GitPython Documentation

Release 0.2.0 Beta

Michael Trier

November 19, 2014

Contents

CHAPTER 1

Overview / Install

GitPython is a python library used to interact with Git repositories.

GitPython was a port of the grit library in Ruby created by Tom Preston-Werner and Chris Wanstrath, but grew beyond its heritage through its improved design and performance.

1.1 Requirements

- Git tested with 1.5.3.7
- Requires Git 1.6.5.4 or newer if index.add function is to be used
- Python Nose - used for running the tests
- Mock by Michael Foord used for tests. Requires 0.5

1.2 Installing GitPython

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

```
# easy_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 `easy_install` and pypi can be found here:

- setuptools
- install setuptools
- pypi

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

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

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 Gitorious and GitHub, which can be browsed at:

- http://gitorious.org/projects/git-python/
- http://github.com/Byron/GitPython

and cloned using:

```
$ git clone git://gitorious.org/git-python/mainline.git git-python
$ git clone git://github.com/Byron/GitPython.git git-python
```

1.6 Mailing List

http://groups.google.com/group/git-python

1.7 Issue Tracker

http://byronimo.lighthouseapp.com/projects/51787-gitpython/milestones

1.8 License Information

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

CHAPTER 2

GitPython Tutorial

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

2.1 Initialize a Repo object

The first step is to create a `Repo` object to represent your repository:

```
from git import *
repo = Repo("/Users/mtrier/Development/git-python")
```

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

```
repo = Repo.create("/var/git/git-python.git")
```

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
repo.config_writer()        # get a config writer to change configuration
```

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

```
repo.is_dirty()
False
repo.untracked_files
['my_untracked_file']
```

Clone from existing repositories or initialize new empty ones:

```
cloned_repo = repo.clone("to/this/path")
new_repo = repo.init("path/for/new/repo")
```

Archive the repository contents to a tar file:

```
repo.archive(open("repo.tar",'w'))
```

2.2 Examining References

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

```
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
```

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
```

2.3 Modifying References

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

```
repo.delete_head('master')           # delete an existing head
master = repo.create_head('master')  # create a new one
master.commit = 'HEAD~10'            # set branch to another commit without changing index or working
```

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

```
new_tag = repo.create_tag('my_tag', 'my message')
repo.delete_tag(new_tag)
```

Change the symbolic reference to switch branches cheaply (without adjusting the index or the working copy):

```
new_branch = repo.create_head('new_branch')
repo.head.reference = new_branch
```

2.4 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 SHA1 hash, being 40 hexadecimal characters in size or 20 bytes in size.

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

In Git-Pyhton, all objects can be accessed through their common base, compared and hashed, as shown in the following example:

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

Basic fields are:

```
hct.type
'tree'
hct.size
166
hct.sha
'a95eeb2a7082212c197cabbf2539185ec74ed0e8'
hct.data        # returns string with pure uncompressed data
'...'
len(hct.data) == hct.size
```

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

```
hct.path            # root tree has no path
''
hct.trees[0].path   # the first subdirectory has one though
'dir'
htc.mode            # trees have mode 0
0
'%o' % htc.blobs[0].mode    # blobs have a specific mode though comparable to a standard linux fs
100644
```

Access blob data (or any object data) directly or using streams:

```
htc.data            # binary tree data as string ( inefficient )
htc.blobs[0].data_stream            # stream object to read data from
htc.blobs[0].stream_data(my_stream) # write data to given stream
```

2.5 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('v0.1')
repo.commit('HEAD~10')
```

Iterate 100 commits:

```
repo.iter_commits('master', max_count=100)
```

If you need paging, you can specify a number of commits to skip:

```
repo.iter_commits('master', max_count=10, skip=20)
```

The above will return commits 21-30 from the commit list.:

```
headcommit = repo.head.commit

headcommit.sha
'207c0c4418115df0d30820ab1a9acd2ea4bf4431'

headcommit.parents
[<git.Commit "a91c45eee0b41bf3cdaad3418ca3850664c4a4b4">]

headcommit.tree
```

```
<git.Tree "563413aedbeda425d8d9dcbb744247d0c3e8a0ac">

headcommit.author
<git.Actor "Michael Trier <mtrier@gmail.com>">

headcommit.authored_date        # seconds since epoch
1256291446

headcommit.committer
<git.Actor "Michael Trier <mtrier@gmail.com>">

headcommit.committed_date
1256291446

headcommit.message
'cleaned up a lot of test information. Fixed escaping so it works with
subprocess.'
```

