GitPython Documentation

Release 3.1.43

Michael Trier

Mar 31, 2024

CONTENTS

CHAPTER

ONE

OVERVIEW / INSTALL

GitPython is a python library used to interact with git repositories, high-level like git-porcelain, or low-level like git-plumbing.

It provides abstractions of git objects for easy access of repository data, and additionally allows you to access the git repository more directly using either a pure python implementation, or the faster, but more resource intensive git command implementation.

The object database implementation is optimized for handling large quantities of objects and large datasets, which is achieved by using low-level structures and data streaming.

1.1 Requirements

- Python >= 3.7
- **Git 1.7.0 or newer** It should also work with older versions, but it may be that some operations involving remotes will not work as expected.
- GitDB - a pure python git database implementation
- typing_extensions >= 3.7.3.4 (if python < 3.10)

1.2 Installing GitPython

Installing GitPython is easily done using pip. Assuming it is installed, just run the following from the command-line:

```
# pip install GitPython
```

This command will download the latest version of GitPython from the Python Package Index and install it to your system. More information about `pip` and pypi can be found here:

- install pip
- pypi

Alternatively, you can install from the distribution using the `setup.py` script:

```
# python setup.py install
```

Note: In this case, you have to manually install GitDB as well. It would be recommended to use the *git source repository* in that case.

1.3 Limitations

1.3.1 Leakage of System Resources

GitPython is not suited for long-running processes (like daemons) as it tends to leak system resources. It was written in a time where destructors (as implemented in the *__del__* method) still ran deterministically.

In case you still want to use it in such a context, you will want to search the codebase for *__del__* implementations and call these yourself when you see fit.

Another way assure proper cleanup of resources is to factor out GitPython into a separate process which can be dropped periodically.

1.4 Getting Started

- *GitPython Tutorial* - This tutorial provides a walk-through of some of the basic functionality and concepts used in GitPython. It, however, is not exhaustive so you are encouraged to spend some time in the *API Reference*.

1.5 API Reference

An organized section of the GitPython API is at *API Reference*.

1.6 Source Code

GitPython's git repo is available on GitHub, which can be browsed at:

- https://github.com/gitpython-developers/GitPython

and cloned using:

```
$ git clone https://github.com/gitpython-developers/GitPython git-python
```

Initialize all submodules to obtain the required dependencies with:

```
$ cd git-python
$ git submodule update --init --recursive
```

Finally verify the installation by running unit tests:

```
$ python -m unittest
```

1.7 Questions and Answers

Please use stackoverflow for questions, and don't forget to tag it with *gitpython* to assure the right people see the question in a timely manner.

http://stackoverflow.com/questions/tagged/gitpython

1.8 Issue Tracker

The issue tracker is hosted by GitHub:

https://github.com/gitpython-developers/GitPython/issues

1.9 License Information

GitPython is licensed under the New BSD License. See the LICENSE file for more information.

CHAPTER

TWO

GITPYTHON QUICK START TUTORIAL

Welcome to the GitPython Quickstart Guide! Designed for developers seeking a practical and interactive learning experience, this concise resource offers step-by-step code snippets to swiftly initialize/clone repositories, perform essential Git operations, and explore GitPython's capabilities. Get ready to dive in, experiment, and unleash the power of GitPython in your projects!

2.1 git.Repo

There are a few ways to create a *git.Repo* object

2.1.1 Initialize a new git Repo

```
# $ git init <path/to/dir>

from git import Repo

repo = Repo.init(path_to_dir)
```

2.1.2 Existing local git Repo

```
repo = Repo(path_to_dir)
```

2.1.3 Clone from URL

For the rest of this tutorial we will use a clone from https://github.com/gitpython-developers/QuickStartTutorialFiles.git

```
# $ git clone <url> <local_dir>

repo_url = "https://github.com/gitpython-developers/QuickStartTutorialFiles.git
↪"

repo = Repo.clone_from(repo_url, local_dir)
```

2.2 Trees & Blobs

2.2.1 Latest Commit Tree

```
tree = repo.head.commit.tree
```

2.2.2 Any Commit Tree

```
prev_commits = list(repo.iter_commits(all=True, max_count=10))  # Last 10␣
↪commits from all branches.
tree = prev_commits[0].tree
```

2.2.3 Display level 1 Contents

```
files_and_dirs = [(entry, entry.name, entry.type) for entry in tree]
files_and_dirs

# Output
# [(< git.Tree "SHA1-HEX_HASH" >, 'Downloads', 'tree'),
#  (< git.Tree "SHA1-HEX_HASH" >, 'dir1', 'tree'),
#  (< git.Blob "SHA1-HEX_HASH" >, 'file4.txt', 'blob')]
```

2.2.4 Recurse through the Tree

```
def print_files_from_git(root, level=0):
    for entry in root:
        print(f'{"-" * 4 * level}| {entry.path}, {entry.type}')
        if entry.type == "tree":
            print_files_from_git(entry, level + 1)
```

```
print_files_from_git(tree)

# Output
# | Downloads, tree
# ----| Downloads / file3.txt, blob
# | dir1, tree
# ----| dir1 / file1.txt, blob
# ----| dir1 / file2.txt, blob
# | file4.txt, blob
```

2.3 Usage

2.3.1 Add file to staging area

```
# We must make a change to a file so that we can add the update to git

update_file = "dir1/file2.txt"  # we'll use local_dir/dir1/file2.txt
with open(f"{local_dir}/{update_file}", "a") as f:
    f.write("\nUpdate version 2")
```

Now lets add the updated file to git

```
# $ git add <file>
add_file = [update_file]  # relative path from git root
repo.index.add(add_file)  # notice the add function requires a list of paths
```

Notice the add method requires a list as a parameter

Warning: If you experience any trouble with this, try to invoke *git* instead via repo.git.add(path)

2.3.2 Commit

```
# $ git commit -m <message>
repo.index.commit("Update to file2")
```

2.3.3 List of commits associated with a file

```
# $ git log <file>

# Relative path from git root
repo.iter_commits(all=True, max_count=10, paths=update_file)  # Gets the last␣
↪10 commits from all branches.

# Outputs: <generator object Commit._iter_from_process_or_stream at␣
↪0x7fb66c186cf0>
```

Notice this returns a generator object

```
commits_for_file_generator = repo.iter_commits(all=True, max_count=10,␣
↪paths=update_file)
commits_for_file = list(commits_for_file_generator)
commits_for_file

# Outputs: [<git.Commit "SHA1-HEX_HASH-2">,
# <git.Commit "SHA1-HEX-HASH-1">]
```

returns list of *Commit* objects

2.3.4 Printing text files

Lets print the latest version of *<local_dir>/dir1/file2.txt*

```
print_file = "dir1/file2.txt"
tree[print_file]  # The head commit tree.

# Output <git.Blob "SHA1-HEX-HASH">
```

```
blob = tree[print_file]
print(blob.data_stream.read().decode())

# Output
# File 2 version 1
# Update version 2
```

Previous version of *<local_dir>/dir1/file2.txt*

```
commits_for_file = list(repo.iter_commits(all=True, paths=print_file))
tree = commits_for_file[-1].tree  # Gets the first commit tree.
blob = tree[print_file]

print(blob.data_stream.read().decode())

# Output
# File 2 version 1
```

2.3.5 Status

- Untracked files

  Lets create a new file

  ```
  f = open(f"{local_dir}/untracked.txt", "w")  # Creates an empty file.
  f.close()
  ```

  ```
  repo.untracked_files
  # Output: ['untracked.txt']
  ```

- Modified files

  ```
  # Let's modify one of our tracked files.

  with open(f"{local_dir}/Downloads/file3.txt", "w") as f:
      f.write("file3 version 2")  # Overwrite file 3.
  ```

  ```
  repo.index.diff(None)  # Compares staging area to working directory.

  # Output: [<git.diff.Diff object at 0x7fb66c076e50>,
  # <git.diff.Diff object at 0x7fb66c076ca0>]
  ```

  returns a list of *Diff* objects

```
diffs = repo.index.diff(None)
for d in diffs:
    print(d.a_path)

# Output
# Downloads/file3.txt
```

2.3.6 Diffs

Compare staging area to head commit

```
diffs = repo.index.diff(repo.head.commit)
for d in diffs:
    print(d.a_path)

# Output
```

```
# Let's add untracked.txt.
repo.index.add(["untracked.txt"])
diffs = repo.index.diff(repo.head.commit)
for d in diffs:
    print(d.a_path)

# Output
# untracked.txt
```

Compare commit to commit

```
first_commit = list(repo.iter_commits(all=True))[-1]
diffs = repo.head.commit.diff(first_commit)
for d in diffs:
    print(d.a_path)

# Output
# dir1/file2.txt
```

2.4 More Resources

Remember, this is just the beginning! There's a lot more you can achieve with GitPython in your development workflow. To explore further possibilities and discover advanced features, check out the full *GitPython tutorial* and the *API Reference*. Happy coding!

CHAPTER

THREE

GITPYTHON TUTORIAL

GitPython provides object model access to your git repository. This tutorial is composed of multiple sections, most of which explain a real-life use case.

All code presented here originated from test_docs.py to assure correctness. Knowing this should also allow you to more easily run the code for your own testing purposes. All you need is a developer installation of git-python.

3.1 Meet the Repo type

The first step is to create a *git.Repo* object to represent your repository.

```
from git import Repo

# rorepo is a Repo instance pointing to the git-python repository.
# For all you know, the first argument to Repo is a path to the repository you
# want to work with.
repo = Repo(self.rorepo.working_tree_dir)
assert not repo.bare
```

In the above example, the directory `self.rorepo.working_tree_dir` equals `/Users/mtrier/Development/git-python` and is my working repository which contains the `.git` directory. You can also initialize GitPython with a *bare* repository.

```
bare_repo = Repo.init(os.path.join(rw_dir, "bare-repo"), bare=True)
assert bare_repo.bare
```

A repo object provides high-level access to your data, it allows you to create and delete heads, tags and remotes and access the configuration of the repository.

```
repo.config_reader()  # Get a config reader for read-only access.
with repo.config_writer():  # Get a config writer to change configuration.
    pass  # Call release() to be sure changes are written and locks are released.
```

Query the active branch, query untracked files or whether the repository data has been modified.

```
assert not bare_repo.is_dirty()  # Check the dirty state.
repo.untracked_files  # Retrieve a list of untracked files.
# ['my_untracked_file']
```

Clone from existing repositories or initialize new empty ones.

```
cloned_repo = repo.clone(os.path.join(rw_dir, "to/this/path"))
assert cloned_repo.__class__ is Repo  # Clone an existing repository.
assert Repo.init(os.path.join(rw_dir, "path/for/new/repo")).__class__ is Repo
```

Archive the repository contents to a tar file.

```
with open(os.path.join(rw_dir, "repo.tar"), "wb") as fp:
    repo.archive(fp)
```

3.1.1 Advanced Repo Usage

And of course, there is much more you can do with this type, most of the following will be explained in greater detail in specific tutorials. Don't worry if you don't understand some of these examples right away, as they may require a thorough understanding of git's inner workings.

Query relevant repository paths ...

```
assert os.path.isdir(cloned_repo.working_tree_dir)  # Directory with your work files.
assert cloned_repo.git_dir.startswith(cloned_repo.working_tree_dir)  # Directory␣
↪containing the git repository.
assert bare_repo.working_tree_dir is None  # Bare repositories have no working tree.
```

Heads Heads are branches in git-speak. *References* are pointers to a specific commit or to other references. Heads and *Tags* are a kind of references. GitPython allows you to query them rather intuitively.

```
self.assertEqual(
    repo.head.ref,
    repo.heads.master,  # head is a sym-ref pointing to master.
    "It's ok if TC not running from `master`.",
)
self.assertEqual(repo.tags["0.3.5"], repo.tag("refs/tags/0.3.5"))  # You can access tags␣
↪in various ways too.
self.assertEqual(repo.refs.master, repo.heads["master"])  # .refs provides all refs, i.e.
↪ heads...

if "TRAVIS" not in os.environ:
    self.assertEqual(repo.refs["origin/master"], repo.remotes.origin.refs.master)  # ...␣
↪remotes ...
self.assertEqual(repo.refs["0.3.5"], repo.tags["0.3.5"])  # ... and tags.
```

You can also create new heads ...

```
new_branch = cloned_repo.create_head("feature")  # Create a new branch ...
assert cloned_repo.active_branch != new_branch  # which wasn't checked out yet ...
self.assertEqual(new_branch.commit, cloned_repo.active_branch.commit)  # pointing to the␣
↪checked-out commit.
# It's easy to let a branch point to the previous commit, without affecting anything else.
# Each reference provides access to the git object it points to, usually commits.
assert new_branch.set_commit("HEAD~1").commit == cloned_repo.active_branch.commit.
↪parents[0]
```

... and tags ...

```
past = cloned_repo.create_tag(
    "past",
    ref=new_branch,
    message="This is a tag-object pointing to %s" % new_branch.name,
)
self.assertEqual(past.commit, new_branch.commit)  # The tag points to the specified␣
↪commit
assert past.tag.message.startswith("This is")  # and its object carries the message␣
↪provided.

now = cloned_repo.create_tag("now")  # This is a tag-reference. It may not carry meta-
↪data.
assert now.tag is None
```

You can traverse down to *git objects* through references and other objects. Some objects like *commits* have additional meta-data to query.

```
assert now.commit.message != past.commit.message
# You can read objects directly through binary streams, no working tree required.
assert (now.commit.tree / "VERSION").data_stream.read().decode("ascii").startswith("3")

# You can traverse trees as well to handle all contained files of a particular commit.
file_count = 0
tree_count = 0
tree = past.commit.tree
for item in tree.traverse():
    file_count += item.type == "blob"
    tree_count += item.type == "tree"
assert file_count and tree_count  # We have accumulated all directories and files.
self.assertEqual(len(tree.blobs) + len(tree.trees), len(tree))  # A tree is iterable on␣
↪its children.
```

Remotes allow to handle fetch, pull and push operations, while providing optional real-time progress information to *progress delegates*.

```
from git import RemoteProgress

class MyProgressPrinter(RemoteProgress):
    def update(self, op_code, cur_count, max_count=None, message=""):
        print(
            op_code,
            cur_count,
            max_count,
            cur_count / (max_count or 100.0),
            message or "NO MESSAGE",
        )

self.assertEqual(len(cloned_repo.remotes), 1)  # We have been cloned, so should be one␣
↪remote.
self.assertEqual(len(bare_repo.remotes), 0)  # This one was just initialized.
origin = bare_repo.create_remote("origin", url=cloned_repo.working_tree_dir)
assert origin.exists()
for fetch_info in origin.fetch(progress=MyProgressPrinter()):
```

(continues on next page)

(continued from previous page)

```
    print("Updated %s to %s" % (fetch_info.ref, fetch_info.commit))
# Create a local branch at the latest fetched master. We specify the name
# statically, but you have all information to do it programmatically as well.
bare_master = bare_repo.create_head("master", origin.refs.master)
bare_repo.head.set_reference(bare_master)
assert not bare_repo.delete_remote(origin).exists()
# push and pull behave very similarly.
```

The *index* is also called stage in git-speak. It is used to prepare new commits, and can be used to keep results of merge operations. Our index implementation allows to stream date into the index, which is useful for bare repositories that do not have a working tree.

```
self.assertEqual(new_branch.checkout(), cloned_repo.active_branch)  # Checking out␣
↪branch adjusts the wtree.
self.assertEqual(new_branch.commit, past.commit)  # Now the past is checked out.

new_file_path = os.path.join(cloned_repo.working_tree_dir, "my-new-file")
open(new_file_path, "wb").close()  # Create new file in working tree.
cloned_repo.index.add([new_file_path])  # Add it to the index.
# Commit the changes to deviate masters history.
cloned_repo.index.commit("Added a new file in the past - for later merge")

# Prepare a merge.
master = cloned_repo.heads.master  # Right-hand side is ahead of us, in the future.
merge_base = cloned_repo.merge_base(new_branch, master)  # Allows for a three-way merge.
cloned_repo.index.merge_tree(master, base=merge_base)  # Write the merge result into␣
↪index.
cloned_repo.index.commit(
    "Merged past and now into future ;)",
    parent_commits=(new_branch.commit, master.commit),
)

# Now new_branch is ahead of master, which probably should be checked out and reset␣
↪softly.
# Note that all these operations didn't touch the working tree, as we managed it␣
↪ourselves.
# This definitely requires you to know what you are doing! :)
assert os.path.basename(new_file_path) in new_branch.commit.tree  # New file is now in␣
↪tree.
master.commit = new_branch.commit  # Let master point to most recent commit.
cloned_repo.head.reference = master  # We adjusted just the reference, not the working␣
↪tree or index.
```

Submodules represent all aspects of git submodules, which allows you query all of their related information, and manipulate in various ways.

```
# Create a new submodule and check it out on the spot, setup to track master
# branch of `bare_repo`. As our GitPython repository has submodules already that
# point to GitHub, make sure we don't interact with them.
for sm in cloned_repo.submodules:
    assert not sm.remove().exists()  # after removal, the sm doesn't exist anymore
sm = cloned_repo.create_submodule("mysubrepo", "path/to/subrepo", url=bare_repo.git_dir,␣
↪branch="master")
```

(continues on next page)

(continued from previous page)

```
# .gitmodules was written and added to the index, which is now being committed.
cloned_repo.index.commit("Added submodule")
assert sm.exists() and sm.module_exists()  # This submodule is definitely available.
sm.remove(module=True, configuration=False)  # Remove the working tree.
assert sm.exists() and not sm.module_exists()  # The submodule itself is still available.

# Update all submodules, non-recursively to save time. This method is very powerful, go␣
↪have a look.
cloned_repo.submodule_update(recursive=False)
assert sm.module_exists()  # The submodule's working tree was checked out by update.
```

3.2 Examining References

References are the tips of your commit graph from which you can easily examine the history of your project.

```
import git

repo = git.Repo.clone_from(self._small_repo_url(), os.path.join(rw_dir, "repo"), branch=
↪"master")

heads = repo.heads
master = heads.master  # Lists can be accessed by name for convenience.
master.commit  # the commit pointed to by head called master.
master.rename("new_name")  # Rename heads.
master.rename("master")
```

Tags are (usually immutable) references to a commit and/or a tag object.

```
tags = repo.tags
tagref = tags[0]
tagref.tag  # Tags may have tag objects carrying additional information
tagref.commit  # but they always point to commits.
repo.delete_tag(tagref)  # Delete or
repo.create_tag("my_tag")  # create tags using the repo for convenience.
```

A *symbolic reference* is a special case of a reference as it points to another reference instead of a commit.

```
head = repo.head  # The head points to the active branch/ref.
master = head.reference  # Retrieve the reference the head points to.
master.commit  # From here you use it as any other reference.
```

Access the *reflog* easily.

```
log = master.log()
log[0]  # first (i.e. oldest) reflog entry
log[-1]  # last (i.e. most recent) reflog entry
```

3.3 Modifying References

You can easily create and delete *reference types* or modify where they point to.

```
new_branch = repo.create_head("new")  # Create a new one.
new_branch.commit = "HEAD~10"  # Set branch to another commit without changing index or␣
↪working trees.
repo.delete_head(new_branch)  # Delete an existing head - only works if it is not␣
↪checked out.
```

Create or delete *tags* the same way except you may not change them afterwards.

```
new_tag = repo.create_tag("my_new_tag", message="my message")
# You cannot change the commit a tag points to. Tags need to be re-created.
self.assertRaises(AttributeError, setattr, new_tag, "commit", repo.commit("HEAD~1"))
repo.delete_tag(new_tag)
```

Change the *symbolic reference* to switch branches cheaply (without adjusting the index or the working tree).

```
new_branch = repo.create_head("another-branch")
repo.head.reference = new_branch
```

3.4 Understanding Objects

An Object is anything storable in git's object database. Objects contain information about their type, their uncompressed size as well as the actual data. Each object is uniquely identified by a binary SHA1 hash, being 20 bytes in size, or 40 bytes in hexadecimal notation.

Git only knows 4 distinct object types being *Blobs*, *Trees*, *Commits* and *Tags*.

In GitPython, all objects can be accessed through their common base, can be compared and hashed. They are usually not instantiated directly, but through references or specialized repository functions.

```
hc = repo.head.commit
hct = hc.tree
assert hc != hct
assert hc != repo.tags[0]
assert hc == repo.head.reference.commit
```

Common fields are ...

```
self.assertEqual(hct.type, "tree")  # Preset string type, being a class attribute.
assert hct.size > 0  # size in bytes
assert len(hct.hexsha) == 40
assert len(hct.binsha) == 20
```

Index objects are objects that can be put into git's index. These objects are trees, blobs and submodules which additionally know about their path in the file system as well as their mode.

```
self.assertEqual(hct.path, "")  # Root tree has no path.
assert hct.trees[0].path != ""  # The first contained item has one though.
self.assertEqual(hct.mode, 0o40000)  # Trees have the mode of a Linux directory.
self.assertEqual(hct.blobs[0].mode, 0o100644)  # Blobs have specific mode, comparable to␣
↪a standard Linux fs.
```

(continues on next page)

(continued from previous page)

Access *blob* data (or any object data) using streams.

```
hct.blobs[0].data_stream.read()  # Stream object to read data from.
hct.blobs[0].stream_data(open(os.path.join(rw_dir, "blob_data"), "wb"))  # Write data to␣
↪a given stream.
```

3.5 The Commit object

Commit objects contain information about a specific commit. Obtain commits using references as done in *Examining References* or as follows.

Obtain commits at the specified revision

```
repo.commit("master")
repo.commit("v0.8.1")
repo.commit("HEAD~10")
```

Iterate 50 commits, and if you need paging, you can specify a number of commits to skip.

```
fifty_first_commits = list(repo.iter_commits("master", max_count=50))
assert len(fifty_first_commits) == 50
# This will return commits 21-30 from the commit list as traversed backwards master.
ten_commits_past_twenty = list(repo.iter_commits("master", max_count=10, skip=20))
assert len(ten_commits_past_twenty) == 10
assert fifty_first_commits[20:30] == ten_commits_past_twenty
```

A commit object carries all sorts of meta-data

```
headcommit = repo.head.commit
assert len(headcommit.hexsha) == 40
assert len(headcommit.parents) > 0
assert headcommit.tree.type == "tree"
assert len(headcommit.author.name) != 0
assert isinstance(headcommit.authored_date, int)
assert len(headcommit.committer.name) != 0
assert isinstance(headcommit.committed_date, int)
assert headcommit.message != ""
```

Note: date time is represented in a `seconds since epoch` format. Conversion to human readable form can be accomplished with the various time module methods.

```
import time

time.asctime(time.gmtime(headcommit.committed_date))
time.strftime("%a, %d %b %Y %H:%M", time.gmtime(headcommit.committed_date))
```

You can traverse a commit's ancestry by chaining calls to `parents`

```
assert headcommit.parents[0].parents[0].parents[0] == repo.commit("master^^^")
```

The above corresponds to `master^^^` or `master~3` in git parlance.

3.6 The Tree object

A *tree* records pointers to the contents of a directory. Let's say you want the root tree of the latest commit on the master branch

```
tree = repo.heads.master.commit.tree
assert len(tree.hexsha) == 40
```

Once you have a tree, you can get its contents

```
assert len(tree.trees) > 0  # Trees are subdirectories.
assert len(tree.blobs) > 0  # Blobs are files.
assert len(tree.blobs) + len(tree.trees) == len(tree)
```

It is useful to know that a tree behaves like a list with the ability to query entries by name

```
self.assertEqual(tree["smmap"], tree / "smmap")  # Access by index and by sub-path.
for entry in tree:  # Intuitive iteration of tree members.
    print(entry)
blob = tree.trees[1].blobs[0]  # Let's get a blob in a sub-tree.
assert blob.name
assert len(blob.path) < len(blob.abspath)
self.assertEqual(tree.trees[1].name + "/" + blob.name, blob.path)  # This is how␣
↪relative blob path generated.
self.assertEqual(tree[blob.path], blob)  # You can use paths like 'dir/file' in tree,
```

There is a convenience method that allows you to get a named sub-object from a tree with a syntax similar to how paths are written in a posix system

```
assert tree / "smmap" == tree["smmap"]
assert tree / blob.path == tree[blob.path]
```

You can also get a commit's root tree directly from the repository

```
# This example shows the various types of allowed ref-specs.
assert repo.tree() == repo.head.commit.tree
past = repo.commit("HEAD~5")
assert repo.tree(past) == repo.tree(past.hexsha)
self.assertEqual(repo.tree("v0.8.1").type, "tree")  # Yes, you can provide any refspec -␣
↪works everywhere.
```

As trees allow direct access to their intermediate child entries only, use the traverse method to obtain an iterator to retrieve entries recursively

```
assert len(tree) < len(list(tree.traverse()))
```

Note: If trees return Submodule objects, they will assume that they exist at the current head's commit. The tree it originated from may be rooted at another commit though, that it doesn't know. That is why the caller would have to set the submodule's owning or parent commit using the `set_parent_commit(my_commit)` method.

3.7 The Index Object

The git index is the stage containing changes to be written with the next commit or where merges finally have to take place. You may freely access and manipulate this information using the *IndexFile* object. Modify the index with ease

```
index = repo.index
# The index contains all blobs in a flat list.
assert len(list(index.iter_blobs())) == len([o for o in repo.head.commit.tree.traverse()␣
↪if o.type == "blob"])
# Access blob objects.
for (_path, _stage), _entry in index.entries.items():
    pass
new_file_path = os.path.join(repo.working_tree_dir, "new-file-name")
open(new_file_path, "w").close()
index.add([new_file_path])  # Add a new file to the index.
index.remove(["LICENSE"])  # Remove an existing one.
assert os.path.isfile(os.path.join(repo.working_tree_dir, "LICENSE"))  # Working tree is␣
↪untouched.

self.assertEqual(index.commit("my commit message").type, "commit")  # Commit changed␣
↪index.
repo.active_branch.commit = repo.commit("HEAD~1")  # Forget last commit.

from git import Actor

author = Actor("An author", "author@example.com")
committer = Actor("A committer", "committer@example.com")
# Commit with a commit message, author, and committer.
index.commit("my commit message", author=author, committer=committer)
```

Create new indices from other trees or as result of a merge. Write that result to a new index file for later inspection.

```
from git import IndexFile

# Load a tree into a temporary index, which exists just in memory.
IndexFile.from_tree(repo, "HEAD~1")
# Merge two trees three-way into memory...
merge_index = IndexFile.from_tree(repo, "HEAD~10", "HEAD", repo.merge_base("HEAD~10",
↪"HEAD"))
# ...and persist it.
merge_index.write(os.path.join(rw_dir, "merged_index"))
```

3.8 Handling Remotes

Remotes are used as alias for a foreign repository to ease pushing to and fetching from them

```
empty_repo = git.Repo.init(os.path.join(rw_dir, "empty"))
origin = empty_repo.create_remote("origin", repo.remotes.origin.url)
assert origin.exists()
assert origin == empty_repo.remotes.origin == empty_repo.remotes["origin"]
origin.fetch()  # Ensure we actually have data. fetch() returns useful information.
# Set up a local tracking branch of a remote branch.
empty_repo.create_head("master", origin.refs.master)  # Create local branch "master"␣
↪from remote "master".
empty_repo.heads.master.set_tracking_branch(origin.refs.master)  # Set local "master" to␣
↪track remote "master.
empty_repo.heads.master.checkout()  # Check out local "master" to working tree.
# Three above commands in one:
empty_repo.create_head("master", origin.refs.master).set_tracking_branch(origin.refs.
↪master).checkout()
# Rename remotes.
origin.rename("new_origin")
# Push and pull behaves similarly to `git push|pull`.
origin.pull()
origin.push()  # Attempt push, ignore errors.
origin.push().raise_if_error()  # Push and raise error if it fails.
# assert not empty_repo.delete_remote(origin).exists()     # Create and delete remotes.
```

You can easily access configuration information for a remote by accessing options as if they were attributes. The modification of remote configuration is more explicit though.

```
assert origin.url == repo.remotes.origin.url
with origin.config_writer as cw:
    cw.set("pushurl", "other_url")

# Please note that in Python 2, writing origin.config_writer.set(...) is totally
# safe. In py3 __del__ calls can be delayed, thus not writing changes in time.
```

You can also specify per-call custom environments using a new context manager on the Git command, e.g. for using a specific SSH key. The following example works with *git* starting at *v2.3*:

```
ssh_cmd = 'ssh -i id_deployment_key'
with repo.git.custom_environment(GIT_SSH_COMMAND=ssh_cmd):
    repo.remotes.origin.fetch()
```

This one sets a custom script to be executed in place of *ssh*, and can be used in *git* prior to *v2.3*:

```
ssh_executable = os.path.join(rw_dir, 'my_ssh_executable.sh')
with repo.git.custom_environment(GIT_SSH=ssh_executable):
    repo.remotes.origin.fetch()
```

Here's an example executable that can be used in place of the *ssh_executable* above:

```
#!/bin/sh
ID_RSA=/var/lib/openshift/5562b947ecdd5ce939000038/app-deployments/id_rsa
exec /usr/bin/ssh -o StrictHostKeyChecking=no -i $ID_RSA "$@"
```

Please note that the script must be executable (i.e. *chmod +x script.sh*). *StrictHostKeyChecking=no* is used to avoid prompts asking to save the hosts key to *~/.ssh/known_hosts*, which happens in case you run this as daemon.

You might also have a look at *Git.update_environment(...)* in case you want to setup a changed environment more permanently.

3.9 Submodule Handling

Submodules can be conveniently handled using the methods provided by GitPython, and as an added benefit, GitPython provides functionality which behave smarter and less error prone than its original c-git implementation, that is GitPython tries hard to keep your repository consistent when updating submodules recursively or adjusting the existing configuration.

```
repo = self.rorepo
sms = repo.submodules

assert len(sms) == 1
sm = sms[0]
self.assertEqual(sm.name, "gitdb")  # GitPython has gitdb as its one and only (direct)␣
↪submodule...
self.assertEqual(sm.children()[0].name, "smmap")  # ...which has smmap as its one and␣
↪only submodule.

# The module is the repository referenced by the submodule.
assert sm.module_exists()  # The module is available, which doesn't have to be the case.
assert sm.module().working_tree_dir.endswith("gitdb")
# The submodule's absolute path is the module's path.
assert sm.abspath == sm.module().working_tree_dir
self.assertEqual(len(sm.hexsha), 40)  # Its sha defines the commit to check out.
assert sm.exists()  # Yes, this submodule is valid and exists.
# Read its configuration conveniently.
assert sm.config_reader().get_value("path") == sm.path
self.assertEqual(len(sm.children()), 1)  # Query the submodule hierarchy.
```

In addition to the query functionality, you can move the submodule's repository to a different path `<move(...)>`, write its configuration `<config_writer().set_value(...).release()>`, update its working tree `<update(...)>`, and remove or add them `<remove(...)`, `add(...)>`.

If you obtained your submodule object by traversing a tree object which is not rooted at the head's commit, you have to inform the submodule about its actual commit to retrieve the data from by using the `set_parent_commit(...)` method.

The special *RootModule* type allows you to treat your superproject (master repository) as root of a hierarchy of submodules, which allows very convenient submodule handling. Its `update(...)` method is reimplemented to provide an advanced way of updating submodules as they change their values over time. The update method will track changes and make sure your working tree and submodule checkouts stay consistent, which is very useful in case submodules get deleted or added to name just two of the handled cases.

Additionally, GitPython adds functionality to track a specific branch, instead of just a commit. Supported by customized update methods, you are able to automatically update submodules to the latest revision available in the remote repository, as well as to keep track of changes and movements of these submodules. To use it, set the name of the branch you want to track to the `submodule.$name.branch` option of the *.gitmodules* file, and use GitPython update methods on the resulting repository with the `to_latest_revision` parameter turned on. In the latter case, the sha of your submodule will be ignored, instead a local tracking branch will be updated to the respective remote branch

automatically, provided there are no local changes. The resulting behaviour is much like the one of svn::externals, which can be useful in times.

3.10 Obtaining Diff Information

Diffs can generally be obtained by subclasses of *Diffable* as they provide the `diff` method. This operation yields a *DiffIndex* allowing you to easily access diff information about paths.

Diffs can be made between the Index and Trees, Index and the working tree, trees and trees as well as trees and the working copy. If commits are involved, their tree will be used implicitly.

```
hcommit = repo.head.commit
hcommit.diff()  # diff tree against index.
hcommit.diff("HEAD~1")  # diff tree against previous tree.
hcommit.diff(None)  # diff tree against working tree.

index = repo.index
index.diff()  # diff index against itself yielding empty diff.
index.diff(None)  # diff index against working copy.
index.diff("HEAD")  # diff index against current HEAD tree.
```

The item returned is a DiffIndex which is essentially a list of Diff objects. It provides additional filtering to ease finding what you might be looking for.

```
# Traverse added Diff objects only
for diff_added in hcommit.diff("HEAD~1").iter_change_type("A"):
    print(diff_added)
```

Use the diff framework if you want to implement git-status like functionality.

- A diff between the index and the commit's tree your HEAD points to
- use `repo.index.diff(repo.head.commit)`
- A diff between the index and the working tree
- use `repo.index.diff(None)`
- A list of untracked files
- use `repo.untracked_files`

3.11 Switching Branches

To switch between branches similar to `git checkout`, you effectively need to point your HEAD symbolic reference to the new branch and reset your index and working copy to match. A simple manual way to do it is the following one

```
# Reset our working tree 10 commits into the past.
past_branch = repo.create_head("past_branch", "HEAD~10")
repo.head.reference = past_branch
assert not repo.head.is_detached
# Reset the index and working tree to match the pointed-to commit.
repo.head.reset(index=True, working_tree=True)

```

(continues on next page)

(continued from previous page)

```
# To detach your head, you have to point to a commit directly.
repo.head.reference = repo.commit("HEAD~5")
assert repo.head.is_detached
# Now our head points 15 commits into the past, whereas the working tree
# and index are 10 commits in the past.
```

The previous approach would brutally overwrite the user's changes in the working copy and index though and is less sophisticated than a `git-checkout`. The latter will generally prevent you from destroying your work. Use the safer approach as follows.

```
# Check out the branch using git-checkout.
# It will fail as the working tree appears dirty.
self.assertRaises(git.GitCommandError, repo.heads.master.checkout)
repo.heads.past_branch.checkout()
```

3.12 Initializing a repository

In this example, we will initialize an empty repository, add an empty file to the index, and commit the change.

```
import git

repo_dir = os.path.join(rw_dir, "my-new-repo")
file_name = os.path.join(repo_dir, "new-file")

r = git.Repo.init(repo_dir)
# This function just creates an empty file.
open(file_name, "wb").close()
r.index.add([file_name])
r.index.commit("initial commit")
```

Please have a look at the individual methods as they usually support a vast amount of arguments to customize their behavior.

3.13 Using git directly

In case you are missing functionality as it has not been wrapped, you may conveniently use the *`git`* command directly. It is owned by each repository instance.

```
git = repo.git
git.checkout("HEAD", b="my_new_branch")  # Create a new branch.
git.branch("another-new-one")
git.branch("-D", "another-new-one")  # Pass strings for full control over argument order.
git.for_each_ref()  # '-' becomes '_' when calling it.
```

The return value will by default be a string of the standard output channel produced by the command.

Keyword arguments translate to short and long keyword arguments on the command-line. The special notion `git.command(flag=True)` will create a flag without value like `command --flag`.

