202

C

call_git() (*in module datalad_next.runners*), 186
call_git_lines() (*in module datalad_next.runners*), 186
call_git_oneline() (*in module datalad_next.runners*), 186
call_git_success() (*in module datalad_next.runners*), 187
can_verify() (*datalad_next.annexbackends.base.Backend method*), 230
can_verify() (*datalad_next.annexbackends.xdlra.DataladRepoAnnexBac method*), 234
CapturedException (*class in datalad_next.exceptions*), 165
cfg (*datalad_next.url_operations.UrlOperations property*), 212
check_gitconfig_global() (*in module datalad_next.tests.fixtures*), 206

check_plaintext_keyring() (*in module datalad_next.tests.fixtures*), 206
check_symlink_capability() (*in module datalad_next.utils*), 221
checkpresent() (*datalad_next.annexremotes.archivist.ArchivistRemote method*), 236
checkpresent() (*datalad_next.annexremotes.uncurl.UncurlRemote method*), 241
checkurl() (*datalad_next.annexremotes.archivist.ArchivistRemote method*), 237
checkurl() (*datalad_next.annexremotes.uncurl.UncurlRemote method*), 241
chpwd (*class in datalad_next.utils*), 222
chunknumber (*datalad_next.types.AnnexKey attribute*), 209
chunksize (*datalad_next.types.AnnexKey attribute*), 209
claimurl() (*datalad_next.annexremotes.archivist.ArchivistRemote method*), 237
claimurl() (*datalad_next.annexremotes.uncurl.UncurlRemote method*), 241
clean() (*datalad_next.datasets.Dataset method*), 74
clone() (*datalad_next.datasets.Dataset method*), 76
close() (*datalad_next.archive_operations.TarArchiveOperations method*), 37
close() (*datalad_next.archive_operations.ZipArchiveOperations method*), 38
close() (*datalad_next.datasets.Dataset method*), 79
close() (*datalad_next.shell.ShellCommandExecutor method*), 193
command() (*datalad_next.annexbackends.base.Protocol method*), 233
command_zero() (*datalad_next.shell.ShellCommandExecutor method*), 193
CommandError, 187
commanderror_getattr() (*in module datalad_next.patches.commanderror*), 244
commanderror_repr() (*in module datalad_next.patches.commanderror*), 244
commanderror_setattr() (*in module datalad_next.patches.commanderror*), 244
CommandParametrizationError, 51
CommandResult (*class in datalad_next.commands*), 39
CommandResultStatus (*class in datalad_next.commands*), 40
communicate() (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote method*), 228
config (*datalad_next.datasets.Dataset property*), 79
ConfigManager (*class in datalad_next.config*), 42
Configuration (*class in datalad_next.patches.configuration*), 244
configuration() (*datalad_next.datasets.Dataset method*), 79
configuration() (*in module datalad_next.patches.configuration*), 246
Constraint (*class in datalad_next.constraints*), 48
constraint (*datalad_next.constraints.WithDescription property*), 50
ConstraintError, 51
copy (*datalad_next.iter_collections.GitDiffStatus attribute*), 183
copy_file() (*datalad_next.datasets.Dataset method*), 81
create() (*datalad_next.datasets.Dataset method*), 83
create_info_file() (*datalad_next.tests.BasicGitTestRepo method*), 202
create_sibling() (*datalad_next.datasets.Dataset method*), 85
create_sibling_gin() (*datalad_next.datasets.Dataset method*), 88
create_sibling_gitea() (*datalad_next.datasets.Dataset method*), 90
create_sibling_github() (*datalad_next.datasets.Dataset method*), 92
create_sibling_gitlab() (*datalad_next.datasets.Dataset method*), 95
create_sibling_gogs() (*datalad_next.datasets.Dataset method*), 98
create_sibling_ria() (*datalad_next.datasets.Dataset method*), 100
create_sibling_webdav() (*datalad_next.datasets.Dataset method*), 103
create_sibling_webdav() (*in module datalad.api*), 5
create_tree() (*in module datalad_next.tests*), 204
CredentialManager (*class in datalad_next.credman*), 62
credentials() (*datalad_next.datasets.Dataset method*), 106
credentials() (*in module datalad.api*), 8
credman() (*in module datalad_next.tests.fixtures*), 206

D

datalad_cfg() (*in module datalad_next.tests.fixtures*), 206
datalad_interactive_ui() (*in module datalad_next.tests.fixtures*), 206
datalad_next.annexbackends.base
    module, 230
datalad_next.annexbackends.xdlra
    module, 233
datalad_next.annexremotes.archivist
    module, 236
datalad_next.annexremotes.uncurl
    module, 238