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

```
import time
time.asctime(time.gmtime(headcommit.committed_date))
'Wed May 7 05:56:02 2008'

time.strftime("%a, %d %b %Y %H:%M", time.gmtime(headcommit.committed_date))
'Wed, 7 May 2008 05:56'
```

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

```
headcommit.parents[0].parents[0].parents[0]
```

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

2.6 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
<git.Tree "a006b5b1a8115185a228b7514cdcd46fed90dc92">

tree.sha
'a006b5b1a8115185a228b7514cdcd46fed90dc92'
```

Once you have a tree, you can get the contents:

```
tree.trees          # trees are subdirectories
[<git.Tree "f7eb5df2e465ab621b1db3f5714850d6732cfed2">]

tree.blobs          # blobs are files
[<git.Blob "a871e79d59cf8488cac4af0c8f990b7a989e2b53">,
<git.Blob "3594e94c04db171e2767224db355f514b13715c5">,
<git.Blob "e79b05161e4836e5fbf197aeb52515753e8d6ab6">,
<git.Blob "94954abda49de8615a048f8d2e64b5de848e27a1">]
```

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

```
tree[0] == tree['dir']                      # access by index and by sub-path
<git.Tree "f7eb5df2e465ab621b1db3f5714850d6732cfed2">
for entry in tree: do_something_with(entry)

blob = tree[0][0]
blob.name
'file'
blob.path
'dir/file'
blob.abspath
'/Users/mtrier/Development/git-python/dir/file'
>>>tree['dir/file'].sha == blob.sha
```

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 an unix system:

```
tree/"lib"
<git.Tree "c1c7214dde86f76bc3e18806ac1f47c38b2b7a30">
tree/"dir/file" == blob.sha
```

You can also get a tree directly from the repository if you know its name:

```
repo.tree()
<git.Tree "master">

repo.tree("c1c7214dde86f76bc3e18806ac1f47c38b2b7a30")
<git.Tree "c1c7214dde86f76bc3e18806ac1f47c38b2b7a30">
repo.tree('0.1.6')
<git.Tree "6825a94104164d9f0f5632607bebd2a32a3579e5">
```

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

```
tree.traverse()
<generator object at 0x7f6598bd65a8>
for entry in traverse(): do_something_with(entry)
```

2.7 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:

```
index = repo.index
```

Access objects and add/remove entries. Commit the changes:

```
for stage,blob in index.iter_blobs(): do_something(...)
Access blob objects
for (path,stage),entry in index.entries.iteritems: pass
Access the entries directly
index.add(['my_new_file'])      # add a new file to the index
index.remove(['dir/existing_file'])
new_commit = index.commit("my commit message")
```

Create new indices from other trees or as result of a merge. Write that result to a new index:

```
tmp_index = Index.from_tree(repo, 'HEAD~1') # load a tree into a temporary index
merge_index = Index.from_tree(repo, 'base', 'HEAD', 'some_branch') # merge two trees three-way
merge_index.write("merged_index")
```

2.8 Handling Remotes

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

```
test_remote = repo.create_remote('test', 'git@server:repo.git')
repo.delete_remote(test_remote) # create and delete remotes
origin = repo.remotes.origin    # get default remote by name
origin.refs                     # local remote references
o = origin.rename('new_origin') # rename remotes
o.fetch()                       # fetch, pull and push from and to the remote
o.pull()
o.push()
```

You can easily access configuration information for a remote by accessing options as if they where attributes:

```
o.url
'git@server:dummy_repo.git'
```

Change configuration for a specific remote only:

```
o.config_writer.set("pushurl", "other_url")
```

2.9 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
idiff = hcommit.diff()          # diff tree against index
tdiff = hcommit.diff('HEAD~1')  # diff tree against previous tree
wdiff = 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:

```
for diff_added in wdiff.iter_change_type('A'): do_something_with(diff_added)
```

2.10 Switching Branches

To switch between branches, you effectively need to point your HEAD to the new branch head and reset your index and working copy to match. A simple manual way to do it is the following one:

```
repo.head.reference = repo.heads.other_branch
repo.head.reset(index=True, working_tree=True)
```

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 for instance which generally prevents you from destroying your work. Use the safer approach as follows:

```
repo.heads.master.checkout()                    # checkout the branch using git-checkout
repo.heads.other_branch.checkout()
```

2.11 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")         # default command
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 commandline. 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 to individual arguments. Objects are converted to strings using the str(...) function.

2.12 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.

API Reference

3.1 Actor

class `git.actor.`**`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.

`name_email_regex` = <_sre.SRE_Pattern object at 0x7fe75493e540>

`name_only_regex` = <_sre.SRE_Pattern object at 0x7fe7548b6310>

3.2 Objects.Base

class `git.objects.base.`**`IndexObject`**(*repo*, *sha*, *mode=None*, *path=None*)

Base for all objects that can be part of the index file , namely Tree, Blob and SubModule objects

`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`

class `git.objects.base.`**`Object`**(*repo*, *id*)

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

This Object also serves as a constructor for instances of the correct type:

```
inst = Object.new(repo,id)
inst.sha        # objects sha in hex
inst.size    # objects uncompressed data size
inst.data    # byte string containing the whole data of the object
```

`NULL_HEX_SHA` = ‘0000000000000000000000000000000000000000’

`TYPES` = (‘blob’, ‘tree’, ‘commit’, ‘tag’)

`data`

`data_stream`

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

classmethod `new` (*repo*, *id*)

Return New Object instance of a type appropriate to the object type behind id. The id of the newly created object will be a hexsha even though the input id may have been a Reference or Rev-Spec

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

`repo`

`sha`

`size`

`stream_data` (*ostream*)
Writes our data directly to the given output stream

`ostream` File object compatible stream object.

Returns self

`type` **= None**

3.3 Objects.Blob

class `git.objects.blob.`**`Blob`** (*repo*, *sha*, *mode=None*, *path=None*)
A Blob encapsulates a git blob object

`DEFAULT_MIME_TYPE` **= 'text/plain'**

`mime_type`
The mime type of this file (based on the filename)

Returns str

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

`type` **= 'blob'**

3.4 Objects.Commit

class `git.objects.commit.`**`Commit`** (*repo*, *sha*, *tree=None*, *author=None*, *authored_date=None*, *author_tz_offset=None*, *committer=None*, *committed_date=None*, *committer_tz_offset=None*, *message=None*, *parents=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.

`author`

`author_tz_offset`

`authored_date`

`committed_date`

`committer`

`committer_tz_offset`

`count` (*paths=''*, ***kwargs*)
Count the number of commits reachable from this commit

`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

classmethod `create_from_tree` (*repo*, *tree*, *message*, *parent_commits=None*, *head=False*)
Commit the given tree, creating a commit object.

`repo` is the Repo

`tree` Sha of a tree or a tree object to become 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.

Returns Commit object representing the new commit

Note: Additional information about hte committer and Author are taken from the environment or from the git configuration, see git-commit-tree for more information

classmethod `iter_items` (*repo*, *rev*, *paths=''*, ***kwargs*)
Find all commits matching the given criteria.

`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.

`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

Return: 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.

Return git.Stats

summary

Returns First line of the commit message.

tree

type = 'commit'

3.5 Objects.Tag

Module containing all object based types.

class `git.objects.tag.`**`TagObject`**(*repo*, *sha*, *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.

message

object

tag

tagged_date

tagger

tagger_tz_offset

type = 'tag'

3.6 Objects.Tree

class `git.objects.tree.`**`Tree`**(*repo*, *sha*, *mode=0*, *path=None*)

Tress represent a ordered list of Blobs and other Trees. Hence it can be accessed like a list.

Tree's will cache their contents after first retrieval to improve efficiency.

```
Tree as a list:
```

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

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

blob_id = 8

blobs

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

commit_id = 14

symlink_id = 10

traverse(*predicate=<function <lambda> at 0x7fe7548626e0>*, *prune=<function <lambda> at 0x7fe754862758>*, *depth=-1*, *branch_first=True*, *visit_once=False*, *ignore_self=1*)
For documentation, see utils.Traversable.traverse

Trees are set to visist_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'

git.objects.tree.**sha_to_hex**(*sha*)
Takes a string and returns the hex of the sha within

3.7 Objects.Utils

Module for general utility functions

class git.objects.utils.**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.

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

traverse(*predicate=<function <lambda> at 0x7fe7548c46e0>*, *prune=<function <lambda> at 0x7fe7548c4758>*, *depth=-1*, *branch_first=True*, *visit_once=True*, *ignore_self=1*, *as_edge=False*)

Returns iterator yieling of items found when traversing self

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.utils.**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.

object_type_name Member of TYPES

Raises ValueError: In case object_type_name is unknown

git.objects.utils.**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]

3.8 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.

args

proc

wait()
Wait for the process and return its status code.
Raise GitCommandError if the return status is not 0

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

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.

Returns:

```
str(output)                                    # extended_output = False (Default)
tuple(int(status), str(stdout), str(stderr)) # extended_output = True

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

Raise 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)

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.

