GitPython Documentation

Release 1.0.2

Michael Trier

March 15, 2016

Contents

CHAPTER 1

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

- **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
- Python Nose - used for running the tests
- Mock by Michael Foord used for tests. Requires version 0.5

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 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*.

1.4 API Reference

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

1.5 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 the nose powered unit tests:

```
$ nosetests
```

1.6 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

1.7 Issue Tracker

The issue tracker is hosted by github:

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

1.8 License Information

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

CHAPTER 2

Whats New in 0.3

GitPython 0.3 is the first step in creating a hybrid which uses a pure python implementations for all simple git features which can be implemented without significant performance penalties. Everything else is still performed using the git command, which is nicely integrated and easy to use.

Its biggest strength, being the support for all git features through the git command itself, is a weakness as well considering the possibly vast amount of times the git command is being started up. Depending on the actual command being performed, the git repository will be initialized on many of these invocations, causing additional overhead for possibly tiny operations.

Keeping as many major operations in the python world will result in improved caching benefits as certain data structures just have to be initialized once and can be reused multiple times. This mode of operation may improve performance when altering the git database on a low level, and is clearly beneficial on operating systems where command invocations are very slow.

2.1 Object Databases

An object database provides a simple interface to query object information or to write new object data. Objects are generally identified by their 20 byte binary sha1 value during query.

GitPython uses the `gitdb` project to provide a pure-python implementation of the git database, which includes reading and writing loose objects, reading pack files and handling alternate repositories.

The great thing about this is that `Repo` objects can use any object database, hence it easily supports different implementations with different performance characteristics. If you are thinking in extremes, you can implement your own database representation, which may be more efficient for what you want to do specifically, like handling big files more efficiently.

2.2 Reduced Memory Footprint

Objects, such as commits, tags, trees and blobs now use 20 byte sha1 signatures internally, reducing their memory demands by 20 bytes per object, allowing you to keep more objects in memory at the same time.

The internal caches of tree objects were improved to use less memory as well.

CHAPTER 3

GitPython Tutorial

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

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
join = os.path.join

# 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(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
cw = repo.config_writer()         # get a config writer to change configuration
cw.release()                      # call release() to be sure changes are written and locks a
```

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

Archive the repository contents to a tar file.

```
        repo.archive(open(join(rw_dir, 'repo.tar'), 'wb'))
```

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 gits inner workings.

Query relevant repository paths ...

```
        assert os.path.isdir(cloned_repo.working_tree_dir)                    # directory with your w
        assert cloned_repo.git_dir.startswith(cloned_repo.working_tree_dir)   # directory containing
        assert bare_repo.working_tree_dir is None                             # bare repositories hav
```

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.

```
        assert repo.head.ref == repo.heads.master                   # head is a symbolic reference po
        assert repo.tags['0.3.5'] == repo.tag('refs/tags/0.3.5')    # you can access tags in various
        assert repo.refs.master == repo.heads['master']             # .refs provides access to all re
        assert repo.refs['origin/master'] == repo.remotes.origin.refs.master  # ... remotes ...
        assert 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
        assert new_branch.commit == cloned_repo.active_branch.commit  # and which points to the check
        # 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
        assert 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 me
        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('1')

        # 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
        assert len(tree.blobs) + len(tree.trees) == len(tree)   # a tree is iterable itself to trave
```

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
        # end

        assert len(cloned_repo.remotes) == 1                    # we have been cloned, so there shou
        assert 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()):
            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
        # information to do it programatically 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.

```
        assert new_branch.checkout() == cloned_repo.active_branch     # checking out a branch adjusts
        assert 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 ti
        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 merege")

        # prepare a merge
        master = cloned_repo.heads.master                           # right-hand side is ahead of us,
        merge_base = cloned_repo.merge_base(new_branch, master)     # allwos for a three-way merge
        cloned_repo.index.merge_tree(master, base=merge_base)       # write the merge result into inde
        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 worki
```

`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_
        # 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 a
        sm = cloned_repo.create_submodule('mysubrepo', 'path/to/subrepo', url=bare_repo.git_dir, bran

        # .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 defintely available
        sm.remove(module=True, configuration=False)         # remove the working tree
        assert sm.exists() and not sm.module_exists()       # the submodule itself is still availab

        # update all submodules, non-recursively to save time, this method is very powerful, go have
        cloned_repo.submodule_update(recursive=False)
        assert sm.module_exists()                           # The submodules working tree was check
```

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='mast

        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 inde
repo.delete_head(new_branch)              # delete an existing head - only works if it is not
```

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.failUnlessRaises(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
hc != hct
hc != repo.tags[0]
hc == repo.head.reference.commit
```

Common fields are ...

```
assert 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.

```
assert hct.path == ''                   # root tree has no path
assert hct.trees[0].path != ''          # the first contained item has one though
assert hct.mode == 0o40000               # trees have the mode of a linux directory
assert hct.blobs[0].mode == 0o100644    # blobs have a specific mode though comparable to a s
```

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 give
```

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 headcommit.author.name == 'Sebastian Thiel'
assert isinstance(headcommit.authored_date, int)
assert headcommit.committer.name == 'Sebastian Thiel'
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 it's 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

```
assert tree['smmap'] == tree / 'smmap'          # access by index and by sub-path
for entry in tree:                                  # intuitive iteration of tree mem
    print(entry)
blob = tree.trees[0].blobs[0]                       # let's get a blob in a sub-tree
assert blob.name
assert len(blob.path) < len(blob.abspath)
assert tree.trees[0].name + '/' + blob.name == blob.path    # this is how the relative blob pa
assert tree[blob.path] == blob                      # you can use paths like 'dir/file
```

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)
assert repo.tree('v0.8.1').type == 'tree'               # yes, you can provide any refspec -
```

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
# 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 t
index.remove(['LICENSE'])                                           # remove an existing
assert os.path.isfile(os.path.join(repo.working_tree_dir, 'LICENSE'))   # working tree is unt

assert index.commit("my commit message").type == 'commit'               # commit changed inde
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 by commit message and 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
        # loads 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()                  # assure we actually have data. fetch() returns useful infor
        # Setup a local tracking branch of a remote branch
        empty_repo.create_head('master', origin.refs.master).set_tracking_branch(origin.refs.master)
        origin.rename('new_origin')    # rename remotes
        # push and pull behaves similarly to `git push|pull`
        origin.pull()
        origin.push()
        # 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 where attributes. The modification of remote configuration is more explicit though.

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

        # 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. *chomd +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]
        assert sm.name == 'gitdb'                           # git-python has gitdb as single submodule
        assert sm.children()[0].name == 'smmap'             # ... which has smmap as single submodule

        # The module is the repository referenced by the submodule
        assert sm.module_exists()                           # the module is available, which doesn't ha
        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
        assert len(sm.hexsha) == 40                         # Its sha defines the commit to checkout
        assert sm.exists()                                  # yes, this submodule is valid and exists
        # read its configuration conveniently
        assert sm.config_reader().get_value('path') == sm.path
        assert 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 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)

# To detach your head, you have to point to a commit directy
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.

```
        # checkout the branch using git-checkout. It will fail as the working tree appears dirty
        self.failUnlessRaises(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 argumen
        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. Its uses less memory when handling huge files, but will be 2 to 5 times slower when extracting large quantities small of 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 the one of the `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 logged using a python logger.
- if set to *full*, the executed git command and its output on stdout and stderr will be logged using a python logger.
- **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 4

API Reference

4.1 Objects.Base

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

Implements an Object which may be Blobs, Trees, Commits and Tags

NULL_BIN_SHA = '\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 = ('blob', 'tree', 'commit', 'tag')

__eq__(*other*)

Returns True if the objects have the same SHA1

__hash__()

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

__init__(*repo*, *binsha*)

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*)

Returns True if the objects do not have the same SHA1

__repr__()

Returns string with pythonic representation of our object

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

__str__()

Returns string of our SHA1 as understood by all git commands

binsha

data_stream

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

Note returned streams must be read in order

hexsha

Returns 40 byte hex version of our 20 byte binary sha

classmethod new(*repo*, *id*)

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 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*, *sha1*)

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*)
Writes our data directly to the given output stream :param ostream: File object compatible stream object. :return: self

type = None

class git.objects.base.**IndexObject**(*repo*, *binsha*, *mode=None*, *path=None*)
Base for all objects that can be part of the index file , namely Tree, Blob and SubModule objects

__hash__()

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

__init__(*repo*, *binsha*, *mode=None*, *path=None*)
Initialize a newly instanced IndexObject

Parameters

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

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

__module__ = 'git.objects.base'

__slots__ = ('path', 'mode')

abspath

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

name

Returns Name portion of the path, effectively being the basename

path

4.2 Objects.Blob

class git.objects.blob.**Blob**(*repo*, *binsha*, *mode=None*, *path=None*)

A Blob encapsulates a git blob object

DEFAULT_MIME_TYPE = 'text/plain'

__module__ = 'git.objects.blob'

__slots__ = ()

executable_mode = 33261

file_mode = 33188

link_mode = 40960

mime_type

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 = 'blob'

4.3 Objects.Commit

class git.objects.commit.**Commit**(*repo*, *binsha*, *tree=None*, *author=None*, *authored_date=None*, *author_tz_offset=None*, *committer=None*, *committed_date=None*, *committer_tz_offset=None*, *message=None*, *parents=None*, *encoding=None*, *gpgsig=None*)

Wraps a git Commit object.