If `None` is found in the arguments, it will be dropped silently. Lists and tuples passed as arguments will be unpacked recursively to individual arguments. Objects are converted to strings using the `str(...)` function.

3.14 Object Databases

git.Repo instances are powered by its object database instance which will be used when extracting any data, or when writing new objects.

The type of the database determines certain performance characteristics, such as the quantity of objects that can be read per second, the resource usage when reading large data files, as well as the average memory footprint of your application.

3.14.1 GitDB

The GitDB is a pure-python implementation of the git object database. It is the default database to use in GitPython 0.3. It uses less memory when handling huge files, but will be 2 to 5 times slower when extracting large quantities of small objects from densely packed repositories:

```
repo = Repo("path/to/repo", odbt=GitDB)
```

3.14.2 GitCmdObjectDB

The git command database uses persistent git-cat-file instances to read repository information. These operate very fast under all conditions, but will consume additional memory for the process itself. When extracting large files, memory usage will be much higher than `GitDB`:

```
repo = Repo("path/to/repo", odbt=GitCmdObjectDB)
```

3.15 Git Command Debugging and Customization

Using environment variables, you can further adjust the behaviour of the git command.

- **GIT_PYTHON_TRACE**
  - If set to non-0, all executed git commands will be shown as they happen
  - If set to *full*, the executed git command _and_ its entire output on stdout and stderr will be shown as they happen

  NOTE: All logging is outputted using a Python logger, so make sure your program is configured to show INFO-level messages. If this is not the case, try adding the following to your program:

  ```
  import logging
  logging.basicConfig(level=logging.INFO)
  ```

- **GIT_PYTHON_GIT_EXECUTABLE**
- If set, it should contain the full path to the git executable, e.g. *c:\Program Files (x86)\Git\bin\git.exe* on windows or */usr/bin/git* on linux.

3.16 And even more ...

There is more functionality in there, like the ability to archive repositories, get stats and logs, blame, and probably a few other things that were not mentioned here.

Check the unit tests for an in-depth introduction on how each function is supposed to be used.

CHAPTER

FOUR

API REFERENCE

4.1 Top-Level

git.**__version__**
Current GitPython version.

git.**refresh**(*path: Optional[Union[str, os.PathLike[str]]] = None*) → None
Convenience method for setting the git executable path.

Parameters **path** – Optional path to the Git executable. If not absolute, it is resolved immediately, relative to the current directory.

Note The *path* parameter is usually omitted and cannot be used to specify a custom command whose location is looked up in a path search on each call. See *Git.refresh* for details on how to achieve this.

Note This calls *Git.refresh* and sets other global configuration according to the effect of doing so. As such, this function should usually be used instead of using *Git.refresh* or *FetchInfo.refresh* directly.

Note This function is called automatically, with no arguments, at import time.

4.2 Objects.Base

class git.objects.base.**IndexObject**(*repo: Repo*, *binsha: bytes*, *mode: Union[None, int] = None*, *path: Union[None, str, os.PathLike[str]] = None*)
Base for all objects that can be part of the index file.

The classes representing git object types that can be part of the index file are `Blob`. In addition, *Submodule*, which is not really a git object type but can be part of an index file, is also a subclass.

__annotations__ = {}

__hash__() → int

Returns Hash of our path as index items are uniquely identifiable by path, not by their data!

__init__(*repo: Repo*, *binsha: bytes*, *mode: Union[None, int] = None*, *path: Union[None, str, os.PathLike[str]] = None*) → None
Initialize a newly instanced *IndexObject*.

Parameters

- **repo** – The *Repo* we are located in.

- **binsha** – 20 byte sha1.
- **mode** – The stat-compatible file mode as `int`. Use the `stat` module to evaluate the information.
- **path** – The path to the file in the file system, relative to the git repository root, like `file.ext` or `folder/other.ext`.

Note Path may not be set if the index object has been created directly, as it cannot be retrieved without knowing the parent tree.

__module__ = 'git.objects.base'

__slots__ = ('path', 'mode')

property abspath: Union[str, os.PathLike[str]]

Returns

Absolute path to this index object in the file system (as opposed to the *path* field which is a path relative to the git repository).

The returned path will be native to the system and contains \ on Windows.

mode

property name: str

Returns Name portion of the path, effectively being the basename

path

class git.objects.base.**Object**(*repo: Repo*, *binsha: bytes*)

Base class for classes representing git object types.

The following four leaf classes represent specific kinds of git objects:

- *Blob*
- *Tree*
- *Commit*
- *TagObject*

See *gitglossary(7)* on:

- "object": https://git-scm.com/docs/gitglossary#def_object
- "object type": https://git-scm.com/docs/gitglossary#def_object_type
- "blob": https://git-scm.com/docs/gitglossary#def_blob_object
- "tree object": https://git-scm.com/docs/gitglossary#def_tree_object
- "commit object": https://git-scm.com/docs/gitglossary#def_commit_object
- "tag object": https://git-scm.com/docs/gitglossary#def_tag_object

Note See the *AnyGitObject* union type of the four leaf subclasses that represent actual git object types.

Note *Submodule* is defined under the hierarchy rooted at this *Object* class, even though submodules are not really a type of git object. (This also applies to its *RootModule* subclass.)

Note This *Object* class should not be confused with `object` (the root of the class hierarchy in Python).

NULL_BIN_SHA = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

NULL_HEX_SHA = '0000000000000000000000000000000000000000'

TYPES = (b'blob', b'tree', b'commit', b'tag')

__annotations__ = {'type': typing.Optional[typing.Literal['commit', 'tag', 'blob', 'tree']]}

__eq__(*other: Any*) → bool

Returns True if the objects have the same SHA1

__hash__() → int

Returns Hash of our id allowing objects to be used in dicts and sets

__init__(*repo: Repo*, *binsha: bytes*) → None

Initialize an object by identifying it by its binary sha.

All keyword arguments will be set on demand if None.

Parameters

- **repo** – Repository this object is located in.
- **binsha** – 20 byte SHA1

__module__ = 'git.objects.base'

__ne__(*other: Any*) → bool

Returns True if the objects do not have the same SHA1

__repr__() → str

Returns String with pythonic representation of our object

__slots__ = ('repo', 'binsha', 'size')

__str__() → str

Returns String of our SHA1 as understood by all git commands

binsha

property data_stream: OStream

Returns File-object compatible stream to the uncompressed raw data of the object

Note Returned streams must be read in order.

property hexsha: str

Returns 40 byte hex version of our 20 byte binary sha

classmethod new(*repo: Repo*, *id: Union[str, Reference]*) → Union[Commit, Tree, TagObject, Blob]

Returns New *Object* instance of a type appropriate to the object type behind *id*. The id of the newly created object will be a binsha even though the input id may have been a *~git.refs.reference.Reference* or rev-spec.

Parameters **id** – *Reference*, rev-spec, or hexsha.

Note This cannot be a __new__ method as it would always call *__init__()* with the input id which is not necessarily a binsha.

classmethod new_from_sha(*repo: Repo*, *sha1: bytes*) → Union[Commit, Tree, TagObject, Blob]

Returns New object instance of a type appropriate to represent the given binary sha1

Parameters **sha1** – 20 byte binary sha1.

repo

size

stream_data(*ostream: OStream*) → *Object*

Write our data directly to the given output stream.

Parameters **ostream** – File-object compatible stream object.

Returns self

type: Optional[Literal['commit', 'tag', 'blob', 'tree']] = None

String identifying (a concrete *Object* subtype for) a git object type.

The subtypes that this may name correspond to the kinds of git objects that exist, i.e., the objects that may be present in a git repository.

Note Most subclasses represent specific types of git objects and override this class attribute accordingly. This attribute is None in the *Object* base class, as well as the *IndexObject* intermediate subclass, but never None in concrete leaf subclasses representing specific git object types.

Note See also *GitObjectTypeString*.

4.3 Objects.Blob

class git.objects.blob.**Blob**(*repo: Repo*, *binsha: bytes*, *mode: Union[None, int] = None*, *path: Union[None, str, os.PathLike[str]] = None*)

A Blob encapsulates a git blob object.

See *gitglossary(7)* on "blob": https://git-scm.com/docs/gitglossary#def_blob_object

DEFAULT_MIME_TYPE = 'text/plain'

__annotations__ = {'type': typing.Literal['blob']}

__module__ = 'git.objects.blob'

__slots__ = ()

executable_mode = 33261

file_mode = 33188

link_mode = 40960

property mime_type: str

Returns String describing the mime type of this file (based on the filename)

Note Defaults to text/plain in case the actual file type is unknown.

type: Literal['blob'] = 'blob'

String identifying (a concrete `Object` subtype for) a git object type.

The subtypes that this may name correspond to the kinds of git objects that exist, i.e., the objects that may be present in a git repository.

Note Most subclasses represent specific types of git objects and override this class attribute accordingly. This attribute is `None` in the `Object` base class, as well as the `IndexObject` intermediate subclass, but never `None` in concrete leaf subclasses representing specific git object types.

Note See also *GitObjectTypeString*.

4.4 Objects.Commit

class git.objects.commit.**Commit**(*repo: Repo*, *binsha: bytes*, *tree: Optional[*git.objects.tree.Tree*] = None*, *author: Optional[*git.util.Actor*] = None*, *authored_date: Optional[int] = None*, *author_tz_offset: Union[None, float] = None*, *committer: Optional[*git.util.Actor*] = None*, *committed_date: Optional[int] = None*, *committer_tz_offset: Union[None, float] = None*, *message: Optional[Union[str, bytes]] = None*, *parents: Optional[Sequence[*Commit*]] = None*, *encoding: Optional[str] = None*, *gpgsig: Optional[str] = None*)

Wraps a git commit object.

See *gitglossary(7)* on "commit object": https://git-scm.com/docs/gitglossary#def_commit_object

Note This class will act lazily on some of its attributes and will query the value on demand only if it involves calling the git binary.

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': 'str', 'parents': typing.Sequence[ForwardRef('Commit')], 'repo': "'Repo'", 'type': typing.Literal['commit']}

__init__(*repo: Repo*, *binsha: bytes*, *tree: Optional[*git.objects.tree.Tree*] = None*, *author: Optional[*git.util.Actor*] = None*, *authored_date: Optional[int] = None*, *author_tz_offset: Union[None, float] = None*, *committer: Optional[*git.util.Actor*] = None*, *committed_date: Optional[int] = None*, *committer_tz_offset: Union[None, float] = None*, *message: Optional[Union[str, bytes]] = None*, *parents: Optional[Sequence[*Commit*]] = None*, *encoding: Optional[str] = None*, *gpgsig: Optional[str] = None*) → None

Instantiate a new *Commit*. All keyword arguments taking `None` as default will be implicitly set on first query.

Parameters

- **binsha** – 20 byte sha1.
- **tree** – A *Tree* object.
- **author** – The author *Actor* object.
- **authored_date** – int_seconds_since_epoch The authored DateTime - use `time.gmtime()` to convert it into a different format.
- **author_tz_offset** – int_seconds_west_of_utc The timezone that the *authored_date* is in.
- **committer** – The committer string, as an *Actor* object.

- **committed_date** – int_seconds_since_epoch The committed DateTime - use `time.gmtime()` to convert it into a different format.
- **committer_tz_offset** – int_seconds_west_of_utc The timezone that the *committed_date* is in.
- **message** – string The commit message.
- **encoding** – string Encoding of the message, defaults to UTF-8.
- **parents** – List or tuple of *Commit* objects which are our parent(s) in the commit dependency graph.

Returns *Commit*

Note Timezone information is in the same format and in the same sign as what `time.altzone()` returns. The sign is inverted compared to git's UTC timezone.

__module__ = 'git.objects.commit'

__parameters__ = ()

__slots__ = ('tree', 'author', 'authored_date', 'author_tz_offset', 'committer', 'committed_date', 'committer_tz_offset', 'message', 'parents', 'encoding', 'gpgsig')

classmethod __subclasshook__(*other*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

author

author_tz_offset

authored_date

property authored_datetime: datetime.datetime

property co_authors: List[*git.util.Actor*]

Search the commit message for any co-authors of this commit.

Details on co-authors: https://github.blog/2018-01-29-commit-together-with-co-authors/

Returns List of co-authors for this commit (as *Actor* objects).

committed_date

property committed_datetime: datetime.datetime

committer

committer_tz_offset

conf_encoding = 'i18n.commitencoding'

count(*paths: Union[str, os.PathLike[str], Sequence[Union[str, os.PathLike[str]]]] = ''*, ***kwargs: Any*) → int

Count the number of commits reachable from this commit.

Parameters

- **paths** – An optional path or a list of paths restricting the return value to commits actually containing the paths.
- **kwargs** – Additional options to be passed to *git-rev-list(1)*. They must not alter the output style of the command, or parsing will yield incorrect results.

Returns An int defining the number of reachable commits

classmethod create_from_tree(*repo: Repo*, *tree: Union[git.objects.tree.Tree, str]*, *message: str*, *parent_commits: Union[None, List[Commit]] = None*, *head: bool = False*, *author: Union[None, git.util.Actor] = None*, *committer: Union[None, git.util.Actor] = None*, *author_date: Union[None, str, datetime.datetime] = None*, *commit_date: Union[None, str, datetime.datetime] = None*) → *Commit*

Commit the given tree, creating a *Commit* object.

Parameters

- **repo** – *Repo* object the commit should be part of.
- **tree** – *Tree* object or hex or bin sha. The tree of the new commit.
- **message** – Commit message. It may be an empty string if no message is provided. It will be converted to a string, in any case.
- **parent_commits** – Optional *Commit* objects to use as parents for the new commit. If empty list, the commit will have no parents at all and become a root commit. If `None`, the current head commit will be the parent of the new commit object.
- **head** – If `True`, the HEAD will be advanced to the new commit automatically. Otherwise the HEAD will remain pointing on the previous commit. This could lead to undesired results when diffing files.
- **author** – The name of the author, optional. If unset, the repository configuration is used to obtain this value.
- **committer** – The name of the committer, optional. If unset, the repository configuration is used to obtain this value.
- **author_date** – The timestamp for the author field.
- **commit_date** – The timestamp for the committer field.

Returns *Commit* object representing the new commit.

Note Additional information about the committer and author are taken from the environment or from the git configuration. See *git-commit-tree(1)* for more information.

default_encoding = 'UTF-8'

encoding

env_author_date = 'GIT_AUTHOR_DATE'

env_committer_date = 'GIT_COMMITTER_DATE'

gpgsig

classmethod iter_items(*repo: Repo*, *rev: Union[str, Commit, SymbolicReference]*, *paths: Union[str, os.PathLike[str], Sequence[Union[str, os.PathLike[str]]]] = ''*, ***kwargs: Any*) → Iterator[*Commit*]

Find all commits matching the given criteria.

Parameters

- **repo** – The *Repo*.
- **rev** – Revision specifier. See *git-rev-parse(1)* for viable options.
- **paths** – An optional path or list of paths. If set only *Commit*s that include the path or paths will be considered.

- **kwargs** – Optional keyword arguments to *git-rev-list(1)* where:
  - `max_count` is the maximum number of commits to fetch.
  - `skip` is the number of commits to skip.
  - `since` selects all commits since some date, e.g. `"1970-01-01"`.

Returns Iterator yielding *Commit* items.

iter_parents(*paths: Union[str, os.PathLike[str], Sequence[Union[str, os.PathLike[str]]]] = ''*, ***kwargs: Any*) → Iterator[*git.objects.commit.Commit*]

Iterate _all_ parents of this commit.

Parameters

- **paths** – Optional path or list of paths limiting the *Commit*s to those that contain at least one of the paths.
- **kwargs** – All arguments allowed by *git-rev-list(1)*.

Returns Iterator yielding *Commit* objects which are parents of `self`

message

property name_rev: str

Returns String describing the commits hex sha based on the closest ~*git.refs.reference.Reference*.

Note Mostly useful for UI purposes.

parents: Sequence[*Commit*]

replace(***kwargs: Any*) → *git.objects.commit.Commit*

Create new commit object from an existing commit object.

Any values provided as keyword arguments will replace the corresponding attribute in the new object.

property stats: *git.util.Stats*

Create a git stat from changes between this commit and its first parent or from all changes done if this is the very first commit.

Returns `Stats`

property summary: Union[str, bytes]

Returns First line of the commit message

property trailers: Dict[str, str]

Deprecated. Get the trailers of the message as a dictionary.

Note This property is deprecated, please use either *trailers_list* or *trailers_dict*.

Returns Dictionary containing whitespace stripped trailer information. Only contains the latest instance of each trailer key.

property trailers_dict: Dict[str, List[str]]

Get the trailers of the message as a dictionary.

Git messages can contain trailer information that are similar to **RFC 822** e-mail headers. See *git-interpret-trailers(1)*.

This function calls `git interpret-trailers --parse` onto the message to extract the trailer information. The key value pairs are stripped of leading and trailing whitespaces before they get saved into a dictionary.

Valid message with trailer:

```
Subject line

some body information

another information

key1: value1.1
key1: value1.2
key2 :    value 2 with inner spaces
```

Returned dictionary will look like this:

```
{
    "key1": ["value1.1", "value1.2"],
    "key2": ["value 2 with inner spaces"],
}
```

Returns Dictionary containing whitespace stripped trailer information, mapping trailer keys to a list of their corresponding values.

property trailers_list: List[Tuple[str, str]]

Get the trailers of the message as a list.

Git messages can contain trailer information that are similar to **RFC 822** e-mail headers. See *git-interpret-trailers(1)*.

This function calls `git interpret-trailers --parse` onto the message to extract the trailer information, returns the raw trailer data as a list.

Valid message with trailer:

```
Subject line

some body information

another information

key1: value1.1
key1: value1.2
key2 :    value 2 with inner spaces
```

Returned list will look like this:

```
[
    ("key1", "value1.1"),
    ("key1", "value1.2"),
    ("key2", "value 2 with inner spaces"),
]
```

Returns List containing key-value tuples of whitespace stripped trailer information.

tree

type: Literal['commit'] = 'commit'

String identifying (a concrete `Object` subtype for) a git object type.

The subtypes that this may name correspond to the kinds of git objects that exist, i.e., the objects that may be present in a git repository.

Note Most subclasses represent specific types of git objects and override this class attribute accordingly. This attribute is `None` in the `Object` base class, as well as the `IndexObject` intermediate subclass, but never `None` in concrete leaf subclasses representing specific git object types.

Note See also *GitObjectTypeString*.

4.5 Objects.Tag

Provides an *Object*-based type for annotated tags.

This defines the *TagObject* class, which represents annotated tags. For lightweight tags, see the *git.refs.tag* module.

class git.objects.tag.**TagObject**(*repo: Repo*, *binsha: bytes*, *object: Union[None,* git.objects.base.Object*] = None*, *tag: Union[None, str] = None*, *tagger: Union[None, Actor] = None*, *tagged_date: Optional[int] = None*, *tagger_tz_offset: Optional[int] = None*, *message: Optional[str] = None*)

Annotated (i.e. non-lightweight) tag carrying additional information about an object we are pointing to.

See *gitglossary(7)* on "tag object": https://git-scm.com/docs/gitglossary#def_tag_object

__annotations__ = {'type': typing.Literal['tag']}

__init__(*repo: Repo*, *binsha: bytes*, *object: Union[None,* git.objects.base.Object*] = None*, *tag: Union[None, str] = None*, *tagger: Union[None, Actor] = None*, *tagged_date: Optional[int] = None*, *tagger_tz_offset: Optional[int] = None*, *message: Optional[str] = None*) → None

Initialize a tag object with additional data.

Parameters

- **repo** – Repository this object is located in.
- **binsha** – 20 byte SHA1.
- **object** – *Object* instance of object we are pointing to.
- **tag** – Name of this tag.
- **tagger** – *Actor* identifying the tagger.
- **tagged_date** – int_seconds_since_epoch The DateTime of the tag creation. Use `time.gmtime()` to convert it into a different format.
- **tagger_tz_offset** – int_seconds_west_of_utc The timezone that the *tagged_date* is in, in a format similar to `time.altzone`.

__module__ = 'git.objects.tag'

__slots__ = ('object', 'tag', 'tagger', 'tagged_date', 'tagger_tz_offset', 'message')

message

object: Union['Commit', 'Blob', 'Tree', 'TagObject']

tag

tagged_date

tagger

tagger_tz_offset

type: Literal['tag'] = 'tag'

String identifying (a concrete `Object` subtype for) a git object type.

The subtypes that this may name correspond to the kinds of git objects that exist, i.e., the objects that may be present in a git repository.

Note Most subclasses represent specific types of git objects and override this class attribute accordingly. This attribute is `None` in the `Object` base class, as well as the `IndexObject` intermediate subclass, but never `None` in concrete leaf subclasses representing specific git object types.

Note See also *GitObjectTypeString*.

4.6 Objects.Tree

class git.objects.tree.**Tree**(*repo: Repo*, *binsha: bytes*, *mode: int = 16384*, *path: Optional[Union[str, os.PathLike[str]]] = None*)

Tree objects represent an ordered list of *Blob*s and other *Tree*s.

See *gitglossary(7)* on "tree object": https://git-scm.com/docs/gitglossary#def_tree_object

Subscripting is supported, as with a list or dict:

- Access a specific blob using the `tree["filename"]` notation.
- You may likewise access by index, like `blob = tree[0]`.

__abstractmethods__ = frozenset({})

__annotations__ = {'_map_id_to_type': typing.Dict[int, typing.Type[typing.Union[ForwardRef('Tree'), ForwardRef('Blob'), ForwardRef('Submodule')]]], 'repo': "'Repo'", 'type': typing.Literal['tree']}

__contains__(*item: Union[*Tree*, Blob, Submodule, str, os.PathLike[str]]*) → bool

__getitem__(*item: Union[str, int, slice]*) → Union[*git.objects.tree.Tree*, *git.objects.blob.Blob*, *git.objects.submodule.base.Submodule*]

__getslice__(*i: int*, *j: int*) → List[Union[*git.objects.tree.Tree*, *git.objects.blob.Blob*, *git.objects.submodule.base.Submodule*]]

__init__(*repo: Repo*, *binsha: bytes*, *mode: int = 16384*, *path: Optional[Union[str, os.PathLike[str]]] = None*)

Initialize a newly instanced `IndexObject`.

Parameters

- **repo** – The *Repo* we are located in.
- **binsha** – 20 byte sha1.
- **mode** – The stat-compatible file mode as `int`. Use the `stat` module to evaluate the information.
- **path** – The path to the file in the file system, relative to the git repository root, like `file.ext` or `folder/other.ext`.

Note Path may not be set if the index object has been created directly, as it cannot be retrieved without knowing the parent tree.

__iter__() → Iterator[Union[*git.objects.tree.Tree*, *git.objects.blob.Blob*, *git.objects.submodule.base.Submodule*]]

__len__() → int

__module__ = 'git.objects.tree'

__reversed__() → Iterator[Union[*git.objects.tree.Tree*, *git.objects.blob.Blob*, *git.objects.submodule.base.Submodule*]]

__slots__ = ('_cache',)

__truediv__(*file: str*) → Union[*git.objects.tree.Tree*, *git.objects.blob.Blob*, *git.objects.submodule.base.Submodule*]

The / operator is another syntax for joining.

See *join()* for details.

blob_id = 8

property blobs: List[*git.objects.blob.Blob*]

Returns list(Blob, ...) List of blobs directly below this tree

property cache: *git.objects.tree.TreeModifier*

Returns An object allowing modification of the internal cache. This can be used to change the tree's contents. When done, make sure you call *set_done()* on the tree modifier, or serialization behaviour will be incorrect.

Note See *TreeModifier* for more information on how to alter the cache.

commit_id = 14

join(*file: str*) → Union[*git.objects.tree.Tree*, *git.objects.blob.Blob*, *git.objects.submodule.base.Submodule*]

Find the named object in this tree's contents.

Returns *Blob*, *Tree*, or *Submodule*

Raises **KeyError** – If the given file or tree does not exist in this tree.

list_traverse(**args: Any*, ***kwargs: Any*) → *git.util.IterableList*[Union[*git.objects.tree.Tree*, *git.objects.blob.Blob*, *git.objects.submodule.base.Submodule*]]

Returns

IterableList with the results of the traversal as produced by *traverse()*

Tree -> IterableList[Union[Submodule, Tree, Blob]]

symlink_id = 10

traverse(*predicate: Callable[[Union[git.objects.tree.Tree, git.objects.blob.Blob, git.objects.submodule.base.Submodule, Tuple[Optional[git.objects.tree.Tree], Union[git.objects.tree.Tree, git.objects.blob.Blob, git.objects.submodule.base.Submodule], Tuple[git.objects.submodule.base.Submodule, git.objects.submodule.base.Submodule]]], int], bool] = <function Tree.<lambda>>, prune: Callable[[Union[git.objects.tree.Tree, git.objects.blob.Blob, git.objects.submodule.base.Submodule, Tuple[Optional[git.objects.tree.Tree], Union[git.objects.tree.Tree, git.objects.blob.Blob, git.objects.submodule.base.Submodule], Tuple[git.objects.submodule.base.Submodule, git.objects.submodule.base.Submodule]]], int], bool] = <function Tree.<lambda>>, depth: int = -1, branch_first: bool = True, visit_once: bool = False, ignore_self: int = 1, as_edge: bool = False*) → Union[Iterator[Union[*git.objects.tree.Tree*, *git.objects.blob.Blob*, *git.objects.submodule.base.Submodule*]], Iterator[Tuple[Optional[*git.objects.tree.Tree*], Union[*git.objects.tree.Tree*, *git.objects.blob.Blob*, *git.objects.submodule.base.Submodule*], Tuple[*git.objects.submodule.base.Submodule*, *git.objects.submodule.base.Submodule*]]]]

For documentation, see *Traversable._traverse() <git.objects.util.Traversable._traverse>*.

Trees are set to `visit_once = False` to gain more performance in the traversal.

tree_id = 4

property trees: List[*git.objects.tree.Tree*]

Returns list(Tree, …) List of trees directly below this tree

type: Literal['tree'] = 'tree'

String identifying (a concrete `Object` subtype for) a git object type.

The subtypes that this may name correspond to the kinds of git objects that exist, i.e., the objects that may be present in a git repository.

Note Most subclasses represent specific types of git objects and override this class attribute accordingly. This attribute is `None` in the `Object` base class, as well as the `IndexObject` intermediate subclass, but never `None` in concrete leaf subclasses representing specific git object types.

Note See also *GitObjectTypeString*.

class git.objects.tree.**TreeModifier**(*cache: List[Tuple[bytes, int, str]]*)

A utility class providing methods to alter the underlying cache in a list-like fashion.

Once all adjustments are complete, the `_cache`, which really is a reference to the cache of a tree, will be sorted. This ensures it will be in a serializable state.

__delitem__(*name: str*) → None

Delete an item with the given name if it exists.

__init__(*cache: List[Tuple[bytes, int, str]]*) → None

__module__ = 'git.objects.tree'

__slots__ = ('_cache',)

add(*sha: bytes*, *mode: int*, *name: str*, *force: bool = False*) → *git.objects.tree.TreeModifier*

Add the given item to the tree.

If an item with the given name already exists, nothing will be done, but a `ValueError` will be raised if the sha and mode of the existing item do not match the one you add, unless *force* is `True`.

Parameters

- **sha** – The 20 or 40 byte sha of the item to add.
- **mode** – `int` representing the stat-compatible mode of the item.

- **force** – If `True`, an item with your name and information will overwrite any existing item with the same name, no matter which information it has.

Returns self

add_unchecked(*binsha: bytes*, *mode: int*, *name: str*) → None
Add the given item to the tree. Its correctness is assumed, so it is the caller's responsibility to ensure that the input is correct.

For more information on the parameters, see *add()*.

Parameters **binsha** – 20 byte binary sha.

set_done() → *git.objects.tree.TreeModifier*
Call this method once you are done modifying the tree information.

This may be called several times, but be aware that each call will cause a sort operation.

Returns self

4.7 Objects.Functions

Functions that are supposed to be as fast as possible.

git.objects.fun.**traverse_tree_recursive**(*odb: GitCmdObjectDB*, *tree_sha: bytes*, *path_prefix: str*) → List[Tuple[bytes, int, str]]

Returns

List of entries of the tree pointed to by the binary *tree_sha*.

An entry has the following format:

- [0] 20 byte sha
- [1] mode as int
- [2] path relative to the repository

Parameters **path_prefix** – Prefix to prepend to the front of all returned paths.

git.objects.fun.**traverse_trees_recursive**(*odb: GitCmdObjectDB*, *tree_shas: Sequence[Optional[bytes]]*, *path_prefix: str*) → List[Tuple[Optional[Tuple[bytes, int, str]], ...]]

Returns

List of list with entries according to the given binary tree-shas.

The result is encoded in a list of n tuple|None per blob/commit, (n == len(tree_shas)), where:

- [0] == 20 byte sha
- [1] == mode as int
- [2] == path relative to working tree root

The entry tuple is `None` if the respective blob/commit did not exist in the given tree.

Parameters

- **tree_shas** – Iterable of shas pointing to trees. All trees must be on the same level. A tree-sha may be `None`, in which case `None`.

- **path_prefix** – A prefix to be added to the returned paths on this level. Set it "" for the first iteration.

Note The ordering of the returned items will be partially lost.

git.objects.fun.**tree_entries_from_data**(*data: bytes*) → List[Tuple[bytes, int, str]]

Read the binary representation of a tree and returns tuples of *Tree* items.

Parameters **data** – Data block with tree data (as bytes).

Returns list(tuple(binsha, mode, tree_relative_path), ...)

git.objects.fun.**tree_to_stream**(*entries: Sequence[Tuple[bytes, int, str]]*, *write: Callable[[ReadableBuffer], Optional[int]]*) → None

Write the given list of entries into a stream using its `write` method.

Parameters

- **entries** – **Sorted** list of tuples with (binsha, mode, name).
- **write** – A `write` method which takes a data string.

4.8 Objects.Submodule.base

class git.objects.submodule.base.**Submodule**(*repo: Repo*, *binsha: bytes*, *mode: Optional[int] = None*, *path: Optional[Union[str, os.PathLike[str]]] = None*, *name: Optional[str] = None*, *parent_commit: Optional[Commit] = None*, *url: Optional[str] = None*, *branch_path: Optional[Union[str, os.PathLike[str]]] = None*)

Implements access to a git submodule. They are special in that their sha represents a commit in the submodule's repository which is to be checked out at the path of this instance.

The submodule type does not have a string type associated with it, as it exists solely as a marker in the tree and index.

All methods work in bare and non-bare repositories.

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': 'str', 'type': typing.Literal['submodule']}

__eq__(*other: Any*) → bool

Compare with another submodule.

__hash__() → int

Hash this instance using its logical id, not the sha.

__init__(*repo: Repo*, *binsha: bytes*, *mode: Optional[int] = None*, *path: Optional[Union[str, os.PathLike[str]]] = None*, *name: Optional[str] = None*, *parent_commit: Optional[Commit] = None*, *url: Optional[str] = None*, *branch_path: Optional[Union[str, os.PathLike[str]]] = None*) → None

Initialize this instance with its attributes.

We only document the parameters that differ from *IndexObject*.

Parameters

- **repo** – Our parent repository.
- **binsha** – Binary sha referring to a commit in the remote repository. See the *url* parameter.

- **parent_commit** – The *Commit* whose tree is supposed to contain the `.gitmodules` blob, or `None` to always point to the most recent commit. See *set_parent_commit()* for details.
- **url** – The URL to the remote repository which is the submodule.
- **branch_path** – Full repository-relative path to ref to checkout when cloning the remote repository.

__module__ = 'git.objects.submodule.base'

__ne__(*other: object*) → bool
Compare with another submodule for inequality.

__parameters__ = ()

__repr__() → str

Returns String with pythonic representation of our object

__slots__ = ('_parent_commit', '_url', '_branch_path', '_name', '__weakref__')

__str__() → str

Returns String of our SHA1 as understood by all git commands

classmethod __subclasshook__(*other*)
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__

classmethod add(*repo: Repo*, *name: str*, *path: Union[str, os.PathLike[str]]*, *url: Optional[str] = None*, *branch: Optional[str] = None*, *no_checkout: bool = False*, *depth: Optional[int] = None*, *env: Optional[Mapping[str, str]] = None*, *clone_multi_options: Optional[Sequence[Any]] = None*, *allow_unsafe_options: bool = False*, *allow_unsafe_protocols: bool = False*) → *Submodule*
Add a new submodule to the given repository. This will alter the index as well as the `.gitmodules` file, but will not create a new commit. If the submodule already exists, no matter if the configuration differs from the one provided, the existing submodule will be returned.

Parameters

- **repo** – Repository instance which should receive the submodule.
- **name** – The name/identifier for the submodule.
- **path** – Repository-relative or absolute path at which the submodule should be located. It will be created as required during the repository initialization.
- **url** – `git clone ...`-compatible URL. See *git-clone(1)* for more information. If `None`, the repository is assumed to exist, and the URL of the first remote is taken instead. This is useful if you want to make an existing repository a submodule of another one.
- **branch** – Name of branch at which the submodule should (later) be checked out. The given branch must exist in the remote repository, and will be checked out locally as a tracking branch. It will only be written into the configuration if it not `None`, which is when the checked out branch will be the one the remote HEAD pointed to. The result you get in

these situation is somewhat fuzzy, and it is recommended to specify at least `master` here. Examples are `master` or `feature/new`.

- **no_checkout** – If `True`, and if the repository has to be cloned manually, no checkout will be performed.
- **depth** – Create a shallow clone with a history truncated to the specified number of commits.
- **env** – Optional dictionary containing the desired environment variables.

  Note: Provided variables will be used to update the execution environment for `git`. If some variable is not specified in *env* and is defined in attr:*os.environ*, the value from attr:*os.environ* will be used. If you want to unset some variable, consider providing an empty string as its value.
- **clone_multi_options** – A list of clone options. Please see *Repo.clone* for details.
- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.
- **allow_unsafe_options** – Allow unsafe options to be used, like `--upload-pack`.

Returns The newly created *Submodule* instance.

Note Works atomically, such that no change will be done if, for example, the repository update fails.

property branch: Head

Returns The branch instance that we are to checkout

Raises *git.exc.InvalidGitRepositoryError* – If our module is not yet checked out.

property branch_name: str

Returns The name of the branch, which is the shortest possible branch name

property branch_path: Union[str, os.PathLike[str]]

Returns Full repository-relative path as string to the branch we would checkout from the remote and track

children() → *git.util.IterableList*[*git.objects.submodule.base.Submodule*]

Returns IterableList(Submodule, ...) An iterable list of *Submodule* instances which are children of this submodule or 0 if the submodule is not checked out.

config_reader() → *git.config.SectionConstraint*[*git.objects.submodule.util.SubmoduleConfigParser*]

Returns ConfigReader instance which allows you to query the configuration values of this submodule, as provided by the `.gitmodules` file.

Note The config reader will actually read the data directly from the repository and thus does not need nor care about your working tree.

Note Should be cached by the caller and only kept as long as needed.

Raises **IOError** – If the `.gitmodules` file/blob could not be read.

config_writer(*index: Optional[IndexFile] = None*, *write: bool = True*) → *git.config.SectionConstraint*[SubmoduleConfigParser]

Returns A config writer instance allowing you to read and write the data belonging to this submodule into the `.gitmodules` file.

Parameters

- **index** – If not `None`, an *IndexFile* instance which should be written. Defaults to the index of the *Submodule*'s parent repository.
- **write** – If `True`, the index will be written each time a configuration value changes.