datalad_next.archive_operations
    module, 37
datalad_next.commands
    module, 38
datalad_next.config
    module, 41
datalad_next.constraints
    module, 46
datalad_next.consts
    module, 61
datalad_next.credman
    module, 62
datalad_next.datasets
    module, 67
datalad_next.exceptions
    module, 164
datalad_next.gitremotes.datalad_annex
    module, 226
datalad_next.iter_collections
    module, 174
datalad_next.iterable_subprocess
    module, 166
datalad_next.itertools
    module, 166
datalad_next.patches.annexrepo
    module, 243
datalad_next.patches.cli_configoverrides
    module, 244
datalad_next.patches.commanderror
    module, 244
datalad_next.patches.common_cfg
    module, 244
datalad_next.patches.configuration
    module, 244
datalad_next.patches.create_sibling_ghlike
    module, 247
datalad_next.patches.create_sibling_gitlab
    module, 247
datalad_next.patches.customremotes_main
    module, 247
datalad_next.patches.distribution_dataset
    module, 248
datalad_next.patches.interface_utils
    module, 249
datalad_next.patches.push_optimize
    module, 249
datalad_next.patches.push_to_export_remote
    module, 250
datalad_next.patches.run
    module, 250
datalad_next.patches.siblings
    module, 251
datalad_next.patches.test_keyring
    module, 251
datalad_next.patches.update
    module, 251
datalad_next.repo_utils
    module, 184
datalad_next.runners
    module, 184
datalad_next.shell
    module, 189
datalad_next.tests
    module, 201
datalad_next.tests.fixtures
    module, 206
datalad_next.types
    module, 209
datalad_next.uis
    module, 211
datalad_next.uis.ansi_colors
    module, 211
datalad_next.url_operations
    module, 211
datalad_next.utils
    module, 219
datalad_noninteractive_ui() (*in module datalad_next.tests.fixtures*), 206
DataladAuth (*class in datalad_next.utils*), 220
DataladRepoAnnexBackend (*class in datalad_next.annexbackends.xdlra*), 233
Dataset (*class in datalad_next.datasets*), 67
dataset() (*in module datalad_next.tests.fixtures*), 207
DatasetParameter (*class in datalad_next.constraints*), 52
debug() (*datalad_next.annexbackends.base.Master method*), 232
decode_bytes() (*in module datalad_next.itertools*), 168
DEFAULT_BRANCH (*in module datalad_next.tests*), 203
DEFAULT_REMOTE (*in module datalad_next.tests*), 203
delete() (*datalad_next.url_operations.AnyUrlOperations method*), 215
delete() (*datalad_next.url_operations.FileUrlOperations method*), 216
delete() (*datalad_next.url_operations.UrlOperations method*), 212
delete() (*in module datalad_next.shell.operations.posix*), 198
deletion (*datalad_next.iter_collections.GitDiffStatus attribute*), 183
description (*datalad_next.constraints.ParameterConstraintContext attribute*), 51
diff() (*datalad_next.datasets.Dataset method*), 109
diff_state (*datalad_next.commands.status.StatusResult attribute*), 40
directory (*datalad_next.iter_collections.FileSystemItemType attribute*), 182

directory (*datalad_next.iter_collections.GitTreeItemType attribute*), 182
do_CANVERIFY() (*datalad_next.annexbackends.base.Protocol method*), 233
do_ERROR() (*datalad_next.annexbackends.base.Protocol method*), 233
do_GENKEY() (*datalad_next.annexbackends.base.Protocol method*), 233
do_GETVERSION() (*datalad_next.annexbackends.base.Protocol method*), 233
do_ISCRYPTOGRAPHICALLYSECURE() (*datalad_next.annexbackends.base.Protocol method*), 233
do_ISSTABLE() (*datalad_next.annexbackends.base.Protocol method*), 233
do_VERIFYKEYCONTENT() (*datalad_next.annexbackends.base.Protocol method*), 233
download() (*datalad_next.datasets.Dataset method*), 111
download() (*datalad_next.url_operations.AnyUrlOperations method*), 215
download() (*datalad_next.url_operations.FileUrlOperations method*), 216
download() (*datalad_next.url_operations.HttpUrlOperations method*), 217
download() (*datalad_next.url_operations.SshUrlOperations method*), 218
download() (*datalad_next.url_operations.UrlOperations method*), 213
download() (*in module datalad.api*), 11
download() (*in module datalad_next.shell.operations.posix*), 197
download_url() (*datalad_next.datasets.Dataset method*), 113
DownloadResponseGenerator (*class in datalad_next.shell*), 196
DownloadResponseGeneratorPosix (*class in datalad_next.shell*), 196
drop() (*datalad_next.datasets.Dataset method*), 114