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.

__init__(*repo*, *binsha*, *tree=None*, *author=None*, *authored_date=None*, *author_tz_offset=None*, *committer=None*, *committed_date=None*, *committer_tz_offset=None*, *message=None*, *parents=None*, *encoding=None*, *gpgsig=None*)

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

Parameters

- **binsha** – 20 byte sha1

- **parents** – tuple(Commit, ...) is a tuple of commit ids or actual Commits
- **tree** – Tree Tree object
- **author** – Actor is the author string (will be implicitly converted into an Actor object)
- **authored_date** – int_seconds_since_epoch is the authored DateTime - use time.gmtime() to convert it into a different format
- **author_tz_offset** – int_seconds_west_of_utc is the timezone that the authored_date is in
- **committer** – Actor is the committer string
- **committed_date** – int_seconds_since_epoch is the committed DateTime - use time.gmtime() to convert it into a different format
- **committer_tz_offset** – int_seconds_west_of_utc is the timezone that the authored_date is in
- **message** – string is 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 git.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'

__slots__ = ('tree', 'author', 'authored_date', 'author_tz_offset', 'committer', 'committed_date', 'committer_tz_offset

author

author_tz_offset

authored_date

committed_date

committer

committer_tz_offset

conf_encoding = 'i18n.commitencoding'

count(*paths=''*, ***kwargs*)

Count the number of commits reachable from this commit

Parameters

- **paths** – is an optinal 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. They must not alter the ouput style of the command, or parsing will yield incorrect results

Returns int defining the number of reachable commits

classmethod create_from_tree(*repo*, *tree*, *message*, *parent_commits=None*, *head=False*, *author=None*, *committer=None*, *author_date=None*, *commit_date=None*)

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. Else 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 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*, *rev*, *paths=''*, ***kwargs*)
Find all commits matching the given criteria.

Parameters

- **repo** – is the Repo
- **rev** – revision specifier, see git-rev-parse for viable options
- **paths** – is an optinal path or list of paths, if set only Commits that include the path or paths will be considered
- **kwargs** – optional keyword arguments to git rev-list where `max_count` is the maximum number of commits to fetch `skip` is the number of commits to skip `since` all commits since i.e. '1970-01-01'

Returns iterator yielding Commit items

iter_parents(*paths=''*, ***kwargs*)
Iterate _all_ parents of this commit.

Parameters

- **paths** – Optional path or list of paths limiting the Commits to those that contain at least one of the paths

- **kwargs** – All arguments allowed by git-rev-list

Returns Iterator yielding Commit objects which are parents of self

message

name_rev

Returns String describing the commits hex sha based on the closest Reference. Mostly useful for UI purposes

parents

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 git.Stats

summary

Returns First line of the commit message

tree

type = 'commit'

4.4 Objects.Tag

Module containing all object based types.

class git.objects.tag.**TagObject**(*repo*, *binsha*, *object=None*, *tag=None*, *tagger=None*, *tagged_date=None*, *tagger_tz_offset=None*, *message=None*)

Non-Lightweight tag carrying additional information about an object we are pointing to.

__init__(*repo*, *binsha*, *object=None*, *tag=None*, *tagger=None*, *tagged_date=None*, *tagger_tz_offset=None*, *message=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 is the DateTime of the tag creation - use time.gmtime to convert it into a different format
- **tagged_tz_offset** – int_seconds_west_of_utc is the timezone that the authored_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

tag

tagged_date

tagger

tagger_tz_offset

type = 'tag'

4.5 Objects.Tree

class git.objects.tree.**TreeModifier**(*cache*)

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 refernce to the cache of a tree, will be sorted. Assuring it will be in a serializable state

__delitem__(*name*)

Deletes an item with the given name if it exists

__init__(*cache*)

__module__ = 'git.objects.tree'

__slots__ = '_cache'

add(*sha*, *mode*, *name*, *force=False*)

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*, *mode*, *name*)

Add the given item to the tree, its correctness is assumed, which puts the caller into responsibility to assure the input is correct. For more information on the parameters, see `add` :param binsha: 20 byte binary sha

set_done()

Call this method once you are done modifying the tree information. It may be called several times, but be aware that each call will cause a sort operation :return self:

class git.objects.tree.**Tree**(*repo*, *binsha*, *mode=16384*, *path=None*)

Tree objects represent an ordered list of Blobs and other Trees.

`Tree as a list:`

```
Access a specific blob using the
tree['filename'] notation.

You may as well access by index
blob = tree[0]
```

__contains__(*item*)

__div__(*file*)
For PY2 only

__getitem__(*item*)

__getslice__(*i*, *j*)

__init__(*repo*, *binsha*, *mode=16384*, *path=None*)

__iter__()

__len__()

__module__ = 'git.objects.tree'

__reversed__()

__slots__ = '_cache'

__truediv__(*file*)
For PY3 only

blob_id = 8

blobs

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

cache

Returns An object allowing to modify 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. See the `TreeModifier` for more information on how to alter the cache

commit_id = 14

join(*file*)
Find the named object in this tree's contents :return: `git.Blob` or `git.Tree` or `git.Submodule`

Raises **KeyError** – if given file or tree does not exist in tree

symlink_id = 10

traverse(*predicate=<function <lambda>>*, *prune=<function <lambda>>*, *depth=-1*, *branch_first=True*, *visit_once=False*, *ignore_self=1*)
For documentation, see util.Traversable.traverse Trees are set to visit_once = False to gain more performance in the traversal

tree_id = 4

trees

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

type = 'tree'

4.6 Objects.Functions

Module with functions which are supposed to be as fast as possible

`git.objects.fun.`**tree_to_stream**(*entries*, *write*)
Write the give list of entries into a stream using its write method :param entries: **sorted** list of tuples with (binsha, mode, name) :param write: write method which takes a data string

git.objects.fun.**tree_entries_from_data**(*data*)
Reads the binary representation of a tree and returns tuples of Tree items :param data: data block with tree data (as bytes) :return: list(tuple(binsha, mode, tree_relative_path), ...)

git.objects.fun.**traverse_trees_recursive**(*odb*, *tree_shas*, *path_prefix*)

Returns 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.**traverse_tree_recursive**(*odb*, *tree_sha*, *path_prefix*)

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

4.7 Objects.Submodule.base

class git.objects.submodule.base.**Submodule**(*repo*, *binsha*, *mode=None*, *path=None*, *name=None*, *parent_commit=None*, *url=None*, *branch_path=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.

__eq__(*other*)
Compare with another submodule

__hash__()
Hash this instance using its logical id, not the sha

__init__(*repo*, *binsha*, *mode=None*, *path=None*, *name=None*, *parent_commit=None*, *url=None*, *branch_path=None*)
Initialize this instance with its attributes. We only document the ones that differ from IndexObject

Parameters

- **repo** – Our parent repository
- **binsha** – binary sha referring to a commit in the remote repository, see url parameter
- **parent_commit** – see set_parent_commit()
- **url** – The url to the remote repository which is the submodule
- **branch_path** – full (relative) path to ref to checkout when cloning the remote repository

__module__ = 'git.objects.submodule.base'

__ne__(*other*)
Compare with another submodule for inequality

__repr__()

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

__str__()

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

classmethod add(*repo*, *name*, *path*, *url=None*, *branch=None*, *no_checkout=False*)
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 reference 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 anotherone.
- **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

Returns The newly created submodule instance

Note works atomically, such that no change will be done if the repository update fails for instance

branch

Returns The branch instance that we are to checkout

Raises **InvalidGitRepositoryError** – if our module is not yet checked out

branch_name

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

branch_path

Returns full (relative) path as string to the branch we would checkout from the remote and track

children()

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

config_reader()

Returns ConfigReader instance which allows you to qurey 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(**args*, ***kwargs*)

exists()

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*, *parent_commit='HEAD'*)

Returns iterator yielding Submodule instances available in the given repository

k_default_mode = 57344

k_head_default = 'master'

k_head_option = 'branch'

k_modules_file = '.gitmodules'

module(**args*, ***kwargs*)

module_exists()

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

move(**args*, ***kwargs*)

name

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

Note by default, the name is the path at which to find the submodule, but in git-python it should be a unique identifier similar to the identifiers used for remotes, which allows to change the path of the submodule easily

parent_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(**args*, ***kwargs*)

rename(**args*, ***kwargs*)

set_parent_commit(*commit*, *check=True*)