Return: (hexsha, type_string, size_as_int)

Git.**transform_kwargs**(***kwargs*)
Transforms Python style kwargs into git command line options.

Git.**working_dir**

Returns Git directory we are working on

git.cmd.**dashify**(*string*)

3.9 Config

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

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

3.10 Diff

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
```

`Deleted File`:

```
b_mode is None
b_blob 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.

`a_blob`

`a_mode`

`b_blob`

`b_mode`

`deleted_file`

`diff`

`new_file`

`null_hex_sha` = **'0000000000000000000000000000000000000000'**

`re_header` = **<_sre.SRE_Pattern object at 0x28e2cd0>**

`rename_from`

`rename_to`

`renamed`

Returns: True if the blob of our diff has been renamed

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

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

`iter_change_type`(*change_type*)

Return iterator yieling Diff instances that match the given change_type

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.`**`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`**

`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.

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 Rename detection will only work if create_patch is True.

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

3.11 Errors

Module containing all exceptions thrown througout the git package,

exception `git.errors.`**`GitCommandError`**(*command*, *status*, *stderr=None*)

Thrown if execution of the git command fails with non-zero status code.

exception `git.errors.`**`InvalidGitRepositoryError`**

Thrown if the given repository appears to have an invalid format.

exception `git.errors.`**`NoSuchPathError`**

Thrown if a path could not be access by the system.

3.12 Index

Module containing Index implementation, allowing to perform all kinds of index manipulations such as querying and merging.

class `git.index.`**`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.

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

Returns Fully equipped BaseIndexEntry at the given stage

`mode`

File Mode, compatible to stat module constants

`path`

`sha`

hex sha of the blob

`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

class `git.index.`**`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.

`paths`

exception `git.index.`**`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

class `git.index.`**`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.

`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

`base` Instance of type BaseIndexEntry

classmethod `from_blob`(*blob*)

Returns Minimal entry resembling the given blob objecft

`gid`
Group ID

`inode`
Inode ID

`mtime`
See ctime property, but returns modification time

`size`
Uncompressed size of the blob

Note Will be 0 if the stage is not 0 (hence it is an unmerged entry)

`uid`
User ID

class `git.index.`**`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.get_entries_key(index_entry_instance)] = index_entry_instance

Otherwise changes to it will be lost when changing the index using its methods.

`S_IFGITLINK` = 57344

`add`(**args*, ***kwargs*)

`checkout`(**args*, ***kwargs*)

`commit`(**args*, ***kwargs*)

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

`entries`

classmethod `from_tree`(*repo*, **treeish*, ***kwargs*)
Merge the given treeish revisions into a new index which is returned. The original index will remain unaltered

`repo` The repository treeish are located in.

`*treeish` One, two or three Tree Objects or Commits. 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.

classmethod `get_entries_key`(**entry*)

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

`entry` One instance of type BaseIndexEntry or the path and the stage

`iter_blobs`(*predicate=<function <lambda> at 0x7fe754818668>*)

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

`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*)

`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.

Raise 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_tree_extension_data=False*)

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

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_tree_extension_data If True, the TREE type extension data read in the index will not be written to disk. 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

Note Index writing based on the dulwich implementation

write_tree(*missing_ok=False*)
Writes the Index in self to a corresponding Tree file into the repository object database and returns it as corresponding Tree object.

missing_ok If True, missing objects referenced by this index will not result in an error.

Returns Tree object representing this index

git.index.**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.**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.

3.13 Refs

Module containing all ref based objects

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

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.

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 allow to partially reset individual files.

kwargs Additional arguments passed to git-reset.