E

emit() (*datalad_next.patches.customremotes_main.AnnexProgressLogHandler method*), 248
ensure_list() (*in module datalad_next.utils*), 222
EnsureBool (*class in datalad_next.constraints*), 53
EnsureCallable (*class in datalad_next.constraints*), 53
EnsureChoice (*class in datalad_next.constraints*), 53
EnsureCommandParameterization (*class in datalad_next.constraints*), 59
EnsureDataset (*class in datalad_next.constraints*), 52
EnsureDType (*class in datalad_next.constraints*), 54
EnsureFloat (*class in datalad_next.constraints*), 53
EnsureGeneratorFromFileLike (*class in datalad_next.constraints*), 57
EnsureGitRefName (*class in datalad_next.constraints*), 58
EnsureHashAlgorithm (*class in datalad_next.constraints*), 54
EnsureInt (*class in datalad_next.constraints*), 54
EnsureIterableOf (*class in datalad_next.constraints*), 56
EnsureJSON (*class in datalad_next.constraints*), 57
EnsureKeyChoice (*class in datalad_next.constraints*), 54
EnsureListOf (*class in datalad_next.constraints*), 56
EnsureMapping (*class in datalad_next.constraints*), 57
EnsureNone (*class in datalad_next.constraints*), 55
EnsureParsedURL (*class in datalad_next.constraints*), 58
EnsurePath (*class in datalad_next.constraints*), 55
EnsureRange (*class in datalad_next.constraints*), 56
EnsureRemoteName (*class in datalad_next.constraints*), 59
EnsureSiblingName (*class in datalad_next.constraints*), 59
EnsureStr (*class in datalad_next.constraints*), 55
EnsureStrPrefix (*class in datalad_next.constraints*), 55
EnsureTupleOf (*class in datalad_next.constraints*), 57
EnsureURL (*class in datalad_next.constraints*), 58
EnsureValue (*class in datalad_next.constraints*), 56
eq_() (*in module datalad_next.tests*), 204
error (*datalad_next.commands.CommandResultStatus attribute*), 40
error() (*datalad_next.annexbackends.base.Backend method*), 230
error() (*datalad_next.annexbackends.base.Master method*), 232
error_message (*datalad_next.commands.CommandResult attribute*), 39
exception (*datalad_next.commands.CommandResult attribute*), 39
executablefile (*datalad_next.iter_collections.GitTreeItemType attribute*), 182
existing_dataset() (*in module datalad_next.tests.fixtures*), 207
existing_noannex_dataset() (*in module datalad_next.tests.fixtures*), 207
export_archive() (*datalad_next.datasets.Dataset method*), 117
export_archive_ora() (*datalad_next.datasets.Dataset method*), 118

export_to_figshare() (*datalad_next.datasets.Dataset method*), 119
external_versions (*in module datalad_next.utils*), 222
extract_tmpl_props() (*datalad_next.annexremotes.uncurl.UncurlRemote method*), 241

F

file (*datalad_next.iter_collections.FileSystemItemType attribute*), 182
file (*datalad_next.iter_collections.GitTreeItemType attribute*), 182
FileSystemItem (*class in datalad_next.iter_collections*), 181
FileSystemItemType (*class in datalad_next.iter_collections*), 182
FileUrlOperations (*class in datalad_next.url_operations*), 216
FixedLengthResponseGenerator (*class in datalad_next.shell*), 195
FixedLengthResponseGeneratorPosix (*class in datalad_next.shell*), 196
FixedLengthResponseGeneratorPowerShell (*class in datalad_next.shell*), 196
for_dataset() (*datalad_next.constraints.Constraint method*), 48
for_dataset() (*datalad_next.constraints.EnsureMapping method*), 57
for_dataset() (*datalad_next.constraints.EnsurePath method*), 55
for_dataset() (*datalad_next.constraints.EnsureRemoteName method*), 59
for_dataset() (*datalad_next.constraints.WithDescription method*), 50
foreach_dataset() (*datalad_next.datasets.Dataset method*), 120
format_command() (*in module datalad_next.patches.run*), 250
format_oneline_tb() (*datalad_next.exceptions.CapturedException method*), 165
format_short() (*datalad_next.exceptions.CapturedException method*), 165
format_standard() (*datalad_next.exceptions.CapturedException method*), 165
format_with_cause() (*datalad_next.exceptions.CapturedException method*), 165
fp (*datalad_next.iter_collections.FileSystemItem attribute*), 181
from_path() (*datalad_next.iter_collections.FileSystemItem class method*), 181
from_str() (*datalad_next.types.AnnexKey class method*), 209
from_str() (*datalad_next.types.ArchivistLocator class method*), 210

G

gen_key() (*datalad_next.annexbackends.base.Backend method*), 231
gen_key() (*datalad_next.annexbackends.xdlra.DataladRepoAnnexBacken method*), 234
get() (*datalad_next.commands.CommandResult method*), 39
get() (*datalad_next.config.ConfigManager method*), 43
get() (*datalad_next.credman.CredentialManager method*), 62
get() (*datalad_next.datasets.Dataset method*), 123
get_allargs_as_kwargs() (*in module datalad_next.patches.interface_utils*), 249
get_deeply_nested_structure() (*in module datalad_next.tests*), 204
get_export_records() (*in module datalad_next.patches.push_to_export_remote*), 250
get_final_command() (*datalad_next.shell.DownloadResponseGeneratorPosix method*), 196
get_final_command() (*datalad_next.shell.FixedLengthResponseGeneratorPosix method*), 196
get_final_command() (*datalad_next.shell.FixedLengthResponseGeneratorPowerShell method*), 196
get_final_command() (*datalad_next.shell.ShellCommandResponseGenerator method*), 194
get_final_command() (*datalad_next.shell.VariableLengthResponseGeneratorPosix method*), 195
get_final_command() (*datalad_next.shell.VariableLengthResponseGeneratorPowerShell method*), 195
get_from_source() (*datalad_next.config.ConfigManager method*), 43
get_headers() (*datalad_next.url_operations.HttpUrlOperations method*), 217
get_hexdigest() (*datalad_next.utils.MultiHash method*), 221