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 = 'submodule'

update(*recursive=False*, *init=True*, *to_latest_revision=False*, *progress=None*, *dry_run=False*, *force=False*, *keep_going=False*)

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 of 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. Additinoally 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

Note does nothing in bare repositories

Note method is definitely not atomic if recurisve is True

Returns self

url

Returns The url to the repository which our module-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

__module__ = 'git.objects.submodule.base'

__slots__ = ()

x = 11

4.8 Objects.Submodule.root

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

A (virtual) Root of all submodules in the given repository. It can be used to more easily traverse all submodules of the master repository

__init__(*repo*)

__module__ = 'git.objects.submodule.root'

__slots__ = ()

k_root_name = '__ROOT__'

module()

Returns the actual repository containing the submodules

update(*previous_commit=None*, *recursive=True*, *force_remove=False*, *init=True*, *to_latest_revision=False*, *progress=None*, *dry_run=False*, *force_reset=False*, *keep_going=False*)

Update the submodules of this repository to the current HEAD commit. This method behaves smartly by determining changes of the path of a submodules repository, next to changes to the to-be-checked-out commit or the branch to be checked out. This works if the submodules 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, it 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 the UpdateProgress

BRANCHCHANGE = 16384

PATHCHANGE = 8192

REMOVE = 4096

URLCHANGE = 32768

__module__ = 'git.objects.submodule.root'

__slots__ = ()

x = 15

4.9 Objects.Submodule.util

git.objects.submodule.util.**sm_section**(*name*)

Returns section title used in .gitmodules configuration file

git.objects.submodule.util.**sm_name**(*section*)

Returns name of the submodule as parsed from the section name

git.objects.submodule.util.**mkhead**(*repo*, *path*)

Returns New branch/head instance

git.objects.submodule.util.**find_first_remote_branch**(*remotes*, *branch_name*)
Find the remote branch matching the name of the given branch or raise InvalidGitRepositoryError

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

__module__ = 'git.objects.submodule.util'

flush_to_index()
Flush changes in our configuration file to the index

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

write()

4.10 Objects.Util

Module for general utility functions

git.objects.util.**get_object_type_by_name**(*object_type_name*)

Returns type suitable to handle the given object type name. Use the type to create new instances.

Parameters **object_type_name** – Member of TYPES

Raises **ValueError** – In case object_type_name is unknown

git.objects.util.**parse_date**(*string_date*)
Parse the given date as one of the following

- 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.

git.objects.util.**parse_actor_and_date**(*line*)
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]

class git.objects.util.**ProcessStreamAdapter**(*process*, *stream_name*)
Class wireing 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*)

__init__(*process*, *stream_name*)

__module__ = 'git.objects.util'

__slots__ = ('_proc', '_stream')

class git.objects.util.**Traversable**
Simple interface to perform depth-first or breadth-first traversals into one direction. Subclasses only need to implement one function. Instances of the Subclass must be hashable

__module__ = 'git.objects.util'

__slots__ = ()

list_traverse(**args*, ***kwargs*)

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

traverse(*predicate=<function <lambda>>*, *prune=<function <lambda>>*, *depth=-1*, *branch_first=True*, *visit_once=True*, *ignore_self=1*, *as_edge=False*)

Returns iterator yieling of items found when traversing self

Parameters

- **predicate** – f(i,d) returns False if item i at depth d should not be included in the result
- **prune** – f(i,d) return True if the search should stop at item i at depth d. Item i will not be returned.

- **depth** – define at which level the iteration should not go deeper if -1, there is no limit if 0, you would effectively only get self, the root of the iteration i.e. if 1, you would only get the first level of predessessors/successors
- **branch_first** – if True, items will be returned branch first, otherwise depth first
- **visit_once** – if True, items will only be returned once, although they might be encountered several times. Loops are prevented that way.
- **ignore_self** – if True, self will be ignored and automatically pruned from the result. Otherwise it will be the first item to be returned. If as_edge is True, the source of the first edge is None
- **as_edge** – if True, return a pair of items, first being the source, second the destinatination, i.e. tuple(src, dest) with the edge spanning from source to destination

git.objects.util.**altz_to_utctz_str**(*altz*)
As above, but inverses the operation, returning a string that can be used in commit objects

git.objects.util.**utctz_to_altz**(*utctz*)
we convert utctz to the timezone in seconds, it is the format time.altzone returns. Git stores it as UTC timezone which has the opposite sign as well, which explains the -1 * (that was made explicit here) :param utctz: git utc timezone string, i.e. +0200

git.objects.util.**verify_utctz**(*offset*)

Raises **ValueError** – if offset is incorrect

Returns offset

class git.objects.util.**Actor**(*name*, *email*)
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*)

__hash__()

__init__(*name*, *email*)

__module__ = 'git.util'

__ne__(*other*)

__repr__()

__slots__ = ('name', 'email')

__str__()

classmethod author(*config_reader=None*)
Same as committer(), but defines the main author. It may be specified in the environment, but defaults to the committer

classmethod committer(*config_reader=None*)

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 = <_sre.SRE_Pattern object>

name_only_regex = <_sre.SRE_Pattern object>

4.11 Index.Base

class git.index.base.**IndexFile**(*repo*, *file_path=None*)

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

It provides custom merging facilities allowing to merge without actually changing your index or your working tree. This way you can perform 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

__init__(*repo*, *file_path=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*, *force=True*, *fprogress=<function <lambda>>*, *path_rewriter=None*, *write=True*, *write_extension_data=False*)

Add files from the working tree, specific blobs or BaseIndexEntries 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, i.e. 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 can be directories like 'lib', the latter ones will add all the files within the dirctory and subdirectories.

This equals a straight git-add.

They are added at stage 0

– **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 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 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) function 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* 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: http://opensource.apple.com/source/Git/Git-26/src/git-htmldocs/technical/index-format.txt

Returns List(BaseIndexEntries) 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(**args*, ***kwargs*)

commit(*message*, *parent_commits=None*, *head=True*, *author=None*, *committer=None*, *author_date=None*, *commit_date=None*)

Commit the current default index file, creating a commit object. For more information on the arguments, see tree.commit.

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

Returns Commit object representing the new commit

diff(**args*, ***kwargs*)

entries

classmethod entry_key(**entry*)

classmethod from_tree(*repo*, **treeish*, ***kwargs*)

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, Commits or 40 byte hexshas. The result changes according to the amount of trees. If 1 Tree is given, it will just be read into a new index 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. 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

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 kwarg to override that.

As the underlying git-read-tree command takes into account the current index, it will be temporarily moved out of the way to assure there are no unsuspected interferences.

iter_blobs(*predicate=<function <lambda>>*)

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 BlobFilter, allows you to yield blobs only if they match a given list of paths.

merge_tree(**args*, ***kwargs*)

move(**args*, ***kwargs*)

classmethod new(*repo*, **tree_sha*)

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.

path

Returns Path to the index file we are representing

remove(**args*, ***kwargs*)

repo

reset(**args*, ***kwargs*)

resolve_blobs(*iter_blobs*)

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()

Returns Iterator yielding 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. I.e. a file removed on the 'other' branch whose entries are at stage 3 will not have a stage 3 entry.

update()

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

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

Returns self

version

write(*file_path=None*, *ignore_extension_data=False*)

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 afterwards to create a tree representing your written changes. If this data is present in the written index, git-write-tree will instead write the stored/cached tree. Alternatively, use IndexFile.write_tree() to handle this case automatically

Returns self

write_tree()
Writes 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
- **UnmergedEntriesError** –

exception git.index.base.**CheckoutError**(*message*, *failed_files*, *valid_files*, *failed_reasons*)
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*, *failed_files*, *valid_files*, *failed_reasons*)

__module__ = 'git.exc'

__str__()

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

4.12 Index.Functions

git.index.fun.**write_cache**(*entries*, *stream*, *extension_data=None*, *ShaStreamCls=<class 'git.util.IndexFileSHA1Writer'>*)
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.**read_cache**(*stream*)
Read a cache file from the given stream :return: 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.**write_tree_from_cache**(*entries*, *odb*, *sl*, *si=0*)
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 IndexEntries
- **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

git.index.fun.**entry_key**(**entry*)

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.**stat_mode_to_index_mode**(*mode*)
Convert the given mode from a stat call to the corresponding index mode and return it

git.index.fun.**run_commit_hook**(*name*, *index*)
Run the commit hook of the given name. Silently ignores hooks that do not exist. :param name: name of hook, like 'pre-commit' :param index: IndexFile instance :raises HookExecutionError:

git.index.fun.**hook_path**(*name*, *git_dir*)

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

4.13 Index.Types

Module with additional types used by the index

class git.index.typ.**BlobFilter**(*paths*)
Predicate to be used by 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*)

__init__(*paths*)

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.**BaseIndexEntry**
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 requried.