Note The parameters allow for a more efficient writing of the index, as you can pass in a modified index on your own, prevent automatic writing, and write yourself once the whole operation is complete.

Raises

- **ValueError** – If trying to get a writer on a parent_commit which does not match the current head commit.
- **IOError** – If the `.gitmodules` file/blob could not be read.

exists() → bool

Returns `True` if the submodule exists, `False` otherwise. Please note that a submodule may exist (in the `.gitmodules` file) even though its module doesn't exist on disk.

classmethod iter_items(*repo: Repo*, *parent_commit: Union[Commit, TagObject, str] = 'HEAD'*, **args: Any*, ***kwargs: Any*) → Iterator[*Submodule*]

Returns Iterator yielding *Submodule* instances available in the given repository

k_default_mode = 57344
Submodule flags. Submodules are directories with link-status.

k_head_default = 'master'

k_head_option = 'branch'

k_modules_file = '.gitmodules'

module() → Repo

Returns *Repo* instance initialized from the repository at our submodule path

Raises ***git.exc.InvalidGitRepositoryError*** – If a repository was not available. This could also mean that it was not yet initialized.

module_exists() → bool

Returns `True` if our module exists and is a valid git repository. See the *module()* method.

move(*module_path: Union[str, os.PathLike[str]]*, *configuration: bool = True*, *module: bool = True*) → *git.objects.submodule.base.Submodule*
Move the submodule to a another module path. This involves physically moving the repository at our current path, changing the configuration, as well as adjusting our index entry accordingly.

Parameters

- **module_path** – The path to which to move our module in the parent repository's working tree, given as repository-relative or absolute path. Intermediate directories will be created accordingly. If the path already exists, it must be empty. Trailing (back)slashes are removed automatically.

- **configuration** – If `True`, the configuration will be adjusted to let the submodule point to the given path.
- **module** – If `True`, the repository managed by this submodule will be moved as well. If `False`, we don't move the submodule's checkout, which may leave the parent repository in an inconsistent state.

Returns self

Raises **ValueError** – If the module path existed and was not empty, or was a file.

Note Currently the method is not atomic, and it could leave the repository in an inconsistent state if a sub-step fails for some reason.

property name: str

Returns The name of this submodule. It is used to identify it within the `.gitmodules` file.

Note By default, this is the name is the path at which to find the submodule, but in GitPython it should be a unique identifier similar to the identifiers used for remotes, which allows to change the path of the submodule easily.

property parent_commit: Commit

Returns *Commit* instance with the tree containing the `.gitmodules` file

Note Will always point to the current head's commit if it was not set explicitly.

remove(*module: bool = True*, *force: bool = False*, *configuration: bool = True*, *dry_run: bool = False*) → *git.objects.submodule.base.Submodule*

Remove this submodule from the repository. This will remove our entry from the `.gitmodules` file and the entry in the `.git/config` file.

Parameters

- **module** – If `True`, the checked out module we point to will be deleted as well. If that module is currently on a commit outside any branch in the remote, or if it is ahead of its tracking branch, or if there are modified or untracked files in its working tree, then the removal will fail. In case the removal of the repository fails for these reasons, the submodule status will not have been altered.

  If this submodule has child modules of its own, these will be deleted prior to touching the direct submodule.
- **force** – Enforces the deletion of the module even though it contains modifications. This basically enforces a brute-force file system based deletion.
- **configuration** – If `True`, the submodule is deleted from the configuration, otherwise it isn't. Although this should be enabled most of the time, this flag enables you to safely delete the repository of your submodule.
- **dry_run** – If `True`, we will not actually do anything, but throw the errors we would usually throw.

Returns self

Note Doesn't work in bare repositories.

Note Doesn't work atomically, as failure to remove any part of the submodule will leave an inconsistent state.

Raises

- *git.exc.InvalidGitRepositoryError* – Thrown if the repository cannot be deleted.

- **OSError** – If directories or files could not be removed.

rename(*new_name: str*) → *git.objects.submodule.base.Submodule*

Rename this submodule.

Note This method takes care of renaming the submodule in various places, such as:

- `$parent_git_dir / config`
- `$working_tree_dir / .gitmodules`
- (git >= v1.8.0: move submodule repository to new name)

As `.gitmodules` will be changed, you would need to make a commit afterwards. The changed `.gitmodules` file will already be added to the index.

Returns This *Submodule* instance

set_parent_commit(*commit: Optional[Union[Commit, TagObject, str]]*, *check: bool = True*) → *Submodule*

Set this instance to use the given commit whose tree is supposed to contain the `.gitmodules` blob.

Parameters

- **commit** – Commit-ish reference pointing at the root tree, or `None` to always point to the most recent commit.
- **check** – If `True`, relatively expensive checks will be performed to verify validity of the submodule.

Raises

- **ValueError** – If the commit's tree didn't contain the `.gitmodules` blob.
- **ValueError** – If the parent commit didn't store this submodule under the current path.

Returns self

type: Literal['submodule'] = 'submodule'

This is a bogus type string for base class compatibility.

update(*recursive: bool = False*, *init: bool = True*, *to_latest_revision: bool = False*, *progress: Optional[git.objects.submodule.base.UpdateProgress] = None*, *dry_run: bool = False*, *force: bool = False*, *keep_going: bool = False*, *env: Optional[Mapping[str, str]] = None*, *clone_multi_options: Optional[Sequence[Any]] = None*, *allow_unsafe_options: bool = False*, *allow_unsafe_protocols: bool = False*) → *git.objects.submodule.base.Submodule*

Update the repository of this submodule to point to the checkout we point at with the binsha of this instance.

Parameters

- **recursive** – If `True`, we will operate recursively and update child modules as well.
- **init** – If `True`, the module repository will be cloned into place if necessary.
- **to_latest_revision** – If `True`, the submodule's sha will be ignored during checkout. Instead, the remote will be fetched, and the local tracking branch updated. This only works if we have a local tracking branch, which is the case if the remote repository had a master branch, or if the `branch` option was specified for this submodule and the branch existed remotely.
- **progress** – *UpdateProgress* instance, or `None` if no progress should be shown.
- **dry_run** – If `True`, the operation will only be simulated, but not performed. All performed operations are read-only.

- **force** – If `True`, we may reset heads even if the repository in question is dirty. Additionally we will be allowed to set a tracking branch which is ahead of its remote branch back into the past or the location of the remote branch. This will essentially 'forget' commits.

  If `False`, local tracking branches that are in the future of their respective remote branches will simply not be moved.

- **keep_going** – If `True`, we will ignore but log all errors, and keep going recursively. Unless *dry_run* is set as well, *keep_going* could cause subsequent/inherited errors you wouldn't see otherwise. In conjunction with *dry_run*, it can be useful to anticipate all errors when updating submodules.

- **env** – Optional dictionary containing the desired environment variables.

  Note: Provided variables will be used to update the execution environment for `git`. If some variable is not specified in *env* and is defined in attr:*os.environ*, value from attr:*os.environ* will be used.

  If you want to unset some variable, consider providing the empty string as its value.

- **clone_multi_options** – List of *git-clone(1)* options. Please see *Repo.clone* for details. They only take effect with the *init* option.

- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.

- **allow_unsafe_options** – Allow unsafe options to be used, like `--upload-pack`.

Note Does nothing in bare repositories.

Note This method is definitely not atomic if *recursive* is `True`.

Returns self

property url: str

Returns The url to the repository our submodule's repository refers to

class git.objects.submodule.base.**UpdateProgress**

Class providing detailed progress information to the caller who should derive from it and implement the *update(...)* message.

CLONE = 512

FETCH = 1024

UPDWKTREE = 2048

__annotations__ = {'_cur_line': 'Optional[str]', '_num_op_codes': <class 'int'>, '_seen_ops': 'List[int]', 'error_lines': 'List[str]', 'other_lines': 'List[str]'}

__module__ = 'git.objects.submodule.base'

__slots__ = ()

4.9 Objects.Submodule.root

class git.objects.submodule.root.**RootModule**(*repo: Repo*)

A (virtual) root of all submodules in the given repository.

This can be used to more easily traverse all submodules of the superproject (master repository).

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': 'str', 'type': "Literal['submodule']"}

__init__(*repo: Repo*) → None

Initialize this instance with its attributes.

We only document the parameters that differ from *IndexObject*.

Parameters

- **repo** – Our parent repository.
- **binsha** – Binary sha referring to a commit in the remote repository. See the *url* parameter.
- **parent_commit** – The *Commit* whose tree is supposed to contain the `.gitmodules` blob, or `None` to always point to the most recent commit. See `set_parent_commit()` for details.
- **url** – The URL to the remote repository which is the submodule.
- **branch_path** – Full repository-relative path to ref to checkout when cloning the remote repository.

__module__ = 'git.objects.submodule.root'

__parameters__ = ()

__slots__ = ()

classmethod __subclasshook__(*other*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

k_root_name = '__ROOT__'

module() → Repo

Returns The actual repository containing the submodules

update(*previous_commit: Optional[Union[Commit, TagObject, str]] = None*, *recursive: bool = True*, *force_remove: bool = False*, *init: bool = True*, *to_latest_revision: bool = False*, *progress: Union[None, RootUpdateProgress] = None*, *dry_run: bool = False*, *force_reset: bool = False*, *keep_going: bool = False*) → *RootModule*

Update the submodules of this repository to the current HEAD commit.

This method behaves smartly by determining changes of the path of a submodule's repository, next to changes to the to-be-checked-out commit or the branch to be checked out. This works if the submodule's ID does not change.

Additionally it will detect addition and removal of submodules, which will be handled gracefully.

Parameters

- **previous_commit** – If set to a commit-ish, the commit we should use as the previous commit the HEAD pointed to before it was set to the commit it points to now. If `None`, it defaults to `HEAD@{1}` otherwise.
- **recursive** – If `True`, the children of submodules will be updated as well using the same technique.
- **force_remove** – If submodules have been deleted, they will be forcibly removed. Otherwise the update may fail if a submodule's repository cannot be deleted as changes have been made to it. (See *Submodule.update* for more information.)
- **init** – If we encounter a new module which would need to be initialized, then do it.
- **to_latest_revision** – If `True`, instead of checking out the revision pointed to by this submodule's sha, the checked out tracking branch will be merged with the latest remote branch fetched from the repository's origin.

  Unless *force_reset* is specified, a local tracking branch will never be reset into its past, therefore the remote branch must be in the future for this to have an effect.
- **force_reset** – If `True`, submodules may checkout or reset their branch even if the repository has pending changes that would be overwritten, or if the local tracking branch is in the future of the remote tracking branch and would be reset into its past.
- **progress** – *RootUpdateProgress* instance, or `None` if no progress should be sent.
- **dry_run** – If `True`, operations will not actually be performed. Progress messages will change accordingly to indicate the WOULD DO state of the operation.
- **keep_going** – If `True`, we will ignore but log all errors, and keep going recursively. Unless *dry_run* is set as well, *keep_going* could cause subsequent/inherited errors you wouldn't see otherwise. In conjunction with *dry_run*, this can be useful to anticipate all errors when updating submodules.

Returns self

class git.objects.submodule.root.**RootUpdateProgress**

Utility class which adds more opcodes to *UpdateProgress*.

BRANCHCHANGE = 16384

PATHCHANGE = 8192

REMOVE = 4096

URLCHANGE = 32768

__annotations__ = {'_cur_line': 'Optional[str]', '_num_op_codes': 'int', '_seen_ops': 'List[int]', 'error_lines': 'List[str]', 'other_lines': 'List[str]'}

__module__ = 'git.objects.submodule.root'

__slots__ = ()

4.10 Objects.Submodule.util

class git.objects.submodule.util.**SubmoduleConfigParser**(**args: Any*, ***kwargs: Any*)

Catches calls to `write()`, and updates the `.gitmodules` blob in the index with the new data, if we have written into a stream.

Otherwise it would add the local file to the index to make it correspond with the working tree. Additionally, the cache must be cleared.

Please note that no mutating method will work in bare mode.

__abstractmethods__ = frozenset({})

__init__(**args: Any*, ***kwargs: Any*) → None

Initialize a configuration reader to read the given *file_or_files* and to possibly allow changes to it by setting *read_only* False.

Parameters

- **file_or_files** – A file path or file object, or a sequence of possibly more than one of them.
- **read_only** – If `True`, the ConfigParser may only read the data, but not change it. If `False`, only a single file path or file object may be given. We will write back the changes when they happen, or when the ConfigParser is released. This will not happen if other configuration files have been included.
- **merge_includes** – If `True`, we will read files mentioned in `[include]` sections and merge their contents into ours. This makes it impossible to write back an individual configuration file. Thus, if you want to modify a single configuration file, turn this off to leave the original dataset unaltered when reading it.
- **repo** – Reference to repository to use if `[includeIf]` sections are found in configuration files.

__module__ = 'git.objects.submodule.util'

flush_to_index() → None

Flush changes in our configuration file to the index.

set_submodule(*submodule: Submodule*) → None

Set this instance's submodule. It must be called before the first write operation begins.

write() → None

Write changes to our file, if there are changes at all.

Raises **IOError** – If this is a read-only writer instance or if we could not obtain a file lock.

git.objects.submodule.util.**find_first_remote_branch**(*remotes: Sequence[Remote]*, *branch_name: str*) → RemoteReference

Find the remote branch matching the name of the given branch or raise *InvalidGitRepositoryError*.

git.objects.submodule.util.**mkhead**(*repo: Repo*, *path: Union[str, os.PathLike[str]]*) → Head

Returns New branch/head instance

git.objects.submodule.util.**sm_name**(*section: str*) → str

Returns Name of the submodule as parsed from the section name

git.objects.submodule.util.**sm_section**(*name: str*) → str

Returns Section title used in `.gitmodules` configuration file

4.11 Objects.Util

Utility functions for working with git objects.

class git.objects.util.**Actor**(*name: Optional[str]*, *email: Optional[str]*)

Actors hold information about a person acting on the repository. They can be committers and authors or anything with a name and an email as mentioned in the git log entries.

__eq__(*other: Any*) → bool
Return self==value.

__hash__() → int
Return hash(self).

__init__(*name: Optional[str]*, *email: Optional[str]*) → None

__module__ = 'git.util'

__ne__(*other: Any*) → bool
Return self!=value.

__repr__() → str
Return repr(self).

__slots__ = ('name', 'email')

__str__() → str
Return str(self).

classmethod author(*config_reader: Union[None, GitConfigParser, SectionConstraint] = None*) → *Actor*
Same as *committer()*, but defines the main author. It may be specified in the environment, but defaults to the committer.

classmethod committer(*config_reader: Union[None, GitConfigParser, SectionConstraint] = None*) → *Actor*

Returns *Actor* instance corresponding to the configured committer. It behaves similar to the git implementation, such that the environment will override configuration values of *config_reader*. If no value is set at all, it will be generated.

Parameters **config_reader** – ConfigReader to use to retrieve the values from in case they are not set in the environment.

conf_email = 'email'

conf_name = 'name'

email

env_author_email = 'GIT_AUTHOR_EMAIL'

env_author_name = 'GIT_AUTHOR_NAME'

env_committer_email = 'GIT_COMMITTER_EMAIL'

env_committer_name = 'GIT_COMMITTER_NAME'

name

name_email_regex = re.compile('(.*) <(.*?)>')

name_only_regex = re.compile('<(.*)>')

class git.objects.util.**ProcessStreamAdapter**(*process: Popen*, *stream_name: str*)
Class wiring all calls to the contained Process instance.

Use this type to hide the underlying process to provide access only to a specified stream. The process is usually wrapped into an *AutoInterrupt* class to kill it if the instance goes out of scope.

__getattr__(*attr: str*) → Any

__init__(*process: Popen*, *stream_name: str*) → None

__module__ = 'git.objects.util'

__slots__ = ('_proc', '_stream')

class git.objects.util.**Traversable**
Simple interface to perform depth-first or breadth-first traversals in one direction.

Subclasses only need to implement one function.

Instances of the subclass must be hashable.

Defined subclasses:

- Commit
- *Tree*
- *Submodule*

__abstractmethods__ = frozenset({'_get_intermediate_items', 'list_traverse', 'traverse'})

__annotations__ = {}

__module__ = 'git.objects.util'

__slots__ = ()

abstract list_traverse(**args: Any*, ***kwargs: Any*) → Any
Traverse self and collect all items found.

Calling this directly on the abstract base class, including via a `super()` proxy, is deprecated. Only overridden implementations should be called.

abstract traverse(**args: Any*, ***kwargs: Any*) → Any
Iterator yielding items found when traversing self.

Calling this directly on the abstract base class, including via a `super()` proxy, is deprecated. Only overridden implementations should be called.

git.objects.util.**altz_to_utctz_str**(*altz: float*) → str
Convert a timezone offset west of UTC in seconds into a Git timezone offset string.

Parameters **altz** – Timezone offset in seconds west of UTC.

git.objects.util.**get_object_type_by_name**(*object_type_name: bytes*) → Union[Type[Commit], Type[TagObject], Type[Tree], Type[Blob]]
Retrieve the Python class GitPython uses to represent a kind of Git object.

Returns A type suitable to handle the given as *object_type_name*. This type can be called create new instances.

Parameters **object_type_name** – Member of *Object.TYPES*.

Raises **ValueError** – If *object_type_name* is unknown.

git.objects.util.**parse_actor_and_date**(*line: str*) → Tuple[*git.util.Actor*, int, int]
Parse out the actor (author or committer) info from a line like:

```
author Tom Preston-Werner <tom@mojombo.com> 1191999972 -0700
```

Returns [Actor, int_seconds_since_epoch, int_timezone_offset]

git.objects.util.**parse_date**(*string_date: Union[str, datetime.datetime]*) → Tuple[int, int]
Parse the given date as one of the following:

- Aware datetime instance
- Git internal format: timestamp offset
- RFC 2822: `Thu, 07 Apr 2005 22:13:13 +0200`
- ISO 8601: `2005-04-07T22:13:13` - The `T` can be a space as well.

Returns Tuple(int(timestamp_UTC), int(offset)), both in seconds since epoch

Raises **ValueError** – If the format could not be understood.

Note Date can also be `YYYY.MM.DD`, `MM/DD/YYYY` and `DD.MM.YYYY`.

class git.objects.util.**tzoffset**(*secs_west_of_utc: float*, *name: Union[None, str] = None*)

__dict__ = mappingproxy({'__module__': 'git.objects.util', '__init__': <function tzoffset.__init__>, '__reduce__': <function tzoffset.__reduce__>, 'utcoffset': <function tzoffset.utcoffset>, 'tzname': <function tzoffset.tzname>, 'dst': <function tzoffset.dst>, '__dict__': <attribute '__dict__' of 'tzoffset' objects>, '__weakref__': <attribute '__weakref__' of 'tzoffset' objects>, '__doc__': None, '__annotations__': {}})

__init__(*secs_west_of_utc: float*, *name: Union[None, str] = None*) → None

__module__ = 'git.objects.util'

__reduce__() → Tuple[Type[*git.objects.util.tzoffset*], Tuple[float, str]]
-> (cls, state)

__weakref__
list of weak references to the object (if defined)

dst(*dt: Optional[datetime.datetime]*) → datetime.timedelta
datetime -> DST offset as timedelta positive east of UTC.

tzname(*dt: Optional[datetime.datetime]*) → str
datetime -> string name of time zone.

utcoffset(*dt: Optional[datetime.datetime]*) → datetime.timedelta
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

git.objects.util.**utctz_to_altz**(*utctz: str*) → int
Convert a git timezone offset into a timezone offset west of UTC in seconds (compatible with `time.altzone`).

Parameters **utctz** – git utc timezone string, e.g. +0200

git.objects.util.**verify_utctz**(*offset: str*) → str

Raises **ValueError** – If *offset* is incorrect.

Returns *offset*

4.12 Index.Base

Module containing *IndexFile*, an Index implementation facilitating all kinds of index manipulations such as querying and merging.

exception git.index.base.**CheckoutError**(*message: str*, *failed_files: Sequence[Union[str, os.PathLike[str]]]*, *valid_files: Sequence[Union[str, os.PathLike[str]]]*, *failed_reasons: List[str]*)

Thrown if a file could not be checked out from the index as it contained changes.

The `failed_files` attribute contains a list of relative paths that failed to be checked out as they contained changes that did not exist in the index.

The `failed_reasons` attribute contains a string informing about the actual cause of the issue.

The `valid_files` attribute contains a list of relative paths to files that were checked out successfully and hence match the version stored in the index.

__init__(*message: str*, *failed_files: Sequence[Union[str, os.PathLike[str]]]*, *valid_files: Sequence[Union[str, os.PathLike[str]]]*, *failed_reasons: List[str]*) → None

__module__ = 'git.exc'

__str__() → str

Return str(self).

class git.index.base.**IndexFile**(*repo: Repo*, *file_path: Optional[Union[str, os.PathLike[str]]] = None*)

An Index that can be manipulated using a native implementation in order to save git command function calls wherever possible.

This provides custom merging facilities allowing to merge without actually changing your index or your working tree. This way you can perform your own test merges based on the index only without having to deal with the working copy. This is useful in case of partial working trees.

Entries:

The index contains an entries dict whose keys are tuples of type *IndexEntry* to facilitate access.

You may read the entries dict or manipulate it using IndexEntry instance, i.e.:

```
index.entries[index.entry_key(index_entry_instance)] = index_entry_instance
```

Make sure you use *index.write()* once you are done manipulating the index directly before operating on it using the git command.

S_IFGITLINK = 57344

Flags for a submodule.

__abstractmethods__ = frozenset({})

__annotations__ = {'_file_path': 'PathLike', 'repo': "'Repo'"}

__init__(*repo: Repo*, *file_path: Optional[Union[str, os.PathLike[str]]] = None*) → None
Initialize this Index instance, optionally from the given *file_path*.

If no *file_path* is given, we will be created from the current index file.

If a stream is not given, the stream will be initialized from the current repository's index on demand.

__module__ = 'git.index.base'

__slots__ = ('repo', 'version', 'entries', '_extension_data', '_file_path')

add(*items: Sequence[Union[str, os.PathLike[str], git.objects.blob.Blob, git.index.typ.BaseIndexEntry, git.objects.submodule.base.Submodule]], force: bool = True, fprogress: Callable = <function IndexFile.<lambda>>, path_rewriter: Optional[Callable[[...], Union[str, os.PathLike[str]]]] = None, write: bool = True, write_extension_data: bool = False*) → List[*git.index.typ.BaseIndexEntry*]
Add files from the working tree, specific blobs, or `BaseIndexEntry`s to the index.

Parameters

- **items** – Multiple types of items are supported, types can be mixed within one call. Different types imply a different handling. File paths may generally be relative or absolute.
  - path string

    Strings denote a relative or absolute path into the repository pointing to an existing file, e.g., `CHANGES`, *lib/myfile.ext*`, `/home/gitrepo/lib/myfile.ext`.

    Absolute paths must start with working tree directory of this index's repository to be considered valid. For example, if it was initialized with a non-normalized path, like `/root/repo/../repo`, absolute paths to be added must start with `/root/repo/../repo`.

    Paths provided like this must exist. When added, they will be written into the object database.

    PathStrings may contain globs, such as `lib/__init__*`. Or they can be directories like `lib`, which will add all the files within the directory and subdirectories.

    This equals a straight *git-add(1)*.

    They are added at stage 0.

  - :class:~`git.objects.blob.Blob` or `Submodule` object

    Blobs are added as they are assuming a valid mode is set.

    The file they refer to may or may not exist in the file system, but must be a path relative to our repository.

    If their sha is null (40*0), their path must exist in the file system relative to the git repository as an object will be created from the data at the path.

    The handling now very much equals the way string paths are processed, except that the mode you have set will be kept. This allows you to create symlinks by settings the mode respectively and writing the target of the symlink directly into the file. This equals a default Linux symlink which is not dereferenced automatically, except that it can be created on filesystems not supporting it as well.

    Please note that globs or directories are not allowed in `Blob` objects.

    They are added at stage 0.

  - `BaseIndexEntry` or type

Handling equals the one of :class:~`git.objects.blob.Blob` objects, but the stage may be explicitly set. Please note that Index Entries require binary sha's.

- **force** – **CURRENTLY INEFFECTIVE** If `True`, otherwise ignored or excluded files will be added anyway. As opposed to the *git-add(1)* command, we enable this flag by default as the API user usually wants the item to be added even though they might be excluded.
- **fprogress** – Function with signature `f(path, done=False, item=item)` called for each path to be added, one time once it is about to be added where `done=False` and once after it was added where `done=True`.

  `item` is set to the actual item we handle, either a path or a *BaseIndexEntry*.

  Please note that the processed path is not guaranteed to be present in the index already as the index is currently being processed.
- **path_rewriter** – Function, with signature `(string) func(BaseIndexEntry)`, returning a path for each passed entry which is the path to be actually recorded for the object created from `entry.path`. This allows you to write an index which is not identical to the layout of the actual files on your hard-disk. If not `None` and *items* contain plain paths, these paths will be converted to Entries beforehand and passed to the path_rewriter. Please note that `entry.path` is relative to the git repository.
- **write** – If `True`, the index will be written once it was altered. Otherwise the changes only exist in memory and are not available to git commands.
- **write_extension_data** – If `True`, extension data will be written back to the index. This can lead to issues in case it is containing the 'TREE' extension, which will cause the *git-commit(1)* command to write an old tree, instead of a new one representing the now changed index.

  This doesn't matter if you use *IndexFile.commit()*, which ignores the 'TREE' extension altogether. You should set it to `True` if you intend to use *IndexFile.commit()* exclusively while maintaining support for third-party extensions. Besides that, you can usually safely ignore the built-in extensions when using GitPython on repositories that are not handled manually at all.

  All current built-in extensions are listed here: https://git-scm.com/docs/index-format

Returns List of *BaseIndexEntry*s representing the entries just actually added.

Raises **OSError** – If a supplied path did not exist. Please note that *BaseIndexEntry* objects that do not have a null sha will be added even if their paths do not exist.

checkout(*paths: Union[None, Iterable[Union[str, os.PathLike[str]]]] = None*, *force: bool = False*, *fprogress: Callable = <function IndexFile.<lambda>>*, ***kwargs: Any*) → Union[None, Iterator[Union[str, os.PathLike[str]]], Sequence[Union[str, os.PathLike[str]]]]

Check out the given paths or all files from the version known to the index into the working tree.

Note Be sure you have written pending changes using the *write()* method in case you have altered the entries dictionary directly.

Parameters

- **paths** – If `None`, all paths in the index will be checked out. Otherwise an iterable of relative or absolute paths or a single path pointing to files or directories in the index is expected.
- **force** – If `True`, existing files will be overwritten even if they contain local modifications. If `False`, these will trigger a *CheckoutError*.
- **fprogress** – See *IndexFile.add()* for signature and explanation.

The provided progress information will contain `None` as path and item if no explicit paths are given. Otherwise progress information will be send prior and after a file has been checked out.

- **kwargs** – Additional arguments to be passed to *git-checkout-index(1)*.

Returns Iterable yielding paths to files which have been checked out and are guaranteed to match the version stored in the index.

Raises

- *git.exc.CheckoutError* –
  - If at least one file failed to be checked out. This is a summary, hence it will checkout as many files as it can anyway.
  - If one of files or directories do not exist in the index (as opposed to the original git command, which ignores them).
- *git.exc.GitCommandError* – If error lines could not be parsed - this truly is an exceptional state.

Note The checkout is limited to checking out the files in the index. Files which are not in the index anymore and exist in the working tree will not be deleted. This behaviour is fundamentally different to `head.checkout`, i.e. if you want *git-checkout(1)*-like behaviour, use `head.checkout` instead of `index.checkout`.

commit(*message: str*, *parent_commits: Optional[List[*git.objects.commit.Commit*]] = None*, *head: bool = True*, *author: Union[None, Actor] = None*, *committer: Union[None, Actor] = None*, *author_date: Optional[Union[datetime.datetime, str]] = None*, *commit_date: Optional[Union[datetime.datetime, str]] = None*, *skip_hooks: bool = False*) → *git.objects.commit.Commit*

Commit the current default index file, creating a *Commit* object.

For more information on the arguments, see *Commit.create_from_tree*.

Note If you have manually altered the *entries* member of this instance, don't forget to *write()* your changes to disk beforehand.

Note Passing `skip_hooks=True` is the equivalent of using `-n` or `--no-verify` on the command line.

Returns *Commit* object representing the new commit

diff(*other: Optional[Union[Literal[<DiffConstants.INDEX: 2>], git.objects.tree.Tree, git.objects.commit.Commit, str]] = DiffConstants.INDEX*, *paths: Optional[Union[str, os.PathLike[str], List[Union[str, os.PathLike[str]]], Tuple[Union[str, os.PathLike[str]], ...]]] = None*, *create_patch: bool = False*, ***kwargs: Any*) → *git.diff.DiffIndex*

Diff this index against the working copy or a *Tree* or *Commit* object.

For documentation of the parameters and return values, see *Diffable.diff*.

Note Will only work with indices that represent the default git index as they have not been initialized with a stream.

entries

classmethod entry_key(**entry: Union[*git.index.typ.BaseIndexEntry*, str, os.PathLike[str], int]*) → Tuple[Union[str, os.PathLike[str]], int]

classmethod from_tree(*repo: Repo*, **treeish: Union[*git.objects.tree.Tree*, *git.objects.commit.Commit*, str, bytes]*, ***kwargs: Any*) → *IndexFile*

Merge the given treeish revisions into a new index which is returned. The original index will remain unaltered.

Parameters

- **repo** – The repository treeish are located in.
- **treeish** – One, two or three *Tree* objects, *Commit*s or 40 byte hexshas.

  The result changes according to the amount of trees:

  1. If 1 Tree is given, it will just be read into a new index.
  2. If 2 Trees are given, they will be merged into a new index using a two way merge algorithm. Tree 1 is the 'current' tree, tree 2 is the 'other' one. It behaves like a fast-forward.
  3. If 3 Trees are given, a 3-way merge will be performed with the first tree being the common ancestor of tree 2 and tree 3. Tree 2 is the 'current' tree, tree 3 is the 'other' one.
- **kwargs** – Additional arguments passed to *git-read-tree(1)*.

Returns New *IndexFile* instance. It will point to a temporary index location which does not exist anymore. If you intend to write such a merged Index, supply an alternate `file_path` to its *write()* method.

Note In the three-way merge case, `--aggressive` will be specified to automatically resolve more cases in a commonly correct manner. Specify `trivial=True` as a keyword argument to override that.

As the underlying *git-read-tree(1)* command takes into account the current index, it will be temporarily moved out of the way to prevent any unexpected interference.

iter_blobs(*predicate: Callable[[Tuple[int, git.objects.blob.Blob]], bool] = <function IndexFile.<lambda>>*) → Iterator[Tuple[int, *git.objects.blob.Blob*]]

Returns Iterator yielding tuples of *Blob* objects and stages, tuple(stage, Blob).

Parameters **predicate** – Function(t) returning `True` if tuple(stage, Blob) should be yielded by the iterator. A default filter, the *~git.index.typ.BlobFilter*, allows you to yield blobs only if they match a given list of paths.

merge_tree(*rhs: Union[*git.objects.tree.Tree, git.objects.commit.Commit*, str, bytes]*, *base: Union[None,* git.objects.tree.Tree, git.objects.commit.Commit*, str, bytes] = None*) → *git.index.base.IndexFile*

Merge the given *rhs* treeish into the current index, possibly taking a common base treeish into account.

As opposed to the *from_tree()* method, this allows you to use an already existing tree as the left side of the merge.

Parameters

- **rhs** – Treeish reference pointing to the 'other' side of the merge.
- **base** – Optional treeish reference pointing to the common base of *rhs* and this index which equals lhs.

Returns self (containing the merge and possibly unmerged entries in case of conflicts)

Raises *git.exc.GitCommandError* – If there is a merge conflict. The error will be raised at the first conflicting path. If you want to have proper merge resolution to be done by yourself, you have to commit the changed index (or make a valid tree from it) and retry with a three-way *index.from_tree* call.

move(*items: Sequence[Union[str, os.PathLike[str], git.objects.blob.Blob, git.index.typ.BaseIndexEntry, git.objects.submodule.base.Submodule]]*, *skip_errors: bool = False*, ***kwargs: Any*) → List[Tuple[str, str]]

Rename/move the items, whereas the last item is considered the destination of the move operation.

If the destination is a file, the first item (of two) must be a file as well.

If the destination is a directory, it may be preceded by one or more directories or files.

The working tree will be affected in non-bare repositories.

Parameters

- **items** – Multiple types of items are supported, please see the *remove()* method for reference.
- **skip_errors** – If `True`, errors such as ones resulting from missing source files will be skipped.
- **kwargs** – Additional arguments you would like to pass to *git-mv(1)*, such as `dry_run` or `force`.

Returns

List(tuple(source_path_string, destination_path_string), ...)

A list of pairs, containing the source file moved as well as its actual destination. Relative to the repository root.

Raises

- **ValueError** – If only one item was given.
- *git.exc.GitCommandError* – If git could not handle your request.

classmethod new(*repo: Repo*, **tree_sha: Union[str, git.objects.tree.Tree]*) → *IndexFile*

Merge the given treeish revisions into a new index which is returned.

This method behaves like `git-read-tree --aggressive` when doing the merge.

Parameters

- **repo** – The repository treeish are located in.
- **tree_sha** – 20 byte or 40 byte tree sha or tree objects.

Returns New *IndexFile* instance. Its path will be undefined. If you intend to write such a merged Index, supply an alternate `file_path` to its *write()* method.

property path: Union[str, os.PathLike[str]]

Returns Path to the index file we are representing

remove(*items: Sequence[Union[str, os.PathLike[str], git.objects.blob.Blob, git.index.typ.BaseIndexEntry, git.objects.submodule.base.Submodule]]*, *working_tree: bool = False*, ***kwargs: Any*) → List[str]

Remove the given items from the index and optionally from the working tree as well.

Parameters

- **items** – Multiple types of items are supported which may be be freely mixed.
  - path string

    Remove the given path at all stages. If it is a directory, you must specify the `r=True` keyword argument to remove all file entries below it. If absolute paths are given, they will be converted to a path relative to the git repository directory containing the working tree

The path string may include globs, such as `*.c`.

- :class:~`git.objects.blob.Blob` object

  Only the path portion is used in this case.

- *BaseIndexEntry* or compatible type

  The only relevant information here is the path. The stage is ignored.

- **working_tree** – If `True`, the entry will also be removed from the working tree, physically removing the respective file. This may fail if there are uncommitted changes in it.
- **kwargs** – Additional keyword arguments to be passed to *git-rm(1)*, such as `r` to allow recursive removal.

Returns

List(path_string, ...) list of repository relative paths that have been removed effectively.

This is interesting to know in case you have provided a directory or globs. Paths are relative to the repository.

repo: Repo

Repository to operate on. Must be provided by subclass or sibling class.

reset(*commit: Union[git.objects.commit.Commit, Reference, str] = 'HEAD'*, *working_tree: bool = False*, *paths: Union[None, Iterable[Union[str, os.PathLike[str]]]] = None*, *head: bool = False*, ***kwargs: Any*) → *IndexFile*

Reset the index to reflect the tree at the given commit. This will not adjust our HEAD reference by default, as opposed to *HEAD.reset*.

Parameters

- **commit** – Revision, *Reference* or *Commit* specifying the commit we should represent.

  If you want to specify a tree only, use *IndexFile.from_tree()* and overwrite the default index.