get_key_urls() (datalad_next.annexremotes.uncurl.UncurlRemote method), 241
get_label_with_parameter_values() (datalad_next.constraints.ParameterConstraintContext method), 52
get_mangled_url() (datalad_next.annexremotes.uncurl.UncurlRemote method), 241
get_mirror_refs() (datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote method), 228
get_parameter_validator() (datalad_next.commands.ValidatedInterface class method), 41
get_remote_gitcfg() (datalad_next.annexremotes.SpecialRemote method), 235
get_remote_refs() (datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote method), 228
get_specialremote_credential_envpatch() (in module datalad_next.utils), 225
get_specialremote_credential_properties() (in module datalad_next.utils), 224
get_specialremote_param_dict() (in module datalad_next.utils), 224
get_superdataset() (datalad_next.datasets.Dataset method), 126
get_value() (datalad_next.config.ConfigManager method), 43
get_worktree_head() (in module datalad_next.repo_utils), 184
getbool() (datalad_next.config.ConfigManager method), 43
getfloat() (datalad_next.config.ConfigManager method), 43
getint() (datalad_next.config.ConfigManager method), 43
gid (datalad_next.iter_collections.FileSystemItem attribute), 181
GitContainerModificationType (class in datalad_next.iter_collections), 184
GitDiffItem (class in datalad_next.iter_collections), 183
GitDiffStatus (class in datalad_next.iter_collections), 183
GitRunner (in module datalad_next.runners), 188
gitsha (datalad_next.iter_collections.GitWorktreeFileSystemItem attribute), 183
GitTreeItemType (class in datalad_next.iter_collections), 182
gittype (datalad_next.commands.status.StatusResult attribute), 40
gittype (datalad_next.iter_collections.GitWorktreeFileSystemItem attribute), 183
GitWorktreeFileSystemItem (class in datalad_next.iter_collections), 182
GitWorktreeItem (class in datalad_next.iter_collections), 182

H

handle_401() (datalad_next.utils.DataladAuth method), 220
handle_redirect() (datalad_next.utils.DataladAuth method), 221
hardlink (datalad_next.iter_collections.FileSystemItemType attribute), 182
has_option() (datalad_next.config.ConfigManager method), 43
has_section() (datalad_next.config.ConfigManager method), 43
http_credential() (in module datalad_next.tests.fixtures), 207
http_server() (in module datalad_next.tests.fixtures), 207
http_server_with_basicauth() (in module datalad_next.tests.fixtures), 207
httpbin() (in module datalad_next.tests.fixtures), 207
httpbin_service() (in module datalad_next.tests.fixtures), 207
HttpUrlOperations (class in datalad_next.url_operations), 217

I

id (datalad_next.datasets.Dataset property), 126
impossible (datalad_next.commands.CommandResultStatus attribute), 40
IncompleteResultsError, 165
initremote() (datalad_next.annexremotes.archivist.ArchivistRemote method), 237
initremote() (datalad_next.annexremotes.uncurl.UncurlRemote method), 242
input (datalad_next.annexbackends.base.Master attribute), 231
input_description (datalad_next.constraints.Constraint property), 48
input_description (datalad_next.constraints.WithDescription property), 50
input_synopsis (datalad_next.constraints.Constraint property), 48
input_synopsis (datalad_next.constraints.WithDescription property), 50
install() (datalad_next.datasets.Dataset method), 126

internal_parameters (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote attribute*), 229
is_cryptographically_secure() (*datalad_next.annexbackends.base.Backend method*), 231
is_cryptographically_secure() (*datalad_next.annexbackends.xdlra.DataladRepoAnnexBackend method*), 234
is_installed() (*datalad_next.datasets.Dataset method*), 129
is_recognized_url() (*datalad_next.annexremotes.uncurl.UncurlRemote method*), 242
is_stable() (*datalad_next.annexbackends.base.Backend method*), 231
is_stable() (*datalad_next.annexbackends.xdlra.DataladRepoAnnexBackend method*), 234
is_supported_url() (*datalad_next.url_operations.AnyUrlOperations method*), 215
item_constraint (*datalad_next.constraints.EnsureIterableOf property*), 56
itemize() (*in module datalad_next.itertools*), 169
items() (*datalad_next.commands.CommandResult method*), 39
items() (*datalad_next.config.ConfigManager method*), 43
iter_annexworktree() (*in module datalad_next.iter_collections*), 175
iter_dir() (*in module datalad_next.iter_collections*), 176
iter_git_subproc() (*in module datalad_next.runners*), 187
iter_gitdiff() (*in module datalad_next.iter_collections*), 176
iter_gitstatus() (*in module datalad_next.iter_collections*), 177
iter_gittree() (*in module datalad_next.iter_collections*), 178
iter_gitworktree() (*in module datalad_next.iter_collections*), 178
iter_submodules() (*in module datalad_next.iter_collections*), 179
iter_subproc() (*in module datalad_next.runners*), 185
iter_tar() (*in module datalad_next.iter_collections*), 179
iter_zip() (*in module datalad_next.iter_collections*), 180
iterable_subprocess() (*in module datalad_next.iterable_subprocess*), 166

J

joint_validation() (*datalad_next.constraints.EnsureCommandParameterization method*), 60

K

keys() (*datalad_next.config.ConfigManager method*), 43
KillOutput (*class in datalad_next.runners*), 188