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

__dict__ = dict_proxy({'__module__': 'git.index.typ', '__str__': <function __str__ at 0x7fcc53881050>, 'binsha': <prop

__module__ = 'git.index.typ'

__repr__()

__str__()

binsha
binary sha of the blob

flags

Returns flags stored with this entry

classmethod from_blob(*blob*, *stage=0*)

Returns Fully equipped BaseIndexEntry at the given stage

hexsha
hex version of our sha

mode
File Mode, compatible to stat module constants

path

Returns our path relative to the repository working tree root

stage
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 http://www.kernel.org/pub/software/scm/git/docs/git-read-tree.html

to_blob(*repo*)

Returns Blob using the information of this index entry

class git.index.typ.**IndexEntry**
Allows convenient access to IndexEntry data without completely unpacking it.

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

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

__module__ = 'git.index.typ'

ctime

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

dev
Device ID

classmethod from_base(*base*)

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*, *stage=0*)

Returns Minimal entry resembling the given blob object

gid
Group ID

inode
Inode ID

mtime
See ctime property, but returns modification time

size

Returns Uncompressed size of the blob

uid
User ID

4.14 Index.Util

Module containing index utilities

class git.index.util.**TemporaryFileSwap**(*file_path*)
Utility class moving a file to a temporary location within the same directory and moving it back on to where on object deletion.

__del__()

__init__(*file_path*)

__module__ = 'git.index.util'

__slots__ = ('file_path', 'tmp_file_path')

file_path

tmp_file_path

git.index.util.**post_clear_cache**(*func*)
Decorator for functions that alter the index using the git command. This would invalidate our possibly existing entries dictionary which is why it must be deleted to allow it to be lazily reread later.

Note This decorator will not be required once all functions are implemented natively which in fact is possible, but probably not feasible performance wise.

git.index.util.**default_index**(*func*)
Decorator assuring 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*)
Decorator which changes the current working dir to the one of the git repository in order to assure relative paths are handled correctly

4.15 GitCmd

class git.cmd.**Git**(*working_dir=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 print each invocation of the command to stdout. Set its value to 'full' to see details about the returned values.

class AutoInterrupt(*proc*, *args*)

Kill/Interrupt the stored process instance once this instance goes out of scope. It is used to prevent processes piling up in case iterators stop reading. Besides all attributes are wired through to the contained process object.

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

__del__()

__getattr__(*attr*)

__init__(*proc*, *args*)

__module__ = 'git.cmd'

__slots__ = ('proc', 'args')

args

proc

wait()

Wait for the process and return its status code.

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

Raises **GitCommandError** – if the return status is not 0

class Git.**CatFileContentStream**(*size*, *stream*)

Object representing a sized read-only stream returning the contents of an object. It 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 is read to the end of the objects's lifetime, we read the rest to assure the underlying stream continues to work

__del__()

__init__(*size*, *stream*)

__iter__()

__module__ = 'git.cmd'

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

next()

read(*size=-1*)

readline(*size=-1*)

readlines(*size=-1*)

Git.**GIT_PYTHON_GIT_EXECUTABLE = 'git'**

Git.**GIT_PYTHON_TRACE = False**

Git.**USE_SHELL = False**

Git.**__call__**(***kwargs*)
Specify command line options to the git executable for a subcommand call

Parameters **kwargs** – is 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()

Git.**__getattr__**(*name*)
A convenience method as it allows to call the command as if it was an object. :return: Callable object that will execute call _call_process with your arguments.

Git.**__init__**(*working_dir=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(). It is meant to be the working tree directory if available, or the .git directory in case of bare repositories.

Git.**__module__ = 'git.cmd'**

Git.**__slots__ = ('_working_dir', 'cat_file_all', 'cat_file_header', '_version_info', '_git_options', '_environment')**

Git.**cat_file_all**

Git.**cat_file_header**

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

Currently persistent commands will be interrupted.

Returns self

Git.**custom_environment**(**args*, ***kwds*)
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

Git.**environment**()

Git.**execute**(*command*, *istream=None*, *with_keep_cwd=False*, *with_extended_output=False*, *with_exceptions=True*, *as_process=False*, *output_stream=None*, *stdout_as_string=True*, *kill_after_timeout=None*, *with_stdout=True*, ***subprocess_kwargs*)
Handles executing the command on the shell and consumes and returns the returned information (stdout)

Parameters

- **command** – The command argument list to execute. It should be a string, or a sequence of program arguments. The program to execute is the first item in the args sequence or string.
- **istream** – Standard input filehandle passed to subprocess.Popen.

- **with_keep_cwd** – Whether to use the current working directory from os.getcwd(). The cmd otherwise uses its own working_dir that it has been initialized with if possible.
- **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 himself. 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 output to the given stream directly. This feature only has any effect if as_process is False. Processes will always be created with a pipe due to issues with subprocess. This merely is a workaround as data will be copied from the output pipe to the given output stream directly. Judging from the implementation, you shouldn't use this flag !
- **stdout_as_string** – if False, the commands 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.
- **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.
- **with_stdout** – If True, default True, we open stdout on the created process
- **kill_after_timeout** – To specify 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. This feature is not supported on Windows. It's also worth noting that kill_after_timeout uses SIGKILL, which can have negative side effects on a repository. For example, stale locks in case of git gc could render the repository incapable of accepting changes until the lock is manually removed.

Returns

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

if ouput_stream is True, the stdout value will be your output stream: * output_stream if extended_output = False * tuple(int(status), output_stream, str(stderr)) if extended_output = True

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

Raises **GitCommandError** –

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

Git.**get_object_data**(*ref*)

As get_object_header, but returns object data as well :return: (hexsha, type_string, size_as_int,data_string) :note: not threadsafe

Git.**get_object_header**(*ref*)
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.**git_exec_name = 'git'**

Git.**git_exec_name_win = 'git.cmd'**

Git.**max_chunk_size = 65536**

Git.**stream_object_data**(*ref*)
As get_object_header, 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 Command instance per thread to be safe !

Git.**transform_kwargs**(*split_single_char_options=True*, ***kwargs*)
Transforms Python style kwargs into git command line options.

Git.**update_environment**(***kwargs*)
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

Git.**version_info**

Returns tuple(int, int, int, int) tuple with integers representing the major, minor and additional version numbers as parsed from git version. This value is generated on demand and is cached

Git.**working_dir**

Returns Git directory we are working on

4.16 Config

Module containing module parser implementation able to properly read and write configuration files

git.config.**GitConfigParser**
alias of write

class git.config.**SectionConstraint**(*config*, *section*)
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

__del__()

__getattr__(*attr*)

__init__(*config*, *section*)

__module__ = 'git.config'

__slots__ = ('_config', '_section_name')

config
return: Configparser instance we constrain

release()
Equivalent to GitConfigParser.release(), which is called on our underlying parser instance

4.17 Diff

class git.diff.**Diffable**
Common interface for all object 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.

class Index

__dict__ = dict_proxy({'__dict__': <attribute '__dict__' of 'Index' objects>, '__module__': 'git.diff', '__weakref

__module__ = 'git.diff'

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

Diffable.**__module__ = 'git.diff'**

Diffable.**__slots__ = ()**

Diffable.**diff**(*other=<class 'git.diff.Index'>*, *paths=None*, *create_patch=False*, ***kwargs*)
Creates diffs between two items being trees, trees and index or an index and the working tree. It will detect renames automatically.

Parameters

- **other** – Is the item to compare us with. If None, we will be compared to the working tree. If Treeish, it will be compared against the respective tree If Index (type), it will be compared against the index. It defaults to Index to assure the method will not by-default fail on bare repositories.
- **paths** – is a list of paths or a single path to limit the diff to. It will only include at least one of the givne path or paths.
- **create_patch** – If True, the returned Diff contains a detailed patch that if applied makes the self to other. Patches are somwhat costly as blobs have to be read and diffed.
- **kwargs** – Additional arguments passed to git-diff, such as R=True to swap both sides of the diff.

Returns git.DiffIndex

Note On a bare repository, 'other' needs to be provided as Index or as as Tree/Commit, or a git command error will occour

class git.diff.**DiffIndex**

Implements an Index for diffs, allowing a list of Diffs to be queried by the diff properties.

The class improves the diff handling convenience

__dict__ = dict_proxy({'iter_change_type': <function iter_change_type at 0x7fcc538b20c8>, '__module__': 'git.diff', '

__module__ = 'git.diff'

__weakref__

list of weak references to the object (if defined)

change_type = ('A', 'D', 'R', 'M')

iter_change_type(*change_type*)

Returns iterator yieling 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

class git.diff.**Diff**(*repo*, *a_path*, *b_path*, *a_blob_id*, *b_blob_id*, *a_mode*, *b_mode*, *new_file*, *deleted_file*, *rename_from*, *rename_to*, *diff*)

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 inidcate 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. But the path will be available though. If it is listed in a diff the working tree version of the file must be different to the version in the index or tree, and hence has been modified.