- **working_tree** – If `True`, the files in the working tree will reflect the changed index. If `False`, the working tree will not be touched. Please note that changes to the working copy will be discarded without warning!
- **head** – If `True`, the head will be set to the given commit. This is `False` by default, but if `True`, this method behaves like *HEAD.reset*.
- **paths** – If given as an iterable of absolute or repository-relative paths, only these will be reset to their state at the given commit-ish. The paths need to exist at the commit, otherwise an exception will be raised.
- **kwargs** – Additional keyword arguments passed to *git-reset(1)*.

Note *IndexFile.reset()*, as opposed to *HEAD.reset*, will not delete any files in order to maintain a consistent working tree. Instead, it will just check out the files according to their state in the index. If you want *git-reset(1)*-like behaviour, use *HEAD.reset* instead.

Returns self

resolve_blobs(*iter_blobs: Iterator[git.objects.blob.Blob]*) → *git.index.base.IndexFile*

Resolve the blobs given in blob iterator.

This will effectively remove the index entries of the respective path at all non-null stages and add the given blob as new stage null blob.

For each path there may only be one blob, otherwise a `ValueError` will be raised claiming the path is already at stage 0.

Raises `ValueError` – If one of the blobs already existed at stage 0.

Returns self

Note You will have to write the index manually once you are done, i.e. `index.resolve_blobs(blobs).write()`.

`unmerged_blobs()` → Dict[Union[str, os.PathLike[str]], List[Tuple[int, *git.objects.blob.Blob*]]]

Returns Dict(path : list(tuple(stage, Blob, ...))), being a dictionary associating a path in the index with a list containing sorted stage/blob pairs.

Note Blobs that have been removed in one side simply do not exist in the given stage. That is, a file removed on the 'other' branch whose entries are at stage 3 will not have a stage 3 entry.

`update()` → *git.index.base.IndexFile*

Reread the contents of our index file, discarding all cached information we might have.

Note This is a possibly dangerous operations as it will discard your changes to *index.entries*.

Returns self

`version`

`write`(*file_path: Union[None, str, os.PathLike[str]] = None*, *ignore_extension_data: bool = False*) → None

Write the current state to our file path or to the given one.

Parameters

- `file_path` – If `None`, we will write to our stored file path from which we have been initialized. Otherwise we write to the given file path. Please note that this will change the *file_path* of this index to the one you gave.
- `ignore_extension_data` – If `True`, the TREE type extension data read in the index will not be written to disk. NOTE that no extension data is actually written. Use this if you have altered the index and would like to use *git-write-tree(1)* afterwards to create a tree representing your written changes. If this data is present in the written index, *git-write-tree(1)* will instead write the stored/cached tree. Alternatively, use *write_tree()* to handle this case automatically.

`write_tree()` → *git.objects.tree.Tree*

Write this index to a corresponding *Tree* object into the repository's object database and return it.

Returns *Tree* object representing this index.

Note The tree will be written even if one or more objects the tree refers to does not yet exist in the object database. This could happen if you added entries to the index directly.

Raises

- `ValueError` – If there are no entries in the cache.
- *git.exc.UnmergedEntriesError* –

`git.index.base.StageType`

alias of `int`

4.13 Index.Functions

Standalone functions to accompany the index implementation and make it more versatile.

git.index.fun.**S_IFGITLINK = 57344**
Flags for a submodule.

git.index.fun.**entry_key**(**entry: Union[*git.index.typ.BaseIndexEntry*, str, os.PathLike[str], int]*) → Tuple[Union[str, os.PathLike[str]], int]

Returns Key suitable to be used for the *index.entries* dictionary.

Parameters **entry** – One instance of type BaseIndexEntry or the path and the stage.

git.index.fun.**hook_path**(*name: str*, *git_dir: Union[str, os.PathLike[str]]*) → str

Returns path to the given named hook in the given git repository directory

git.index.fun.**read_cache**(*stream: IO[bytes]*) → Tuple[int, Dict[Tuple[Union[str, os.PathLike[str]], int], *git.index.typ.IndexEntry*], bytes, bytes]
Read a cache file from the given stream.

Returns

tuple(version, entries_dict, extension_data, content_sha)

- *version* is the integer version number.
- *entries_dict* is a dictionary which maps IndexEntry instances to a path at a stage.
- *extension_data* is "" or 4 bytes of type + 4 bytes of size + size bytes.
- *content_sha* is a 20 byte sha on all cache file contents.

git.index.fun.**run_commit_hook**(*name: str*, *index: IndexFile*, **args: str*) → None
Run the commit hook of the given name. Silently ignore hooks that do not exist.

Parameters

- **name** – Name of hook, like `pre-commit`.
- **index** – *IndexFile* instance.
- **args** – Arguments passed to hook file.

Raises *git.exc.HookExecutionError* –

git.index.fun.**stat_mode_to_index_mode**(*mode: int*) → int
Convert the given mode from a stat call to the corresponding index mode and return it.

git.index.fun.**write_cache**(*entries: Sequence[Union[git.index.typ.BaseIndexEntry, git.index.typ.IndexEntry]]*, *stream: IO[bytes]*, *extension_data: Union[None, bytes] = None*, *ShaStreamCls: Type[git.util.IndexFileSHA1Writer] = <class 'git.util.IndexFileSHA1Writer'>*) → None
Write the cache represented by entries to a stream.

Parameters

- **entries** – **Sorted** list of entries.
- **stream** – Stream to wrap into the AdapterStreamCls - it is used for final output.
- **ShaStreamCls** – Type to use when writing to the stream. It produces a sha while writing to it, before the data is passed on to the wrapped stream.

- **extension_data** – Any kind of data to write as a trailer, it must begin a 4 byte identifier, followed by its size (4 bytes).

git.index.fun.**write_tree_from_cache**(*entries: List[*git.index.typ.IndexEntry*]*, *odb: GitCmdObjectDB*, *sl: slice*, *si: int = 0*) → Tuple[bytes, List[TreeCacheTup]]

Create a tree from the given sorted list of entries and put the respective trees into the given object database.

Parameters

- **entries** – **Sorted** list of *IndexEntry*s.
- **odb** – Object database to store the trees in.
- **si** – Start index at which we should start creating subtrees.
- **sl** – Slice indicating the range we should process on the entries list.

Returns

tuple(binsha, list(tree_entry, ...))

A tuple of a sha and a list of tree entries being a tuple of hexsha, mode, name.

4.14 Index.Types

Additional types used by the index.

class git.index.typ.**BaseIndexEntry**(*inp_tuple: Union[Tuple[int, bytes, int, Union[str, os.PathLike[str]]], Tuple[int, bytes, int, Union[str, os.PathLike[str]], bytes, bytes, int, int, int, int, int]]*)

Small brother of an index entry which can be created to describe changes done to the index in which case plenty of additional information is not required.

As the first 4 data members match exactly to the *IndexEntry* type, methods expecting a *BaseIndexEntry* can also handle full *IndexEntry*s even if they use numeric indices for performance reasons.

__annotations__ = {}

__dict__ = mappingproxy({'__module__': 'git.index.typ', '__doc__': 'Small brother of an index entry which can be created to describe changes\n done to the index in which case plenty of additional information is not required.\n\n As the first 4 data members match exactly to the :class:`IndexEntry` type, methods\n expecting a :class:`BaseIndexEntry` can also handle full :class:`IndexEntry`\\s even\n if they use numeric indices for performance reasons.\n ', '__new__': <staticmethod(<function BaseIndexEntry.__new__>)>, '__str__': <function BaseIndexEntry.__str__>, '__repr__': <function BaseIndexEntry.__repr__>, 'hexsha': <property object>, 'stage': <property object>, 'from_blob': <classmethod(<function BaseIndexEntry.from_blob>)>, 'to_blob': <function BaseIndexEntry.to_blob>, '__dict__': <attribute '__dict__' of 'BaseIndexEntry' objects>, '__annotations__': {'mode': 'int', 'binsha': 'bytes', 'flags': 'int', 'path': 'PathLike', 'ctime_bytes': 'bytes', 'mtime_bytes': 'bytes', 'dev': 'int', 'inode': 'int', 'uid': 'int', 'gid': 'int', 'size': 'int'}})

__module__ = 'git.index.typ'

static __new__(*cls*, *inp_tuple: Union[Tuple[int, bytes, int, Union[str, os.PathLike[str]]], Tuple[int, bytes, int, Union[str, os.PathLike[str]], bytes, bytes, int, int, int, int, int]]*) → *BaseIndexEntry*

Override __new__ to allow construction from a tuple for backwards compatibility.

__repr__() → str
Return a nicely formatted representation string

__str__() → str
Return str(self).

classmethod from_blob(*blob:* git.objects.blob.Blob, *stage: int = 0*) → *git.index.typ.BaseIndexEntry*

Returns Fully equipped BaseIndexEntry at the given stage

property hexsha: str
hex version of our sha

property stage: int
Stage of the entry, either:

- 0 = default stage
- 1 = stage before a merge or common ancestor entry in case of a 3 way merge
- 2 = stage of entries from the 'left' side of the merge
- 3 = stage of entries from the 'right' side of the merge

Note For more information, see *git-read-tree(1)*.

to_blob(*repo: Repo*) → *git.objects.blob.Blob*

Returns Blob using the information of this index entry

class git.index.typ.BlobFilter(*paths: Sequence[Union[str, os.PathLike[str]]]*)
Predicate to be used by *IndexFile.iter_blobs* allowing to filter only return blobs which match the given list of directories or files.

The given paths are given relative to the repository.

__call__(*stage_blob: Tuple[int,* git.objects.blob.Blob*]*) → bool
Call self as a function.

__init__(*paths: Sequence[Union[str, os.PathLike[str]]]*) → None

Parameters **paths** – Tuple or list of paths which are either pointing to directories or to files relative to the current repository.

__module__ = 'git.index.typ'

__slots__ = ('paths',)

paths

class git.index.typ.IndexEntry(*inp_tuple: Union[Tuple[int, bytes, int, Union[str, os.PathLike[str]]], Tuple[int, bytes, int, Union[str, os.PathLike[str]], bytes, bytes, int, int, int, int, int]]*)
Allows convenient access to index entry data as defined in *BaseIndexEntry* without completely unpacking it.

Attributes usually accessed often are cached in the tuple whereas others are unpacked on demand.

See the properties for a mapping between names and tuple indices.

__annotations__ = {}

__module__ = 'git.index.typ'

property ctime: Tuple[int, int]

Returns Tuple(int_time_seconds_since_epoch, int_nano_seconds) of the file's creation time

classmethod from_base(*base:* git.index.typ.BaseIndexEntry) → *git.index.typ.IndexEntry*

Returns Minimal entry as created from the given *BaseIndexEntry* instance. Missing values will be set to null-like values.

Parameters **base** – Instance of type *BaseIndexEntry*.

classmethod from_blob(*blob:* git.objects.blob.Blob, *stage: int = 0*) → *git.index.typ.IndexEntry*

Returns Minimal entry resembling the given blob object

property mtime: Tuple[int, int]

See *ctime* property, but returns modification time.

git.index.typ.**StageType**

alias of `int`

4.15 Index.Util

Index utilities.

class git.index.util.**TemporaryFileSwap**(*file_path: Union[str, os.PathLike[str]]*)

Utility class moving a file to a temporary location within the same directory and moving it back on to where on object deletion.

__enter__() → *git.index.util.TemporaryFileSwap*

__exit__(*exc_type: Optional[Type[BaseException]]*, *exc_val: Optional[BaseException]*, *exc_tb: Optional[types.TracebackType]*) → Literal[False]

__init__(*file_path: Union[str, os.PathLike[str]]*) → None

__module__ = 'git.index.util'

__slots__ = ('file_path', 'tmp_file_path')

file_path

tmp_file_path

git.index.util.**default_index**(*func: Callable[[...], git.types._T]*) → Callable[[...], git.types._T]

Decorator ensuring the wrapped method may only run if we are the default repository index.

This is as we rely on git commands that operate on that index only.

git.index.util.**git_working_dir**(*func: Callable[[...], git.types._T]*) → Callable[[...], git.types._T]

Decorator which changes the current working dir to the one of the git repository in order to ensure relative paths are handled correctly.

git.index.util.**post_clear_cache**(*func: Callable[[...], git.types._T]*) → Callable[[...], git.types._T]

Decorator for functions that alter the index using the git command.

When a git command alters the index, this invalidates our possibly existing entries dictionary, which is why it must be deleted to allow it to be lazily reread later.

4.16 GitCmd

class git.cmd.**Git**(*working_dir: Union[None, str, os.PathLike[str]] = None*)

The Git class manages communication with the Git binary.

It provides a convenient interface to calling the Git binary, such as in:

```
g = Git( git_dir )
g.init()                  # calls 'git init' program
rval = g.ls_files()       # calls 'git ls-files' program
```

Debugging:

- Set the `GIT_PYTHON_TRACE` environment variable to print each invocation of the command to stdout.
- Set its value to `full` to see details about the returned values.

class AutoInterrupt(*proc: Union[None, subprocess.Popen]*, *args: Any*)

Process wrapper that terminates the wrapped process on finalization.

This kills/interrupts the stored process instance once this instance goes out of scope. It is used to prevent processes piling up in case iterators stop reading.

All attributes are wired through to the contained process object.

The wait method is overridden to perform automatic status code checking and possibly raise.

__annotations__ = {'_status_code_if_terminate': 'int'}

__del__() → None

__getattr__(*attr: str*) → Any

__init__(*proc: Union[None, subprocess.Popen]*, *args: Any*) → None

__module__ = 'git.cmd'

__slots__ = ('proc', 'args', 'status')

args

proc

status: Optional[int]

wait(*stderr: Union[None, str, bytes] = b''*) → int

Wait for the process and return its status code.

Parameters **stderr** – Previously read value of stderr, in case stderr is already closed.

Warn May deadlock if output or error pipes are used and not handled separately.

Raises *git.exc.GitCommandError* – If the return status is not 0.

class CatFileContentStream(*size: int*, *stream: IO[bytes]*)

Object representing a sized read-only stream returning the contents of an object.

This behaves like a stream, but counts the data read and simulates an empty stream once our sized content region is empty.

If not all data are read to the end of the object's lifetime, we read the rest to ensure the underlying stream continues to work.

__del__() → None

__init__(*size: int*, *stream: IO[bytes]*) → None

__iter__() → *git.cmd.Git.CatFileContentStream*

__module__ = 'git.cmd'

__next__() → bytes

__slots__ = ('_stream', '_nbr', '_size')

next() → bytes

read(*size: int = - 1*) → bytes

readline(*size: int = - 1*) → bytes

readlines(*size: int = - 1*) → List[bytes]

GIT_PYTHON_GIT_EXECUTABLE = 'git'

Provide the full path to the git executable. Otherwise it assumes git is in the executable search path.

> **Note** The git executable is actually found during the refresh step in the top level `__init__`. It can also be changed by explicitly calling *git.refresh()*.

GIT_PYTHON_TRACE = False

Enables debugging of GitPython's git commands.

USE_SHELL: bool = False

Deprecated. If set to `True`, a shell will be used when executing git commands.

Code that uses `USE_SHELL = True` or that passes `shell=True` to any GitPython functions should be updated to use the default value of `False` instead. `True` is unsafe unless the effect of syntax treated specially by the shell is fully considered and accounted for, which is not possible under most circumstances. As detailed below, it is also no longer needed, even where it had been in the past.

It is in many if not most cases a command injection vulnerability for an application to set *USE_SHELL* to `True`. Any attacker who can cause a specially crafted fragment of text to make its way into any part of any argument to any git command (including paths, branch names, etc.) can cause the shell to read and write arbitrary files and execute arbitrary commands. Innocent input may also accidentally contain special shell syntax, leading to inadvertent malfunctions.

In addition, how a value of `True` interacts with some aspects of GitPython's operation is not precisely specified and may change without warning, even before GitPython 4.0.0 when *USE_SHELL* may be removed. This includes:

- Whether or how GitPython automatically customizes the shell environment.
- Whether, outside of Windows (where `subprocess.Popen` supports lists of separate arguments even when `shell=True`), this can be used with any GitPython functionality other than direct calls to the *execute()* method.
- Whether any GitPython feature that runs git commands ever attempts to partially sanitize data a shell may treat specially. Currently this is not done.

Prior to GitPython 2.0.8, this had a narrow purpose in suppressing console windows in graphical Windows applications. In 2.0.8 and higher, it provides no benefit, as GitPython solves that problem more robustly and safely by using the `CREATE_NO_WINDOW` process creation flag on Windows.

Because Windows path search differs subtly based on whether a shell is used, in rare cases changing this from `True` to `False` may keep an unusual git "executable", such as a batch file, from being found. To fix this, set the command name or full path in the `GIT_PYTHON_GIT_EXECUTABLE` environment variable or pass the full path to *git.refresh()* (or invoke the script using a `.exe` shim).

Further reading:

- *Git.execute()* (on the `shell` parameter).

- https://github.com/gitpython-developers/GitPython/commit/0d9390866f9ce42870d3116094cd49e0019a970a
- https://learn.microsoft.com/en-us/windows/win32/procthread/process-creation-flags
- https://github.com/python/cpython/issues/91558#issuecomment-1100942950
- https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessw

__annotations__ = {'USE_SHELL': 'bool', '_environment': 'Dict[str, str]', '_git_options': 'Union[List[str], Tuple[str, ...]]', '_persistent_git_options': 'List[str]', '_version_info': 'Union[Tuple[int, ...], None]', '_version_info_token': 'object', 'cat_file_all': 'Union[None, TBD]', 'cat_file_header': 'Union[None, TBD]'}

__call__(***kwargs: Any*) → *git.cmd.Git*

Specify command line options to the git executable for a subcommand call.

Parameters **kwargs** – A dict of keyword arguments. These arguments are passed as in `_call_process()`, but will be passed to the git command rather than the subcommand.

Examples:

```
git(work_tree='/tmp').difftool()
```

__getattr__(*name: str*) → Any

A convenience method as it allows to call the command as if it was an object.

Returns Callable object that will execute call `_call_process()` with your arguments.

__getattribute__(*name: str*) → Any

Return getattr(self, name).

__getstate__() → Dict[str, Any]

Helper for pickle.

__init__(*working_dir: Union[None, str, os.PathLike[str]] = None*) → None

Initialize this instance with:

Parameters **working_dir** – Git directory we should work in. If `None`, we always work in the current directory as returned by `os.getcwd()`. This is meant to be the working tree directory if available, or the `.git` directory in case of bare repositories.

__module__ = 'git.cmd'

__setstate__(*d: Dict[str, Any]*) → None

__slots__ = ('_working_dir', 'cat_file_all', 'cat_file_header', '_version_info', '_version_info_token', '_git_options', '_persistent_git_options', '_environment')

cat_file_all: Union[None, Any]

cat_file_header: Union[None, Any]

classmethod check_unsafe_options(*options: List[str]*, *unsafe_options: List[str]*) → None

Check for unsafe options.

Some options that are passed to `git <command>` can be used to execute arbitrary commands. These are blocked by default.

classmethod check_unsafe_protocols(*url: str*) → None

Check for unsafe protocols.

Apart from the usual protocols (http, git, ssh), Git allows "remote helpers" that have the form `<transport>::<address>`. One of these helpers (`ext::`) can be used to invoke any arbitrary command.

See:

- https://git-scm.com/docs/gitremote-helpers
- https://git-scm.com/docs/git-remote-ext

clear_cache() → *git.cmd.Git*
Clear all kinds of internal caches to release resources.

Currently persistent commands will be interrupted.

Returns self

custom_environment(**kwargs: Any*) → Iterator[None]
A context manager around the above *update_environment()* method to restore the environment back to its previous state after operation.

Examples:

```
with self.custom_environment(GIT_SSH='/bin/ssh_wrapper'):
    repo.remotes.origin.fetch()
```

Parameters **kwargs** – See *update_environment()*.

environment() → Dict[str, str]

execute(*command: Union[str, Sequence[Any]]*, *, *as_process: Literal[True]*) → *AutoInterrupt*
execute(*command: Union[str, Sequence[Any]]*, *, *as_process: Literal[False] = 'False'*, *stdout_as_string: Literal[True]*) → Union[str, Tuple[int, str, str]]
execute(*command: Union[str, Sequence[Any]]*, *, *as_process: Literal[False] = 'False'*, *stdout_as_string: Literal[False] = 'False'*) → Union[bytes, Tuple[int, bytes, str]]
execute(*command: Union[str, Sequence[Any]]*, *, *with_extended_output: Literal[False]*, *as_process: Literal[False]*, *stdout_as_string: Literal[True]*) → str
execute(*command: Union[str, Sequence[Any]]*, *, *with_extended_output: Literal[False]*, *as_process: Literal[False]*, *stdout_as_string: Literal[False]*) → bytes
Handle executing the command, and consume and return the returned information (stdout).

Parameters

- **command** – The command argument list to execute. It should be a sequence of program arguments, or a string. The program to execute is the first item in the args sequence or string.
- **istream** – Standard input filehandle passed to `subprocess.Popen`.
- **with_extended_output** – Whether to return a (status, stdout, stderr) tuple.
- **with_exceptions** – Whether to raise an exception when git returns a non-zero status.
- **as_process** – Whether to return the created process instance directly from which streams can be read on demand. This will render *with_extended_output* and *with_exceptions* ineffective - the caller will have to deal with the details. It is important to note that the process will be placed into an *AutoInterrupt* wrapper that will interrupt the process once it goes out of scope. If you use the command in iterators, you should pass the whole process instance instead of a single stream.

- **output_stream** – If set to a file-like object, data produced by the git command will be copied to the given stream instead of being returned as a string. This feature only has any effect if *as_process* is `False`.
- **stdout_as_string** – If `False`, the command's standard output will be bytes. Otherwise, it will be decoded into a string using the default encoding (usually UTF-8). The latter can fail, if the output contains binary data.
- **kill_after_timeout** – Specifies a timeout in seconds for the git command, after which the process should be killed. This will have no effect if *as_process* is set to `True`. It is set to `None` by default and will let the process run until the timeout is explicitly specified. Uses of this feature should be carefully considered, due to the following limitations:
  1. This feature is not supported at all on Windows.
  2. Effectiveness may vary by operating system. `ps --ppid` is used to enumerate child processes, which is available on most GNU/Linux systems but not most others.
  3. Deeper descendants do not receive signals, though they may sometimes terminate as a consequence of their parent processes being killed.
  4. *kill_after_timeout* uses `SIGKILL`, which can have negative side effects on a repository. For example, stale locks in case of *git-gc(1)* could render the repository incapable of accepting changes until the lock is manually removed.
- **with_stdout** – If `True`, default `True`, we open stdout on the created process.
- **universal_newlines** – If `True`, pipes will be opened as text, and lines are split at all known line endings.
- **shell** – Whether to invoke commands through a shell (see `Popen(..., shell=True)`). If this is not `None`, it overrides *USE_SHELL*.

  Passing `shell=True` to this or any other GitPython function should be avoided, as it is unsafe under most circumstances. This is because it is typically not feasible to fully consider and account for the effect of shell expansions, especially when passing `shell=True` to other methods that forward it to *Git.execute()*. Passing `shell=True` is also no longer needed (nor useful) to work around any known operating system specific issues.
- **env** – A dictionary of environment variables to be passed to `subprocess.Popen`.
- **max_chunk_size** – Maximum number of bytes in one chunk of data passed to the *output_stream* in one invocation of its `write()` method. If the given number is not positive then the default value is used.
- **strip_newline_in_stdout** – Whether to strip the trailing `\n` of the command stdout.
- **subprocess_kwargs** – Keyword arguments to be passed to `subprocess.Popen`. Please note that some of the valid kwargs are already set by this method; the ones you specify may not be the same ones.

Returns

- str(output), if *extended_output* is `False` (Default)
- tuple(int(status), str(stdout), str(stderr)), if *extended_output* is `True`

If *output_stream* is `True`, the stdout value will be your output stream:

- output_stream, if *extended_output* is `False`

- tuple(int(status), output_stream, str(stderr)), if *extended_output* is `True`

Note that git is executed with `LC_MESSAGES="C"` to ensure consistent output regardless of system language.

Raises *git.exc.GitCommandError* –

Note If you add additional keyword arguments to the signature of this method, you must update the `execute_kwargs` variable housed in this module.

get_object_data(*ref: str*) → Tuple[str, str, int, bytes]
Similar to *get_object_header()*, but returns object data as well.

Returns (hexsha, type_string, size_as_int, data_string)

Note Not threadsafe.

get_object_header(*ref: str*) → Tuple[str, str, int]
Use this method to quickly examine the type and size of the object behind the given ref.

Note The method will only suffer from the costs of command invocation once and reuses the command in subsequent calls.

Returns (hexsha, type_string, size_as_int)

git_exec_name = 'git'
Default git command that should work on Linux, Windows, and other systems.

classmethod is_cygwin() → bool

classmethod polish_url(*url: str*, *is_cygwin: Literal[False] = None*) → str
classmethod polish_url(*url: str*, *is_cygwin: Union[None, bool] = None*) → str
Remove any backslashes from URLs to be written in config files.

Windows might create config files containing paths with backslashes, but git stops liking them as it will escape the backslashes. Hence we undo the escaping just to be sure.

re_unsafe_protocol = re.compile('(.+)::.+')

classmethod refresh(*path: Union[None, str, os.PathLike[str]] = None*) → bool
Update information about the git executable *Git* objects will use.

Called by the *git.refresh()* function in the top level `__init__`.

Parameters **path** – Optional path to the git executable. If not absolute, it is resolved immediately, relative to the current directory. (See note below.)

Note The top-level *git.refresh()* should be preferred because it calls this method and may also update other state accordingly.

Note There are three different ways to specify the command that refreshing causes to be used for git:

1. Pass no *path* argument and do not set the `GIT_PYTHON_GIT_EXECUTABLE` environment variable. The command name `git` is used. It is looked up in a path search by the system, in each command run (roughly similar to how git is found when running `git` commands manually). This is usually the desired behavior.

2. Pass no *path* argument but set the `GIT_PYTHON_GIT_EXECUTABLE` environment variable. The command given as the value of that variable is used. This may be a simple command or an arbitrary path. It is looked up in each command run. Setting `GIT_PYTHON_GIT_EXECUTABLE` to `git` has the same effect as not setting it.

3. Pass a *path* argument. This path, if not absolute, is immediately resolved, relative to the current directory. This resolution occurs at the time of the refresh. When git

commands are run, they are run using that previously resolved path. If a *path* argument is passed, the `GIT_PYTHON_GIT_EXECUTABLE` environment variable is not consulted.

Note Refreshing always sets the *Git.GIT_PYTHON_GIT_EXECUTABLE* class attribute, which can be read on the *Git* class or any of its instances to check what command is used to run git. This attribute should not be confused with the related `GIT_PYTHON_GIT_EXECUTABLE` environment variable. The class attribute is set no matter how refreshing is performed.

set_persistent_git_options(***kwargs: Any*) → None
Specify command line options to the git executable for subsequent subcommand calls.

Parameters **kwargs** – A dict of keyword arguments. These arguments are passed as in `_call_process()`, but will be passed to the git command rather than the subcommand.

stream_object_data(*ref: str*) → Tuple[str, str, int, *git.cmd.Git.CatFileContentStream*]
Similar to *get_object_data()*, but returns the data as a stream.

Returns (hexsha, type_string, size_as_int, stream)

Note This method is not threadsafe. You need one independent *Git* instance per thread to be safe!

transform_kwarg(*name: str*, *value: Any*, *split_single_char_options: bool*) → List[str]

transform_kwargs(*split_single_char_options: bool = True*, ***kwargs: Any*) → List[str]
Transform Python-style kwargs into git command line options.

update_environment(***kwargs: Any*) → Dict[str, Optional[str]]
Set environment variables for future git invocations. Return all changed values in a format that can be passed back into this function to revert the changes.

Examples:

```
old_env = self.update_environment(PWD='/tmp')
self.update_environment(**old_env)
```

Parameters **kwargs** – Environment variables to use for git processes.

Returns Dict that maps environment variables to their old values

property version_info: Tuple[int, ...]

Returns

Tuple with integers representing the major, minor and additional version numbers as parsed from *git-version(1)*. Up to four fields are used.

This value is generated on demand and is cached.

property working_dir: Union[None, str, os.PathLike[str]]

Returns Git directory we are working on

git.cmd.**GitMeta**
Alias of *Git*'s metaclass, whether it is `type` or a custom metaclass.

Whether the *Git* class has the default `type` as its metaclass or uses a custom metaclass is not documented and may change at any time. This statically checkable metaclass alias is equivalent at runtime to `type(Git)`. This should almost never be used. Code that benefits from it is likely to be remain brittle even if it is used.

In view of the *Git* class's intended use and *Git* objects' dynamic callable attributes representing git subcommands, it rarely makes sense to inherit from *Git* at all. Using *Git* in multiple inheritance can be especially tricky to do correctly. Attempting uses of *Git* where its metaclass is relevant, such as when a sibling class has

an unrelated metaclass and a shared lower bound metaclass might have to be introduced to solve a metaclass conflict, is not recommended.

Note The correct static type of the *Git* class itself, and any subclasses, is `Type[Git]`. (This can be written as `type[Git]` in Python 3.9 later.)

GitMeta should never be used in any annotation where `Type[Git]` is intended or otherwise possible to use. This alias is truly only for very rare and inherently precarious situations where it is necessary to deal with the metaclass explicitly.

4.17 Config

Parser for reading and writing configuration files.

`git.config.`**GitConfigParser**
alias of *git.config.GitConfigParser*

class `git.config.`**SectionConstraint**(*config: git.config.T_ConfigParser*, *section: str*)
Constrains a ConfigParser to only option commands which are constrained to always use the section we have been initialized with.

It supports all ConfigParser methods that operate on an option.

Note If used as a context manager, will release the wrapped ConfigParser.

__annotations__ = {}

__del__() → None

__enter__() → *git.config.SectionConstraint*[git.config.T_ConfigParser]

__exit__(*exception_type: str*, *exception_value: str*, *traceback: str*) → None

__getattr__(*attr: str*) → Any

__init__(*config: git.config.T_ConfigParser*, *section: str*) → None

__module__ = 'git.config'

__orig_bases__ = (typing.Generic[~T_ConfigParser],)

__parameters__ = (~T_ConfigParser,)

__slots__ = ('_config', '_section_name')

property config: git.config.T_ConfigParser
return: ConfigParser instance we constrain

release() → None
Equivalent to `GitConfigParser.release()`, which is called on our underlying parser instance.

4.18 Diff

class git.diff.**Diff**(*repo: Repo*, *a_rawpath: Optional[bytes]*, *b_rawpath: Optional[bytes]*, *a_blob_id: Optional[Union[str, bytes]]*, *b_blob_id: Optional[Union[str, bytes]]*, *a_mode: Optional[Union[bytes, str]]*, *b_mode: Optional[Union[bytes, str]]*, *new_file: bool*, *deleted_file: bool*, *copied_file: bool*, *raw_rename_from: Optional[bytes]*, *raw_rename_to: Optional[bytes]*, *diff: Optional[Union[str, bytes]]*, *change_type: Optional[Literal['A', 'D', 'C', 'M', 'R', 'T', 'U']]*, *score: Optional[int]*)

A Diff contains diff information between two Trees.

It contains two sides a and b of the diff. Members are prefixed with "a" and "b" respectively to indicate that.

Diffs keep information about the changed blob objects, the file mode, renames, deletions and new files.

There are a few cases where `None` has to be expected as member variable value:

New File:

```
a_mode is None
a_blob is None
a_path is None
```

Deleted File:

```
b_mode is None
b_blob is None
b_path is None
```

Working Tree Blobs:

> When comparing to working trees, the working tree blob will have a null hexsha as a corresponding object does not yet exist. The mode will be null as well. The path will be available, though.
>
> If it is listed in a diff, the working tree version of the file must differ from the version in the index or tree, and hence has been modified.

NULL_BIN_SHA = **b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'**

NULL_HEX_SHA = **'0000000000000000000000000000000000000000'**

__eq__(*other: object*) → bool
Return self==value.

__hash__() → int
Return hash(self).

__init__(*repo: Repo*, *a_rawpath: Optional[bytes]*, *b_rawpath: Optional[bytes]*, *a_blob_id: Optional[Union[str, bytes]]*, *b_blob_id: Optional[Union[str, bytes]]*, *a_mode: Optional[Union[bytes, str]]*, *b_mode: Optional[Union[bytes, str]]*, *new_file: bool*, *deleted_file: bool*, *copied_file: bool*, *raw_rename_from: Optional[bytes]*, *raw_rename_to: Optional[bytes]*, *diff: Optional[Union[str, bytes]]*, *change_type: Optional[Literal['A', 'D', 'C', 'M', 'R', 'T', 'U']]*, *score: Optional[int]*) → None

__module__ = **'git.diff'**

__ne__(*other: object*) → bool
Return self!=value.

__slots__ = ('a_blob', 'b_blob', 'a_mode', 'b_mode', 'a_rawpath', 'b_rawpath', 'new_file', 'deleted_file', 'copied_file', 'raw_rename_from', 'raw_rename_to', 'diff', 'change_type', 'score')

__str__() → str
Return str(self).

a_blob: Union['IndexObject', None]

a_mode

property a_path: Optional[str]

a_rawpath

b_blob: Union['IndexObject', None]

b_mode

property b_path: Optional[str]

b_rawpath

change_type: Union[Lit_change_type, None]

copied_file: bool

deleted_file: bool

diff

new_file: bool

raw_rename_from

raw_rename_to

re_header = re.compile(b'\n ^diff[ ]--git\n [ ](?P<a_path_fallback>"?[ab]/.+?"?)[ ](?P<b_path_fallback>"?[ab]/.+?"?)\\n\n (?:^, re.MULTILINE|re.VERBOSE)

property rename_from: Optional[str]

property rename_to: Optional[str]

property renamed: bool
Deprecated, use *renamed_file* instead.

Returns True if the blob of our diff has been renamed

Note This property is deprecated. Please use the *renamed_file* property instead.

property renamed_file: bool

Returns True if the blob of our diff has been renamed

score

class git.diff.**DiffConstants**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)
Special objects for *Diffable.diff()*.

See the *Diffable.diff()* method's other parameter, which accepts various values including these.

Note These constants are also available as attributes of the *git.diff* module, the *Diffable* class and its subclasses and instances, and the top-level *git* module.

INDEX = 2

Stand-in indicating you want to diff against the index.

Also accessible as `git.INDEX`, *git.diff.INDEX*, and *Diffable.INDEX*, as well as *Diffable.Index*. The latter has been kept for backward compatibility and made an alias of this, so it may still be used.

NULL_TREE = 1

Stand-in indicating you want to compare against the empty tree in diffs.

Also accessible as `git.NULL_TREE`, *git.diff.NULL_TREE*, and *Diffable.NULL_TREE*.

__module__ = 'git.diff'

class git.diff.**DiffIndex**(*iterable=()*, */*)

An index for diffs, allowing a list of *Diff*s to be queried by the diff properties.

The class improves the diff handling convenience.