L

label (*datalad_next.constraints.ParameterConstraintContext property*), 52
LeanAnnexRepo (*class in datalad_next.datasets*), 164
LeanGitRepo (*in module datalad_next.datasets*), 164
LegacyAnnexRepo (*in module datalad_next.datasets*), 164
LegacyGitRepo (*in module datalad_next.datasets*), 164
link_target (*datalad_next.iter_collections.FileSystemItem attribute*), 181
link_target (*datalad_next.iter_collections.TarfileItem attribute*), 180
link_target_path (*datalad_next.iter_collections.TarfileItem property*), 180
link_target_path() (*datalad_next.iter_collections.FileSystemItem method*), 181
LinkBackend() (*datalad_next.annexbackends.base.Master method*), 232
Listen() (*datalad_next.annexbackends.base.Master method*), 232
load_json() (*in module datalad_next.itertools*), 170
load_json_with_flag() (*in module datalad_next.itertools*), 171
log() (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote method*), 229
log_progress() (*in module datalad_next.utils*), 222
logger (*datalad_next.commands.CommandResult attribute*), 39
long_description() (*datalad_next.constraints.AllOf method*), 49
long_description() (*datalad_next.constraints.AnyOf method*), 49
long_description() (*datalad_next.constraints.Constraint method*), 48
long_description() (*datalad_next.constraints.EnsureBool method*), 53
long_description() (*datalad_next.constraints.EnsureCallable method*), 53

long_description() (*datalad_next.constraints.EnsureChoice method*), 53
long_description() (*datalad_next.constraints.EnsureDType method*), 54
long_description() (*datalad_next.constraints.EnsureKeyChoice method*), 54
long_description() (*datalad_next.constraints.EnsureRange method*), 56
long_description() (*datalad_next.constraints.EnsureStr method*), 55
long_description() (*datalad_next.constraints.EnsureStrPrefix method*), 55
long_description() (*datalad_next.constraints.EnsureValue method*), 56
long_description() (*datalad_next.constraints.WithDescription method*), 50
lookupMethod() (*datalad_next.annexbackends.base.Protocol method*), 233
ls_file_collection() (*in module datalad.api*), 13

M

main() (*in module datalad_next.annexbackends.xdlra*), 234
main() (*in module datalad_next.annexremotes.archivist*), 237
main() (*in module datalad_next.annexremotes.uncurl*), 243
Master (*class in datalad_next.annexbackends.base*), 231
member (*datalad_next.types.ArchivistLocator attribute*), 210
message (*datalad_next.commands.CommandResult attribute*), 39
message (*datalad_next.exceptions.CapturedException property*), 165
mirrorrepo (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote property*), 229
mode (*datalad_next.iter_collections.FileSystemItem attribute*), 181
modification (*datalad_next.iter_collections.GitDiffStatus attribute*), 183
modification_types (*datalad_next.commands.status.StatusResult attribute*), 40
modification_types (*datalad_next.iter_collections.GitDiffItem attribute*), 183
modified_content (*datalad_next.iter_collections.GitContainerModificationType attribute*), 184
modified_dataset() (*in module datalad_next.tests.fixtures*), 207
module
    datalad_next.annexbackends.base, 230
    datalad_next.annexbackends.xdlra, 233
    datalad_next.annexremotes.archivist, 236
    datalad_next.annexremotes.uncurl, 238
    datalad_next.archive_operations, 37
    datalad_next.commands, 38
    datalad_next.config, 41
    datalad_next.constraints, 46
    datalad_next.consts, 61
    datalad_next.credman, 62
    datalad_next.datasets, 67
    datalad_next.exceptions, 164
    datalad_next.gitremotes.datalad_annex, 226
    datalad_next.iter_collections, 174
    datalad_next.iterable_subprocess, 166
    datalad_next.itertools, 166
    datalad_next.patches.annexrepo, 243
    datalad_next.patches.cli_configoverrides, 244
    datalad_next.patches.commanderror, 244
    datalad_next.patches.common_cfg, 244
    datalad_next.patches.configuration, 244
    datalad_next.patches.create_sibling_ghlike, 247
    datalad_next.patches.create_sibling_gitlab, 247
    datalad_next.patches.customremotes_main, 247
    datalad_next.patches.distribution_dataset, 248
    datalad_next.patches.interface_utils, 249
    datalad_next.patches.push_optimize, 249
    datalad_next.patches.push_to_export_remote, 250
    datalad_next.patches.run, 250
    datalad_next.patches.siblings, 251
    datalad_next.patches.test_keyring, 251
    datalad_next.patches.update, 251
    datalad_next.repo_utils, 184
    datalad_next.runners, 184
    datalad_next.shell, 189
    datalad_next.tests, 201
    datalad_next.tests.fixtures, 206
    datalad_next.types, 209
    datalad_next.uis, 211
    datalad_next.uis.ansi_colors, 211

datalad_next.url_operations, 211
datalad_next.utils, 219
mtime (*datalad_next.iter_collections.FileSystemItem attribute*), 181
mtime (*datalad_next.types.AnnexKey attribute*), 209
MultiHash (*class in datalad_next.utils*), 221

N

name (*datalad_next.exceptions.CapturedException property*), 165
name (*datalad_next.iter_collections.GitWorktreeFileSystemItem attribute*), 183
name (*datalad_next.iter_collections.GitWorktreeItem attribute*), 182
name (*datalad_next.iter_collections.TarfileItem attribute*), 180
name (*datalad_next.iter_collections.ZipfileItem attribute*), 181
name (*datalad_next.types.AnnexKey attribute*), 210
needs_specialremote_credential_envpatch() (*in module datalad_next.utils*), 225
new_commits (*datalad_next.iter_collections.GitContainerModificationType attribute*), 184
next_status() (*datalad_next.datasets.Dataset method*), 129
next_status() (*in module datalad.api*), 15
no_annex() (*datalad_next.datasets.Dataset method*), 131
no_result_rendering() (*in module datalad_next.tests.fixtures*), 208
NoCapture (*class in datalad_next.runners*), 189
NoConstraint (*class in datalad_next.constraints*), 49
NoDatasetFound, 166
NotLinkedError, 232
notneeded (*datalad_next.commands.CommandResultStatus attribute*), 40

