
Git Interface

Release 0.8.2

Leo Spratt

Mar 25, 2022

CONTENTS

1 Requirements	3
2 Contents	5
Python Module Index	19
Index	21

Use the git cli from Python.

Attention: This project currently is not heavily tested and not fully feature complete

REQUIREMENTS

Git (version 2.30)

CONTENTS

2.1 Quick Start

Get Branches:

```
import asyncio
from git_interface.branch import get_branches

head, other_branches = asyncio.run(get_branches("my_git_repo.git"))

print("HEAD = ", head)
print("OTHER", other_branches)
```

2.2 Reference

2.2.1 git_interface.smart_http

git_interface.smart_http.quart

Smart HTTP Git helpers for quart

async `git_interface.smart_http.quart.get_info_refs_response(repo_path, pack_type)`

Make the response for handling advertisements.

A matching route should be: `'/<repo_name>.git/info/refs'`, accessing the `'service'` argument for `pack_type`.

param repo_path Path to the repo

param pack_type The pack-type

return The created response

Return type `quart.Response`

async `git_interface.smart_http.quart.post_pack_response(repo_path, pack_type)`

Make the response for handling exchange pack responses, uses `'BODY_TIMEOUT'` for a timeout of a request.

A matching route should be: `'/<repo_name>.git/<pack_type>'`.

param repo_path Path to the repo

param pack_type The pack-type

return The created response

Parameters

- **repo_path** (*pathlib.Path*) –
- **pack_type** (*str*) –

Return type `quart.Response`

2.2.2 `git_interface.branch`

Methods for using the ‘branch’ command

async `git_interface.branch.copy_branch`(*git_repo, branch_name, new_branch*)

Copy an existing branch to a new branch in repo (uses `-force`)

param `git_repo` Path to the repo

param `branch_name` Branch name

param `new_branch` The new branch name

raises `NoBranchesException` Branch does not exist

raises `GitException` Error to do with git

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **branch_name** (*str*) –
- **new_branch** (*str*) –

async `git_interface.branch.delete_branch`(*git_repo, branch_name*)

Delete an existing branch (uses `-force`)

param `git_repo` Path to the repo

param `branch_name` Branch name

raises `NoBranchesException` Branch does not exist

raises `GitException` Error to do with git

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **branch_name** (*str*) –

async `git_interface.branch.get_branches`(*git_repo*)

Get the head branch and all others

param `git_repo` Path to the repo

raises `GitException` Error to do with git

raises `NoBranchesException` Repo has no branches

return the head branch and other branches

Parameters `git_repo` (*Union[pathlib.Path, str]*) –

Return type `collections.abc.Coroutine[Any, Any, tuple[str, tuple[str]]]`

async `git_interface.branch.new_branch(git_repo, branch_name)`

Create a new branch in repo

param `git_repo` Path to the repo

param `branch_name` Branch name

raises `AlreadyExistsException` Branch already exists

raises `GitException` Error to do with git

Parameters

- `git_repo` (`Union[pathlib.Path, str]`) –
- `branch_name` (`str`) –

async `git_interface.branch.rename_branch(git_repo, branch_name, new_branch)`

Rename an existing branch (uses `-force`)

param `git_repo` Path to the repo

param `branch_name` Branch name

param `new_branch` The new branch name

raises `NoBranchesException` Branch does not exist

raises `GitException` Error to do with git

Parameters

- `git_repo` (`Union[pathlib.Path, str]`) –
- `branch_name` (`str`) –
- `new_branch` (`str`) –