NULL_BIN_SHA = '\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*)

__hash__()

__init__(*repo*, *a_path*, *b_path*, *a_blob_id*, *b_blob_id*, *a_mode*, *b_mode*, *new_file*, *deleted_file*, *rename_from*, *rename_to*, *diff*)

__module__ = 'git.diff'

__ne__(*other*)

__slots__ = ('a_blob', 'b_blob', 'a_mode', 'b_mode', 'a_path', 'b_path', 'new_file', 'deleted_file', 'rename_from', 'ren

__str__()

a_blob

a_mode

a_path

b_blob

b_mode

b_path

deleted_file

diff

new_file

re_header = <_sre.SRE_Pattern object at 0x2effc90>

rename_from

rename_to

renamed

Returns True if the blob of our diff has been renamed

4.18 Exceptions

Module containing all exceptions thrown througout the git package,

exception git.exc.**CacheError**
Base for all errors related to the git index, which is called cache internally

__module__ = 'git.exc'

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

exception git.exc.**CheckoutError**(*message*, *failed_files*, *valid_files*, *failed_reasons*)
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*, *failed_files*, *valid_files*, *failed_reasons*)

__module__ = 'git.exc'

__str__()

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

exception git.exc.**GitCommandError**(*command*, *status*, *stderr=None*, *stdout=None*)
Thrown if execution of the git command fails with non-zero status code.

__init__(*command*, *status*, *stderr=None*, *stdout=None*)

__module__ = 'git.exc'

__str__()

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

exception git.exc.**GitCommandNotFound**
Thrown if we cannot find the *git* executable in the PATH or at the path given by the GIT_PYTHON_GIT_EXECUTABLE environment variable

__module__ = 'git.exc'

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

exception git.exc.**HookExecutionError**(*command*, *status*, *stdout*, *stderr*)
Thrown if a hook exits with a non-zero exit code. It provides access to the exit code and the string returned via standard output

__init__(*command*, *status*, *stdout*, *stderr*)

__module__ = 'git.exc'

__str__()

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

exception git.exc.**InvalidGitRepositoryError**
Thrown if the given repository appears to have an invalid format.

__module__ = 'git.exc'

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

exception git.exc.**NoSuchPathError**
Thrown if a path could not be access by the system.

__module__ = 'git.exc'

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

exception git.exc.**RepositoryDirtyError**(*repo*, *message*)
Thrown whenever an operation on a repository fails as it has uncommited changes that would be overwritten

__init__(*repo*, *message*)

__module__ = 'git.exc'

__str__()

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

exception git.exc.**UnmergedEntriesError**
Thrown if an operation cannot proceed as there are still unmerged entries in the cache

__module__ = 'git.exc'

exception git.exc.**WorkTreeRepositoryUnsupported**
Thrown to indicate we can't handle work tree repositories

__module__ = 'git.exc'

4.19 Refs.symbolic

class git.refs.symbolic.**SymbolicReference**(*repo*, *path*)
Represents a special case of a reference such that this reference is symbolic. It 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.

__eq__(*other*)

__hash__()

__init__(*repo*, *path*)

__module__ = 'git.refs.symbolic'

__ne__(*other*)

__repr__()

__slots__ = ('repo', 'path')

__str__()

abspath

commit
Query or set commits directly

classmethod create(*repo*, *path*, *reference='HEAD'*, *force=False*, *logmsg=None*)
Create a new symbolic reference, hence 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, i.e. "NEW_HEAD" or "symrefs/my_new_symref"
- **reference** – The reference to 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. Otherwise 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*, *path*)
Delete the reference at the given path

Parameters

- **repo** – Repository to delete the reference from
- **path** – Short or full path pointing to the reference, i.e. refs/myreference or just "myreference", hence 'refs/' is implied. Alternatively the symbolic reference to be deleted

classmethod **dereference_recursive**(*repo*, *ref_path*)

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*, *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

is_detached

Returns True if we are a detached reference, hence we point to a specific commit instead to another reference

is_remote()

Returns True if this symbolic reference points to a remote branch

is_valid()

Returns True if the reference is valid, hence it can be read and points to a valid object or reference.

classmethod **iter_items**(*repo*, *common_path=None*)
Find all refs in the repository

Parameters

- **repo** – is 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 assuring that only refs suitable for the actual class are returned.

Returns

git.SymbolicReference[], each of them is guaranteed to be a symbolic ref which is not detached and pointing to a valid ref

List is lexigraphically sorted The returned objects represent actual subclasses, such as Head or TagReference

log()

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*, *message*, *newbinsha=None*)
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 added RefLogEntry instance

log_entry(*index*)

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 sparringly, or only if you need just one index. In that case, it will be faster than the `log()` method

name

Returns In case of symbolic references, the shortest assumable name is the path itself.

object
Return the object our ref currently refers to

path

ref
Returns the Reference we point to

reference
Returns the Reference we point to

rename(*new_path*, *force=False*)
Rename self to a new path

Parameters

- **new_path** – Either a simple name or a full path, i.e. 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** – In case a file at path but a different contents already exists

repo

set_commit(*commit*, *logmsg=None*)
As set_object, but restricts the type of object to be a Commit

Raises **ValueError** – If commit is not a Commit object or doesn't point to a commit

Returns self

set_object(*object*, *logmsg=None*)
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. SymbolicReferences will be dereferenced beforehand to obtain the object they point to
- **logmsg** – If not None, the message will be used in the reflog entry to be written. Otherwise the reflog is not altered

Note plain SymbolicReferences may not actually point to objects by convention

Returns self

set_reference(*ref*, *logmsg=None*)
Set ourselves to the given ref. It will stay a symbol if the ref is a Reference. Otherwise an Object, given as Object instance or refspec, is assumed and if valid, will be set which effectively detaches the refererence if it was a purely symbolic one.

Parameters

- **ref** – SymbolicReference instance, Object instance or refspec string Only if the ref is a SymbolicRef instance, we will point to it. Everthing 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*)

Returns string with a full repository-relative path which can be used to initialize a Reference instance, for instance by using `Reference.from_path`

4.20 Refs.reference

class `git.refs.reference.`**Reference**(*repo*, *path*, *check_path=True*)
Represents a named reference to any object. Subclasses may apply restrictions though, i.e. Heads can only point to commits.

__init__(*repo*, *path*, *check_path=True*)
Initialize this instance :param repo: Our parent repository

Parameters

- **path** – Path relative to the .git/ directory pointing to the ref in question, i.e. 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'

__slots__ = ()

__str__()

classmethod iter_items(*repo*, *common_path=None*)
Equivalent to SymbolicReference.iter_items, but will return non-detached references as well.

name

Returns (shortest) Name of this reference - it may contain path components

remote_head

remote_name

set_object(*object*, *logmsg=None*)
Special version which checks if the head-log needs an update as well :return: self

4.21 Refs.head

class git.refs.head.**HEAD**(*repo*, *path='HEAD'*)
Special case of a Symbolic Reference as it represents the repository's HEAD reference.

__init__(*repo*, *path='HEAD'*)

__module__ = 'git.refs.head'

__slots__ = ()

orig_head()

Returns SymbolicReference pointing at the ORIG_HEAD, which is maintained to contain the previous value of HEAD

reset(*commit='HEAD'*, *index=True*, *working_tree=False*, *paths=None*, ***kwargs*)
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 object, Reference Object 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.

Returns self

class git.refs.head.**Head**(*repo*, *path*, *check_path=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">

>>> head.commit.hexsha
'1c09f116cbc2cb4100fb6935bb162daa4723f455'
```

__dict__ = dict_proxy({'rename': <function rename at 0x7fcc538cd758>, '__module__': 'git.refs.head', '_common_pat

__module__ = 'git.refs.head'

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

checkout (*force=False*, ***kwargs*)
Checkout 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, i.e. 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.

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()

Returns A configuration parser instance constrained to only read this instance's values

config_writer()

Returns A configuration writer instance with read-and write acccess to options of this head

classmethod delete (*repo*, **heads*, ***kwargs*)
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*, *force=False*)
Rename self to a new path

Parameters

- **new_path** – Either a simple name or a path, i.e. 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*)

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()

Returns The remote_reference we are tracking, or None if we are not a tracking branch

4.22 Refs.tag

class git.refs.tag.**TagReference**(*repo*, *path*, *check_path=True*)

Class representing 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 carray 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)
```

__module__ = 'git.refs.tag'

__slots__ = ()

commit

Returns Commit object the tag ref points to

classmethod create(*repo*, *path*, *ref='HEAD'*, *message=None*, *force=False*, ***kwargs*)

Create a new tag reference.