O

obtain() (*datalad_next.config.ConfigManager method*), 43
obtain() (*datalad_next.credman.CredentialManager method*), 63
ok (*datalad_next.commands.CommandResultStatus attribute*), 40
ok_() (*in module datalad_next.tests*), 204
ok_broken_symlink() (*in module datalad_next.tests*), 204
ok_good_symlink() (*in module datalad_next.tests*), 204
only_progress_logrecords() (*in module datalad_next.patches.customremotes_main*), 248
open() (*datalad_next.archive_operations.TarArchiveOperations method*), 37
open() (*datalad_next.archive_operations.ZipArchiveOperations method*), 38
options() (*datalad_next.config.ConfigManager method*), 44
other (*datalad_next.iter_collections.GitDiffStatus attribute*), 184
output (*datalad_next.annexbackends.base.Master attribute*), 232

P

ParamDictator (*class in datalad_next.utils*), 225
ParameterConstraintContext (*class in datalad_next.constraints*), 51
parameters (*datalad_next.constraints.ParameterConstraintContext attribute*), 52
parse_overrides_from_cmdline() (*in module datalad_next.patches.cli_configoverrides*), 244
parse_www_authenticate() (*in module datalad_next.utils*), 223
patched_env() (*in module datalad_next.utils*), 224
patched_underscore_main() (*in module datalad_next.patches.customremotes_main*), 248
path (*datalad_next.commands.CommandResult attribute*), 39
path (*datalad_next.datasets.Dataset property*), 132
path (*datalad_next.iter_collections.TarfileItem property*), 180
path (*datalad_next.iter_collections.ZipfileItem property*), 181
pathobj (*datalad_next.datasets.Dataset property*), 132
percentage (*datalad_next.iter_collections.GitDiffItem attribute*), 183
pipe_data_received() (*datalad_next.runners.KillOutput method*), 188
pop() (*datalad_next.commands.CommandResult method*), 39
populate() (*datalad_next.tests.BasicGitTestRepo method*), 202
prepare() (*datalad_next.annexremotes.archivist.ArchivistRemote method*), 237
prepare() (*datalad_next.annexremotes.uncurl.UncurlRemote method*), 242
prev_gitsha (*datalad_next.iter_collections.GitDiffItem attribute*), 183
prev_gittype (*datalad_next.commands.status.StatusResult attribute*), 40
prev_gittype (*datalad_next.iter_collections.GitDiffItem attribute*), 183
prev_name (*datalad_next.iter_collections.GitDiffItem attribute*), 183
prev_path (*datalad_next.iter_collections.GitDiffItem property*), 183
prev_type (*datalad_next.commands.status.StatusResult property*), 40
probe_url() (*datalad_next.url_operations.HttpUrlOperations method*), 217

proc_err (*datalad_next.runners.KillOutput attribute*), 188
proc_err (*datalad_next.runners.StdErrCapture attribute*), 189
proc_err (*datalad_next.runners.StdOutErrCapture attribute*), 189
proc_out (*datalad_next.runners.KillOutput attribute*), 188
proc_out (*datalad_next.runners.StdOutCapture attribute*), 189
proc_out (*datalad_next.runners.StdOutErrCapture attribute*), 189
progress() (*datalad_next.annexbackends.base.Master method*), 232
Protocol (*class in datalad_next.annexbackends.base*), 232
ProtocolError, 233
push() (*datalad_next.datasets.Dataset method*), 133

Q

query() (*datalad_next.credman.CredentialManager method*), 64
query_() (*datalad_next.credman.CredentialManager method*), 65

R

raise_for() (*datalad_next.constraints.Constraint method*), 48
recall_state() (*datalad_next.datasets.Dataset method*), 134
reduce_logging() (*in module datalad_next.tests.fixtures*), 208
refds (*datalad_next.commands.CommandResult attribute*), 39
refs_key (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote attribute*), 229
reload() (*datalad_next.config.ConfigManager method*), 44
remotename (*datalad_next.annexremotes.SpecialRemote property*), 235
remove() (*datalad_next.annexremotes.archivist.ArchivistRemote method*), 237
remove() (*datalad_next.annexremotes.uncurl.UncurlRemote method*), 242
remove() (*datalad_next.credman.CredentialManager method*), 65
remove() (*datalad_next.datasets.Dataset method*), 134
remove_section() (*datalad_next.config.ConfigManager method*), 44
rename (*datalad_next.iter_collections.GitDiffStatus attribute*), 184
rename_section() (*datalad_next.config.ConfigManager method*), 45
replace_mirrorrepo_from_remote_deposit() (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote method*), 229
replace_mirrorrepo_from_remote_deposit_if_needed() (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote method*), 229
replace_remote_deposit_from_mirrorrepo() (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote method*), 229
repo (*datalad_next.annexremotes.SpecialRemote property*), 235
repo (*datalad_next.datasets.Dataset property*), 137
REPO_CLASS (*datalad_next.tests.BasicGitTestRepo attribute*), 202
repo_export_key (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote attribute*), 229
repoannex (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote property*), 229
RepoAnnexGitRemote (*class in datalad_next.gitremotes.datalad_annex*), 228
rerun() (*datalad_next.datasets.Dataset method*), 137
resolve_path() (*in module datalad_next.patches.distribution_dataset*), 248
rewrite_url() (*datalad_next.config.ConfigManager method*), 45
rmtree() (*in module datalad_next.utils*), 224
route_in() (*in module datalad_next.itertools*), 173
route_out() (*in module datalad_next.itertools*), 171
run() (*datalad_next.datasets.Dataset method*), 139
run_main() (*in module datalad_next.tests*), 204
run_procedure() (*datalad_next.datasets.Dataset method*), 142
Runner (*in module datalad_next.runners*), 188