__annotations__ = {}

__dict__ = mappingproxy({'__module__': 'git.diff', '__doc__': 'An index for diffs, allowing a list of :class:`Diff`\\s to be queried by the diff\n properties.\n\n The class improves the diff handling convenience.\n ', 'change_type': ('A', 'C', 'D', 'R', 'M', 'T'), 'iter_change_type': <function DiffIndex.iter_change_type>, '__orig_bases__': (typing.List[~T_Diff],), '__dict__': <attribute '__dict__' of 'DiffIndex' objects>, '__weakref__': <attribute '__weakref__' of 'DiffIndex' objects>, '__parameters__': (~T_Diff,), '__annotations__': {}})

__module__ = 'git.diff'

__orig_bases__ = (typing.List[~T_Diff],)

__parameters__ = (~T_Diff,)

__weakref__

list of weak references to the object (if defined)

change_type = ('A', 'C', 'D', 'R', 'M', 'T')

Change type invariant identifying possible ways a blob can have changed:

- `A` = Added
- `D` = Deleted
- `R` = Renamed
- `M` = Modified
- `T` = Changed in the type

iter_change_type(*change_type: Literal['A', 'D', 'C', 'M', 'R', 'T', 'U']*) → Iterator[git.diff.T_Diff]

Returns Iterator yielding *Diff* instances that match the given *change_type*

Parameters **change_type** – Member of *DiffIndex.change_type*, namely:

- 'A' for added paths
- 'D' for deleted paths
- 'R' for renamed paths
- 'M' for paths with modified data
- 'T' for changed in the type paths

class git.diff.**Diffable**

Common interface for all objects that can be diffed against another object of compatible type.

Note Subclasses require a *repo* member, as it is the case for *Object* instances. For practical reasons we do not derive from *Object*.

INDEX = 2

Stand-in indicating you want to diff against the index.

See the *diff()* method, which accepts this as a value of its `other` parameter.

This is the same as *DiffConstants.INDEX*, and may also be accessed as `git.INDEX` and *git.diff.INDEX*, as well as *Diffable.INDEX*, which is kept for backward compatibility (it is now defined an alias of this).

Index = 2

Stand-in indicating you want to diff against the index (same as *INDEX*).

This is an alias of *INDEX*, for backward compatibility. See *INDEX* and *diff()* for details.

Note Although always meant for use as an opaque constant, this was formerly defined as a class. Its usage is unchanged, but static type annotations that attempt to permit only this object must be changed to avoid new mypy errors. This was previously not possible to do, though `Type[Diffable.Index]` approximated it. It is now possible to do precisely, using `Literal[DiffConstants.INDEX]`.

NULL_TREE = 1

Stand-in indicating you want to compare against the empty tree in diffs.

See the *diff()* method, which accepts this as a value of its `other` parameter.

This is the same as *DiffConstants.NULL_TREE*, and may also be accessed as `git.NULL_TREE` and *git.diff.NULL_TREE*.

__annotations__ = {'repo': 'Repo'}

__module__ = 'git.diff'

__slots__ = ()

diff(*other: Optional[Union[*git.diff.DiffConstants*, Tree, Commit, str]] = DiffConstants.INDEX*, *paths: Optional[Union[str, os.PathLike[str], List[Union[str, os.PathLike[str]]], Tuple[Union[str, os.PathLike[str]], ...]]] = None*, *create_patch: bool = False*, ***kwargs: Any*) → *DiffIndex*

Create diffs between two items being trees, trees and index or an index and the working tree. Detects renames automatically.

Parameters

- **other** – This the item to compare us with.
  - If `None`, we will be compared to the working tree.
  - If a *Tree_ish* or string, it will be compared against the respective tree.
  - If *INDEX*, it will be compared against the index.
  - If *NULL_TREE*, it will compare against the empty tree.

  This parameter defaults to *INDEX* (rather than `None`) so that the method will not by default fail on bare repositories.
- **paths** – This a list of paths or a single path to limit the diff to. It will only include at least one of the given path or paths.

- **create_patch** – If `True`, the returned *Diff* contains a detailed patch that if applied makes the self to other. Patches are somewhat costly as blobs have to be read and diffed.
- **kwargs** – Additional arguments passed to *git-diff(1)*, such as `R=True` to swap both sides of the diff.

Returns A *DiffIndex* representing the computed diff.

Note On a bare repository, *other* needs to be provided as *INDEX*, or as an instance of *Tree* or *Commit*, or a git command error will occur.

repo: Repo
Repository to operate on. Must be provided by subclass or sibling class.

git.diff.**INDEX: Literal[<DiffConstants.INDEX: 2>] = DiffConstants.INDEX**
Stand-in indicating you want to diff against the index.

See *Diffable.diff()*, which accepts this as a value of its `other` parameter.

This is an alias of *DiffConstants.INDEX*, which may also be accessed as `git.INDEX` and *Diffable.INDEX*, as well as *Diffable.Index*.

git.diff.**NULL_TREE: Literal[<DiffConstants.NULL_TREE: 1>] = DiffConstants.NULL_TREE**
Stand-in indicating you want to compare against the empty tree in diffs.

See *Diffable.diff()*, which accepts this as a value of its `other` parameter.

This is an alias of *DiffConstants.NULL_TREE*, which may also be accessed as `git.NULL_TREE` and *Diffable.NULL_TREE*.

4.19 Exceptions

Exceptions thrown throughout the git package.

exception git.exc.**AmbiguousObjectName**
Thrown if a possibly shortened name does not uniquely represent a single object in the database

__module__ = 'gitdb.exc'

exception git.exc.**BadName**
A name provided to rev_parse wasn't understood

__annotations__ = {}

__module__ = 'gitdb.exc'

__str__()
Return str(self).

exception git.exc.**BadObject**
The object with the given SHA does not exist. Instantiate with the failed sha

__annotations__ = {}

__module__ = 'gitdb.exc'

__str__()
Return str(self).

exception git.exc.**BadObjectType**
The object had an unsupported type

__annotations__ = {}

__module__ = 'gitdb.exc'

exception git.exc.CacheError
Base for all errors related to the git index, which is called "cache" internally.

__annotations__ = {}

__module__ = 'git.exc'

exception git.exc.CheckoutError(*message: str*, *failed_files: Sequence[Union[str, os.PathLike[str]]]*, *valid_files: Sequence[Union[str, os.PathLike[str]]]*, *failed_reasons: List[str]*)
Thrown if a file could not be checked out from the index as it contained changes.

The failed_files attribute contains a list of relative paths that failed to be checked out as they contained changes that did not exist in the index.

The failed_reasons attribute contains a string informing about the actual cause of the issue.

The valid_files attribute contains a list of relative paths to files that were checked out successfully and hence match the version stored in the index.

__annotations__ = {}

__init__(*message: str*, *failed_files: Sequence[Union[str, os.PathLike[str]]]*, *valid_files: Sequence[Union[str, os.PathLike[str]]]*, *failed_reasons: List[str]*) → None

__module__ = 'git.exc'

__str__() → str
Return str(self).

exception git.exc.CommandError(*command: Union[List[str], Tuple[str, ...], str]*, *status: Union[str, int, None, Exception] = None*, *stderr: Optional[Union[bytes, str]] = None*, *stdout: Optional[Union[bytes, str]] = None*)
Base class for exceptions thrown at every stage of Popen execution.

Parameters **command** – A non-empty list of argv comprising the command-line.

__annotations__ = {}

__init__(*command: Union[List[str], Tuple[str, ...], str]*, *status: Union[str, int, None, Exception] = None*, *stderr: Optional[Union[bytes, str]] = None*, *stdout: Optional[Union[bytes, str]] = None*) → None

__module__ = 'git.exc'

__str__() → str
Return str(self).

exception git.exc.GitCommandError(*command: Union[List[str], Tuple[str, ...], str]*, *status: Union[str, int, None, Exception] = None*, *stderr: Optional[Union[bytes, str]] = None*, *stdout: Optional[Union[bytes, str]] = None*)
Thrown if execution of the git command fails with non-zero status code.

__annotations__ = {}

__init__(*command: Union[List[str], Tuple[str, ...], str]*, *status: Union[str, int, None, Exception] = None*, *stderr: Optional[Union[bytes, str]] = None*, *stdout: Optional[Union[bytes, str]] = None*) → None

__module__ = 'git.exc'

exception git.exc.**GitCommandNotFound**(*command: Union[List[str], Tuple[str], str]*, *cause: Union[str, Exception]*)

Thrown if we cannot find the `git` executable in the `PATH` or at the path given by the `GIT_PYTHON_GIT_EXECUTABLE` environment variable.

__annotations__ = {}

__init__(*command: Union[List[str], Tuple[str], str]*, *cause: Union[str, Exception]*) → None

__module__ = 'git.exc'

exception git.exc.**GitError**

Base class for all package exceptions.

__annotations__ = {}

__module__ = 'git.exc'

__weakref__

list of weak references to the object (if defined)

exception git.exc.**HookExecutionError**(*command: Union[List[str], Tuple[str, ...], str]*, *status: Union[str, int, None, Exception]*, *stderr: Optional[Union[bytes, str]] = None*, *stdout: Optional[Union[bytes, str]] = None*)

Thrown if a hook exits with a non-zero exit code.

This provides access to the exit code and the string returned via standard output.

__annotations__ = {}

__init__(*command: Union[List[str], Tuple[str, ...], str]*, *status: Union[str, int, None, Exception]*, *stderr: Optional[Union[bytes, str]] = None*, *stdout: Optional[Union[bytes, str]] = None*) → None

__module__ = 'git.exc'

exception git.exc.**InvalidDBRoot**

Thrown if an object database cannot be initialized at the given path

__annotations__ = {}

__module__ = 'gitdb.exc'

exception git.exc.**InvalidGitRepositoryError**

Thrown if the given repository appears to have an invalid format.

__annotations__ = {}

__module__ = 'git.exc'

exception git.exc.**NoSuchPathError**

Thrown if a path could not be access by the system.

__annotations__ = {}

__module__ = 'git.exc'

__weakref__

list of weak references to the object (if defined)

exception git.exc.**ODBError**

All errors thrown by the object database

__annotations__ = {}

__module__ = 'gitdb.exc'

__weakref__
list of weak references to the object (if defined)

exception git.exc.ParseError
Thrown if the parsing of a file failed due to an invalid format

__annotations__ = {}

__module__ = 'gitdb.exc'

exception git.exc.RepositoryDirtyError(*repo: Repo*, *message: str*)
Thrown whenever an operation on a repository fails as it has uncommitted changes that would be overwritten.

__annotations__ = {}

__init__(*repo: Repo*, *message: str*) → None

__module__ = 'git.exc'

__str__() → str
Return str(self).

exception git.exc.UnmergedEntriesError
Thrown if an operation cannot proceed as there are still unmerged entries in the cache.

__annotations__ = {}

__module__ = 'git.exc'

exception git.exc.UnsafeOptionError
Thrown if unsafe options are passed without being explicitly allowed.

__annotations__ = {}

__module__ = 'git.exc'

exception git.exc.UnsafeProtocolError
Thrown if unsafe protocols are passed without being explicitly allowed.

__annotations__ = {}

__module__ = 'git.exc'

exception git.exc.UnsupportedOperation
Thrown if the given operation cannot be supported by the object database

__annotations__ = {}

__module__ = 'gitdb.exc'

exception git.exc.WorkTreeRepositoryUnsupported
Thrown to indicate we can't handle work tree repositories.

__annotations__ = {}

__module__ = 'git.exc'

4.20 Refs.symbolic

class git.refs.symbolic.**SymbolicReference**(*repo: Repo*, *path: Union[str, os.PathLike[str]]*, *check_path: bool = False*)

Special case of a reference that is symbolic.

This does not point to a specific commit, but to another *Head*, which itself specifies a commit.

A typical example for a symbolic reference is *HEAD*.

__annotations__ = {'reference': typing.Union[ForwardRef('Head'), ForwardRef('TagReference'), ForwardRef('RemoteReference'), ForwardRef('Reference')]}

__eq__(*other:* object) → bool
Return self==value.

__hash__() → int
Return hash(self).

__init__(*repo: Repo*, *path: Union[str, os.PathLike[str]]*, *check_path: bool = False*) → None

__module__ = 'git.refs.symbolic'

__ne__(*other:* object) → bool
Return self!=value.

__repr__() → str
Return repr(self).

__slots__ = ('repo', 'path')

__str__() → str
Return str(self).

property abspath: Union[str, os.PathLike[str]]

property commit: *git.objects.commit.Commit*
Query or set commits directly

classmethod create(*repo: Repo*, *path: Union[str, os.PathLike[str]]*, *reference: Union[*SymbolicReference*, str] = 'HEAD'*, *logmsg: Optional[str] = None*, *force: bool = False*, ***kwargs: Any*) → git.refs.symbolic.T_References
Create a new symbolic reference: a reference pointing to another reference.

Parameters

- **repo** – Repository to create the reference in.
- **path** – Full path at which the new symbolic reference is supposed to be created at, e.g. `NEW_HEAD` or `symrefs/my_new_symref`.
- **reference** – The reference which the new symbolic reference should point to. If it is a commit-ish, the symbolic ref will be detached.
- **force** – If `True`, force creation even if a symbolic reference with that name already exists. Raise `OSError` otherwise.
- **logmsg** – If not `None`, the message to append to the reflog. If `None`, no reflog entry is written.

Returns Newly created symbolic reference

Raises `OSError` – If a (Symbolic)Reference with the same name but different contents already exists.

Note This does not alter the current HEAD, index or working tree.

classmethod delete(*repo: Repo*, *path: Union[str, os.PathLike[str]]*) → None

Delete the reference at the given path.

Parameters

- **repo** – Repository to delete the reference from.
- **path** – Short or full path pointing to the reference, e.g. `refs/myreference` or just `myreference`, hence `refs/` is implied. Alternatively the symbolic reference to be deleted.

classmethod dereference_recursive(*repo: Repo*, *ref_path: Optional[Union[str, os.PathLike[str]]]*) → str

Returns hexsha stored in the reference at the given *ref_path*, recursively dereferencing all intermediate references as required

Parameters **repo** – The repository containing the reference at *ref_path*.

classmethod from_path(*repo: Repo*, *path: Union[str, os.PathLike[str]]*) → git.refs.symbolic.T_References

Make a symbolic reference from a path.

Parameters **path** – Full `.git`-directory-relative path name to the Reference to instantiate.

Note Use *to_full_path()* if you only have a partial path of a known Reference type.

Returns Instance of type *Reference*, *Head*, or *Tag*, depending on the given path.

property is_detached: bool

Returns `True` if we are a detached reference, hence we point to a specific commit instead to another reference.

is_remote() → bool

Returns True if this symbolic reference points to a remote branch

is_valid() → bool

Returns `True` if the reference is valid, hence it can be read and points to a valid object or reference.

classmethod iter_items(*repo: Repo*, *common_path: Optional[Union[str, os.PathLike[str]]] = None*, **args: Any*, ***kwargs: Any*) → Iterator[git.refs.symbolic.T_References]

Find all refs in the repository.

Parameters

- **repo** – The *Repo*.
- **common_path** – Optional keyword argument to the path which is to be shared by all returned Ref objects. Defaults to class specific portion if `None`, ensuring that only refs suitable for the actual class are returned.

Returns

A list of *SymbolicReference*, each guaranteed to be a symbolic ref which is not detached and pointing to a valid ref.

The list is lexicographically sorted. The returned objects are instances of concrete subclasses, such as *Head* or *TagReference*.

log() → *git.refs.log.RefLog*

Returns *RefLog* for this reference. Its last entry reflects the latest change applied to this reference.

Note As the log is parsed every time, its recommended to cache it for use instead of calling this method repeatedly. It should be considered read-only.

log_append(*oldbinsha: bytes*, *message: Optional[str]*, *newbinsha: Optional[bytes] = None*) → RefLogEntry

Append a logentry to the logfile of this ref.

Parameters

- **oldbinsha** – Binary sha this ref used to point to.
- **message** – A message describing the change.
- **newbinsha** – The sha the ref points to now. If None, our current commit sha will be used.

Returns The added *RefLogEntry* instance.

log_entry(*index: int*) → RefLogEntry

Returns *RefLogEntry* at the given index

Parameters **index** – Python list compatible positive or negative index.

Note This method must read part of the reflog during execution, hence it should be used sparingly, or only if you need just one index. In that case, it will be faster than the *log()* method.

property name: str

Returns In case of symbolic references, the shortest assumable name is the path itself.

property object: Union[Commit, Tree, TagObject, Blob]

Return the object our ref currently refers to

path

property ref: *git.refs.symbolic.SymbolicReference*

Returns the Reference we point to

property reference: *git.refs.symbolic.SymbolicReference*

Returns the Reference we point to

rename(*new_path: Union[str, os.PathLike[str]]*, *force: bool = False*) → *git.refs.symbolic.SymbolicReference*

Rename self to a new path.

Parameters

- **new_path** – Either a simple name or a full path, e.g. `new_name` or `features/new_name`. The prefix `refs/` is implied for references and will be set as needed. In case this is a symbolic ref, there is no implied prefix.
- **force** – If `True`, the rename will succeed even if a head with the target name already exists. It will be overwritten in that case.

Returns self

Raises **OSError** – If a file at path but with different contents already exists.

repo

set_commit(*commit: Union[*git.objects.commit.Commit*,* git.refs.symbolic.SymbolicReference*, str]*, *logmsg: Optional[str] = None*) → *git.refs.symbolic.SymbolicReference*

Like *set_object()*, but restricts the type of object to be a *Commit*.

Raises **ValueError** – If *commit* is not a *Commit* object, nor does it point to a commit.

Returns self

set_object(*object: Union[Commit, Tree, TagObject, Blob,* SymbolicReference*, str]*, *logmsg: Optional[str] = None*) → *SymbolicReference*

Set the object we point to, possibly dereference our symbolic reference first. If the reference does not exist, it will be created.

Parameters

- **object** – A refspec, a *SymbolicReference* or an *Object* instance.
  - *SymbolicReference* instances will be dereferenced beforehand to obtain the git object they point to.
  - *Object* instances must represent git objects (*AnyGitObject*).
- **logmsg** – If not None, the message will be used in the reflog entry to be written. Otherwise the reflog is not altered.

Note Plain *SymbolicReference* instances may not actually point to objects by convention.

Returns self

set_reference(*ref: Union[Commit, Tree, TagObject, Blob,* SymbolicReference*, str]*, *logmsg: Optional[str] = None*) → *SymbolicReference*

Set ourselves to the given *ref*.

It will stay a symbol if the *ref* is a *Reference*.

Otherwise a git object, specified as a *Object* instance or refspec, is assumed. If it is valid, this reference will be set to it, which effectively detaches the reference if it was a purely symbolic one.

Parameters

- **ref** – A *SymbolicReference* instance, an *Object* instance (specifically an *AnyGitObject*), or a refspec string. Only if the ref is a *SymbolicReference* instance, we will point to it. Everything else is dereferenced to obtain the actual object.
- **logmsg** – If set to a string, the message will be used in the reflog. Otherwise, a reflog entry is not written for the changed reference. The previous commit of the entry will be the commit we point to now.

  See also: *log_append()*

Returns self

Note This symbolic reference will not be dereferenced. For that, see *set_object()*.

classmethod to_full_path(*path: Union[str, os.PathLike[str],* git.refs.symbolic.SymbolicReference*]*) → Union[str, os.PathLike[str]]

Returns String with a full repository-relative path which can be used to initialize a *Reference* instance, for instance by using `Reference.from_path`.

4.21 Refs.reference

class git.refs.reference.**Reference**(*repo: Repo*, *path: Union[str, os.PathLike[str]]*, *check_path: bool = True*)

A named reference to any object.

Subclasses may apply restrictions though, e.g., a *Head* can only point to commits.

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': 'str', 'path': 'str', 'reference': "Union['Head', 'TagReference', 'RemoteReference', 'Reference']"}

__init__(*repo: Repo*, *path: Union[str, os.PathLike[str]]*, *check_path: bool = True*) → None

Initialize this instance.

Parameters

- **repo** – Our parent repository.
- **path** – Path relative to the `.git/` directory pointing to the ref in question, e.g. `refs/heads/master`.
- **check_path** – If `False`, you can provide any path. Otherwise the path must start with the default path prefix of this type.

__module__ = 'git.refs.reference'

__parameters__ = ()

__slots__ = ()

__str__() → str

Return str(self).

classmethod __subclasshook__(*other*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

classmethod iter_items(*repo: Repo*, *common_path: Optional[Union[str, os.PathLike[str]]] = None*, **args: Any*, ***kwargs: Any*) → Iterator[git.refs.symbolic.T_References]

Equivalent to *SymbolicReference.iter_items*, but will return non-detached references as well.

property name: str

Returns (shortest) Name of this reference - it may contain path components

property remote_head: git.types._T

property remote_name: git.types._T

set_object(*object: Union[Commit, Tree, TagObject, Blob, SymbolicReference, str]*, *logmsg: Optional[str] = None*) → *Reference*

Special version which checks if the head-log needs an update as well.

Returns self

4.22 Refs.head

Some ref-based objects.

Note the distinction between the *HEAD* and *Head* classes.

class git.refs.head.**HEAD**(*repo: Repo*, *path: Union[str, os.PathLike[str]] = 'HEAD'*)

Special case of a *SymbolicReference* representing the repository's HEAD reference.

__annotations__ = {'commit': 'Commit'}

__init__(*repo: Repo*, *path: Union[str, os.PathLike[str]] = 'HEAD'*) → None

__module__ = 'git.refs.head'

__slots__ = ()

orig_head() → *git.refs.symbolic.SymbolicReference*

Returns *SymbolicReference* pointing at the ORIG_HEAD, which is maintained to contain the previous value of HEAD.

reset(*commit: Union[Commit, TagObject,* git.refs.symbolic.SymbolicReference*, str] = 'HEAD'*, *index: bool = True*, *working_tree: bool = False*, *paths: Optional[Union[str, os.PathLike[str], Sequence[Union[str, os.PathLike[str]]]]] = None*, ***kwargs: Any*) → *HEAD*

Reset our HEAD to the given commit optionally synchronizing the index and working tree. The reference we refer to will be set to commit as well.

Parameters

- **commit** – *Commit*, *Reference*, or string identifying a revision we should reset HEAD to.
- **index** – If `True`, the index will be set to match the given commit. Otherwise it will not be touched.
- **working_tree** – If `True`, the working tree will be forcefully adjusted to match the given commit, possibly overwriting uncommitted changes without warning. If *working_tree* is `True`, *index* must be `True` as well.
- **paths** – Single path or list of paths relative to the git root directory that are to be reset. This allows to partially reset individual files.
- **kwargs** – Additional arguments passed to *git-reset(1)*.

Returns self

class git.refs.head.**Head**(*repo: Repo*, *path: Union[str, os.PathLike[str]]*, *check_path: bool = True*)

A Head is a named reference to a *Commit*. Every Head instance contains a name and a *Commit* object.

Examples:

```
>>> repo = Repo("/path/to/repo")
>>> head = repo.heads[0]

>>> head.name
'master'

>>> head.commit
<git.Commit "1c09f116cbc2cb4100fb6935bb162daa4723f455">
```

(continues on next page)

(continued from previous page)

```
>>> head.commit.hexsha
'1c09f116cbc2cb4100fb6935bb162daa4723f455'
```

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': 'str', 'path': 'str', 'reference': "Union['Head', 'TagReference', 'RemoteReference', 'Reference']"}

__dict__ = mappingproxy({'__module__': 'git.refs.head', '__doc__': 'A Head is a named reference to a :class:`~git.objects.commit.Commit`. Every Head\n instance contains a name and a :class:`~git.objects.commit.Commit` object.\n\n Examples::\n\n >>> repo = Repo("/path/to/repo")\n >>> head = repo.heads[0]\n\n >>> head.name\n \'master\'\n\n >>> head.commit\n <git.Commit "1c09f116cbc2cb4100fb6935bb162daa4723f455">\n\n >>> head.commit.hexsha\n \'1c09f116cbc2cb4100fb6935bb162daa4723f455\'\n ', '_common_path_default': 'refs/heads', 'k_config_remote': 'remote', 'k_config_remote_ref': 'merge', 'delete': <classmethod(<function Head.delete>)>, 'set_tracking_branch': <function Head.set_tracking_branch>, 'tracking_branch': <function Head.tracking_branch>, 'rename': <function Head.rename>, 'checkout': <function Head.checkout>, '_config_parser': <function Head._config_parser>, 'config_reader': <function Head.config_reader>, 'config_writer': <function Head.config_writer>, '__dict__': <attribute '__dict__' of 'Head' objects>, '__weakref__': <attribute '__weakref__' of 'Head' objects>, '__parameters__': (), '_is_protocol': False, '__subclasshook__': <classmethod(<function _proto_hook>)>, '__abstractmethods__': frozenset(), '_abc_impl': <_abc._abc_data object>, '__annotations__': {'path': 'str', 'reference': "Union['Head', 'TagReference', 'RemoteReference', 'Reference']", '_id_attribute_': 'str'}})

__module__ = 'git.refs.head'

__parameters__ = ()

classmethod __subclasshook__(*other*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__

list of weak references to the object (if defined)

checkout(*force: bool = False*, ***kwargs: Any*) → Union[*git.refs.head.HEAD*, *git.refs.head.Head*]

Check out this head by setting the HEAD to this reference, by updating the index to reflect the tree we point to and by updating the working tree to reflect the latest index.

The command will fail if changed working tree files would be overwritten.

Parameters

- **force** – If `True`, changes to the index and the working tree will be discarded. If `False`, *`GitCommandError`* will be raised in that situation.
- **kwargs** – Additional keyword arguments to be passed to git checkout, e.g. b="new_branch" to create a new branch at the given spot.

Returns The active branch after the checkout operation, usually self unless a new branch has been created. If there is no active branch, as the HEAD is now detached, the HEAD reference will be returned instead.

Note By default it is only allowed to checkout heads - everything else will leave the HEAD detached which is allowed and possible, but remains a special state that some tools might not be able to handle.

config_reader() → *git.config.SectionConstraint*[*git.config.GitConfigParser*]

Returns A configuration parser instance constrained to only read this instance's values.

config_writer() → *git.config.SectionConstraint*[*git.config.GitConfigParser*]

Returns A configuration writer instance with read-and write access to options of this head.

classmethod delete(*repo: Repo*, **heads: Union[*Head*, str]*, *force: bool = False*, ***kwargs: Any*) → None
Delete the given heads.

Parameters **force** – If `True`, the heads will be deleted even if they are not yet merged into the main development stream. Default `False`.

k_config_remote = 'remote'

k_config_remote_ref = 'merge'

rename(*new_path: Union[str, os.PathLike[str]]*, *force: bool = False*) → *Head*
Rename self to a new path.

Parameters

- **new_path** – Either a simple name or a path, e.g. `new_name` or `features/new_name`. The prefix `refs/heads` is implied.
- **force** – If `True`, the rename will succeed even if a head with the target name already exists.

Returns self

Note Respects the ref log, as git commands are used.

set_tracking_branch(*remote_reference: Optional[RemoteReference]*) → *Head*
Configure this branch to track the given remote reference. This will alter this branch's configuration accordingly.

Parameters **remote_reference** – The remote reference to track or None to untrack any references.

Returns self

tracking_branch() → Optional[RemoteReference]

Returns The remote reference we are tracking, or `None` if we are not a tracking branch.

4.23 Refs.tag

Provides a *Reference*-based type for lightweight tags.

This defines the *TagReference* class (and its alias *Tag*), which represents lightweight tags. For annotated tags (which are git objects), see the *git.objects.tag* module.

git.refs.tag.**Tag**
alias of *git.refs.tag.TagReference*

class git.refs.tag.**TagReference**(*repo: Repo*, *path: Union[str, os.PathLike[str]]*, *check_path: bool = True*)
A lightweight tag reference which either points to a commit, a tag object or any other object. In the latter case additional information, like the signature or the tag-creator, is available.

This tag object will always point to a commit object, but may carry additional information in a tag object:

```
tagref = TagReference.list_items(repo)[0]
print(tagref.commit.message)
if tagref.tag is not None:
   print(tagref.tag.message)
```

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': 'str', 'path': 'str', 'reference': "Union['Head', 'TagReference', 'RemoteReference', 'Reference']"}

__module__ = 'git.refs.tag'

__parameters__ = ()

__slots__ = ()

classmethod __subclasshook__(*other*)
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

property commit: Commit

Returns Commit object the tag ref points to

Raises **ValueError** – If the tag points to a tree or blob.

classmethod create(*repo: Repo*, *path: Union[str, os.PathLike[str]]*, *reference: Union[str, SymbolicReference] = 'HEAD'*, *logmsg: Optional[str] = None*, *force: bool = False*, ***kwargs: Any*) → *TagReference*
Create a new tag reference.

Parameters

- **repo** – The *Repo* to create the tag in.
- **path** – The name of the tag, e.g. `1.0` or `releases/1.0`. The prefix `refs/tags` is implied.
- **reference** – A reference to the *Object* you want to tag. The referenced object can be a commit, tree, or blob.
- **logmsg** – If not `None`, the message will be used in your tag object. This will also create an additional tag object that allows to obtain that information, e.g.:

```
tagref.tag.message
```

- **message** – Synonym for the *logmsg* parameter. Included for backwards compatibility. *logmsg* takes precedence if both are passed.
- **force** – If `True`, force creation of a tag even though that tag already exists.
- **kwargs** – Additional keyword arguments to be passed to *git-tag(1)*.

Returns A new *TagReference*.

classmethod delete(*repo: Repo*, **tags:* TagReference) → None
Delete the given existing tag or tags.

property object: Union[Commit, Tree, TagObject, Blob]
Return the object our ref currently refers to

property tag: Optional[TagObject]

Returns Tag object this tag ref points to, or `None` in case we are a lightweight tag

4.24 Refs.remote

Module implementing a remote object allowing easy access to git remotes.

class git.refs.remote.**RemoteReference**(*repo: Repo*, *path: Union[str, os.PathLike[str]]*, *check_path: bool = True*)
A reference pointing to a remote head.

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': 'str', 'path': 'str', 'reference': "Union['Head', 'TagReference', 'RemoteReference', 'Reference']"}

__module__ = 'git.refs.remote'

__parameters__ = ()

classmethod __subclasshook__(*other*)
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

classmethod create(**args: Any*, ***kwargs: Any*) → NoReturn
Raise `TypeError`. Defined so the `create` method is disabled.

classmethod delete(*repo: Repo*, **refs:* RemoteReference, ***kwargs: Any*) → None
Delete the given remote references.

Note *kwargs* are given for comparability with the base class method as we should not narrow the signature.

classmethod iter_items(*repo: Repo*, *common_path: Optional[Union[str, os.PathLike[str]]] = None*, *remote: Optional[Remote] = None*, **args: Any*, ***kwargs: Any*) → Iterator[*RemoteReference*]
Iterate remote references, and if given, constrain them to the given remote.

4.25 Refs.log

class git.refs.log.**RefLog**(*filepath: Optional[Union[str, os.PathLike[str]]] = None*)

A reflog contains *RefLogEntry*s, each of which defines a certain state of the head in question. Custom query methods allow to retrieve log entries by date or by other criteria.

Reflog entries are ordered. The first added entry is first in the list. The last entry, i.e. the last change of the head or reference, is last in the list.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__init__(*filepath: Optional[Union[str, os.PathLike[str]]] = None*) → None

Initialize this instance with an optional filepath, from which we will initialize our data. The path is also used to write changes back using the *write()* method.

__module__ = 'git.refs.log'

static __new__(*cls*, *filepath: Optional[Union[str, os.PathLike[str]]] = None*) → *RefLog*

__orig_bases__ = (typing.List[git.refs.log.RefLogEntry], <class 'git.objects.util.Serializable'>)

__parameters__ = ()

__slots__ = ('_path',)

classmethod append_entry(*config_reader: Optional[Union[*git.util.Actor*, GitConfigParser, SectionConstraint]]*, *filepath: Union[str, os.PathLike[str]]*, *oldbinsha: bytes*, *newbinsha: bytes*, *message: str*, *write: bool = True*) → *RefLogEntry*

Append a new log entry to the revlog at filepath.

Parameters

- **config_reader** – Configuration reader of the repository - used to obtain user information. May also be an *Actor* instance identifying the committer directly or `None`.
- **filepath** – Full path to the log file.
- **oldbinsha** – Binary sha of the previous commit.
- **newbinsha** – Binary sha of the current commit.
- **message** – Message describing the change to the reference.
- **write** – If `True`, the changes will be written right away. Otherwise the change will not be written.

Returns *RefLogEntry* objects which was appended to the log.

Note As we are append-only, concurrent access is not a problem as we do not interfere with readers.

classmethod entry_at(*filepath: Union[str, os.PathLike[str]]*, *index: int*) → *RefLogEntry*

Returns *RefLogEntry* at the given index.

Parameters

- **filepath** – Full path to the index file from which to read the entry.
- **index** – Python list compatible index, i.e. it may be negative to specify an entry counted from the end of the list.

Raises **IndexError** – If the entry didn't exist.

Note This method is faster as it only parses the entry at index, skipping all other lines. Nonetheless, the whole file has to be read if the index is negative.

classmethod from_file(*filepath: Union[str, os.PathLike[str]]*) → *RefLog*

Returns A new *RefLog* instance containing all entries from the reflog at the given *filepath*.

Parameters **filepath** – Path to reflog.

Raises **ValueError** – If the file could not be read or was corrupted in some way.

classmethod iter_entries(*stream: Union[str, BytesIO, mmap.mmap]*) → Iterator[*git.refs.log.RefLogEntry*]

Returns Iterator yielding *RefLogEntry* instances, one for each line read from the given stream.

Parameters **stream** – File-like object containing the revlog in its native format or string instance pointing to a file to read.

classmethod path(*ref: SymbolicReference*) → str

Returns String to absolute path at which the reflog of the given ref instance would be found. The path is not guaranteed to point to a valid file though.

Parameters **ref** – *SymbolicReference* instance

to_file(*filepath: Union[str, os.PathLike[str]]*) → None

Write the contents of the reflog instance to a file at the given filepath.

Parameters **filepath** – Path to file. Parent directories are assumed to exist.

write() → *git.refs.log.RefLog*

Write this instance's data to the file we are originating from.

Returns self

class git.refs.log.**RefLogEntry**(*iterable=(), /*)

Named tuple allowing easy access to the revlog data fields.

__annotations__ = {}

__module__ = 'git.refs.log'

__orig_bases__ = (typing.Tuple[str, str, git.util.Actor, typing.Tuple[int, int], str],)

__parameters__ = ()

__repr__() → str

Representation of ourselves in git reflog format.

__slots__ = ()

property actor: *git.util.Actor*

Actor instance, providing access.

format() → str

Returns A string suitable to be placed in a reflog file.

classmethod from_line(*line: bytes*) → *git.refs.log.RefLogEntry*

Returns New *RefLogEntry* instance from the given revlog line.