2.2.3 git_interface.cat_file

Methods for using the ‘cat-file’ command

async `git_interface.cat_file.get_object_size(git_repo, tree_ish, file_path)`

Gets the objects size from repo

param `git_repo` Path to the repo

param `tree_ish` The tree ish (branch name, HEAD)

param `file_path` The file in the repo to read

raises `UnknownRevisionException` Invalid `tree_ish` or `file_path`

raises `GitException` Error to do with git

return The object size

Parameters

- `git_repo` (`Union[pathlib.Path, str]`) –
- `tree_ish` (`str`) –

- **file_path** (*str*) –

Return type int

async `git_interface.cat_file.get_object_type(git_repo, tree_ish, file_path)`

Gets the object type from repo

param `git_repo` Path to the repo

param `tree_ish` The tree ish (branch name, HEAD)

param `file_path` The file in the repo to read

raises `UnknownRevisionException` Invalid `tree_ish` or `file_path`

raises `GitException` Error to do with git

return The object type

Parameters

- **git_repo** (`Union[pathlib.Path, str]`) –

- **tree_ish** (*str*) –

- **file_path** (*str*) –

Return type `collections.abc.Coroutine[Any, Any, git_interface.datatypes.TreeContentTypes]`

async `git_interface.cat_file.get_pretty_print(git_repo, tree_ish, file_path)`

Gets a object from repo

param `git_repo` Path to the repo

param `tree_ish` The tree ish (branch name, HEAD)

param `file_path` The file in the repo to read

raises `UnknownRevisionException` Invalid `tree_ish` or `file_path`

raises `GitException` Error to do with git

return The object type

Parameters

- **git_repo** (`Union[pathlib.Path, str]`) –

- **tree_ish** (*str*) –

- **file_path** (*str*) –

Return type `collections.abc.Coroutine[Any, Any, bytes]`

2.2.4 git_interface.datatypes

Custom types that are used

class git_interface.datatypes.**ArchiveTypes**(*value*)

Bases: enum.Enum

Possible archive types

TAR = 'tar'

TAR_GZ = 'tar.gz'

ZIP = 'zip'

class git_interface.datatypes.**Log**(*commit_hash*, *parent_hash*, *author_email*, *author_name*, *commit_date*, *subject*)

Bases: object

Represents a single git log

Parameters

- **commit_hash** (*str*) –
- **parent_hash** (*str*) –
- **author_email** (*str*) –
- **author_name** (*str*) –
- **commit_date** (*datetime.datetime*) –
- **subject** (*str*) –

Return type None

author_email: str

author_name: str

commit_date: datetime.datetime

commit_hash: str

parent_hash: str

subject: str

2.2.5 git_interface.exceptions

Exceptions that could be raised during one of the git commands

exception git_interface.exceptions.**AlreadyExistsException**

Bases: [git_interface.exceptions.GitException](#)

Raised when something already exists could be repository, branch, etc

exception git_interface.exceptions.**BufferedProcessError**

Bases: Exception

Exception raised when non-zero return code is found

exception git_interface.exceptions.**DoesNotExistException**

Bases: [git_interface.exceptions.GitException](#)

Raised when something does not exist e.g. tag

exception `git_interface.exceptions.GitException`

Bases: `Exception`

Parent exception for all git exceptions, used when there is no other exception that fits error

exception `git_interface.exceptions.NoBranchesException`

Bases: `git_interface.exceptions.GitException`

Raised when a repository has no branches or none that match a filter

exception `git_interface.exceptions.NoCommitsException`

Bases: `git_interface.exceptions.GitException`

Raised when a repository has commits

exception `git_interface.exceptions.NoLogsException`

Bases: `git_interface.exceptions.GitException`

Raised when a repository has no logs available

exception `git_interface.exceptions.PathDoesNotExistInRevException`

Bases: `git_interface.exceptions.GitException`

Raised when a path does not exist in a repository

exception `git_interface.exceptions.UnknownRefException`

Bases: `git_interface.exceptions.GitException`

Raised when a a reference is not found

exception `git_interface.exceptions.UnknownRevisionException`

Bases: `git_interface.exceptions.GitException`

Raised when a revision is not found

2.2.6 `git_interface.log`

Methods for using the 'log' command

async `git_interface.log.get_logs`(*git_repo*, *branch=None*, *max_number=None*, *since=None*, *until=None*)