S

safe_content (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRem attribute*), 229
save() (*datalad_next.datasets.Dataset method*), 145
save_entered_credential() (*datalad_next.utils.DataladAuth method*), 221
secret_names (*datalad_next.credman.CredentialManager attribute*), 65
sections() (*datalad_next.config.ConfigManager method*), 45
send() (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote method*), 230
send() (*datalad_next.shell.DownloadResponseGenerator method*), 196
send() (*datalad_next.shell.FixedLengthResponseGenerator method*), 195
send() (*datalad_next.shell.ShellCommandResponseGenerator method*), 194

send() (*datalad_next.shell.VariableLengthResponseGenerator method*), 194
set() (*datalad_next.config.ConfigManager method*), 45
set() (*datalad_next.credman.CredentialManager method*), 65
shell() (*in module datalad_next.shell*), 198
ShellCommandExecutor (*class in datalad_next.shell*), 192
ShellCommandResponseGenerator (*class in datalad_next.shell*), 194
short_description() (*datalad_next.constraints.AllOf method*), 49
short_description() (*datalad_next.constraints.AnyOf method*), 49
short_description() (*datalad_next.constraints.Constraint method*), 48
short_description() (*datalad_next.constraints.EnsureBool method*), 53
short_description() (*datalad_next.constraints.EnsureCallable method*), 53
short_description() (*datalad_next.constraints.EnsureChoice method*), 53
short_description() (*datalad_next.constraints.EnsureDataset method*), 52
short_description() (*datalad_next.constraints.EnsureDType method*), 54
short_description() (*datalad_next.constraints.EnsureGeneratorFromFileLike method*), 57
short_description() (*datalad_next.constraints.EnsureGitRefName method*), 58
short_description() (*datalad_next.constraints.EnsureIterableOf method*), 56
short_description() (*datalad_next.constraints.EnsureJSON method*), 57
short_description() (*datalad_next.constraints.EnsureKeyChoice method*), 54
short_description() (*datalad_next.constraints.EnsureListOf method*), 56
short_description() (*datalad_next.constraints.EnsureMapping method*), 57
short_description() (*datalad_next.constraints.EnsurePath method*), 55
short_description() (*datalad_next.constraints.EnsureRange method*), 56
short_description() (*datalad_next.constraints.EnsureRemoteName method*), 59
short_description() (*datalad_next.constraints.EnsureStr method*), 55
short_description() (*datalad_next.constraints.EnsureStrPrefix method*), 55
short_description() (*datalad_next.constraints.EnsureTupleOf method*), 57
short_description() (*datalad_next.constraints.EnsureURL method*), 58
short_description() (*datalad_next.constraints.EnsureValue method*), 56
short_description() (*datalad_next.constraints.NoConstraint method*), 49
short_description() (*datalad_next.constraints.WithDescription method*), 50
siblings() (*datalad_next.datasets.Dataset method*), 147
size (*datalad_next.iter_collections.FileSystemItem attribute*), 181
size (*datalad_next.types.AnnexKey attribute*), 210
size (*datalad_next.types.ArchivistLocator attribute*), 210
skip_if_on_windows() (*in module datalad_next.tests*), 205
skip_if_root() (*in module datalad_next.tests*), 205
skip_wo_symlink_capability() (*in module datalad_next.tests*), 205
skipif_no_network (*in module datalad_next.tests*), 205
specialfile (*datalad_next.iter_collections.FileSystemItemType attribute*), 182
SpecialRemote (*class in datalad_next.annexremotes*), 235
specialremote_defaultclose_noop() (*in module datalad_next.patches.customremotes_main*), 248
sshserver() (*in module datalad_next.tests.fixtures*), 209
sshserver_setup() (*in module datalad_next.tests.fixtures*), 209
SshUrlOperations (*class in datalad_next.url_operations*), 218