Parameters **line** – Line bytes without trailing newline

Raises **ValueError** – If *line* could not be parsed.

property message: str
Message describing the operation that acted on the reference.

classmethod new(*oldhexsha: str*, *newhexsha: str*, *actor:* git.util.Actor, *time: int*, *tz_offset: int*, *message: str*) → *git.refs.log.RefLogEntry*

Returns New instance of a *RefLogEntry*

property newhexsha: str
The hexsha to the commit the ref now points to, after the change.

property oldhexsha: str
The hexsha to the commit the ref pointed to before the change.

property time: Tuple[int, int]
Time as tuple:

- [0] = `int(time)`
- [1] = `int(timezone_offset)` in `time.altzone` format

4.26 Remote

Module implementing a remote object allowing easy access to git remotes.

class git.remote.**FetchInfo**(*ref:* git.refs.symbolic.SymbolicReference, *flags: int*, *note: str = ''*, *old_commit: Optional[Union[Commit, Tree, TagObject, Blob]] = None*, *remote_ref_path: Optional[Union[str, os.PathLike[str]]] = None*)
Carries information about the results of a fetch operation of a single head:

```
info = remote.fetch()[0]
info.ref           # Symbolic Reference or RemoteReference to the changed
                   # remote head or FETCH_HEAD
info.flags         # additional flags to be & with enumeration members,
                   # i.e. info.flags & info.REJECTED
                   # is 0 if ref is SymbolicReference
info.note          # additional notes given by git-fetch intended for the user
info.old_commit    # if info.flags & info.FORCED_UPDATE|info.FAST_FORWARD,
                   # field is set to the previous location of ref, otherwise None
info.remote_ref_path # The path from which we fetched on the remote. It's the remote
↪'s version of our info.ref
```

ERROR = 128

FAST_FORWARD = 64

FORCED_UPDATE = 32

HEAD_UPTODATE = 4

NEW_HEAD = 2

NEW_TAG = 1

REJECTED = 16

TAG_UPDATE = 8

__abstractmethods__ = frozenset({})

__annotations__ = {'_flag_map': typing.Dict[typing.Literal[' ', '!', '+', '-', '*', '=', 't', '?'], int], '_id_attribute_': 'str'}

__init__(*ref:* git.refs.symbolic.SymbolicReference, *flags: int*, *note: str = ''*, *old_commit: Optional[Union[Commit, Tree, TagObject, Blob]] = None*, *remote_ref_path: Optional[Union[str, os.PathLike[str]]] = None*) → None
Initialize a new instance.

__module__ = 'git.remote'

__parameters__ = ()

__slots__ = ('ref', 'old_commit', 'flags', 'note', 'remote_ref_path')

__str__() → str
Return str(self).

classmethod __subclasshook__(*other*)
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

property commit: Commit

Returns Commit of our remote ref

flags

classmethod iter_items(*repo: Repo*, **args: Any*, ***kwargs: Any*) → NoReturn
Find (all) items of this type.

Subclasses can specify *args* and *kwargs* differently, and may use them for filtering. However, when the method is called with no additional positional or keyword arguments, subclasses are obliged to to yield all items.

Returns Iterator yielding Items

property name: str

Returns Name of our remote ref

note

old_commit

ref

classmethod refresh() → Literal[True]
Update information about which *git-fetch(1)* flags are supported by the git executable being used.

Called by the *git.refresh()* function in the top level __init__.

remote_ref_path

class git.remote.**PushInfo**(*flags: int*, *local_ref: Optional[*git.refs.symbolic.SymbolicReference*]*, *remote_ref_string: str*, *remote:* git.remote.Remote, *old_commit: Optional[str] = None*, *summary: str = ''*)

Carries information about the result of a push operation of a single head:

```
info = remote.push()[0]
info.flags          # bitflags providing more information about the result
info.local_ref      # Reference pointing to the local reference that was pushed
                    # It is None if the ref was deleted.
info.remote_ref_string # path to the remote reference located on the remote side
info.remote_ref # Remote Reference on the local side corresponding to
                # the remote_ref_string. It can be a TagReference as well.
info.old_commit # commit at which the remote_ref was standing before we pushed
                # it to local_ref.commit. Will be None if an error was indicated
info.summary    # summary line providing human readable english text about the push
```

DELETED = 64

ERROR = 1024

FAST_FORWARD = 256

FORCED_UPDATE = 128

NEW_HEAD = 2

NEW_TAG = 1

NO_MATCH = 4

REJECTED = 8

REMOTE_FAILURE = 32

REMOTE_REJECTED = 16

UP_TO_DATE = 512

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': 'str'}

__init__(*flags: int*, *local_ref: Optional[*git.refs.symbolic.SymbolicReference*]*, *remote_ref_string: str*, *remote:* git.remote.Remote, *old_commit: Optional[str] = None*, *summary: str = ''*) → None

Initialize a new instance.

local_ref: HEAD | Head | RemoteReference | TagReference | Reference | SymbolicReference | None

__module__ = 'git.remote'

__parameters__ = ()

__slots__ = ('local_ref', 'remote_ref_string', 'flags', '_old_commit_sha', '_remote', 'summary')

classmethod __subclasshook__(*other*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

flags

classmethod iter_items(*repo: Repo*, **args: Any*, ***kwargs: Any*) → NoReturn
Find (all) items of this type.

Subclasses can specify *args* and *kwargs* differently, and may use them for filtering. However, when the method is called with no additional positional or keyword arguments, subclasses are obliged to to yield all items.

Returns Iterator yielding Items

local_ref

property old_commit: Optional[Commit]

property remote_ref: Union[*git.refs.remote.RemoteReference*, *git.refs.tag.TagReference*]

Returns Remote *Reference* or *TagReference* in the local repository corresponding to the *remote_ref_string* kept in this instance.

remote_ref_string

summary

class git.remote.**Remote**(*repo: Repo*, *name: str*)
Provides easy read and write access to a git remote.

Everything not part of this interface is considered an option for the current remote, allowing constructs like `remote.pushurl` to query the pushurl.

Note When querying configuration, the configuration accessor will be cached to speed up subsequent accesses.

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': 'str', 'url': <class 'str'>}

__eq__(*other: object*) → bool
Return self==value.

__getattr__(*attr: str*) → Any
Allows to call this instance like `remote.special(*args, **kwargs)` to call `git remote special self.name`.

__hash__() → int
Return hash(self).

__init__(*repo: Repo*, *name: str*) → None
Initialize a remote instance.

Parameters

- **repo** – The repository we are a remote of.
- **name** – The name of the remote, e.g. `origin`.

__module__ = 'git.remote'

__ne__(*other: object*) → bool
Return self!=value.

__parameters__ = ()

__repr__() → str
Return repr(self).

__slots__ = ('repo', 'name', '_config_reader')

__str__() → str
Return str(self).

classmethod __subclasshook__(*other*)
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

classmethod add(*repo: Repo*, *name: str*, *url: str*, ***kwargs: Any*) → *Remote*

add_url(*url: str*, *allow_unsafe_protocols: bool = False*, ***kwargs: Any*) → *git.remote.Remote*
Adds a new url on current remote (special case of `git remote set-url`).

This command adds new URLs to a given remote, making it possible to have multiple URLs for a single remote.

Parameters

- **url** – String being the URL to add as an extra remote URL.
- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.

Returns self

property config_reader: *git.config.SectionConstraint*[*git.config.GitConfigParser*]

Returns *GitConfigParser* compatible object able to read options for only our remote. Hence you may simply type `config.get("pushurl")` to obtain the information.

property config_writer: *git.config.SectionConstraint*

Returns *GitConfigParser*-compatible object able to write options for this remote.

Note You can only own one writer at a time - delete it to release the configuration file and make it usable by others.

To assure consistent results, you should only query options through the writer. Once you are done writing, you are free to use the config reader once again.

classmethod create(*repo: Repo*, *name: str*, *url: str*, *allow_unsafe_protocols: bool = False*, ***kwargs: Any*) → *Remote*
Create a new remote to the given repository.

Parameters

- **repo** – Repository instance that is to receive the new remote.
- **name** – Desired name of the remote.
- **url** – URL which corresponds to the remote's name.
- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.
- **kwargs** – Additional arguments to be passed to the `git remote add` command.

Returns New *Remote* instance

Raises *git.exc.GitCommandError* – In case an origin with that name already exists.

delete_url(*url: str*, ***kwargs: Any*) → *git.remote.Remote*
Deletes a new url on current remote (special case of `git remote set-url`).

This command deletes new URLs to a given remote, making it possible to have multiple URLs for a single remote.

Parameters **url** – String being the URL to delete from the remote.

Returns self

exists() → bool

Returns `True` if this is a valid, existing remote. Valid remotes have an entry in the repository's configuration.

fetch(*refspec: Optional[Union[str, List[str]]] = None*, *progress: Union[*git.util.RemoteProgress*, None, UpdateProgress] = None*, *verbose: bool = True*, *kill_after_timeout: Union[None, float] = None*, *allow_unsafe_protocols: bool = False*, *allow_unsafe_options: bool = False*, ***kwargs: Any*) → *git.util.IterableList*[*git.remote.FetchInfo*]

Fetch the latest changes for this remote.

Parameters

- **refspec** – A "refspec" is used by fetch and push to describe the mapping between remote ref and local ref. They are combined with a colon in the format <src>:<dst>, preceded by an optional plus sign, +. For example: `git fetch $URL refs/heads/master:refs/heads/origin` means "grab the master branch head from the $URL and store it as my origin branch head". And `git push $URL refs/heads/master:refs/heads/to-upstream` means "publish my master branch head as to-upstream branch at $URL". See also *git-push(1)*.

  Taken from the git manual, *gitglossary(7)*.

  Fetch supports multiple refspecs (as the underlying *git-fetch(1)* does) - supplying a list rather than a string for 'refspec' will make use of this facility.

- **progress** – See the *push()* method.
- **verbose** – Boolean for verbose output.
- **kill_after_timeout** – To specify a timeout in seconds for the git command, after which the process should be killed. It is set to `None` by default.
- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.
- **allow_unsafe_options** – Allow unsafe options to be used, like `--upload-pack`.
- **kwargs** – Additional arguments to be passed to *git-fetch(1)*.

Returns IterableList(FetchInfo, ...) list of *FetchInfo* instances providing detailed information about the fetch results

Note As fetch does not provide progress information to non-ttys, we cannot make it available here unfortunately as in the *push()* method.

classmethod iter_items(*repo: Repo*, **args: Any*, ***kwargs: Any*) → Iterator[*Remote*]

Returns Iterator yielding *Remote* objects of the given repository

name

pull(*refspec: Optional[Union[str, List[str]]] = None*, *progress: Optional[Union[*git.util.RemoteProgress*, UpdateProgress]] = None*, *kill_after_timeout: Union[None, float] = None*, *allow_unsafe_protocols: bool = False*, *allow_unsafe_options: bool = False*, ***kwargs: Any*) → *git.util.IterableList*[*git.remote.FetchInfo*]

Pull changes from the given branch, being the same as a fetch followed by a merge of branch with your local branch.

Parameters

- **refspec** – See *fetch()* method.
- **progress** – See *push()* method.
- **kill_after_timeout** – See *fetch()* method.
- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.
- **allow_unsafe_options** – Allow unsafe options to be used, like `--upload-pack`.
- **kwargs** – Additional arguments to be passed to *git-pull(1)*.

Returns Please see *fetch()* method.

push(*refspec: Optional[Union[str, List[str]]] = None*, *progress: Optional[Union[*git.util.RemoteProgress*, UpdateProgress, Callable[[...],* git.util.RemoteProgress*]]] = None*, *kill_after_timeout: Union[None, float] = None*, *allow_unsafe_protocols: bool = False*, *allow_unsafe_options: bool = False*, ***kwargs: Any*) → git.remote.PushInfoList

Push changes from source branch in refspec to target branch in refspec.

Parameters

- **refspec** – See *fetch()* method.
- **progress** – Can take one of many value types:
  - `None`, to discard progress information.
  - A function (callable) that is called with the progress information. Signature: `progress(op_code, cur_count, max_count=None, message='')`. See *RemoteProgress.update* for a description of all arguments given to the function.
  - An instance of a class derived from *RemoteProgress* that overrides the *RemoteProgress.update* method.
- **kill_after_timeout** – To specify a timeout in seconds for the git command, after which the process should be killed. It is set to `None` by default.
- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.
- **allow_unsafe_options** – Allow unsafe options to be used, like `--receive-pack`.
- **kwargs** – Additional arguments to be passed to *git-push(1)*.

Note No further progress information is returned after push returns.

Returns

A `PushInfoList` object, where each list member represents an individual head which had been updated on the remote side.

If the push contains rejected heads, these will have the *PushInfo.ERROR* bit set in their flags.

If the operation fails completely, the length of the returned `PushInfoList` will be 0.

Call `raise_if_error()` on the returned object to raise on any failure.

property refs: *git.util.IterableList*[*git.refs.remote.RemoteReference*]

Returns

IterableList of *RemoteReference* objects.

It is prefixed, allowing you to omit the remote path portion, e.g.:

```
remote.refs.master  # yields RemoteReference('/refs/remotes/origin/
↪master')
```

classmethod remove(*repo: Repo*, *name: str*) → str
Remove the remote with the given name.

Returns The passed remote name to remove

rename(*new_name: str*) → *git.remote.Remote*
Rename self to the given *new_name*.

Returns self

repo

classmethod rm(*repo: Repo*, *name: str*) → str
Remove the remote with the given name.

Returns The passed remote name to remove

set_url(*new_url: str*, *old_url: Optional[str] = None*, *allow_unsafe_protocols: bool = False*, ***kwargs: Any*) → *git.remote.Remote*
Configure URLs on current remote (cf. command `git remote set-url`).

This command manages URLs on the remote.

Parameters

- **new_url** – String being the URL to add as an extra remote URL.
- **old_url** – When set, replaces this URL with *new_url* for the remote.
- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.

Returns self

property stale_refs: *git.util.IterableList*[*git.refs.reference.Reference*]

Returns

IterableList of *RemoteReference* objects that do not have a corresponding head in the remote reference anymore as they have been deleted on the remote side, but are still available locally.

The *IterableList* is prefixed, hence the 'origin' must be omitted. See *refs* property for an example.

To make things more complicated, it can be possible for the list to include other kinds of references, for example, tag references, if these are stale as well. This is a fix for the issue described here: https://github.com/gitpython-developers/GitPython/issues/260

unsafe_git_fetch_options = ['--upload-pack']

unsafe_git_pull_options = ['--upload-pack']

unsafe_git_push_options = ['--receive-pack', '--exec']

update(***kwargs: Any*) → *git.remote.Remote*
Fetch all changes for this remote, including new branches which will be forced in (in case your local remote branch is not part the new remote branch's ancestry anymore).

Parameters **kwargs** – Additional arguments passed to `git remote update`.

Returns self

url: str
The URL configured for the remote.

property urls: Iterator[str]

Returns Iterator yielding all configured URL targets on a remote as strings

class git.remote.**RemoteProgress**
Handler providing an interface to parse progress information emitted by *git-push(1)* and *git-fetch(1)* and to dispatch callbacks allowing subclasses to react to the progress.

BEGIN = 1

CHECKING_OUT = 256

COMPRESSING = 8

COUNTING = 4

DONE_TOKEN = 'done.'

END = 2

FINDING_SOURCES = 128

OP_MASK = -4

RECEIVING = 32

RESOLVING = 64

STAGE_MASK = 3

TOKEN_SEPARATOR = ', '

WRITING = 16

__annotations__ = {'_cur_line': 'Optional[str]', '_num_op_codes': <class 'int'>, '_seen_ops': 'List[int]', 'error_lines': 'List[str]', 'other_lines': 'List[str]'}

__init__() → None

__module__ = 'git.util'

__slots__ = ('_cur_line', '_seen_ops', 'error_lines', 'other_lines')

error_lines: List[str]

line_dropped(*line: str*) → None
Called whenever a line could not be understood and was therefore dropped.

new_message_handler() → Callable[[str], None]

Returns A progress handler suitable for `handle_process_output()`, passing lines on to this progress handler in a suitable format.

other_lines: List[str]

re_op_absolute = re.compile('(remote:)?([\\w\\s]+):\\s+()(\\d+)()(.*)')

re_op_relative = re.compile('(remote:)?([\\w\\s]+):\\s+(\\d+)% \\((\\d+)/(\\d+)\\)(.*)')

update(*op_code: int*, *cur_count: Union[str, float]*, *max_count: Optional[Union[str, float]] = None*, *message: str = ''*) → None
Called whenever the progress changes.

Parameters

- **op_code** – Integer allowing to be compared against Operation IDs and stage IDs.

  Stage IDs are *BEGIN* and *END*. *BEGIN* will only be set once for each Operation ID as well as *END*. It may be that *BEGIN* and *END* are set at once in case only one progress message was emitted due to the speed of the operation. Between *BEGIN* and *END*, none of these flags will be set.

  Operation IDs are all held within the *OP_MASK*. Only one Operation ID will be active per call.

- **cur_count** – Current absolute count of items.
- **max_count** – The maximum count of items we expect. It may be `None` in case there is no maximum number of items or if it is (yet) unknown.
- **message** – In case of the *WRITING* operation, it contains the amount of bytes transferred. It may possibly be used for other purposes as well.

Note You may read the contents of the current line in `self._cur_line`.

4.27 Repo.Base

class git.repo.base.**Repo**(*path: Optional[Union[str, os.PathLike[str]]] = None*, *odbt: Type[gitdb.db.loose.LooseObjectDB] = <class 'git.db.GitCmdObjectDB'>*, *search_parent_directories: bool = False*, *expand_vars: bool = True*)

Represents a git repository and allows you to query references, create commit information, generate diffs, create and clone repositories, and query the log.

The following attributes are worth using:

- *working_dir* is the working directory of the git command, which is the working tree directory if available or the `.git` directory in case of bare repositories.
- *working_tree_dir* is the working tree directory, but will return `None` if we are a bare repository.
- *git_dir* is the `.git` repository directory, which is always set.

DAEMON_EXPORT_FILE = 'git-daemon-export-ok'

GitCommandWrapperType
alias of *git.cmd.Git*

__annotations__ = {'_common_dir': 'PathLike', '_working_tree_dir': 'Optional[PathLike]', 'config_level': 'ConfigLevels_Tup', 'git_dir': 'PathLike', 'working_dir': 'PathLike'}

__del__() → None

```
__dict__ = mappingproxy({'__module__': 'git.repo.base', '__annotations__':
{'working_dir': 'PathLike', '_working_tree_dir': 'Optional[PathLike]', 'git_dir':
'PathLike', '_common_dir': 'PathLike', 'config_level': 'ConfigLevels_Tup'},
'__doc__': 'Represents a git repository and allows you to query references, create
commit\n information, generate diffs, create and clone repositories, and query the
log.\n\n The following attributes are worth using:\n\n * :attr:`working_dir` is the
working directory of the git command, which is the\n working tree directory if
available or the ``.git`` directory in case of bare\n repositories.\n\n *
:attr:`working_tree_dir` is the working tree directory, but will return ``None``\n
if we are a bare repository.\n\n * :attr:`git_dir` is the ``.git`` repository
directory, which is always set.\n ', 'DAEMON_EXPORT_FILE': 'git-daemon-export-ok',
'git': None, '_working_tree_dir': None, '_common_dir': '', 're_whitespace':
re.compile('\\s+'), 're_hexsha_only': re.compile('^[0-9A-Fa-f]{40}$'),
're_hexsha_shortened': re.compile('^[0-9A-Fa-f]{4,40}$'), 're_envvars':
re.compile('(\\$(\\{\\s?)?[a-zA-Z_]\\w*(\\}\\s?)?|%\\s?[a-zA-Z_]\\w*\\s?%)'),
're_author_committer_start': re.compile('^(author|committer)'), 're_tab_full_line':
re.compile('^\\t(.*)$'), 'unsafe_git_clone_options': ['--upload-pack', '-u',
'--config', '-c'], 'config_level': ('system', 'user', 'global', 'repository'),
'GitCommandWrapperType': <class 'git.cmd.Git'>, '__init__': <function
Repo.__init__>, '__enter__': <function Repo.__enter__>, '__exit__': <function
Repo.__exit__>, '__del__': <function Repo.__del__>, 'close': <function
Repo.close>, '__eq__': <function Repo.__eq__>, '__ne__': <function Repo.__ne__>,
'__hash__': <function Repo.__hash__>, 'description': <property object>,
'working_tree_dir': <property object>, 'common_dir': <property object>, 'bare':
<property object>, 'heads': <property object>, 'references': <property object>,
'refs': <property object>, 'branches': <property object>, 'index': <property
object>, 'head': <property object>, 'remotes': <property object>, 'remote':
<function Repo.remote>, 'submodules': <property object>, 'submodule': <function
Repo.submodule>, 'create_submodule': <function Repo.create_submodule>,
'iter_submodules': <function Repo.iter_submodules>, 'submodule_update': <function
Repo.submodule_update>, 'tags': <property object>, 'tag': <function Repo.tag>,
'_to_full_tag_path': <staticmethod(<function Repo._to_full_tag_path>)>,
'create_head': <function Repo.create_head>, 'delete_head': <function
Repo.delete_head>, 'create_tag': <function Repo.create_tag>, 'delete_tag':
<function Repo.delete_tag>, 'create_remote': <function Repo.create_remote>,
'delete_remote': <function Repo.delete_remote>, '_get_config_path': <function
Repo._get_config_path>, 'config_reader': <function Repo.config_reader>,
'_config_reader': <function Repo._config_reader>, 'config_writer': <function
Repo.config_writer>, 'commit': <function Repo.commit>, 'iter_trees': <function
Repo.iter_trees>, 'tree': <function Repo.tree>, 'iter_commits': <function
Repo.iter_commits>, 'merge_base': <function Repo.merge_base>, 'is_ancestor':
<function Repo.is_ancestor>, 'is_valid_object': <function Repo.is_valid_object>,
'daemon_export': <property object>, '_get_alternates': <function
Repo._get_alternates>, '_set_alternates': <function Repo._set_alternates>,
'alternates': <property object>, 'is_dirty': <function Repo.is_dirty>,
'untracked_files': <property object>, '_get_untracked_files': <function
Repo._get_untracked_files>, 'ignored': <function Repo.ignored>, 'active_branch':
<property object>, 'blame_incremental': <function Repo.blame_incremental>, 'blame':
<function Repo.blame>, 'init': <classmethod(<function Repo.init>)>, '_clone':
<classmethod(<function Repo._clone>)>, 'clone': <function Repo.clone>,
'clone_from': <classmethod(<function Repo.clone_from>)>, 'archive': <function
Repo.archive>, 'has_separate_working_tree': <function
Repo.has_separate_working_tree>, 'rev_parse': <function rev_parse>, '__repr__':
<function Repo.__repr__>, 'currently_rebasing_on': <function
Repo.currently_rebasing_on>, '__dict__': <attribute '__dict__' of 'Repo' objects>,
'__weakref__': <attribute '__weakref__' of 'Repo' objects>})
```

__enter__() → *git.repo.base.Repo*

__eq__(*rhs: object*) → bool
Return self==value.

__exit__(**args: Any*) → None

__hash__() → int
Return hash(self).

__init__(*path: Optional[Union[str, os.PathLike[str]]] = None*, *odbt: Type[gitdb.db.loose.LooseObjectDB] = <class 'git.db.GitCmdObjectDB'>*, *search_parent_directories: bool = False*, *expand_vars: bool = True*) → None
Create a new *Repo* instance.

Parameters

- **path** – The path to either the root git directory or the bare git repo:

```
repo = Repo("/Users/mtrier/Development/git-python")
repo = Repo("/Users/mtrier/Development/git-python.git")
repo = Repo("~/Development/git-python.git")
repo = Repo("$REPOSITORIES/Development/git-python.git")
repo = Repo(R"C:\Users\mtrier\Development\git-python\.git")
```

  - In *Cygwin*, *path* may be a `cygdrive/...` prefixed path.
  - If *path* is `None` or an empty string, `GIT_DIR` is used. If that environment variable is absent or empty, the current directory is used.

- **odbt** – Object DataBase type - a type which is constructed by providing the directory containing the database objects, i.e. `.git/objects`. It will be used to access all object data.
- **search_parent_directories** – If `True`, all parent directories will be searched for a valid repo as well.

  Please note that this was the default behaviour in older versions of GitPython, which is considered a bug though.

Raises

- *git.exc.InvalidGitRepositoryError* –
- *git.exc.NoSuchPathError* –

Returns *Repo*

__module__ = 'git.repo.base'

__ne__(*rhs: object*) → bool
Return self!=value.

__repr__() → str
Return repr(self).

__weakref__
list of weak references to the object (if defined)

property active_branch: *git.refs.head.Head*
The name of the currently active branch.

Raises **TypeError** – If HEAD is detached.

Returns *Head* to the active branch

property alternates: List[str]

Retrieve a list of alternates paths or set a list paths to be used as alternates

archive(*ostream: Union[TextIO, BinaryIO]*, *treeish: Optional[str] = None*, *prefix: Optional[str] = None*, ***kwargs: Any*) → *git.repo.base.Repo*

Archive the tree at the given revision.

Parameters

- **ostream** – File-compatible stream object to which the archive will be written as bytes.
- **treeish** – The treeish name/id, defaults to active branch.
- **prefix** – The optional prefix to prepend to each filename in the archive.
- **kwargs** – Additional arguments passed to *git-archive(1)*:
  - Use the format argument to define the kind of format. Use specialized ostreams to write any format supported by Python.
  - You may specify the special path keyword, which may either be a repository-relative path to a directory or file to place into the archive, or a list or tuple of multiple paths.

Raises *git.exc.GitCommandError* – If something went wrong.

Returns self

property bare: bool

Returns True if the repository is bare

blame(*rev: Optional[Union[str,* git.refs.head.HEAD*]]*, *file: str*, *incremental: bool = False*, *rev_opts: Optional[List[str]] = None*, ***kwargs: Any*) →
Optional[Union[List[List[List[Optional[Union[*git.objects.commit.Commit*, List[str | bytes]]]]]], Iterator[git.repo.base.BlameEntry]]]

The blame information for the given file at the given revision.

Parameters **rev** – Revision specifier. If None, the blame will include all the latest uncommitted changes. Otherwise, anything successfully parsed by *git-rev-parse(1)* is a valid option.

Returns

list: [git.Commit, list: [<line>]]

A list of lists associating a *Commit* object with a list of lines that changed within the given commit. The *Commit* objects will be given in order of appearance.

blame_incremental(*rev: str* | git.refs.head.HEAD | *None*, *file: str*, ***kwargs: Any*) →
Iterator[git.repo.base.BlameEntry]

Iterator for blame information for the given file at the given revision.

Unlike *blame()*, this does not return the actual file's contents, only a stream of BlameEntry tuples.

Parameters **rev** – Revision specifier. If None, the blame will include all the latest uncommitted changes. Otherwise, anything successfully parsed by *git-rev-parse(1)* is a valid option.

Returns Lazy iterator of BlameEntry tuples, where the commit indicates the commit to blame for the line, and range indicates a span of line numbers in the resulting file.

If you combine all line number ranges outputted by this command, you should get a continuous range spanning all line numbers in the file.

property branches: IterableList[Head]

A list of *Head* objects representing the branch heads in this repo.

Returns `git.IterableList(Head, ...)`

clone(*path: Union[str, os.PathLike[str]]*, *progress: Optional[Callable[[int, Union[str, float], Optional[Union[str, float]], str], None]] = None*, *multi_options: Optional[List[str]] = None*, *allow_unsafe_protocols: bool = False*, *allow_unsafe_options: bool = False*, ***kwargs: Any*) → *git.repo.base.Repo*

Create a clone from this repository.

Parameters

- **path** – The full path of the new repo (traditionally ends with `./<name>.git`).
- **progress** – See *Remote.push*.
- **multi_options** – A list of *git-clone(1)* options that can be provided multiple times.

  One option per list item which is passed exactly as specified to clone. For example:

```
[
    "--config core.filemode=false",
    "--config core.ignorecase",
    "--recurse-submodule=repo1_path",
    "--recurse-submodule=repo2_path",
]
```

- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.
- **allow_unsafe_options** – Allow unsafe options to be used, like `--upload-pack`.
- **kwargs** –
  - `odbt` = ObjectDatabase Type, allowing to determine the object database implementation used by the returned *Repo* instance.
  - All remaining keyword arguments are given to the *git-clone(1)* command.

Returns *Repo* (the newly cloned repo)

classmethod clone_from(*url: Union[str, os.PathLike[str]]*, *to_path: Union[str, os.PathLike[str]]*, *progress: Optional[Callable[[int, Union[str, float], Optional[Union[str, float]], str], None]] = None*, *env: Optional[Mapping[str, str]] = None*, *multi_options: Optional[List[str]] = None*, *allow_unsafe_protocols: bool = False*, *allow_unsafe_options: bool = False*, ***kwargs: Any*) → *git.repo.base.Repo*

Create a clone from the given URL.

Parameters

- **url** – Valid git url, see: https://git-scm.com/docs/git-clone#URLS
- **to_path** – Path to which the repository should be cloned to.
- **progress** – See *Remote.push*.
- **env** – Optional dictionary containing the desired environment variables.

Note: Provided variables will be used to update the execution environment for `git`. If some variable is not specified in *env* and is defined in `os.environ`, value from `os.environ` will be used. If you want to unset some variable, consider providing empty string as its value.

- **multi_options** – See the *clone()* method.
- **allow_unsafe_protocols** – Allow unsafe protocols to be used, like `ext`.
- **allow_unsafe_options** – Allow unsafe options to be used, like `--upload-pack`.
- **kwargs** – See the *clone()* method.

Returns *Repo* instance pointing to the cloned directory.

close() → None

commit(*rev: Union[str, Commit_ish, None] = None*) → Commit

The *Commit* object for the specified revision.

Parameters **rev** – Revision specifier, see *git-rev-parse(1)* for viable options.

Returns *Commit*

property common_dir: Union[str, os.PathLike[str]]

Returns The git dir that holds everything except possibly HEAD, FETCH_HEAD, ORIG_HEAD, COMMIT_EDITMSG, index, and logs/.

config_level: Tuple[Literal['system'], Literal['user'], Literal['global'], Literal['repository']] = ('system', 'user', 'global', 'repository')

Represents the configuration level of a configuration file.

config_reader(*config_level: Optional[Literal['system', 'global', 'user', 'repository']] = None*) → *git.config.GitConfigParser*

Returns

GitConfigParser allowing to read the full git configuration, but not to write it.

The configuration will include values from the system, user and repository configuration files.

Parameters **config_level** – For possible values, see the *config_writer()* method. If `None`, all applicable levels will be used. Specify a level in case you know which file you wish to read to prevent reading multiple files.

Note On Windows, system configuration cannot currently be read as the path is unknown, instead the global path will be used.

config_writer(*config_level: Literal['system', 'global', 'user', 'repository'] = 'repository'*) → *git.config.GitConfigParser*

Returns A *GitConfigParser* allowing to write values of the specified configuration file level. Config writers should be retrieved, used to change the configuration, and written right away as they will lock the configuration file in question and prevent other's to write it.

Parameters **config_level** – One of the following values:

- `"system"` = system wide configuration file
- `"global"` = user level configuration file

- "`repository" = configuration file for this repository only

create_head(*path: PathLike*, *commit: Union['SymbolicReference', 'str'] = 'HEAD'*, *force: bool = False*, *logmsg: Optional[str] = None*) → Head
Create a new head within the repository.

Note For more documentation, please see the `Head.create` method.

Returns Newly created *Head* Reference.

create_remote(*name: str*, *url: str*, ***kwargs: Any*) → *git.remote.Remote*
Create a new remote.

For more information, please see the documentation of the *Remote.create* method.

Returns *Remote* reference

create_submodule(**args: Any*, ***kwargs: Any*) → *git.objects.submodule.base.Submodule*
Create a new submodule.

Note For a description of the applicable parameters, see the documentation of *Submodule.add*.

Returns The created submodule.

create_tag(*path: PathLike*, *ref: Union[str, 'SymbolicReference'] = 'HEAD'*, *message: Optional[str] = None*, *force: bool = False*, ***kwargs: Any*) → TagReference
Create a new tag reference.

Note For more documentation, please see the *TagReference.create* method.

Returns *TagReference* object

currently_rebasing_on() → *git.objects.commit.Commit* | None

Returns

The commit which is currently being replayed while rebasing.

`None` if we are not currently rebasing.

property daemon_export: bool
If True, git-daemon may export this repository

delete_head(**heads: Union[str,* git.refs.head.Head*]*, ***kwargs: Any*) → None
Delete the given heads.

Parameters **kwargs** – Additional keyword arguments to be passed to *git-branch(1)*.

delete_remote(*remote:* git.remote.Remote) → str
Delete the given remote.

delete_tag(**tags:* git.refs.tag.TagReference) → None
Delete the given tag references.

property description: str
the project's description

git = None

git_dir: Union[str, os.PathLike[str]]
The `.git` repository directory.

has_separate_working_tree() → bool

Returns True if our *git_dir* is not at the root of our *working_tree_dir*, but a `.git` file with a platform-agnostic symbolic link. Our *git_dir* will be wherever the `.git` file points to.

Note Bare repositories will always return `False` here.

property head: *git.refs.head.HEAD*

Returns *HEAD* object pointing to the current head reference

property heads: `IterableList[Head]`

A list of *Head* objects representing the branch heads in this repo.

Returns `git.IterableList(Head, ...)`

ignored(**paths: Union[str, os.PathLike[str]]*) → List[str]

Checks if paths are ignored via `.gitignore`.

This does so using the *git-check-ignore(1)* method.

Parameters **paths** – List of paths to check whether they are ignored or not.

Returns Subset of those paths which are ignored

property index: *git.index.base.IndexFile*

Returns A *IndexFile* representing this repository's index.

Note This property can be expensive, as the returned *IndexFile* will be reinitialized. It is recommended to reuse the object.

classmethod init(*path: Optional[Union[str, os.PathLike[str]]] = None*, *mkdir: bool = True*, *odbt: Type[git.db.GitCmdObjectDB] = <class 'git.db.GitCmdObjectDB'>*, *expand_vars: bool = True*, ***kwargs: Any*) → *git.repo.base.Repo*

Initialize a git repository at the given path if specified.

Parameters

- **path** – The full path to the repo (traditionally ends with `/<name>.git`). Or `None`, in which case the repository will be created in the current working directory.
- **mkdir** – If specified, will create the repository directory if it doesn't already exist. Creates the directory with a mode=0755. Only effective if a path is explicitly given.
- **odbt** – Object DataBase type - a type which is constructed by providing the directory containing the database objects, i.e. `.git/objects`. It will be used to access all object data.
- **expand_vars** – If specified, environment variables will not be escaped. This can lead to information disclosure, allowing attackers to access the contents of environment variables.
- **kwargs** – Keyword arguments serving as additional options to the *git-init(1)* command.

Returns *Repo* (the newly created repo)

is_ancestor(*ancestor_rev:* git.objects.commit.Commit, *rev:* git.objects.commit.Commit) → bool

Check if a commit is an ancestor of another.

Parameters

- **ancestor_rev** – Rev which should be an ancestor.
- **rev** – Rev to test against *ancestor_rev*.

Returns `True` if *ancestor_rev* is an ancestor to *rev*.

is_dirty(*index: bool = True*, *working_tree: bool = True*, *untracked_files: bool = False*, *submodules: bool = True*, *path: Optional[Union[str, os.PathLike[str]]] = None*) → bool

Returns `True` if the repository is considered dirty. By default it will react like a *`git-status(1)`* without untracked files, hence it is dirty if the index or the working copy have changes.

is_valid_object(*sha: str*, *object_type: Optional[str] = None*) → bool

iter_commits(*rev: Union[str, Commit, 'SymbolicReference', None] = None*, *paths: Union[PathLike, Sequence[PathLike]] = ''*, ***kwargs: Any*) → Iterator[Commit]

An iterator of *`Commit`* objects representing the history of a given ref/commit.

Parameters

- **rev** – Revision specifier, see *`git-rev-parse(1)`* for viable options. If `None`, the active branch will be used.
- **paths** – An optional path or a list of paths. If set, only commits that include the path or paths will be returned.
- **kwargs** – Arguments to be passed to *`git-rev-list(1)`*. Common ones are `max_count` and `skip`.

Note To receive only commits between two named revisions, use the `"revA...revB"` revision specifier.