Parameters

- **path** – The name of the tag, i.e. 1.0 or releases/1.0. The prefix refs/tags is implied
- **ref** – A reference to the object you want to tag. It can be a commit, tree or blob.
- **message** – 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, i.e.:

```
tagref.tag.message
```

- **force** – If True, to force creation of a tag even though that tag already exists.
- **kwargs** – Additional keyword arguments to be passed to git-tag

Returns A new TagReference

classmethod delete (*repo*, **tags*)
Delete the given existing tag or tags

object

Returns The object our ref currently refers to. Refs can be cached, they will always point to the actual object as it gets re-created on each query

tag

Returns Tag object this tag ref points to or None in case we are a light weight tag

git.refs.tag.**Tag**
alias of *TagReference*

4.23 Refs.remote

class git.refs.remote.**RemoteReference** (*repo*, *path*, *check_path=True*)
Represents a reference pointing to a remote head.

__module__ = 'git.refs.remote'

classmethod create (**args*, ***kwargs*)
Used to disable this method

classmethod delete (*repo*, **refs*, ***kwargs*)
Delete the given remote references

Note kwargs are given for compatability with the base class method as we should not narrow the signature.

classmethod iter_items (*repo*, *common_path=None*, *remote=None*)
Iterate remote references, and if given, constrain them to the given remote

4.24 Refs.log

class git.refs.log.**RefLog** (*filepath=None*)
A reflog contains reflog entries, 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 orded, 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.

__init__ (*filepath=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__ (*filepath=None*)

__slots__ = ('_path',)

classmethod append_entry (*config_reader*, *filepath*, *oldbinsha*, *newbinsha*, *message*)
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. May also be 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*, *index*)

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 specifiy 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*)

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*)

Returns Iterator yielding RefLogEntry instances, one for each line read sfrom the given stream.

Parameters stream – file-like object containing the revlog in its native format or basestring instance pointing to a file to read

classmethod path (*ref*)

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*)
Write the contents of the reflog instance to a file at the given filepath. :param filepath: path to file, parent directories are assumed to exist

write ()
Write this instance's data to the file we are originating from :return: self

class git.refs.log.**RefLogEntry**
Named tuple allowing easy access to the revlog data fields

__module__ = 'git.refs.log'

__repr__()
Representation of ourselves in git reflog format

__slots__ = ()

actor
Actor instance, providing access

format()

Returns a string suitable to be placed in a reflog file

classmethod from_line(*line*)

Returns New RefLogEntry instance from the given revlog line.

Parameters **line** – line bytes without trailing newline

Raises **ValueError** – If line could not be parsed

message
Message describing the operation that acted on the reference

classmethod new(*oldhexsha*, *newhexsha*, *actor*, *time*, *tz_offset*, *message*)

Returns New instance of a RefLogEntry

newhexsha
The hexsha to the commit the ref now points to, after the change

oldhexsha
The hexsha to the commit the ref pointed to before the change

time
time as tuple:

- [0] = int(time)
- [1] = int(timezone_offset) in time.altzone format

4.25 Remote

class git.remote.**RemoteProgress**
Handler providing an interface to parse progress information emitted by git-push and git-fetch and to dispatch callbacks allowing subclasses to react to the progress.

BEGIN = 1

CHECKING_OUT = 256

COMPRESSING = 8

COUNTING = 4

END = 2

FINDING_SOURCES = 128

OP_MASK = -4

RECEIVING = 32

RESOLVING = 64

STAGE_MASK = 3

WRITING = 16

__init__()

__module__ = 'git.util'

__slots__ = ('_cur_line', '_seen_ops')

line_dropped(*line*)
Called whenever a line could not be understood and was therefore dropped.

new_message_handler()

Returns a progress handler suitable for handle_process_output(), passing lines on to this Progress handler in a suitable format

re_op_absolute = <_sre.SRE_Pattern object>

re_op_relative = <_sre.SRE_Pattern object at 0x2f83830>

update(*op_code*, *cur_count*, *max_count=None*, *message=''*)
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.

You may read the contents of the current line in self._cur_line

x = 8

class git.remote.**PushInfo**(*flags*, *local_ref*, *remote_ref_string*, *remote*, *old_commit=None*, *summary=''*)
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

__init__ (*flags*, *local_ref*, *remote_ref_string*, *remote*, *old_commit=None*, *summary=''*)
Initialize a new instance

__module__ = 'git.remote'

__slots__ = ('local_ref', 'remote_ref_string', 'flags', 'old_commit', '_remote', 'summary')

flags

local_ref

old_commit

remote_ref

Returns Remote Reference or TagReference in the local repository corresponding to the remote_ref_string kept in this instance.

remote_ref_string

summary

x = 10

class git.remote.**FetchInfo**(*ref*, *flags*, *note=''*, *old_commit=None*, *remote_ref_path=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 o
```

ERROR = 128

FAST_FORWARD = 64

FORCED_UPDATE = 32

HEAD_UPTODATE = 4

NEW_HEAD = 2

NEW_TAG = 1

REJECTED = 16

TAG_UPDATE = 8

__init__(*ref*, *flags*, *note=''*, *old_commit=None*, *remote_ref_path=None*)
Initialize a new instance

__module__ = 'git.remote'

__slots__ = ('ref', 'old_commit', 'flags', 'note', 'remote_ref_path')

__str__()

commit

Returns Commit of our remote ref

flags

name

Returns Name of our remote ref

note

old_commit

re_fetch_result = <_sre.SRE_Pattern object at 0x2f90ac0>

ref

remote_ref_path

x = 7

class git.remote.**Remote**(*repo*, *name*)
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.

__eq__(*other*)

__getattr__(*attr*)
Allows to call this instance like remote.special(*args, **kwargs) to call git-remote special self.name

__hash__()

__init__(*repo*, *name*)
Initialize a remote instance

Parameters

- **repo** – The repository we are a remote of
- **name** – the name of the remote, i.e. 'origin'

__module__ = 'git.remote'

__ne__(*other*)

__repr__()

__slots__ = ('repo', 'name', '_config_reader')

__str__()

classmethod add(*repo*, *name*, *url*, ***kwargs*)
Create a new remote to the given repository :param repo: Repository instance that is to receive the new remote :param name: Desired name of the remote :param url: URL which corresponds to the remote's name :param kwargs: Additional arguments to be passed to the git-remote add command :return: New Remote instance :raise GitCommandError: in case an origin with that name already exists

config_reader

Returns GitConfigParser compatible object able to read options for only our remote. Hence you may simple type config.get("pushurl") to obtain the information

config_writer

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 useable 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*, *name*, *url*, ***kwargs*)
Create a new remote to the given repository :param repo: Repository instance that is to receive the new remote :param name: Desired name of the remote :param url: URL which corresponds to the remote's name :param kwargs: Additional arguments to be passed to the git-remote add command :return: New Remote instance :raise GitCommandError: in case an origin with that name already exists

exists()

Returns True if this is a valid, existing remote. Valid remotes have an entry in the repository's configuration

fetch(*refspec=None*, *progress=None*, ***kwargs*)
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

  Fetch supports multiple refspecs (as the underlying git-fetch does) - supplying a list rather than a string for 'refspec' will make use of this facility.

- **progress** – See 'push' method
- **kwargs** – Additional arguments to be passed to git-fetch

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*)

Returns Iterator yielding Remote objects of the given repository

name

pull(*refspec=None*, *progress=None*, ***kwargs*)
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
- **kwargs** – Additional arguments to be passed to git-pull

Returns Please see 'fetch' method

push(*refspec=None*, *progress=None*, ***kwargs*)
Push changes from source branch in refspec to target branch in refspec.

Parameters

- **refspec** – see 'fetch' method
- **progress** – Instance of type RemoteProgress allowing the caller to receive progress information until the method returns. If None, progress information will be discarded
- **kwargs** – Additional arguments to be passed to git-push

Returns IterableList(PushInfo, ...) iterable list of PushInfo instances, each one informing about 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 IterableList will be null.

refs

Returns IterableList of RemoteReference objects. It is prefixed, allowing you to omit the remote path portion, i.e.:: remote.refs.master # yields RemoteReference('/refs/remotes/origin/master')

classmethod **remove**(*repo*, *name*)
Remove the remote with the given name :return: the passed remote name to remove

rename(*new_name*)
Rename self to the given new_name :return: self

repo

classmethod **rm**(*repo*, *name*)
Remove the remote with the given name :return: the passed remote name to remove

stale_refs

Returns

IterableList 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 possble 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

update(***kwargs*)

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 branches ancestry anymore).

Parameters **kwargs** – Additional arguments passed to git-remote update

Returns self

4.26 Repo.Base

class git.repo.base.**Repo**(*path=None*, *odbt=<class 'git.db.GitCmdObjectDB'>*, *search_parent_directories=False*)

Represents a git repository and allows you to query references, gather commit information, generate diffs, create and clone repositories 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 raise AssertionError 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

__del__()

__eq__(*rhs*)

__hash__()

__init__(*path=None*, *odbt=<class 'git.db.GitCmdObjectDB'>*, *search_parent_directories=False*)

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")
```

- **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

- **InvalidGitRepositoryError** –
- **NoSuchPathError** –

Returns git.Repo

__module__ = 'git.repo.base'

__ne__(*rhs*)

__repr__()

__slots__ = ('working_dir', '_working_tree_dir', 'git_dir', '_bare', 'git', 'odb')

active_branch
The name of the currently active branch.