Generate git logs from a repo

param `git_repo` Path to the repo

param `branch` The branch name, defaults to None

param `max_number` max number of logs to get, defaults to None

param `since` Filter logs after given date, defaults to None

param `until` Filter logs before given date defaults to None

raises `NoCommitsException` Repo has no commits

raises `UnknownRevisionException` Unknown revision/branch name

raises `GitException` Error to do with git

raises `NoLogsException` No logs have been generated

return The generated logs

Parameters

- `git_repo` (`pathlib.Path`) –

- **branch** (*Optional[str]*) –
- **max_number** (*Optional[int]*) –
- **since** (*Optional[datetime.datetime]*) –
- **until** (*Optional[datetime.datetime]*) –

Return type `collections.abc.Coroutine[Any, Any, collections.abc.Iterator[git_interface.datatypes.Log]]`

2.2.7 git_interface.ls

Methods for using the ‘ls-tree’ command

async `git_interface.ls.ls_tree(git_repo, tree_ish, recursive, use_long, path=None)`

Get the tree of objects in repo

param git_repo Path to the repo
param tree_ish The tree ish (branch name, HEAD)
param recursive Whether tree is recursive
param use_long Whether to get object sizes
param path Filter path, defaults to None
raises UnknownRevisionException Unknown tree_ish
raises GitException Error to do with git
return The git tree

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **tree_ish** (*str*) –
- **recursive** (*bool*) –
- **use_long** (*bool*) –
- **path** (*Optional[pathlib.Path]*) –

Return type `collections.abc.Coroutine[Any, Any, collections.abc.Iterator[git_interface.datatypes.TreeContent]]`

2.2.8 git_interface.pack

Methods for using commands relating to git packs

`git_interface.pack.advertise_pack(git_repo, pack_type)`

Used to advertise packs between remote and client.

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) – Path to the repo
- **pack_type** (*str*) – The pack-type (‘git-upload-pack’ or ‘git-receive-pack’)

Returns The buffered output stream as a AsyncGenerator

Return type `AsyncGenerator[bytes, None]`

`git_interface.pack.exchange_pack(git_repo, pack_type, input_stream)`

Used to exchange packs between client and remote.

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) – Path to the repo
- **pack_type** (*str*) – The pack-type ('git-upload-pack' or 'git-receive-pack')
- **input_stream** (*AsyncGenerator[bytes, None]*) – The buffered input stream

Returns The buffered output stream as a AsyncGenerator

Return type *AsyncGenerator[bytes, None]*

2.2.9 git_interface.rev_list

Methods for using the 'rev-list' command

`async git_interface.rev_list.get_commit_count(git_repo, branch=None)`

Get a repos commit count

param git_repo Path to the repo

param branch Branch to filter, defaults to None

raises UnknownRevisionException Unknown tree-ish

raises GitException Error to do with git

return The commit count

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **branch** (*Optional[str]*) –

Return type *collections.abc.Coroutine[Any, Any, int]*

`async git_interface.rev_list.get_disk_usage(git_repo, branch=None)`

Get a size of the repo

param git_repo Path to the repo

param branch Branch to filter, defaults to None

raises UnknownRevisionException Unknown tree-ish

raises GitException Error to do with git

return The size of the repo

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **branch** (*Optional[str]*) –

Return type *collections.abc.Coroutine[Any, Any, int]*

`async git_interface.rev_list.get_rev_list(git_repo, branch=None)`

Get a repos revisions

param git_repo Path to the repo

param branch Branch to filter, defaults to None
raises UnknownRevisionException Unknown tree_ish
raises GitException Error to do with git
return The repos revisions

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **branch** (*Optional[str]*) –

Return type collections.abc.Coroutine[*Any, Any*, list[str]]

2.2.10 git_interface.show

Methods for using the ‘show’ command

async git_interface.show.**show_file**(*git_repo, tree_ish, file_path*)
 Read a file from a repository

param git_repo Path to the repo
param tree_ish The tree ish (branch name, HEAD)
param file_path The file in the repo to read
raises UnknownRevisionException Unknown tree_ish
raises PathDoesNotExistInRevException File not found in repo
raises GitException Error to do with git
return The read file