Returns Iterator of *`Commit`* objects

iter_submodules(**args: Any*, ***kwargs: Any*) → Iterator[*git.objects.submodule.base.Submodule*]

An iterator yielding Submodule instances.

See the *~git.objects.util.Traversable* interface for a description of *args* and *kwargs*.

Returns Iterator

iter_trees(**args: Any*, ***kwargs: Any*) → Iterator['Tree']

Returns Iterator yielding *`Tree`* objects

Note Accepts all arguments known to the *`iter_commits()`* method.

merge_base(**rev: Any*, ***kwargs: Any*) → List[*git.objects.commit.Commit*]

Find the closest common ancestor for the given revision (*`Commit`*s, *`Tag`*s, *`Reference`*s, etc.).

Parameters

- **rev** – At least two revs to find the common ancestor for.
- **kwargs** – Additional arguments to be passed to the `repo.git.merge_base()` command which does all the work.

Returns A list of *`Commit`* objects. If `--all` was not passed as a keyword argument, the list will have at max one *`Commit`*, or is empty if no common merge base exists.

Raises **ValueError** – If fewer than two revisions are provided.

re_author_committer_start = re.compile('^(author|committer)')

re_envvars =
re.compile('(\\$(\\{\\s?)?[a-zA-Z_]\\w*(\\}\\s?)?|%\\s?[a-zA-Z_]\\w*\\s?%)')

re_hexsha_only = re.compile('^[0-9A-Fa-f]{40}$')

re_hexsha_shortened = re.compile('^[0-9A-Fa-f]{4,40}$')

re_tab_full_line = re.compile('^\\t(.*)$')

re_whitespace = re.compile('\\s+')

property references: IterableList[Reference]
A list of *Reference* objects representing tags, heads and remote references.

Returns git.IterableList(Reference, ...)

property refs: IterableList[Reference]
A list of *Reference* objects representing tags, heads and remote references.

Returns git.IterableList(Reference, ...)

remote(*name: str = 'origin'*) → *git.remote.Remote*

Returns The remote with the specified name

Raises **ValueError** – If no remote with such a name exists.

property remotes: IterableList[Remote]
A list of *Remote* objects allowing to access and manipulate remotes.

Returns git.IterableList(Remote, ...)

rev_parse(*rev: str*) → AnyGitObject
Parse a revision string. Like *git-rev-parse(1)*.

Returns

~git.objects.base.Object at the given revision.

This may be any type of git object:

- *Commit*
- *TagObject*
- *Tree*
- *Blob*

Parameters **rev** – *git-rev-parse(1)*-compatible revision specification as string. Please see *git-rev-parse(1)* for details.

Raises

- ***gitdb.exc.BadObject*** – If the given revision could not be found.
- **ValueError** – If *rev* couldn't be parsed.
- **IndexError** – If an invalid reflog index is specified.

submodule(*name: str*) → *git.objects.submodule.base.Submodule*

Returns The submodule with the given name

Raises **ValueError** – If no such submodule exists.

submodule_update(**args: Any*, ***kwargs: Any*) → Iterator[*git.objects.submodule.base.Submodule*]
Update the submodules, keeping the repository consistent as it will take the previous state into consideration.

Note For more information, please see the documentation of *RootModule.update*.

property submodules: IterableList[Submodule]

Returns git.IterableList(Submodule, ...) of direct submodules available from the current head

tag(*path: Union[str, os.PathLike[str]]*) → *git.refs.tag.TagReference*

Returns *TagReference* object, reference pointing to a *Commit* or tag

Parameters **path** – Path to the tag reference, e.g. `0.1.5` or `tags/0.1.5`.

property tags: IterableList[TagReference]

A list of *TagReference* objects that are available in this repo.

Returns `git.IterableList(TagReference, ...)`

tree(*rev: Union[Tree_ish, str, None] = None*) → Tree

The *Tree* object for the given tree-ish revision.

Examples:

```
repo.tree(repo.heads[0])
```

Parameters **rev** – A revision pointing to a Treeish (being a commit or tree).

Returns *Tree*

Note If you need a non-root level tree, find it by iterating the root tree. Otherwise it cannot know about its path relative to the repository root and subsequent operations might have unexpected results.

unsafe_git_clone_options = ['--upload-pack', '-u', '--config', '-c']

Options to *git-clone(1)* that allow arbitrary commands to be executed.

The `--upload-pack`/`-u` option allows users to execute arbitrary commands directly: https://git-scm.com/docs/git-clone#Documentation/git-clone.txt—upload-packltupload-packgt

The `--config`/`-c` option allows users to override configuration variables like `protocol.allow` and `core.gitProxy` to execute arbitrary commands: https://git-scm.com/docs/git-clone#Documentation/git-clone.txt—configltkeygtltvaluegt

property untracked_files: List[str]

Returns

list(str,...)

Files currently untracked as they have not been staged yet. Paths are relative to the current working directory of the git command.

Note Ignored files will not appear here, i.e. files mentioned in `.gitignore`.

Note This property is expensive, as no cache is involved. To process the result, please consider caching it yourself.

working_dir: Union[str, os.PathLike[str]]

The working directory of the git command.

property working_tree_dir: Optional[Union[str, os.PathLike[str]]]

Returns The working tree directory of our git repository. If this is a bare repository, `None` is returned.

4.28 Repo.Functions

General repository-related functions.

git.repo.fun.**deref_tag**(*tag: Tag*) → AnyGitObject
Recursively dereference a tag and return the resulting object.

git.repo.fun.**find_submodule_git_dir**(*d: Union[str, os.PathLike[str]]*) → Optional[Union[str, os.PathLike[str]]]
Search for a submodule repo.

git.repo.fun.**find_worktree_git_dir**(*dotgit: Union[str, os.PathLike[str]]*) → Optional[str]
Search for a gitdir for this worktree.

git.repo.fun.**is_git_dir**(*d: Union[str, os.PathLike[str]]*) → bool
This is taken from the git setup.c:is_git_directory function.

Raises *git.exc.WorkTreeRepositoryUnsupported* – If it sees a worktree directory. It's quite hacky to do that here, but at least clearly indicates that we don't support it. There is the unlikely danger to throw if we see directories which just look like a worktree dir, but are none.

git.repo.fun.**name_to_object**(*repo: Repo*, *name: str*, *return_ref: Literal[False] = False*) → AnyGitObject
git.repo.fun.**name_to_object**(*repo: Repo*, *name: str*, *return_ref: Literal[True]*) → Union[AnyGitObject, SymbolicReference]

Returns Object specified by the given name - hexshas (short and long) as well as references are supported.

Parameters **return_ref** – If `True`, and name specifies a reference, we will return the reference instead of the object. Otherwise it will raise *BadObject* or *BadName*.

git.repo.fun.**rev_parse**(*repo: Repo*, *rev: str*) → AnyGitObject
Parse a revision string. Like *git-rev-parse(1)*.

Returns

~git.objects.base.Object at the given revision.

This may be any type of git object:

- *Commit*
- *TagObject*
- *Tree*
- *Blob*

Parameters **rev** – *git-rev-parse(1)*-compatible revision specification as string. Please see *git-rev-parse(1)* for details.

Raises

- *gitdb.exc.BadObject* – If the given revision could not be found.
- **ValueError** – If *rev* couldn't be parsed.
- **IndexError** – If an invalid reflog index is specified.

git.repo.fun.**short_to_long**(*odb: GitCmdObjectDB*, *hexsha: str*) → Optional[bytes]

Returns Long hexadecimal sha1 from the given less than 40 byte hexsha, or `None` if no candidate could be found.

Parameters **hexsha** – hexsha with less than 40 bytes.

git.repo.fun.**to_commit**(*obj: Object*) → Commit
Convert the given object to a commit if possible and return it.

git.repo.fun.**touch**(*filename: str*) → str

4.29 Compat

Utilities to help provide compatibility with Python 3.

This module exists for historical reasons. Code outside GitPython may make use of public members of this module, but is unlikely to benefit from doing so. GitPython continues to use some of these utilities, in some cases for compatibility across different platforms.

git.compat.**__dir__**() → List[str]

git.compat.**__getattr__**(*name: str*) → Any

git.compat.**defenc = 'utf-8'**
The encoding used to convert between Unicode and bytes filenames.

git.compat.**is_darwin: bool = False**
Deprecated alias for `sys.platform == "darwin"` to check for macOS (Darwin).

This is deprecated because it clearer to write out `os.name` or `sys.platform` checks explicitly.

Note For macOS (Darwin), `os.name == "posix"` as in other Unix-like systems, while `sys.platform == "darwin"`.

git.compat.**is_posix: bool = True**
Deprecated alias for `os.name == "posix"` to check for Unix-like ("POSIX") systems.

This is deprecated because it clearer to write out `os.name` or `sys.platform` checks explicitly, especially in cases where it matters which is used.

Note For POSIX systems, more detailed information is available in `sys.platform`, while `os.name` is always `"posix"` on such systems, including macOS (Darwin).

git.compat.**is_win: bool = False**
Deprecated alias for `os.name == "nt"` to check for native Windows.

This is deprecated because it is clearer to write out `os.name` or `sys.platform` checks explicitly, especially in cases where it matters which is used.

Note `is_win` is `False` on Cygwin, but is often wrongly assumed `True`. To detect Cygwin, use `sys.platform == "cygwin"`.

git.compat.**safe_decode**(*s: None*) → None
git.compat.**safe_decode**(*s: AnyStr*) → str
Safely decode a binary string to Unicode.

git.compat.**safe_encode**(*s: None*) → None
git.compat.**safe_encode**(*s: AnyStr*) → bytes
Safely encode a binary string to Unicode.

git.compat.**win_encode**(*s: None*) → None
git.compat.**win_encode**(*s: AnyStr*) → bytes
Encode Unicode strings for process arguments on Windows.

4.30 DB

Module with our own gitdb implementation - it uses the git command.

class git.db.**GitCmdObjectDB**(*root_path: Union[str, os.PathLike[str]]*, *git: Git*)

A database representing the default git object store, which includes loose objects, pack files and an alternates file.

It will create objects only in the loose object database.

__init__(*root_path: Union[str, os.PathLike[str]]*, *git: Git*) → None

Initialize this instance with the root and a git command.

__module__ = 'git.db'

info(*binsha: bytes*) → gitdb.base.OInfo

Get a git object header (using git itself).

partial_to_complete_sha_hex(*partial_hexsha: str*) → bytes

Returns Full binary 20 byte sha from the given partial hexsha

Raises

- *gitdb.exc.AmbiguousObjectName* –
- *gitdb.exc.BadObject* –

Note Currently we only raise *BadObject* as git does not communicate ambiguous objects separately.

stream(*binsha: bytes*) → gitdb.base.OStream

Get git object data as a stream supporting `read()` (using git itself).

class git.db.**GitDB**(*root_path*)

A git-style object database, which contains all objects in the 'objects' subdirectory

IMPORTANT: The usage of this implementation is highly discouraged as it fails to release file-handles. This can be a problem with long-running processes and/or big repositories.

LooseDBCls

alias of `gitdb.db.loose.LooseObjectDB`

PackDBCls

alias of `gitdb.db.pack.PackedDB`

ReferenceDBCls

alias of `gitdb.db.ref.ReferenceDB`

__annotations__ = {}

__init__(*root_path*)

Initialize ourselves on a git objects directory

__module__ = 'gitdb.db.git'

alternates_dir = 'info/alternates'

loose_dir = ''

ostream()

Return the output stream

Returns overridden output stream this instance will write to, or None if it will write to the default stream

`packs_dir = 'pack'`

`set_ostream`(*ostream*)

Adjusts the stream to which all data should be sent when storing new objects

Parameters `stream` – if not None, the stream to use, if None the default stream will be used.

Returns previously installed stream, or None if there was no override

Raises `TypeError` – if the stream doesn't have the supported functionality

`store`(*istream*)

Create a new object in the database :return: the input istream object with its sha set to its corresponding value

Parameters `istream` – IStream compatible instance. If its sha is already set to a value, the object will just be stored in the our database format, in which case the input stream is expected to be in object format (header + contents).

Raises `IOError` – if data could not be written

4.31 Types

`git.types.AnyGitObject`

Union of the *Object*-based types that represent actual git object types.

As noted in *Object*, which has further details, these are:

- *Blob*
- *Tree*
- *Commit*
- *TagObject*

Those GitPython classes represent the four git object types, per *gitglossary(7)*:

- "blob": https://git-scm.com/docs/gitglossary#def_blob_object
- "tree object": https://git-scm.com/docs/gitglossary#def_tree_object
- "commit object": https://git-scm.com/docs/gitglossary#def_commit_object
- "tag object": https://git-scm.com/docs/gitglossary#def_tag_object

For more general information on git objects and their types as git understands them:

- "object": https://git-scm.com/docs/gitglossary#def_object
- "object type": https://git-scm.com/docs/gitglossary#def_object_type

Note See also the *Tree_ish* and *Commit_ish* unions.

alias of `Union`[`Commit`, `Tree`, `TagObject`, `Blob`]

`git.types.CallableProgress`

General type of a function or other callable used as a progress reporter for cloning.

This is the type of a function or other callable that reports the progress of a clone, when passed as a `progress` argument to *Repo.clone* or *Repo.clone_from*.

Note Those *clone()* and *clone_from()* methods also accept *RemoteProgress()* instances, including instances of its *CallableRemoteProgress()* subclass.

Note Unlike objects that match this type, *RemoteProgress()* instances are not directly callable, not even when they are instances of *CallableRemoteProgress()*, which wraps a callable and forwards information to it but is not itself callable.

Note This type also allows `None`, for cloning without reporting progress.

alias of `Optional[Callable[[int, Union[str, float], Optional[Union[str, float]], str], None]]`

`git.types.`**Commit_ish**

Union of *Object*-based types that are typically commit-ish.

See *gitglossary(7)* on "commit-ish": https://git-scm.com/docs/gitglossary#def_commit-ish

Note *Commit* is the only class whose instances are all commit-ish. This union type includes *Commit*, but also *TagObject*, only **most** of whose instances are commit-ish. Whether a particular *TagObject* peels (recursively dereferences) to a commit, rather than a tree or blob, can in general only be known at runtime. In practice, git tag objects are nearly always used for tagging commits, and such tags are of course commit-ish.

Note See also the *AnyGitObject* union of all four classes corresponding to git object types.

alias of `Union[Commit, TagObject]`

`git.types.`**ConfigLevels_Tup**

Static type of a tuple of the four strings representing configuration levels.

alias of `Tuple[Literal['system'], Literal['user'], Literal['global'], Literal['repository']]`

class `git.types.`**Files_TD**

Dictionary with stat counts for the diff of a particular file.

For the *files* attribute of *Stats* objects.

__annotations__ = {'deletions': <class 'int'>, 'insertions': <class 'int'>, 'lines': <class 'int'>}

__dict__ = mappingproxy({'__module__': 'git.types', '__annotations__': {'insertions': <class 'int'>, 'deletions': <class 'int'>, 'lines': <class 'int'>}, '__doc__': 'Dictionary with stat counts for the diff of a particular file.\n\n For the :class:`~git.util.Stats.files` attribute of :class:`~git.util.Stats`\n objects.\n ', '__orig_bases__': (<function TypedDict>,), '__dict__': <attribute '__dict__' of 'Files_TD' objects>, '__weakref__': <attribute '__weakref__' of 'Files_TD' objects>, '__required_keys__': frozenset({'insertions', 'deletions', 'lines'}), '__optional_keys__': frozenset(), '__total__': True})

__module__ = 'git.types'

__optional_keys__ = frozenset({})

__orig_bases__ = (<function TypedDict>,)

__required_keys__ = frozenset({'deletions', 'insertions', 'lines'})

__total__ = True

__weakref__

list of weak references to the object (if defined)

deletions: int

insertions: int

lines: int

git.types.**GitObjectTypeString**
Literal strings identifying git object types and the *Object*-based types that represent them.

See the *Object.type* attribute. These are its values in *Object* subclasses that represent git objects. These literals therefore correspond to the types in the *AnyGitObject* union.

These are the same strings git itself uses to identify its four object types. See *gitglossary(7)* on "object type": https://git-scm.com/docs/gitglossary#def_object_type

alias of `Literal`['commit', 'tag', 'blob', 'tree']

class git.types.**HSH_TD**
Dictionary carrying the same information as a *Stats* object.

__annotations__ = {'files': typing.Dict[typing.Union[str, ForwardRef('os.PathLike[str]')], git.types.Files_TD], 'total': <class 'git.types.Total_TD'>}

__dict__ = mappingproxy({'__module__': 'git.types', '__annotations__': {'total': <class 'git.types.Total_TD'>, 'files': typing.Dict[typing.Union[str, ForwardRef('os.PathLike[str]')], git.types.Files_TD]}, '__doc__': 'Dictionary carrying the same information as a :class:`~git.util.Stats` object.', '__orig_bases__': (<function TypedDict>,), '__dict__': <attribute '__dict__' of 'HSH_TD' objects>, '__weakref__': <attribute '__weakref__' of 'HSH_TD' objects>, '__required_keys__': frozenset({'total', 'files'}), '__optional_keys__': frozenset(), '__total__': True})

__module__ = 'git.types'

__optional_keys__ = frozenset({})

__orig_bases__ = (<function TypedDict>,)

__required_keys__ = frozenset({'files', 'total'})

__total__ = True

__weakref__
list of weak references to the object (if defined)

files: Dict[Union[str, os.PathLike[str]], *git.types.Files_TD*]

total: *git.types.Total_TD*

class git.types.**Has_Repo**(**args*, ***kwargs*)
Protocol for having a *repo* attribute, the repository to operate on.

__abstractmethods__ = frozenset({})

__annotations__ = {'repo': 'Repo'}

__callable_proto_members_only__ = False

__dict__ = mappingproxy({'__module__': 'git.types', '__annotations__': {'repo': 'Repo'}, '__doc__': 'Protocol for having a :attr:`repo` attribute, the repository to operate on.', '__dict__': <attribute '__dict__' of 'Has_Repo' objects>, '__weakref__': <attribute '__weakref__' of 'Has_Repo' objects>, '__parameters__': (), '_is_protocol': True, '__subclasshook__': <classmethod(<function _proto_hook>)>, '__init__': <function _no_init_or_replace_init>, '__abstractmethods__': frozenset(), '_abc_impl': <_abc._abc_data object>, '__protocol_attrs__': {'repo'}, '__callable_proto_members_only__': False, '_is_runtime_protocol': True})

__init__(**args*, ***kwargs*)

__module__ = 'git.types'

__parameters__ = ()

__protocol_attrs__ = {'repo'}

classmethod __subclasshook__(*other*)
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

repo: Repo

class git.types.Has_id_attribute(**args*, ***kwargs*)
Protocol for having _id_attribute_ used in iteration and traversal.

__abstractmethods__ = frozenset({})

__annotations__ = {'_id_attribute_': <class 'str'>}

__callable_proto_members_only__ = False

__dict__ = mappingproxy({'__module__': 'git.types', '__annotations__': {'_id_attribute_': <class 'str'>}, '__doc__': 'Protocol for having :attr:`_id_attribute_` used in iteration and traversal.', '__dict__': <attribute '__dict__' of 'Has_id_attribute' objects>, '__weakref__': <attribute '__weakref__' of 'Has_id_attribute' objects>, '__parameters__': (), '_is_protocol': True, '__subclasshook__': <classmethod(<function _proto_hook>)>, '__init__': <function _no_init_or_replace_init>, '__abstractmethods__': frozenset(), '_abc_impl': <_abc._abc_data object>, '__protocol_attrs__': {'_id_attribute_'}, '__callable_proto_members_only__': False, '_is_runtime_protocol': True})

__init__(**args*, ***kwargs*)

__module__ = 'git.types'

__parameters__ = ()

__protocol_attrs__ = {'_id_attribute_'}

classmethod __subclasshook__(*other*)
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

git.types.Lit_commit_ish
Deprecated. Type of literal strings identifying typically-commitish git object types.

Prior to a bugfix, this type had been defined more broadly. Any usage is in practice ambiguous and likely to be incorrect. This type has therefore been made a static type error to appear in annotations. It is preserved, with a deprecated status, to avoid introducing runtime errors in code that refers to it, but it should not be used.

Instead of this type:

- For the type of the string literals associated with *Commit_ish*, use `Literal["commit", "tag"]` or create a new type alias for it. That is equivalent to this type as currently defined (but usable in statically checked type annotations).
- For the type of all four string literals associated with *AnyGitObject*, use *GitObjectTypeString*. That is equivalent to the old definition of this type prior to the bugfix (and is also usable in statically checked type annotations).

alias of `Literal`['commit', 'tag']

`git.types.`**Lit_config_levels**
Type of literal strings naming git configuration levels.

These strings relate to which file a git configuration variable is in.

alias of `Literal`['system', 'global', 'user', 'repository']

`git.types.`**PathLike**
A `str` (Unicode) based file or directory path.

alias of `Union`[`str`, `os.PathLike`[`str`]]

`git.types.`**TBD**
alias of `Any`

class `git.types.`**Total_TD**
Dictionary with total stats from any number of files.

For the *total* attribute of *Stats* objects.

__annotations__ = {'deletions': <class 'int'>, 'files': <class 'int'>, 'insertions': <class 'int'>, 'lines': <class 'int'>}

__dict__ = mappingproxy({'__module__': 'git.types', '__annotations__': {'insertions': <class 'int'>, 'deletions': <class 'int'>, 'lines': <class 'int'>, 'files': <class 'int'>}, '__doc__': 'Dictionary with total stats from any number of files.\n\n For the :class:`~git.util.Stats.total` attribute of :class:`~git.util.Stats`\n objects.\n ', '__orig_bases__': (<function TypedDict>,), '__dict__': <attribute '__dict__' of 'Total_TD' objects>, '__weakref__': <attribute '__weakref__' of 'Total_TD' objects>, '__required_keys__': frozenset({'files', 'insertions', 'deletions', 'lines'}), '__optional_keys__': frozenset(), '__total__': True})

__module__ = 'git.types'

__optional_keys__ = frozenset({})

__orig_bases__ = (<function TypedDict>,)

__required_keys__ = frozenset({'deletions', 'files', 'insertions', 'lines'})

__total__ = True

__weakref__
list of weak references to the object (if defined)

deletions: int

files: int

insertions: int

lines: int

git.types.**Tree_ish**
Union of *Object*-based types that are typically tree-ish.

See *gitglossary(7)* on "tree-ish": https://git-scm.com/docs/gitglossary#def_tree-ish

Note *Tree* and *Commit* are the classes whose instances are all tree-ish. This union includes them, but also *TagObject*, only **most** of whose instances are tree-ish. Whether a particular *TagObject* peels (recursively dereferences) to a tree or commit, rather than a blob, can in general only be known at runtime. In practice, git tag objects are nearly always used for tagging commits, and such tags are tree-ish because commits are tree-ish.

Note See also the *AnyGitObject* union of all four classes corresponding to git object types.

alias of Union[Commit, Tree, TagObject]

git.types.**__dir__**() → List[str]

git.types.**__getattr__**(*name: str*) → Any

git.types.**assert_never**(*inp: NoReturn*, *raise_error: bool = True*, *exc: Optional[Exception] = None*) → None
For use in exhaustive checking of a literal or enum in if/else chains.

A call to this function should only be reached if not all members are handled, or if an attempt is made to pass non-members through the chain.

Parameters

- **inp** – If all members are handled, the argument for *inp* will have the Never/NoReturn type. Otherwise, the type will mismatch and cause a mypy error.
- **raise_error** – If True, will also raise ValueError with a general "unhandled literal" message, or the exception object passed as *exc*.
- **exc** – It not None, this should be an already-constructed exception object, to be raised if *raise_error* is True.

4.32 Util

class git.util.**Actor**(*name: Optional[str]*, *email: Optional[str]*)
Actors hold information about a person acting on the repository. They can be committers and authors or anything with a name and an email as mentioned in the git log entries.

__annotations__ = {}

__eq__(*other: Any*) → bool
Return self==value.

__hash__() → int
Return hash(self).

__init__(*name: Optional[str]*, *email: Optional[str]*) → None

__module__ = 'git.util'

__ne__(*other: Any*) → bool
Return self!=value.

__repr__() → str
Return repr(self).

__slots__ = ('name', 'email')

__str__() → str
Return str(self).

classmethod author(*config_reader: Union[None, GitConfigParser, SectionConstraint] = None*) → *Actor*
Same as *committer()*, but defines the main author. It may be specified in the environment, but defaults to the committer.

classmethod committer(*config_reader: Union[None, GitConfigParser, SectionConstraint] = None*) → *Actor*

Returns *Actor* instance corresponding to the configured committer. It behaves similar to the git implementation, such that the environment will override configuration values of *config_reader*. If no value is set at all, it will be generated.

Parameters **config_reader** – ConfigReader to use to retrieve the values from in case they are not set in the environment.

conf_email = 'email'

conf_name = 'name'

email

env_author_email = 'GIT_AUTHOR_EMAIL'

env_author_name = 'GIT_AUTHOR_NAME'

env_committer_email = 'GIT_COMMITTER_EMAIL'

env_committer_name = 'GIT_COMMITTER_NAME'

name

name_email_regex = re.compile('(.*) <(.*?)>')

name_only_regex = re.compile('<(.*)>')

class git.util.**BlockingLockFile**(*file_path: Union[str, os.PathLike[str]]*, *check_interval_s: float = 0.3*, *max_block_time_s: int = 9223372036854775807*)
The lock file will block until a lock could be obtained, or fail after a specified timeout.

Note If the directory containing the lock was removed, an exception will be raised during the blocking period, preventing hangs as the lock can never be obtained.

__init__(*file_path: Union[str, os.PathLike[str]]*, *check_interval_s: float = 0.3*, *max_block_time_s: int = 9223372036854775807*) → None
Configure the instance.

Parameters

- **check_interval_s** – Period of time to sleep until the lock is checked the next time. By default, it waits a nearly unlimited time.
- **max_block_time_s** – Maximum amount of seconds we may lock.

__module__ = 'git.util'

__slots__ = ('_check_interval', '_max_block_time')

class git.util.**CallableRemoteProgress**(*fn: Callable*)
A *RemoteProgress* implementation forwarding updates to any callable.

Note Like direct instances of *RemoteProgress*, instances of this *CallableRemoteProgress* class are not themselves directly callable. Rather, instances of this class wrap a callable and forward to it. This should therefore not be confused with *git.types.CallableProgress*.

__annotations__ = {}

__init__(*fn: Callable*) → None

__module__ = 'git.util'

__slots__ = ('_callable',)

update(**args: Any*, ***kwargs: Any*) → None

Called whenever the progress changes.

Parameters

- **op_code** – Integer allowing to be compared against Operation IDs and stage IDs.

  Stage IDs are `BEGIN` and `END`. `BEGIN` will only be set once for each Operation ID as well as `END`. It may be that `BEGIN` and `END` are set at once in case only one progress message was emitted due to the speed of the operation. Between `BEGIN` and `END`, none of these flags will be set.

  Operation IDs are all held within the `OP_MASK`. Only one Operation ID will be active per call.

- **cur_count** – Current absolute count of items.
- **max_count** – The maximum count of items we expect. It may be `None` in case there is no maximum number of items or if it is (yet) unknown.
- **message** – In case of the `WRITING` operation, it contains the amount of bytes transferred. It may possibly be used for other purposes as well.

Note You may read the contents of the current line in `self._cur_line`.

git.util.**HIDE_WINDOWS_KNOWN_ERRORS = False**

We need an easy way to see if Appveyor TCs start failing, so the errors marked with this var are considered "acknowledged" ones, awaiting remedy, till then, we wish to hide them.

class git.util.**IndexFileSHA1Writer**(*f: IO*)

Wrapper around a file-like object that remembers the SHA1 of the data written to it. It will write a sha when the stream is closed or if asked for explicitly using *write_sha()*.

Only useful to the index file.

Note Based on the dulwich project.

__init__(*f: IO*) → None

__module__ = 'git.util'

__slots__ = ('f', 'sha1')

close() → bytes

f

sha1

tell() → int

write(*data: AnyStr*) → int

write_sha() → bytes

class git.util.**IterableList**(*id_attr: str*, *prefix: str = ''*)

List of iterable objects allowing to query an object by id or by named index:

```
heads = repo.heads
heads.master
heads['master']
heads[0]
```

Iterable parent objects:

- Commit
- *Submodule*
- *Reference*
- *FetchInfo*
- *PushInfo*

Iterable via inheritance:

- *Head*
- *TagReference*
- *RemoteReference*

This requires an `id_attribute` name to be set which will be queried from its contained items to have a means for comparison.

A prefix can be specified which is to be used in case the id returned by the items always contains a prefix that does not matter to the user, so it can be left out.

__annotations__ = {}

__contains__(*attr: object*) → bool
Return bool(key in self).

__delitem__(*index: Union[SupportsIndex, int, slice, str]*) → None
Delete self[key].

__getattr__(*attr: str*) → git.util.T_IterableObj

__getitem__(*index: Union[SupportsIndex, int, slice, str]*) → git.util.T_IterableObj
Return self[index].

__init__(*id_attr: str*, *prefix: str = ''*) → None

__module__ = 'git.util'

static __new__(*cls*, *id_attr: str*, *prefix: str = ''*) → *git.util.IterableList*[git.util.T_IterableObj]

__orig_bases__ = (typing.List[+T_IterableObj],)

__parameters__ = (+T_IterableObj,)

__slots__ = ('_id_attr', '_prefix')

class git.util.**IterableObj**(**args*, ***kwargs*)
Defines an interface for iterable items, so there is a uniform way to retrieve and iterate items within the git repository.

Subclasses:

- *Submodule*
- Commit
- *Reference*

- *PushInfo*
- *FetchInfo*
- *Remote*

__abstractmethods__ = frozenset({'iter_items'})

__annotations__ = {'_id_attribute_': <class 'str'>}

__callable_proto_members_only__ = False

__init__(**args*, ***kwargs*)

__module__ = 'git.util'

__parameters__ = ()

__protocol_attrs__ = {'_id_attribute_', 'iter_items', 'list_items'}

__slots__ = ()

classmethod __subclasshook__(*other*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract classmethod iter_items(*repo: Repo*, **args: Any*, ***kwargs: Any*) → Iterator[git.util.T_IterableObj]

Find (all) items of this type.

Subclasses can specify *args* and *kwargs* differently, and may use them for filtering. However, when the method is called with no additional positional or keyword arguments, subclasses are obliged to to yield all items.

Returns Iterator yielding Items

classmethod list_items(*repo: Repo*, **args: Any*, ***kwargs: Any*) → *git.util.IterableList*[git.util.T_IterableObj]

Find (all) items of this type and collect them into a list.

For more information about the arguments, see *iter_items()*.

Note Favor the *iter_items()* method as it will avoid eagerly collecting all items. When there are many items, that can slow performance and increase memory usage.

Returns list(Item,...) list of item instances

class git.util.**LockFile**(*file_path: Union[str, os.PathLike[str]]*)

Provides methods to obtain, check for, and release a file based lock which should be used to handle concurrent access to the same file.

As we are a utility class to be derived from, we only use protected methods.

Locks will automatically be released on destruction.

__annotations__ = {}

__del__() → None

__init__(*file_path: Union[str, os.PathLike[str]]*) → None

__module__ = 'git.util'

__slots__ = ('_file_path', '_owns_lock')

class git.util.**RemoteProgress**

Handler providing an interface to parse progress information emitted by *git-push(1)* and *git-fetch(1)* and to dispatch callbacks allowing subclasses to react to the progress.

BEGIN = 1

CHECKING_OUT = 256

COMPRESSING = 8

COUNTING = 4

DONE_TOKEN = 'done.'

END = 2

FINDING_SOURCES = 128

OP_MASK = -4

RECEIVING = 32

RESOLVING = 64

STAGE_MASK = 3

TOKEN_SEPARATOR = ', '

WRITING = 16

__annotations__ = {'_cur_line': 'Optional[str]', '_num_op_codes': <class 'int'>, '_seen_ops': 'List[int]', 'error_lines': 'List[str]', 'other_lines': 'List[str]'}

__init__() → None

__module__ = 'git.util'

__slots__ = ('_cur_line', '_seen_ops', 'error_lines', 'other_lines')

error_lines: List[str]

line_dropped(*line: str*) → None

Called whenever a line could not be understood and was therefore dropped.

new_message_handler() → Callable[[str], None]

Returns A progress handler suitable for `handle_process_output()`, passing lines on to this progress handler in a suitable format.

other_lines: List[str]

re_op_absolute = re.compile('(remote:)?([\\w\\s]+):\\s+()(\\d+)()(.*)')

re_op_relative = re.compile('(remote:)?([\\w\\s]+):\\s+(\\d+)% \\(\\d+/(\\d+)\\)(.*)')

update(*op_code: int*, *cur_count: Union[str, float]*, *max_count: Optional[Union[str, float]] = None*, *message: str = ''*) → None

Called whenever the progress changes.

Parameters

- **op_code** – Integer allowing to be compared against Operation IDs and stage IDs.

  Stage IDs are *BEGIN* and *END*. *BEGIN* will only be set once for each Operation ID as well as *END*. It may be that *BEGIN* and *END* are set at once in case only one progress

message was emitted due to the speed of the operation. Between *BEGIN* and *END*, none of these flags will be set.

Operation IDs are all held within the *OP_MASK*. Only one Operation ID will be active per call.

- **cur_count** – Current absolute count of items.
- **max_count** – The maximum count of items we expect. It may be None in case there is no maximum number of items or if it is (yet) unknown.
- **message** – In case of the *WRITING* operation, it contains the amount of bytes transferred. It may possibly be used for other purposes as well.

Note You may read the contents of the current line in `self._cur_line`.

class git.util.**Stats**(*total:* git.types.Total_TD, *files: Dict[Union[str, os.PathLike[str]],* git.types.Files_TD*]*)

Represents stat information as presented by git at the end of a merge. It is created from the output of a diff operation.

Example:

```
c = Commit( sha1 )
s = c.stats
s.total         # full-stat-dict
s.files         # dict( filepath : stat-dict )
```

`stat-dict`

A dictionary with the following keys and values:

```
deletions = number of deleted lines as int
insertions = number of inserted lines as int
lines = total number of lines changed as int, or deletions + insertions
```

`full-stat-dict`

In addition to the items in the stat-dict, it features additional information:

```
files = number of changed files as int
```

__init__(*total:* git.types.Total_TD, *files: Dict[Union[str, os.PathLike[str]],* git.types.Files_TD*]*) → None

__module__ = 'git.util'

__slots__ = ('total', 'files')

files

total

git.util.**assure_directory_exists**(*path: Union[str, os.PathLike[str]]*, *is_file: bool = False*) → bool

Make sure that the directory pointed to by path exists.

Parameters **is_file** – If `True`, *path* is assumed to be a file and handled correctly. Otherwise it must be a directory.

Returns `True` if the directory was created, `False` if it already existed.

git.util.**get_user_id**() → str

Returns String identifying the currently active system user as `name@node`

git.util.**join_path**(*a: Union[str, os.PathLike[str]]*, **p: Union[str, os.PathLike[str]]*) → Union[str, os.PathLike[str]]

Join path tokens together similar to osp.join, but always use / instead of possibly \ on Windows.

git.util.**join_path_native**(*a: Union[str, os.PathLike[str]]*, **p: Union[str, os.PathLike[str]]*) → Union[str, os.PathLike[str]]

Like *join_path()*, but makes sure an OS native path is returned.