start() (*datalad_next.shell.ShellCommandExecutor method*), 193
stat() (*datalad_next.url_operations.AnyUrlOperations method*), 216
stat() (*datalad_next.url_operations.FileUrlOperations method*), 216
stat() (*datalad_next.url_operations.HttpUrlOperations method*), 218
stat() (*datalad_next.url_operations.SshUrlOperations method*), 218
stat() (*datalad_next.url_operations.UrlOperations method*), 213
state (*datalad_next.commands.status.StatusResult property*), 40
status (*datalad_next.commands.CommandResult attribute*), 39
status (*datalad_next.iter_collections.GitDiffItem attribute*), 183
status() (*datalad_next.datasets.Dataset method*), 149
StatusResult (*class in datalad_next.commands.status*), 40
StdErrCapture (*class in datalad_next.runners*), 189
StdOutCapture (*class in datalad_next.runners*), 189
StdOutErrCapture (*class in datalad_next.runners*), 189
subdatasets() (*datalad_next.datasets.Dataset method*), 152
submodule (*datalad_next.iter_collections.GitTreeItemType attribute*), 182
support_githelper_options (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote attribute*), 230
swallow_logs() (*in module datalad_next.tests*), 205
symlink (*datalad_next.iter_collections.FileSystemItemType attribute*), 182
symlink (*datalad_next.iter_collections.GitTreeItemType attribute*), 182

T

tar (*datalad_next.types.ArchiveType attribute*), 211
TarArchiveOperations (*class in datalad_next.archive_operations*), 37
tarfile (*datalad_next.archive_operations.TarArchiveOperations property*), 37
TarfileItem (*class in datalad_next.iter_collections*), 180
throw() (*datalad_next.shell.ShellCommandResponseGenerator method*), 194
tmp_keyring() (*in module datalad_next.tests.fixtures*), 209
transfer_retrieve() (*datalad_next.annexremotes.archivist.ArchivistRemote method*), 237
transfer_retrieve() (*datalad_next.annexremotes.uncurl.UncurlRemote method*), 242
transfer_store() (*datalad_next.annexremotes.archivist.ArchivistRemote method*), 237
transfer_store() (*datalad_next.annexremotes.uncurl.UncurlRemote method*), 242
tree() (*datalad_next.datasets.Dataset method*), 155
tree() (*in module datalad.api*), 17
type (*datalad_next.commands.CommandResult attribute*), 39
type (*datalad_next.commands.status.StatusResult property*), 40
type (*datalad_next.iter_collections.FileSystemItem attribute*), 181
type_src (*datalad_next.commands.status.StatusResult property*), 40
typechange (*datalad_next.iter_collections.GitDiffStatus attribute*), 184

U

ui_switcher (*in module datalad_next.uis*), 211
uid (*datalad_next.iter_collections.FileSystemItem attribute*), 181
UncurlRemote (*class in datalad_next.annexremotes.uncurl*), 241
UnexpectedMessage, 233
uninstall() (*datalad_next.datasets.Dataset method*), 158
unknown (*datalad_next.iter_collections.GitDiffStatus attribute*), 184
unlock() (*datalad_next.datasets.Dataset method*), 159
unmerged (*datalad_next.iter_collections.GitDiffStatus attribute*), 184
unset() (*datalad_next.config.ConfigManager method*), 46
UnsupportedRequest, 233
untracked_content (*datalad_next.iter_collections.GitContainerModificationType attribute*), 184
update() (*datalad_next.datasets.Dataset method*), 160
update() (*datalad_next.utils.MultiHash method*), 221
update_specialremote_credential() (*in module datalad_next.utils*), 225
upload() (*datalad_next.url_operations.AnyUrlOperations method*), 216
upload() (*datalad_next.url_operations.FileUrlOperations method*), 216
upload() (*datalad_next.url_operations.SshUrlOperations method*), 218
upload() (*datalad_next.url_operations.UrlOperations method*), 214
upload() (*in module datalad_next.shell.operations.posix*), 197

UrlOperations (*class in datalad_next.url_operations*), 212
UrlOperationsAuthenticationError, 219
UrlOperationsAuthorizationError, 219
UrlOperationsInteractionError, 219
UrlOperationsRemoteError, 219
UrlOperationsResourceUnknown, 219

V

valid_property_names_regex (*datalad_next.credman.CredentialManager attribute*), 66
ValidatedInterface (*class in datalad_next.commands*), 40
VariableLengthResponseGenerator (*class in datalad_next.shell*), 194
VariableLengthResponseGeneratorPosix (*class in datalad_next.shell*), 195
VariableLengthResponseGeneratorPowerShell (*class in datalad_next.shell*), 195
verify_content() (*datalad_next.annexbackends.base.Backend method*), 231
verify_content() (*datalad_next.annexbackends.xdlra.DataladRepoAnnexBackend method*), 234
verify_property_names() (*in module datalad_next.credman*), 67

W

webdav_credential() (*in module datalad_next.tests.fixtures*), 209
webdav_server() (*in module datalad_next.tests.fixtures*), 209
WithDescription (*class in datalad_next.constraints*), 50
wtf() (*datalad_next.datasets.Dataset method*), 162

X

xdlra_key_locations (*datalad_next.gitremotes.datalad_annex.RepoAnnexGitRemote attribute*), 230

Z

zero_command (*datalad_next.shell.VariableLengthResponseGenerator property*), 194
zero_command (*datalad_next.shell.VariableLengthResponseGeneratorPosix property*), 195
zero_command (*datalad_next.shell.VariableLengthResponseGeneratorPowerShell property*), 195
zip (*datalad_next.types.ArchiveType attribute*), 211
ZipArchiveOperations (*class in datalad_next.archive_operations*), 38
zipfile (*datalad_next.archive_operations.ZipArchiveOperations property*), 38
ZipfileItem (*class in datalad_next.iter_collections*), 181