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **tree_ish** (*str*) –
- **file_path** (*str*) –

Return type collections.abc.Coroutine[*Any, Any*, bytes]

async git_interface.show.**show_file_buffered**(*git_repo, tree_ish, file_path*)
 Read a file from a repository, but using a buffered read

param git_repo Path to the repo
param tree_ish The tree ish (branch name, HEAD)
param file_path The file in the repo to read
raises UnknownRevisionException Unknown tree_ish
raises PathDoesNotExistInRevException File not found in repo
raises GitException Error to do with git
yield Each read file section

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **tree_ish** (*str*) –
- **file_path** (*str*) –

Return type collections.abc.AsyncGenerator[bytes, None, None]

2.2.11 git_interface.symbolic_ref

Methods for using git symbolic-ref command

async git_interface.symbolic_ref.**change_active_branch**(*git_repo, branch*)

Change the active (HEAD) reference

param git_repo Path to the repo

param branch The branch name to use

raises UnknownRefException Unknown reference given

raises GitException Error to do with git

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **branch** (*str*) –

async git_interface.symbolic_ref.**change_symbolic_ref**(*git_repo, name, ref*)

Change a symbolic ref in repo

param git_repo Path to the repo

param name The name (for example HEAD)

param ref The reference

raises UnknownRefException Unknown reference given

raises GitException Error to do with git

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **name** (*str*) –
- **ref** (*str*) –

async git_interface.symbolic_ref.**delete_symbolic_ref**(*git_repo, name*)

Delete a symbolic ref in repo

param git_repo Path to the repo

param name The name (for example HEAD)

raises UnknownRefException Unknown reference given

raises GitException Error to do with git

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –

- **name** (*str*) –

async `git_interface.symbolic_ref.get_active_branch(git_repo)`

Get the active (HEAD) reference

param `git_repo` Path to the repo

raises `UnknownRefException` Unknown reference given

raises `GitException` Error to do with git

Parameters `git_repo` (*Union*[*pathlib.Path*, *str*]) –

Return type `collections.abc.Coroutine`[*Any*, *Any*, *str*]

async `git_interface.symbolic_ref.get_symbolic_ref(git_repo, name)`

Get a symbolic ref in repo

param `git_repo` Path to the repo

param `name` The name (for example HEAD)

raises `UnknownRefException` Unknown reference given

raises `GitException` Error to do with git

Parameters

- **git_repo** (*Union*[*pathlib.Path*, *str*]) –

- **name** (*str*) –

Return type `collections.abc.Coroutine`[*Any*, *Any*, *str*]

2.2.12 git_interface.tag

Methods for using the ‘tag’ command

async `git_interface.tag.create_tag(git_repo, tag_name, commit_hash=None)`

Create a new lightweight tag

param `git_repo` Path to the repo

param `tag_name` The tag name to use

param `commit_hash` Create tag on a different commit other than HEAD, defaults to None

raises `AlreadyExistsException` When the tag name already exists

raises `GitException` Error to do with git

Parameters

- **git_repo** (*Union*[*pathlib.Path*, *str*]) –

- **tag_name** (*str*) –

- **commit_hash** (*Optional*[*str*]) –

async `git_interface.tag.delete_tag(git_repo, tag_name)`

Delete a tag

param `git_repo` Path to the repo

param tag_name The tag name to use
raises DoesNotExistException The tag was not found
raises GitException Error to do with git
return Output provided by the git when a tag is removed

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **tag_name** (*str*) –

Return type *collections.abc.Coroutine[Any, Any, str]*

async `git_interface.tag.list_tags(git_repo, tag_pattern=None)`
List all git tags or filter with a wildcard pattern

param git_repo Path to the repo
param tag_pattern Filter the tag list with a wildcard pattern, defaults to None
raises GitException Error to do with git
return List of found git tags

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **tag_pattern** (*Optional[str]*) –