Returns Head to the active branch

alternates
Retrieve a list of alternates paths or set a list paths to be used as alternates

archive(*ostream*, *treeish=None*, *prefix=None*, ***kwargs*)
Archive the tree at the given revision.

Parm ostream file compatible stream object to which the archive will be written as bytes

Parm treeish is the treeish name/id, defaults to active branch

Parm prefix is the optional prefix to prepend to each filename in the archive

Parm kwargs Additional arguments passed to git-archive

- 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 multipe paths.

Raises **GitCommandError** – in case something went wrong

Returns self

bare

Returns True if the repository is bare

blame(*rev*, *file*)
The blame information for the given file at the given revision.

Parm rev revision specifier, see git-rev-parse for viable options.

Returns list: [git.Commit, list: [<line>]] A list of tuples 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.

branches
A list of `Head` objects representing the branch heads in this repo

Returns `git.IterableList(Head, ...)`

clone(*path*, *progress=None*, ***kwargs*)
Create a clone from this repository.

Parameters

- **path** – is the full path of the new repo (traditionally ends with ./<name>.git).
- **progress** – See 'git.remote.Remote.push'.
- **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 command

Returns `git.Repo` (the newly cloned repo)

classmethod clone_from(*url*, *to_path*, *progress=None*, *env=None*, ***kwargs*)
Create a clone from the given URL

Parameters

- **url** – valid git url, see http://www.kernel.org/pub/software/scm/git/docs/git-clone.html#URLS
- **to_path** – Path to which the repository should be cloned to
- **progress** – See 'git.remote.Remote.push'.
- **env** – Optional dictionary containing the desired environment variables.
- **kwargs** – see the `clone` method

Returns Repo instance pointing to the cloned directory

commit(*rev=None*)
The Commit object for the specified revision :param rev: revision specifier, see git-rev-parse for viable options. :return: `git.Commit`

config_level = ('system', 'user', 'global', 'repository')

config_reader(*config_level=None*)

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 config_writer method If None, all applicable levels will be used. Specify a level in case you know which exact file you whish to read to prevent reading multiple files for instance

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='repository'*)

Returns 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 = sytem wide configuration file global = user level configuration file repository = configuration file for this repostory only

create_head(*path*, *commit='HEAD'*, *force=False*, *logmsg=None*)
Create a new head within the repository. For more documentation, please see the Head.create method.

Returns newly created Head Reference

create_remote(*name*, *url*, ***kwargs*)
Create a new remote.

For more information, please see the documentation of the Remote.create methods

Returns Remote reference

create_submodule(**args*, ***kwargs*)
Create a new submodule

Note See the documentation of Submodule.add for a description of the applicable parameters

Returns created submodules

create_tag(*path*, *ref='HEAD'*, *message=None*, *force=False*, ***kwargs*)
Create a new tag reference. For more documentation, please see the TagReference.create method.

Returns TagReference object

daemon_export
If True, git-daemon may export this repository

delete_head(**heads*, ***kwargs*)
Delete the given heads

Parameters **kwargs** – Additional keyword arguments to be passed to git-branch

delete_remote(*remote*)
Delete the given remote.

delete_tag(**tags*)
Delete the given tag references

description
the project's description

git

git_dir

has_separate_working_tree()

Returns True if our git_dir is not at the root of our working_tree_dir, but a .git file with a platform agnositic symbolic link. Our git_dir will be whereever the .git file points to

Note bare repositories will always return False here

head

Returns HEAD Object pointing to the current head reference

heads
A list of `Head` objects representing the branch heads in this repo

Returns `git.IterableList(Head, ...)`

index

Returns IndexFile representing this repository's index.

classmethod init(*path=None*, *mkdir=True*, *odbt=<class 'git.db.GitCmdObjectDB'>*, ***kwargs*)
Initialize a git repository at the given path if specified

Parameters

- **path** – is 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
- **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

Parm mkdir if specified will create the repository directory if it doesn't already exists. Creates the directory with a mode=0755. Only effective if a path is explicitly given

Parm kwargs keyword arguments serving as additional options to the git-init command

Returns `git.Repo` (the newly created repo)

is_ancestor(*ancestor_rev*, *rev*)
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`, ancestor_rev is an accestor to rev.

is_dirty(*index=True*, *working_tree=True*, *untracked_files=False*, *submodules=True*)

Returns `True`, the repository is considered dirty. By default it will react like a git-status without untracked files, hence it is dirty if the index or the working copy have changes.

iter_commits(*rev=None*, *paths=''*, ***kwargs*)
A list of Commit objects representing the history of a given ref/commit

Parm rev revision specifier, see git-rev-parse for viable options. If None, the active branch will be used.

Parm paths is an optional path or a list of paths to limit the returned commits to Commits that do not contain that path or the paths will not be returned.

Parm kwargs Arguments to be passed to git-rev-list - common ones are max_count and skip

Note to receive only commits between two named revisions, use the "revA...revB" revision specifier

:return `git.Commit[]`

iter_submodules(**args*, ***kwargs*)
An iterator yielding Submodule instances, see Traversable interface for a description of args and kwargs
:return: Iterator

iter_trees(**args*, ***kwargs*)

Returns Iterator yielding Tree objects

Note Takes all arguments known to iter_commits method

merge_base(**rev*, ***kwargs*)
Find the closest common ancestor for the given revision (e.g. Commits, Tags, References, 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 specified as kwarg, the list will have at max one Commit, or is empty if no common merge base exists.

Raises **ValueError** – If not at least two revs are provided

odb

re_author_committer_start = <_sre.SRE_Pattern object>

re_hexsha_only = <_sre.SRE_Pattern object>

re_hexsha_shortened = <_sre.SRE_Pattern object>

re_tab_full_line = <_sre.SRE_Pattern object>

re_whitespace = <_sre.SRE_Pattern object>

references
A list of Reference objects representing tags, heads and remote references.

Returns IterableList(Reference, ...)

refs
A list of Reference objects representing tags, heads and remote references.

Returns IterableList(Reference, ...)

remote(*name='origin'*)

Returns Remote with the specified name

Raises **ValueError** – if no remote with such a name exists

remotes
A list of Remote objects allowing to access and manipulate remotes :return: `git.IterableList(Remote, ...)`

rev_parse(*repo*, *rev*)

Returns Object at the given revision, either Commit, Tag, Tree or Blob

Parameters **rev** – git-rev-parse compatible revision specification as string, please see http://www.kernel.org/pub/software/scm/git/docs/git-rev-parse.html for details

Raises

- **BadObject** – if the given revision could not be found
- **ValueError** – If rev couldn't be parsed
- **IndexError** – If invalid reflog index is specified

submodule(*name*)

Returns Submodule with the given name

Raises **ValueError** – If no such submodule exists

submodule_update(**args*, ***kwargs*)
Update the submodules, keeping the repository consistent as it will take the previous state into consideration. For more information, please see the documentation of RootModule.update

submodules

Returns git.IterableList(Submodule, ...) of direct submodules available from the current head

tag(*path*)

Returns TagReference Object, reference pointing to a Commit or Tag

Parameters **path** – path to the tag reference, i.e. 0.1.5 or tags/0.1.5

tags
A list of `Tag` objects that are available in this repo :return: `git.IterableList(TagReference, ...)`

tree(*rev=None*)
The Tree object for the given treeish revision Examples:

```
repo.tree(repo.heads[0])
```

Parameters **rev** – is a revision pointing to a Treeish (being a commit or tree)

Returns `git.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.

untracked_files

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

working_dir

working_tree_dir

Returns The working tree directory of our git repository. If this is a bare repository, None is returned.

4.27 Repo.Functions

Package with general repository related functions

`git.repo.fun.`**rev_parse**(*repo*, *rev*)

Returns Object at the given revision, either Commit, Tag, Tree or Blob

Parameters **rev** – git-rev-parse compatible revision specification as string, please see http://www.kernel.org/pub/software/scm/git/docs/git-rev-parse.html for details

Raises

- **BadObject** – if the given revision could not be found
- **ValueError** – If rev couldn't be parsed
- **IndexError** – If invalid reflog index is specified

`git.repo.fun.`**is_git_dir**(*d*)
This is taken from the git setup.c:is_git_directory function.