This is only needed to play it safe on Windows and to ensure nice paths that only use \.

git.util.**rmtree**(*path: Union[str, os.PathLike[str]]*) → None

Remove the given directory tree recursively.

Note We use `shutil.rmtree()` but adjust its behaviour to see whether files that couldn't be deleted are read-only. Windows will not remove them in that case.

git.util.**stream_copy**(*source: BinaryIO*, *destination: BinaryIO*, *chunk_size: int = 524288*) → int

Copy all data from the *source* stream into the *destination* stream in chunks of size *chunk_size*.

Returns Number of bytes written

git.util.**to_native_path_linux**(*path: Union[str, os.PathLike[str]]*) → str

git.util.**unbare_repo**(*func: Callable[[...], git.util.T]*) → Callable[[...], git.util.T]

Methods with this decorator raise *InvalidGitRepositoryError* if they encounter a bare repository.

CHAPTER

FIVE

ROADMAP

The full list of milestones including associated tasks can be found on GitHub: https://github.com/gitpython-developers/GitPython/issues

Select the respective milestone to filter the list of issues accordingly.

CHAPTER

SIX

CHANGELOG

6.1 3.1.43

A major visible change will be the added deprecation- or user-warnings, and greatly improved typing.

See the following for all changes. https://github.com/gitpython-developers/GitPython/releases/tag/3.1.43

6.2 3.1.42

See the following for all changes. https://github.com/gitpython-developers/GitPython/releases/tag/3.1.42

6.3 3.1.41

This release is relevant for security as it fixes a possible arbitary code execution on Windows.

See this PR for details: https://github.com/gitpython-developers/GitPython/pull/1792 An advisory is available soon at: https://github.com/gitpython-developers/GitPython/security/advisories/GHSA-2mqj-m65w-jghx

See the following for all changes. https://github.com/gitpython-developers/GitPython/releases/tag/3.1.41

6.4 3.1.40

See the following for all changes. https://github.com/gitpython-developers/GitPython/releases/tag/3.1.40

6.5 3.1.38

See the following for all changes. https://github.com/gitpython-developers/GitPython/releases/tag/3.1.38

6.6 3.1.37

This release contains another security fix that further improves validation of symbolic references and thus properly fixes this CVE: https://github.com/advisories/GHSA-cwvm-v4w8-q58c .

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/67?closed=1

6.7 3.1.36

Note that this release should be a no-op, it's mainly for testing the changed release-process.

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/66?closed=1

6.8 3.1.35

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/65?closed=1

6.9 3.1.34

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/64?closed=1

6.10 3.1.33

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/63?closed=1

6.11 3.1.32

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/62?closed=1

6.12 3.1.31

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/61?closed=1

6.13 3.1.30

- Make injections of command-invocations harder or impossible for clone and others. See https://github.com/gitpython-developers/GitPython/pull/1518 for details. Note that this might constitute a breaking change for some users, and if so please let us know and we add an opt-out to this.
- Prohibit insecure options and protocols by default, which is potentially a breaking change, but a necessary fix for https://github.com/gitpython-developers/GitPython/issues/1515. Please take a look at the PR for more information and how to bypass these protections in case they cause breakage: https://github.com/gitpython-developers/GitPython/pull/1521.

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/60?closed=1

6.14 3.1.29

- Make the git.__version__ re-appear.

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/59?closed=1

6.15 3.1.28

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/58?closed=1

6.16 3.1.27

- Reduced startup time due to optimized imports.
- Fix a vulenerability that could cause great slowdowns when encountering long remote path names when pulling/fetching.

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/57?closed=1

6.17 3.1.26

- Fixes a leaked file descriptor when reading the index, which would cause make writing a previously read index on windows impossible. See https://github.com/gitpython-developers/GitPython/issues/1395 for details.

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/56?closed=1

6.18 3.1.25

See the following for all changes. https://github.com/gitpython-developers/gitpython/milestone/55?closed=1

6.19 3.1.24

- Newly added timeout flag is not be enabled by default, and was renamed to kill_after_timeout

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/54?closed=1 https://github.com/gitpython-developers/gitpython/milestone/53?closed=1

6.20 3.1.23 (YANKED)

- This is the second typed release with a lot of improvements under the hood.
- General:
  - Remove python 3.6 support
  - Remove distutils ahead of deprecation in standard library.
  - Update sphinx to 4.1.12 and use autodoc-typehints.
  - Include README as long_description on PyPI
  - Test against earliest and latest minor version available on Github Actions (e.g. 3.9.0 and 3.9.7)
- Typing:
  - Add types to ALL functions.
  - Ensure py.typed is collected.
  - Increase mypy strictness with disallow_untyped_defs, warn_redundant_casts, warn_unreachable.
  - Use typing.NamedTuple and typing.OrderedDict now 3.6 dropped.
  - Make Protocol classes ABCs at runtime due to new behaviour/bug in 3.9.7 & 3.10.0-rc1
  - Remove use of typing.TypeGuard until later release, to allow dependent libs time to update.
  - Tracking issue: https://github.com/gitpython-developers/GitPython/issues/1095
- Runtime improvements:
  - Add clone_multi_options support to submodule.add()
  - Delay calling get_user_id() unless essential, to support sand-boxed environments.
  - Add timeout to handle_process_output(), in case thread.join() hangs.

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/53?closed=1

6.21 3.1.20 (YANKED)

- This is the second typed release with a lot of improvements under the hood. * Tracking issue: https://github.com/gitpython-developers/GitPython/issues/1095

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/52?closed=1

6.22 3.1.19 (YANKED)

- This is the second typed release with a lot of improvements under the hood. * Tracking issue: https://github.com/gitpython-developers/GitPython/issues/1095

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/51?closed=1

6.23 3.1.18

- drop support for python 3.5 to reduce maintenance burden on typing. Lower patch levels of python 3.5 would break, too.

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/50?closed=1

6.24 3.1.17

- Fix issues from 3.1.16 (see https://github.com/gitpython-developers/GitPython/issues/1238)
- Fix issues from 3.1.15 (see https://github.com/gitpython-developers/GitPython/issues/1223)
- Add more static typing information

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/49?closed=1

6.25 3.1.16 (YANKED)

- Fix issues from 3.1.15 (see https://github.com/gitpython-developers/GitPython/issues/1223)
- Add more static typing information

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/48?closed=1

6.26 3.1.15 (YANKED)

- add deprecation warning for python 3.5

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/47?closed=1

6.27 3.1.14

- git.Commit objects now have a `replace` method that will return a copy of the commit with modified attributes.
- Add python 3.9 support
- Drop python 3.4 support

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/46?closed=1

6.28 3.1.13

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/45?closed=1

6.29 3.1.12

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/44?closed=1

6.30 3.1.11

Fixes regression of 3.1.10.

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/43?closed=1

6.31 3.1.10

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/42?closed=1

6.32 3.1.9

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/41?closed=1

6.33 3.1.8

- support for 'includeIf' in git configuration files
- tests are now excluded from the package, making it conisderably smaller

See the following for more details: https://github.com/gitpython-developers/gitpython/milestone/40?closed=1

6.34 3.1.7

- Fix tutorial examples, which disappeared in 3.1.6 due to a missed path change.

6.35 3.1.6

- Greatly reduced package size, see https://github.com/gitpython-developers/GitPython/pull/1031

6.36 3.1.5

- rollback: package size was reduced significantly not placing tests into the package anymore. See https://github.com/gitpython-developers/GitPython/issues/1030

6.37 3.1.4

- all exceptions now keep track of their cause
- package size was reduced significantly not placing tests into the package anymore.

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/39?closed=1

6.38 3.1.3

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/38?closed=1

6.39 3.1.2

- Re-release of 3.1.1, with known signature

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/37?closed=1

6.40 3.1.1

- support for PyOxidizer, which previously failed due to usage of *__file__*.

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/36?closed=1

6.41 3.1.0

- Switched back to using gitdb package as requirement (gitdb#59)

6.42 3.0.9

- Restricted GitDB (gitdb2) version requirement to < 4
- Removed old nose library from test requirements

6.42.1 Bugfixes

- Changed to use UTF-8 instead of default encoding when getting information about a symbolic reference (#774)
- Fixed decoding of tag object message so as to replace invalid bytes (#943)

6.43 3.0.8

- Added support for Python 3.8
- Bumped GitDB (gitdb2) version requirement to > 3

6.43.1 Bugfixes

- Fixed Repo.__repr__ when subclassed (#968)
- Removed compatibility shims for Python < 3.4 and old mock library
- Replaced usage of deprecated unittest aliases and Logger.warn
- Removed old, no longer used assert methods
- Replaced usage of nose assert methods with unittest

6.44 3.0.7

Properly signed re-release of v3.0.6 with new signature (See #980)

6.45 3.0.6

Note: There was an issue that caused this version to be released to PyPI without a signature
See the changelog for v3.0.7 and #980

6.45.1 Bugfixes

- Fixed warning for usage of environment variables for paths containing $ or % (#832, #961)
- Added support for parsing Git internal date format (@<unix timestamp> <timezone offset>) (#965)
- Removed Python 2 and < 3.3 compatibility shims (#979)
- Fixed GitDB (gitdb2) requirement version specifier formatting in requirements.txt (#979)

6.46 3.0.5 - Bugfixes

see the following for details: https://github.com/gitpython-developers/gitpython/milestone/32?closed=1

6.47 3.0.4 - Bugfixes

see the following for details: https://github.com/gitpython-developers/gitpython/milestone/31?closed=1

6.48 3.0.3 - Bugfixes

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/30?closed=1

6.49 3.0.2 - Bugfixes

- fixes an issue with installation

6.50 3.0.1 - Bugfixes and performance improvements

- Fix a performance regression which could make certain workloads 50% slower
- Add *currently_rebasing_on* method on *Repo*, see the PR
- Fix incorrect *requirements.txt* which could lead to broken installations, see this issue for details.

6.51 3.0.0 - Remove Python 2 support

Motivation for this is a patch which improves unicode handling when dealing with filesystem paths. Python 2 compatibility was introduced to deal with differences, and I thought it would be a good idea to 'just' drop support right now, mere 5 months away from the official maintenance stop of python 2.7.

The underlying motivation clearly is my anger when thinking python and unicode, which was a hassle from the start, at least in a codebase as old as GitPython, which totally doesn't handle encodings correctly in many cases.

Having migrated to using *Rust* exclusively for tooling, I still see that correct handling of encodings isn't entirely trivial, but at least *Rust* makes clear what has to be done at compile time, allowing to write software that is pretty much guaranteed to work once it compiles.

Again, my apologies if removing Python 2 support caused inconveniences, please see release 2.1.13 which returns it.

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/27?closed=1

or run have a look at the difference between tags v2.1.12 and v3.0.0: https://github.com/gitpython-developers/GitPython/compare/2.1.12...3.0.0.

6.52 2.1.15

- Fixed GitDB (gitdb2) requirement version specifier formatting in requirements.txt (Backported from #979)
- Restricted GitDB (gitdb2) version requirement to < 3 (#897)

6.53 2.1.14

- Fixed handling of 0 when transforming kwargs into Git command arguments (Backported from #899)

6.54 2.1.13 - Bring back Python 2.7 support

My apologies for any inconvenience this may have caused. Following semver, backward incompatible changes will be introduced in a minor version.

6.55 2.1.12 - Bugfixes and Features

- Multi-value support and interface improvements for Git configuration. Thanks to A. Jesse Jiryu Davis.

or run have a look at the difference between tags v2.1.11 and v2.1.12: https://github.com/gitpython-developers/GitPython/compare/2.1.11...2.1.12

6.56 2.1.11 - Bugfixes

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/26?closed=1

or run have a look at the difference between tags v2.1.10 and v2.1.11: https://github.com/gitpython-developers/GitPython/compare/2.1.10...2.1.11

6.57 2.1.10 - Bugfixes

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/25?closed=1

or run have a look at the difference between tags v2.1.9 and v2.1.10: https://github.com/gitpython-developers/GitPython/compare/2.1.9...2.1.10

6.58 2.1.9 - Dropping support for Python 2.6

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/24?closed=1

or run have a look at the difference between tags v2.1.8 and v2.1.9: https://github.com/gitpython-developers/GitPython/compare/2.1.8...2.1.9

6.59 2.1.8 - bugfixes

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/23?closed=1

or run have a look at the difference between tags v2.1.7 and v2.1.8: https://github.com/gitpython-developers/GitPython/compare/2.1.7...2.1.8

6.60 2.1.6 - bugfixes

- support for worktrees

6.61 2.1.3 - Bugfixes

All issues and PRs can be viewed in all detail when following this URL: https://github.com/gitpython-developers/GitPython/milestone/21?closed=1

6.62 2.1.1 - Bugfixes

All issues and PRs can be viewed in all detail when following this URL: https://github.com/gitpython-developers/GitPython/issues?q=is%3Aclosed+milestone%3A%22v2.1.1+-+Bugfixes%22

6.63 2.1.0 - Much better windows support!

Special thanks to @ankostis, who made this release possible (nearly) single-handedly. GitPython is run by its users, and their PRs make all the difference, they keep GitPython relevant. Thank you all so much for contributing !

6.63.1 Notable fixes

- The *GIT_DIR* environment variable does not override the *path* argument when initializing a *Repo* object anymore. However, if said *path* unset, *GIT_DIR* will be used to fill the void.

All issues and PRs can be viewed in all detail when following this URL: https://github.com/gitpython-developers/GitPython/issues?q=is%3Aclosed+milestone%3A%22v2.1.0+-+proper+windows+support%22

6.64 2.0.9 - Bugfixes

- *tag.commit* will now resolve commits deeply.
- *Repo* objects can now be pickled, which helps with multi-processing.
- *Head.checkout()* now deals with detached heads, which is when it will return the *HEAD* reference instead.
- *DiffIndex.iter_change_type(...)* produces better results when diffing

6.65 2.0.8 - Features and Bugfixes

- *DiffIndex.iter_change_type(…)* produces better results when diffing an index against the working tree.
- *Repo().is_dirty(…)* now supports the *path* parameter, to specify a single path by which to filter the output. Similar to *git status <path>*
- Symbolic refs created by this library will now be written with a newline character, which was previously missing.
- *blame()* now properly preserves multi-line commit messages.
- No longer corrupt ref-logs by writing multi-line comments into them.

6.66 2.0.7 - New Features

- ***IndexFile.commit(…,skip_hooks=False)* added. This parameter emulates the** behaviour of *–no-verify* on the command-line.

6.67 2.0.6 - Fixes and Features

- Fix: remote output parser now correctly matches refs with non-ASCII chars in them
- API: Diffs now have *a_rawpath*, *b_rawpath*, *raw_rename_from*, *raw_rename_to* properties, which are the raw-bytes equivalents of their unicode path counterparts.
- Fix: TypeError about passing keyword argument to string decode() on Python 2.6.
- Feature: setUrl API on Remotes

6.68 2.0.5 - Fixes

- Fix: parser of fetch info lines choked on some legitimate lines

6.69 2.0.4 - Fixes

- Fix: parser of commit object data is now robust against cases where commit object contains invalid bytes. The invalid characters are now replaced rather than choked on.
- Fix: non-ASCII paths are now properly decoded and returned in `.diff()` output
- Fix: *RemoteProgress* will now strip the ‘, ‘ prefix or suffix from messages.
- API: Remote.[fetch|push|pull](…) methods now allow the `progress` argument to be a callable. This saves you from creating a custom type with usually just one implemented method.

6.70 2.0.3 - Fixes

- Fix: bug in `git-blame --incremental` output parser that broken when commit messages contained `\r` characters
- Fix: progress handler exceptions are not caught anymore, which would usually just hide bugs previously.
- Fix: The *Git.execute* method will now redirect *stdout* to *devnull* if *with_stdout* is false, which is the intended behaviour based on the parameter's documentation.

6.71 2.0.2 - Fixes

- Fix: source package does not include *.pyc files
- Fix: source package does include doc sources

6.72 2.0.1 - Fixes

- Fix: remote output parser now correctly matches refs with "@" in them

6.73 2.0.0 - Features

Please note that due to breaking changes, we have to increase the major version.

- **IMPORTANT**: This release drops support for python 2.6, which is officially deprecated by the python maintainers.
- **CRITICAL**: *Diff* objects created with patch output will now not carry the — and +++ header lines anymore. All diffs now start with the @@ header line directly. Users that rely on the old behaviour can now (reliably) read this information from the a_path and b_path properties without having to parse these lines manually.
- *Commit* now has extra properties *authored_datetime* and *committer_datetime* (to get Python datetime instances rather than timestamps)
- *Commit.diff()* now supports diffing the root commit via *Commit.diff(NULL_TREE)*.
- *Repo.blame()* now respects *incremental=True*, supporting incremental blames. Incremental blames are slightly faster since they don't include the file's contents in them.
- Fix: *Diff* objects created with patch output will now have their *a_path* and *b_path* properties parsed out correctly. Previously, some values may have been populated incorrectly when a file was added or deleted.
- Fix: diff parsing issues with paths that contain "unsafe" chars, like spaces, tabs, backslashes, etc.

6.74 1.0.2 - Fixes

- IMPORTANT: Changed default object database of *Repo* objects to *GitCmdObjectDB*. The pure-python implementation used previously usually fails to release its resources (i.e. file handles), which can lead to problems when working with large repositories.
- CRITICAL: fixed incorrect *Commit* object serialization when authored or commit date had timezones which were not divisiblej by 3600 seconds. This would happen if the timezone was something like *+0530* for instance.
- A list of all additional fixes can be found on GitHub
- CRITICAL: *Tree.cache* was removed without replacement. It is technically impossible to change individual trees and expect their serialization results to be consistent with what *git* expects. Instead, use the *IndexFile* facilities to adjust the content of the staging area, and write it out to the respective tree objects using *IndexFile.write_tree()* instead.

6.75 1.0.1 - Fixes

- A list of all issues can be found on GitHub

6.76 1.0.0 - Notes

This version is equivalent to v0.3.7, but finally acknowledges that GitPython is stable and production ready.

It follows the semantic version scheme, and thus will not break its existing API unless it goes 2.0.

6.77 0.3.7 - Fixes

- *IndexFile.add()* will now write the index without any extension data by default. However, you may override this behaviour with the new *write_extension_data* keyword argument.
  - Renamed *ignore_tree_extension_data* keyword argument in *IndexFile.write(...)* to *ignore_extension_data*
- If the git command executed during *Remote.push(...)|fetch(...)* returns with an non-zero exit code and GitPython didn't obtain any head-information, the corresponding *GitCommandError* will be raised. This may break previous code which expected these operations to never raise. However, that behaviour is undesirable as it would effectively hide the fact that there was an error. See this issue for more information.
- If the git executable can't be found in the PATH or at the path provided by *GIT_PYTHON_GIT_EXECUTABLE*, this is made obvious by throwing *GitCommandNotFound*, both on unix and on windows.
  - Those who support **GUI on windows** will now have to set *git.Git.USE_SHELL = True* to get the previous behaviour.
- A list of all issues can be found on GitHub

6.78 0.3.6 - Features

- **DOCS**
  - special members like *__init__* are now listed in the API documentation
  - tutorial section was revised entirely, more advanced examples were added.
- **POSSIBLY BREAKING CHANGES**
  - As *rev_parse* will now throw *BadName* as well as *BadObject*, client code will have to catch both exception types.
  - Repo.working_tree_dir now returns None if it is bare. Previously it raised AssertionError.
  - IndexFile.add() previously raised AssertionError when paths where used with bare repository, now it raises InvalidGitRepositoryError
- Added *Repo.merge_base()* implementation. See the respective issue on GitHub
- *[include]* sections in git configuration files are now respected
- Added *GitConfigParser.rename_section()*
- Added *Submodule.rename()*
- A list of all issues can be found on GitHub

6.79 0.3.5 - Bugfixes

- push/pull/fetch operations will not block anymore
- diff() can now properly detect renames, both in patch and raw format. Previously it only worked when create_patch was True.
- repo.odb.update_cache() is now called automatically after fetch and pull operations. In case you did that in your own code, you might want to remove your line to prevent a double-update that causes unnecessary IO.
- *Repo(path)* will not automatically search upstream anymore and find any git directory on its way up. If you need that behaviour, you can turn it back on using the new *search_parent_directories=True* flag when constructing a *Repo* object.
- IndexFile.commit() now runs the *pre-commit* and *post-commit* hooks. Verified to be working on posix systems only.
- A list of all fixed issues can be found here: https://github.com/gitpython-developers/GitPython/issues?q=milestone%3A%22v0.3.5+-+bugfixes%22+

6.80 0.3.4 - Python 3 Support

- Internally, hexadecimal SHA1 are treated as ascii encoded strings. Binary SHA1 are treated as bytes.
- Id attribute of Commit objects is now *hexsha*, instead of *binsha*. The latter makes no sense in python 3 and I see no application of it anyway besides its artificial usage in test cases.
- **IMPORTANT**: If you were using the config_writer(), you implicitly relied on __del__ to work as expected to flush changes. To be sure changes are flushed under PY3, you will have to call the new *release()* method to trigger a flush. For some reason, __del__ is not called necessarily anymore when a symbol goes out of scope.

- The *Tree* now has a *.join('name')* method which is equivalent to *tree / 'name'*

6.81 0.3.3

- When fetching, pulling or pushing, and an error occurs, it will not be reported on stdout anymore. However, if there is a fatal error, it will still result in a GitCommandError to be thrown. This goes hand in hand with improved fetch result parsing.
- Code Cleanup (in preparation for python 3 support)
  - Applied autopep8 and cleaned up code
  - Using python logging module instead of print statements to signal certain kinds of errors

6.82 0.3.2.1

- Fix for #207

6.83 0.3.2

- Release of most recent version as non-RC build, just to allow pip to install the latest version right away.
- Have a look at the milestones (https://github.com/gitpython-developers/GitPython/milestones) to see what's next.

6.84 0.3.2 RC1

- **git** command wrapper
- Added `version_info` property which returns a tuple of integers representing the installed git version.
- Added GIT_PYTHON_GIT_EXECUTABLE environment variable, which can be used to set the desired git executable to be used. despite of what would be found in the path.
- **Blob** Type
- Added mode constants to ease the manual creation of blobs
- **IterableList**
- Added __contains__ and __delitem__ methods
- **More Changes**
- Configuration file parsing is more robust. It should now be able to handle everything that the git command can parse as well.
- The progress parsing was updated to support git 1.7.0.3 and newer. Previously progress was not enabled for the git command or only worked with ssh in case of older git versions.
- Parsing of tags was improved. Previously some parts of the name could not be parsed properly.
- The rev-parse pure python implementation now handles branches correctly if they look like hexadecimal sha's.
- GIT_PYTHON_TRACE is now set on class level of the Git type, previously it was a module level global variable.

- GIT_PYTHON_GIT_EXECUTABLE is a class level variable as well.

6.85 0.3.1 Beta 2

- Added **reflog support** (reading and writing)
  - New types: `RefLog` and `RefLogEntry`
  - Reflog is maintained automatically when creating references and deleting them
  - Non-intrusive changes to `SymbolicReference`, these don't require your code to change. They allow to append messages to the reflog.
    - `abspath` property added, similar to `abspath` of Object instances
    - `log()` method added
    - `log_append(...)` method added
    - `set_reference(...)` method added (reflog support)
    - `set_commit(...)` method added (reflog support)
    - `set_object(...)` method added (reflog support)
  - **Intrusive Changes** to `Head` type
  - `create(...)` method now supports the reflog, but will not raise `GitCommandError` anymore as it is a pure python implementation now. Instead, it raises `OSError`.
  - **Intrusive Changes** to `Repo` type
  - `create_head(...)` method does not support kwargs anymore, instead it supports a logmsg parameter
- Repo.rev_parse now supports the [ref]@{n} syntax, where *n* is the number of steps to look into the reference's past
- **BugFixes**
  - Removed incorrect ORIG_HEAD handling
- **Flattened directory** structure to make development more convenient.
- ---

  Note: This alters the way projects using git-python as a submodule have to adjust their sys.path to be able to import git-python successfully.

- Misc smaller changes and bugfixes

6.86 0.3.1 Beta 1

- Full Submodule-Support
- Added unicode support for author names. Commit.author.name is now unicode instead of string.
- Head Type changes
- config_reader() & config_writer() methods added for access to head specific options.
- tracking_branch() & set_tracking_branch() methods added for easy configuration of tracking branches.

6.87 0.3.0 Beta 2

- Added python 2.4 support

6.88 0.3.0 Beta 1

6.88.1 Renamed Modules

- For consistency with naming conventions used in sub-modules like gitdb, the following modules have been renamed
  - git.utils -> git.util
  - git.errors -> git.exc
  - git.objects.utils -> git.objects.util

6.88.2 General

- Object instances, and everything derived from it, now use binary sha's internally. The 'sha' member was removed, in favor of the 'binsha' member. An 'hexsha' property is available for convenient conversions. They may only be initialized using their binary shas, reference names or revision specs are not allowed anymore.
- IndexEntry instances contained in IndexFile.entries now use binary sha's. Use the .hexsha property to obtain the hexadecimal version. The .sha property was removed to make the use of the respective sha more explicit.
- If objects are instantiated explicitly, a binary sha is required to identify the object, where previously any rev-spec could be used. The ref-spec compatible version still exists as Object.new or Repo.commit|Repo.tree respectively.
- The .data attribute was removed from the Object type, to obtain plain data, use the data_stream property instead.
- ConcurrentWriteOperation was removed, and replaced by LockedFD
- IndexFile.get_entries_key was renamed to entry_key
- IndexFile.write_tree: removed missing_ok keyword, its always True now. Instead of raising GitCommandError it raises UnmergedEntriesError. This is required as the pure-python implementation doesn't support the missing_ok keyword yet.
- diff.Diff.null_hex_sha renamed to NULL_HEX_SHA, to be conforming with the naming in the Object base class

6.89 0.2 Beta 2

- Commit objects now carry the 'encoding' information of their message. It wasn't parsed previously, and defaults to UTF-8
- Commit.create_from_tree now uses a pure-python implementation, mimicking git-commit-tree

6.90 0.2

6.90.1 General

- file mode in Tree, Blob and Diff objects now is an int compatible to definitions in the stat module, allowing you to query whether individual user, group and other read, write and execute bits are set.
- Adjusted class hierarchy to generally allow comparison and hash for Objects and Refs
- Improved Tag object which now is a Ref that may contain a tag object with additional Information
- id_abbrev method has been removed as it could not assure the returned short SHA's where unique
- removed basename method from Objects with path's as it replicated features of os.path
- from_string and list_from_string methods are now private and were renamed to _from_string and _list_from_string respectively. As part of the private API, they may change without prior notice.
- Renamed all find_all methods to list_items - this method is part of the Iterable interface that also provides a more efficients and more responsive iter_items method
- All dates, like authored_date and committer_date, are stored as seconds since epoch to consume less memory - they can be converted using time.gmtime in a more suitable presentation format if needed.
- Named method parameters changed on a wide scale to unify their use. Now git specific terms are used everywhere, such as "Reference" (ref) and "Revision" (rev). Previously multiple terms where used making it harder to know which type was allowed or not.
- Unified diff interface to allow easy diffing between trees, trees and index, trees and working tree, index and working tree, trees and index. This closely follows the git-diff capabilities.
- Git.execute does not take the with_raw_output option anymore. It was not used by anyone within the project and False by default.

6.90.2 Item Iteration

- Previously one would return and process multiple items as list only which can hurt performance and memory consumption and reduce response times. iter_items method provide an iterator that will return items on demand as parsed from a stream. This way any amount of objects can be handled.
- list_items method returns IterableList allowing to access list members by name

6.90.3 objects Package

- blob, tree, tag and commit module have been moved to new objects package. This should not affect you though unless you explicitly imported individual objects. If you just used the git package, names did not change.

6.90.4 Blob

- former 'name' member renamed to path as it suits the actual data better

6.90.5 GitCommand

- git.subcommand call scheme now prunes out None from the argument list, allowing to be called more comfortably as None can never be a valid to the git command if converted to a string.
- Renamed 'git_dir' attribute to 'working_dir' which is exactly how it is used

6.90.6 Commit

- 'count' method is not an instance method to increase its ease of use
- 'name_rev' property returns a nice name for the commit's sha

6.90.7 Config

- The git configuration can now be read and manipulated directly from within python using the GitConfigParser
- Repo.config_reader() returns a read-only parser
- Repo.config_writer() returns a read-write parser

6.90.8 Diff

- Members a a_commit and b_commit renamed to a_blob and b_blob - they are populated with Blob objects if possible
- Members a_path and b_path removed as this information is kept in the blobs
- Diffs are now returned as DiffIndex allowing to more quickly find the kind of diffs you are interested in

6.90.9 Diffing

- Commit and Tree objects now support diffing natively with a common interface to compare against other Commits or Trees, against the working tree or against the index.

6.90.10 Index

- A new Index class allows to read and write index files directly, and to perform simple two and three way merges based on an arbitrary index.

6.90.11 References

- References are object that point to a Commit
- SymbolicReference are a pointer to a Reference Object, which itself points to a specific Commit
- They will dynamically retrieve their object at the time of query to assure the information is actual. Recently objects would be cached, hence ref object not be safely kept persistent.

6.90.12 Repo

- Moved blame method from Blob to repo as it appeared to belong there much more.
- active_branch method now returns a Head object instead of a string with the name of the active branch.
- tree method now requires a Ref instance as input and defaults to the active_branch instead of master
- is_dirty now takes additional arguments allowing fine-grained control about what is considered dirty
- Removed the following methods:
  - 'log' method as it as effectively the same as the 'commits' method
  - 'commits_since' as it is just a flag given to rev-list in Commit.iter_items
  - 'commit_count' as it was just a redirection to the respective commit method
  - 'commits_between', replaced by a note on the iter_commits method as it can achieve the same thing
  - 'commit_delta_from' as it was a very special case by comparing two different repjrelated repositories, i.e. clones, git-rev-list would be sufficient to find commits that would need to be transferred for example.
  - 'create' method which equals the 'init' method's functionality
  - 'diff' - it returned a mere string which still had to be parsed
  - 'commit_diff' - moved to Commit, Tree and Diff types respectively
- Renamed the following methods:
  - commits to iter_commits to improve the performance, adjusted signature
  - init_bare to init, implying less about the options to be used
  - fork_bare to clone, as it was to represent general clone functionality, but implied a bare clone to be more versatile
  - archive_tar_gz and archive_tar and replaced by archive method with different signature
- 'commits' method has no max-count of returned commits anymore, it now behaves like git-rev-list
- The following methods and properties were added
  - 'untracked_files' property, returning all currently untracked files
  - 'head', creates a head object
  - 'tag', creates a tag object
  - 'iter_trees' method
  - 'config_reader' method
  - 'config_writer' method
  - 'bare' property, previously it was a simple attribute that could be written

- Renamed the following attributes
  - 'path' is now 'git_dir'
  - 'wd' is now 'working_dir'
- Added attribute
  - 'working_tree_dir' which may be None in case of bare repositories

6.90.13 Remote

- Added Remote object allowing easy access to remotes
- Repo.remotes lists all remotes
- Repo.remote returns a remote of the specified name if it exists

6.90.14 Test Framework

- Added support for common TestCase base class that provides additional functionality to receive repositories tests can also write to. This way, more aspects can be tested under real-world (un-mocked) conditions.

6.90.15 Tree

- former 'name' member renamed to path as it suits the actual data better
- added traverse method allowing to recursively traverse tree items
- deleted blob method
- added blobs and trees properties allowing to query the respective items in the tree
- now mimics behaviour of a read-only list instead of a dict to maintain order.
- content_from_string method is now private and not part of the public API anymore

6.91 0.1.6

6.91.1 General

- Added in Sphinx documentation.
- Removed ambiguity between paths and treeishs. When calling commands that accept treeish and path arguments and there is a path with the same name as a treeish git cowardly refuses to pick one and asks for the command to use the unambiguous syntax where '–' separates the treeish from the paths.
- `Repo.commits`, `Repo.commits_between`, `Repo.commits_since`, `Repo.commit_count`, `Repo.commit`, `Commit.count` and `Commit.find_all` all now optionally take a path argument which constrains the lookup by path. This changes the order of the positional arguments in `Repo.commits` and `Repo.commits_since`.

6.91.2 Commit

- `Commit.message` now contains the full commit message (rather than just the first line) and a new property `Commit.summary` contains the first line of the commit message.
- Fixed a failure when trying to lookup the stats of a parentless commit from a bare repo.

6.91.3 Diff

- The diff parser is now far faster and also addresses a bug where sometimes b_mode was not set.
- Added support for parsing rename info to the diff parser. Addition of new properties `Diff.renamed`, `Diff.rename_from`, and `Diff.rename_to`.

6.91.4 Head

- Corrected problem where branches was only returning the last path component instead of the entire path component following refs/heads/.

6.91.5 Repo

- Modified the gzip archive creation to use the python gzip module.
- Corrected `commits_between` always returning None instead of the reversed list.

6.92 0.1.5

6.92.1 General

- upgraded to Mock 0.4 dependency.
- Replace GitPython with git in repr() outputs.
- Fixed packaging issue caused by ez_setup.py.

6.92.2 Blob

- No longer strip newlines from Blob data.

6.92.3 Commit

- Corrected problem with git-rev-list –bisect-all. See http://groups.google.com/group/git-python/browse_thread/thread/aed1d5c4b31d5027

6.92.4 Repo

- Corrected problems with creating bare repositories.
- Repo.tree no longer accepts a path argument. Use:

```
>>> dict(k, o for k, o in tree.items() if k in paths)
```

- Made daemon export a property of Repo. Now you can do this:

```
>>> exported = repo.daemon_export
>>> repo.daemon_export = True
```

- Allows modifying the project description. Do this:

```
>>> repo.description = "Foo Bar"
>>> repo.description
'Foo Bar'
```

- Added a read-only property Repo.is_dirty which reflects the status of the working directory.
- Added a read-only Repo.active_branch property which returns the name of the currently active branch.

6.92.5 Tree

- Switched to using a dictionary for Tree contents since you will usually want to access them by name and order is unimportant.
- Implemented a dictionary protocol for Tree objects. The following:

  child = tree.contents[‘grit’]

  becomes:

  child = tree[‘grit’]

- Made Tree.content_from_string a static method.

6.93 0.1.4.1

- removed `method_missing` stuff and replaced with a `__getattr__` override in `Git`.

6.94 0.1.4

- renamed `git_python` to `git`. Be sure to delete all pyc files before testing.

6.94.1 Commit

- Fixed problem with commit stats not working under all conditions.

6.94.2 Git

- Renamed module to cmd.
- Removed shell escaping completely.
- Added support for `stderr`, `stdin`, and `with_status`.
- `git_dir` is now optional in the constructor for `git.Git`. Git now falls back to `os.getcwd()` when git_dir is not specified.
- add a `with_exceptions` keyword argument to git commands. `GitCommandError` is raised when the exit status is non-zero.
- add support for a `GIT_PYTHON_TRACE` environment variable. `GIT_PYTHON_TRACE` allows us to debug GitPython's usage of git through the use of an environment variable.

6.94.3 Tree

- Fixed up problem where `name` doesn't exist on root of tree.

6.94.4 Repo

- Corrected problem with creating bare repo. Added `Repo.create` alias.

6.95 0.1.2

6.95.1 Tree

- Corrected problem with `Tree.__div__` not working with zero length files. Removed `__len__` override and replaced with size instead. Also made size cache properly. This is a breaking change.

6.96 0.1.1

Fixed up some urls because I'm a moron

6.97 0.1.0

initial release

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W