Return type *collections.abc.Coroutine[Any, Any, list[str]]*

2.2.13 git_interface.utils

Methods that don't fit in their own file

async `git_interface.utils.clone_repo(git_repo, src, bare=False, mirror=False)`
Clone an existing repo, please note this method has no way of passing passwords+usernames

param git_repo Repo path to clone into
param src Where to clone from
param bare Use `--bare` git argument, defaults to False
param mirror Use `--mirror` git argument, defaults to False
raises ValueError Both bare and mirror are True
raises GitException Error to do with git

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **src** (*str*) –

async `git_interface.utils.get_archive(git_repo, archive_type, tree_ish='HEAD')`
get a archive of a git repo

param git_repo Where the repo is

param archive_type What archive type will be created
param tree_ish What commit/branch to save, defaults to “HEAD”
raises GitException Error to do with git
return The content of the archive ready to write to a file

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **archive_type** (*git_interface.datatypes.ArchiveTypes*) –
- **tree_ish** (*str*) –

Return type bytes

async `git_interface.utils.get_archive_buffered(git_repo, archive_type, tree_ish='HEAD')`
 get a archive of a git repo, but using a buffered read

param git_repo Where the repo is
param archive_type What archive type will be created
param tree_ish What commit/branch to save, defaults to “HEAD”
raises GitException Error to do with git
yield Each read content section

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **archive_type** (*git_interface.datatypes.ArchiveTypes*) –
- **tree_ish** (*str*) –

Return type *AsyncGenerator*[bytes, None]

async `git_interface.utils.get_description(git_repo)`
 Gets the set description for a repo

param git_repo Path to the repo
return The description

Parameters **git_repo** (*Union[pathlib.Path, str]*) –

Return type *collections.abc.Coroutine*[*Any*, *Any*, str]

async `git_interface.utils.get_version()`
 Gets the git version

raises GitException Error to do with git
return The version

Return type *collections.abc.Coroutine*[*Any*, *Any*, str]

async `git_interface.utils.init_repo(repo_dir, repo_name, bare=True, default_branch=None)`
 Creates a new git repo in the directory with the given name, if bare the repo name will have .git added at the end.

param repo_dir Where the repo will be

param repo_name The name of the repo
param bare Whether the repo is bare, defaults to True
param default_branch The branch name to use, defaults to None
raises AlreadyExistsException A repo already exists
raises GitException Error to do with git

Parameters

- **repo_dir** (*pathlib.Path*) –
- **repo_name** (*str*) –
- **bare** (*bool*) –
- **default_branch** (*Optional[str]*) –

async `git_interface.utils.run_maintenance(git_repo)`
Run a maintenance git command to specified repo

param git_repo Where the repo is
raises GitException Error to do with git

Parameters `git_repo` (*Union[pathlib.Path, str]*) –

async `git_interface.utils.set_description(git_repo, description)`
Sets the set description for a repo

param git_repo Path to the repo

Parameters

- **git_repo** (*Union[pathlib.Path, str]*) –
- **description** (*str*) –

- `genindex`

PYTHON MODULE INDEX

g

- `git_interface.branch`, 6
- `git_interface.cat_file`, 7
- `git_interface.datatypes`, 9
- `git_interface.exceptions`, 9
- `git_interface.log`, 10
- `git_interface.ls`, 11
- `git_interface.pack`, 11
- `git_interface.rev_list`, 12
- `git_interface.show`, 13
- `git_interface.smart_http.quart`, 5
- `git_interface.symbolic_ref`, 14
- `git_interface.tag`, 15
- `git_interface.utils`, 16

A

advertise_pack() (in module *git_interface.pack*), 11
 AlreadyExistsException, 9
 ArchiveTypes (class in *git_interface.datatypes*), 9
 author_email (*git_interface.datatypes.Log* attribute), 9
 author_name (*git_interface.datatypes.Log* attribute), 9