@throws 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.**touch**(*filename*)

git.repo.fun.**find_git_dir**(*d*)

git.repo.fun.**name_to_object**(*repo*, *name*, *return_ref=False*)

Returns object specified by the given name, hexshas (short and long) as well as references are supported

Parameters **return_ref** – if name specifies a reference, we will return the reference instead of the object. Otherwise it will raise BadObject or BadName

git.repo.fun.**short_to_long**(*odb*, *hexsha*)

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 byte

git.repo.fun.**deref_tag**(*tag*)
Recursively dereference a tag and return the resulting object

git.repo.fun.**to_commit**(*obj*)
Convert the given object to a commit if possible and return it

4.28 Util

git.util.**stream_copy**(*source*, *destination*, *chunk_size=524288*)
Copy all data from the source stream into the destination stream in chunks of size chunk_size

Returns amount of bytes written

git.util.**join_path**(*a*, **p*)
Join path tokens together similar to os.path.join, but always use '/' instead of possibly '' on windows.

git.util.**to_native_path_linux**(*path*)

git.util.**join_path_native**(*a*, **p*)

As join path, but makes sure an OS native path is returned. This is only needed to play it safe on my dear windows and to assure nice paths that only use ''

class git.util.**Stats**(*total*, *files*)
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*, *files*)

__module__ = 'git.util'

__slots__ = ('total', 'files')

files

total

class git.util.**IndexFileSHA1Writer**(*f*)

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 the asked for explicitly usign write_sha.

Only useful to the indexfile

Note Based on the dulwich project

__init__(*f*)

__module__ = 'git.util'

__slots__ = ('f', 'sha1')

close()

f

sha1

tell()

write(*data*)

write_sha()

class git.util.**Iterable**

Defines an interface for iterable items which is to assure a uniform way to retrieve and iterate items within the git repository

__module__ = 'git.util'

__slots__ = ()

classmethod iter_items(*repo*, **args*, ***kwargs*)

For more information about the arguments, see list_items :return: iterator yielding Items

classmethod list_items(*repo*, **args*, ***kwargs*)

Find all items of this type - subclasses can specify args and kwargs differently. If no args are given, subclasses are obliged to return all items if no additional arguments arg given.

Note Favor the iter_items method as it will

:return:list(Item,...) list of item instances

class git.util.**IterableList**(*id_attr*, *prefix=''*)

List of iterable objects allowing to query an object by id or by named index:

```
heads = repo.heads
heads.master
heads['master']
heads[0]
```

It 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.

__contains__(*attr*)

__delitem__(*index*)

__getattr__(*attr*)

__getitem__(*index*)

__init__(*id_attr*, *prefix=''*)

__module__ = 'git.util'

static __new__(*id_attr*, *prefix=''*)

__slots__ = ('_id_attr', '_prefix')

class git.util.**BlockingLockFile**(*file_path*, *check_interval_s=0.3*, *max_block_time_s=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*, *check_interval_s=0.3*, *max_block_time_s=9223372036854775807*)

Configure the instance

Parm check_interval_s Period of time to sleep until the lock is checked the next time. By default, it waits a nearly unlimited time

Parm max_block_time_s Maximum amount of seconds we may lock

__module__ = 'git.util'

__slots__ = ('_check_interval', '_max_block_time')

class git.util.**LockFile**(*file_path*)

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

__del__()

__init__(*file_path*)

__module__ = 'git.util'

__slots__ = ('_file_path', '_owns_lock')

class git.util.**Actor**(*name*, *email*)

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*)

__hash__()

__init__(*name*, *email*)

__module__ = 'git.util'

__ne__(*other*)

__repr__()

__slots__ = ('name', 'email')

__str__()

classmethod author(*config_reader=None*)
Same as committer(), but defines the main author. It may be specified in the environment, but defaults to the committer

classmethod committer(*config_reader=None*)

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 = <_sre.SRE_Pattern object>

name_only_regex = <_sre.SRE_Pattern object>

git.util.**get_user_id**()

Returns string identifying the currently active system user as name@node

git.util.**assure_directory_exists**(*path*, *is_file=False*)
Assure 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

class git.util.**RemoteProgress**
Handler providing an interface to parse progress information emitted by git-push and git-fetch and to dispatch callbacks allowing subclasses to react to the progress.

BEGIN = 1

CHECKING_OUT = 256

COMPRESSING = 8

COUNTING = 4

END = 2

FINDING_SOURCES = 128

OP_MASK = -4

RECEIVING = 32

RESOLVING = 64

STAGE_MASK = 3

WRITING = 16

__init__()

__module__ = 'git.util'

__slots__ = ('_cur_line', '_seen_ops')

line_dropped(*line*)
Called whenever a line could not be understood and was therefore dropped.

new_message_handler()

Returns a progress handler suitable for handle_process_output(), passing lines on to this Progress handler in a suitable format

re_op_absolute = <_sre.SRE_Pattern object>

re_op_relative = <_sre.SRE_Pattern object at 0x2f83830>

update(*op_code*, *cur_count*, *max_count=None*, *message=''*)
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.

You may read the contents of the current line in self._cur_line

x = 8

git.util.**rmtree**(*path*)
Remove the given 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

class git.util.**WaitGroup**
WaitGroup is like Go sync.WaitGroup.

Without all the useful corner cases. By Peter Teichman, taken from https://gist.github.com/pteichman/84b92ae7cef0ab98f5a8

__dict__ = dict_proxy({'__module__': 'git.util', 'done': <function done at 0x7fcc53910398>, '__dict__': <attribute '__d

__init__()

__module__ = 'git.util'

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

add(*n*)

done()

wait()

git.util.**unbare_repo**(*func*)
Methods with this decorator raise InvalidGitRepositoryError if they encounter a bare repository

CHAPTER 5

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 6

Changelog

6.1 1.0.2 - Fixes

- IMPORTANT: Changed default object database of *Repo* objects to *GitComdObjectDB*. 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.

6.2 1.0.1 - Fixes

- A list of all issues can be found on github

6.3 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.

6.4 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 behavious 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

6.5 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

6.6 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+

6.7 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'*

6.8 0.3.3

- When fetching, pulling or pushing, and an error occours, 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 statments to signal certain kinds of errors

6.9 0.3.2.1

- Fix for #207

6.10 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.

6.11 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.

6.12 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

6.13 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 addded for easy configuration of tracking branches.

6.14 0.3.0 Beta 2

- Added python 2.4 support

6.15 0.3.0 Beta 1

6.15.1 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

6.15.2 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

6.16 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, mimicing git-commit-tree

6.17 0.2

6.17.1 General

- file mode in Tree, Blob and Diff objects now is an int compatible to definintiions 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 epoc 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). Prevously 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.

6.17.2 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

6.17.3 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.

6.17.4 Blob

- former 'name' member renamed to path as it suits the actual data better

6.17.5 GitCommand

- git.subcommand call scheme now prunes out None from the argument list, allowing to be called more confortably 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

6.17.6 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

6.17.7 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

6.17.8 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

6.17.9 Diffing

- Commit and Tree objects now support diffing natively with a common interface to compare agains other Commits or Trees, against the working tree or against the index.

6.17.10 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.

6.17.11 Referernces

- 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 dynmically 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.

6.17.12 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_branche 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

6.17.13 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

6.17.14 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.

6.17.15 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

6.18 0.1.6

6.18.1 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 '–' seperates 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`.

6.18.2 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.

6.18.3 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`.

6.18.4 Head

- Corrected problem where branches was only returning the last path component instead of the entire path component following refs/heads/.

6.18.5 Repo

- Modified the gzip archive creation to use the python gzip module.
- Corrected `commits_between` always returning None instead of the reversed list.

6.19 0.1.5

6.19.1 General

- upgraded to Mock 0.4 dependency.
- Replace GitPython with git in repr() outputs.
- Fixed packaging issue caused by ez_setup.py.

6.19.2 Blob

- No longer strip newlines from Blob data.

6.19.3 Commit

- Corrected problem with git-rev-list –bisect-all. See http://groups.google.com/group/git-python/browse_thread/thread/aed1d5c4b31d5027

6.19.4 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.

6.19.5 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.

6.20 0.1.4.1

- removed `method_missing` stuff and replaced with a `__getattr__` override in `Git`.

6.21 0.1.4

- renamed `git_python` to `git`. Be sure to delete all pyc files before testing.

6.21.1 Commit

- Fixed problem with commit stats not working under all conditions.

6.21.2 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.

6.21.3 Tree

- Fixed up problem where `name` doesn't exist on root of tree.

6.21.4 Repo

- Corrected problem with creating bare repo. Added `Repo.create` alias.

6.22 0.1.2

6.22.1 Tree

- Corrected problem with `Tree.__div__` not working with zero length files. Removed `__len__` override and replaced with size instead. Also made size cach properly. This is a breaking change.

6.23 0.1.1

Fixed up some urls because I'm a moron

6.24 0.1.0

initial release

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

g

Symbols

A

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X