B

BufferedProcessError, 9

C

change_active_branch() (in module *git_interface.symbolic_ref*), 14
 change_symbolic_ref() (in module *git_interface.symbolic_ref*), 14
 clone_repo() (in module *git_interface.utils*), 16
 commit_date (*git_interface.datatypes.Log* attribute), 9
 commit_hash (*git_interface.datatypes.Log* attribute), 9
 copy_branch() (in module *git_interface.branch*), 6
 create_tag() (in module *git_interface.tag*), 15

D

delete_branch() (in module *git_interface.branch*), 6
 delete_symbolic_ref() (in module *git_interface.symbolic_ref*), 14
 delete_tag() (in module *git_interface.tag*), 15
 DoesNotExistException, 9

E

exchange_pack() (in module *git_interface.pack*), 11

G

get_active_branch() (in module *git_interface.symbolic_ref*), 15
 get_archive() (in module *git_interface.utils*), 16
 get_archive_buffered() (in module *git_interface.utils*), 17
 get_branches() (in module *git_interface.branch*), 6
 get_commit_count() (in module *git_interface.rev_list*), 12
 get_description() (in module *git_interface.utils*), 17

get_disk_usage() (in module *git_interface.rev_list*), 12
 get_info_refs_response() (in module *git_interface.smart_http.quart*), 5
 get_logs() (in module *git_interface.log*), 10
 get_object_size() (in module *git_interface.cat_file*), 7
 get_object_type() (in module *git_interface.cat_file*), 8
 get_pretty_print() (in module *git_interface.cat_file*), 8
 get_rev_list() (in module *git_interface.rev_list*), 12
 get_symbolic_ref() (in module *git_interface.symbolic_ref*), 15
 get_version() (in module *git_interface.utils*), 17
 git_interface.branch module, 6
 git_interface.cat_file module, 7
 git_interface.datatypes module, 9
 git_interface.exceptions module, 9
 git_interface.log module, 10
 git_interface.ls module, 11
 git_interface.pack module, 11
 git_interface.rev_list module, 12
 git_interface.show module, 13
 git_interface.smart_http.quart module, 5
 git_interface.symbolic_ref module, 14
 git_interface.tag module, 15
 git_interface.utils module, 16
 GitException, 10

I

`init_repo()` (in module `git_interface.utils`), 17

L

`list_tags()` (in module `git_interface.tag`), 16

`Log` (class in `git_interface.datatypes`), 9

`ls_tree()` (in module `git_interface.ls`), 11

M

module

- `git_interface.branch`, 6
- `git_interface.cat_file`, 7
- `git_interface.datatypes`, 9
- `git_interface.exceptions`, 9
- `git_interface.log`, 10
- `git_interface.ls`, 11
- `git_interface.pack`, 11
- `git_interface.rev_list`, 12
- `git_interface.show`, 13
- `git_interface.smart_http.quart`, 5
- `git_interface.symbolic_ref`, 14
- `git_interface.tag`, 15
- `git_interface.utils`, 16

N

`new_branch()` (in module `git_interface.branch`), 7

`NoBranchesException`, 10

`NoCommitsException`, 10

`NoLogsException`, 10

P

`parent_hash` (`git_interface.datatypes.Log` attribute), 9

`PathDoesNotExistInRevException`, 10

`post_pack_response()` (in module `git_interface.smart_http.quart`), 5

R

`rename_branch()` (in module `git_interface.branch`), 7

`run_maintenance()` (in module `git_interface.utils`), 18

S

`set_description()` (in module `git_interface.utils`), 18

`show_file()` (in module `git_interface.show`), 13

`show_file_buffered()` (in module `git_interface.show`), 13

`subject` (`git_interface.datatypes.Log` attribute), 9

T

`TAR` (`git_interface.datatypes.ArchiveTypes` attribute), 9

`TAR_GZ` (`git_interface.datatypes.ArchiveTypes` attribute), 9

U

`UnknownRefException`, 10

`UnknownRevisionException`, 10

Z

`ZIP` (`git_interface.datatypes.ArchiveTypes` attribute), 9