Returns self

class git.refs.**Head**(*repo*, *path*)
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.sha
'1c09f116cbc2cb4100fb6935bb162daa4723f455'
```

`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.

`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.

classmethod `create` (*repo*, *path*, *commit='HEAD'*, *force=False*, ***kwargs*)

Create a new head. `repo`

Repository to create the head in

`path` The name or path of the head, i.e. 'new_branch' or feature/feature1. The prefix refs/heads is implied.

`commit` Commit to which the new head should point, defaults to the current HEAD

`force` if True, force creation even if branch with that name already exists.

`**kwargs` Additional keyword arguments to be passed to git-branch, i.e. track, no-track, l

Returns Newly created Head

Note This does not alter the current HEAD, index or Working Tree

classmethod `delete` (*repo*, **heads*, ***kwargs*)

Delete the given heads

`force` If True, the heads will be deleted even if they are not yet merged into the main development stream. Default False

`rename` (*new_path*, *force=False*)

Rename self to a new path

`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

class `git.refs.`**`Reference`**(*repo*, *path*)

Represents a named reference to any object. Subclasses may apply restrictions though, i.e. Heads can only point to commits.

classmethod **`create`**(*repo*, *path*, *commit='HEAD'*, *force=False*)

Create a new reference. `repo`

Repository to create the reference in

`path` The relative path of the reference, i.e. 'new_branch' or feature/feature1. The path prefix 'refs/' is implied if not given explicitly

`commit` Commit to which the new reference should point, defaults to the current HEAD

`force` if True, force creation even if a reference with that name already exists. Raise OSError otherwise

Returns Newly created Reference

Note This does not alter the current HEAD, index or Working Tree

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

`object`

Return the object our ref currently refers to

class `git.refs.`**`RemoteReference`**(*repo*, *path*)

Represents a reference pointing to a remote head.

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.

`remote_head`

Returns Name of the remote head itself, i.e. master.

NOTE: The returned name is usually not qualified enough to uniquely identify a branch

`remote_name`

Returns Name of the remote we are a reference of, such as 'origin' for a reference named 'origin/master'

class `git.refs.`**`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.

`commit`

Query or set commits directly

classmethod **`create`**(*repo*, *path*, *reference='HEAD'*, *force=False*)

Create a new symbolic reference, hence a reference pointing to another reference. `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

force if True, force creation even if a symbolic reference with that name already exists. Raise OSError otherwise

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

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 `from_path` (*repo*, *path*)

Return 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_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

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.

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

`name`

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

`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

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

classmethod to_full_path(*path*)

Returns string with a full path name which can be used to initialize

a Reference instance, for instance by using `Reference.from_path`

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

class `git.refs.`**`TagReference`**(*repo*, *path*)
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
```

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.

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

tag

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

3.14 Remote

Module implementing a remote object allowing easy access to git remotes

class `git.remote.`**`FetchInfo`**(*ref*, *flags*, *note=''*, *old_commit=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
```

`ERROR` = **128**

`FAST_FORWARD` = **64**

`FORCED_UPDATE` = **32**

`HEAD_UPTODATE` = **4**

`NEW_HEAD` = **2**

`NEW_TAG` = **1**

`REJECTED` = **16**

`TAG_UPDATE` = **8**

`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 0x2859790>**

`ref`

`x` = **7**

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

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.**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.

classmethod add(*repo*, *name*, *url*, ***kwargs*)

Create a new remote to the given repository `repo`

Repository instance that is to receive the new remote

name Desired name of the remote

url URL which corresponds to the remote's name

****kwargs** Additional arguments to be passed to the git-remote add command

Returns 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`

Return 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 `repo`

Repository instance that is to receive the new remote

`name` Desired name of the remote

`url` URL which corresponds to the remote's name

`**kwargs` Additional arguments to be passed to the git-remote add command

Returns New Remote instance

Raise GitCommandError in case an origin with that name already exists

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

Fetch the latest changes for this remote

`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

`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.

`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.

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

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

Returns self

repo

classmethod rm(*repo*, *name*)
Remove the remote with the given name

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.

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).

kwargs Additional arguments passed to git-remote update

Returns self

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

COMPRESSING = 8

COUNTING = 4

END = 2

OP_MASK = 28

`STAGE_MASK` = 3

`WRITING` = 16

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

`re_op_absolute` = <_sre.SRE_Pattern object at 0x7fe754853750>

`re_op_relative` = <_sre.SRE_Pattern object at 0x28e8960>

`update`(*op_code*, *cur_count*, *max_count=None*, *message=''*)
Called whenever the progress changes

`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` = 4

3.15 Repo

class `git.repo.`**`Repo`**(*path=None*)
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, wich 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 directoy, which is always set.

`DAEMON_EXPORT_FILE` = 'git-daemon-export-ok'

`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. `ostream`

file compatible stream object to which the archive will be written

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

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

kwargs Additional arguments passed to git-archive NOTE: Use the 'format' argument to define the kind of format. Use specialized ostreams to write any format supported by python

Examples:

```
>>> repo.archive(open("archive"))
<String containing tar.gz archive>
```

Raise 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.

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*, ***kwargs*)
Create a clone from this repository.

path is the full path of the new repo (traditionally ends with ./<name>.git).

kwargs keyword arguments to be given to the git-clone command

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

commit(*rev=None*)
The Commit object for the specified revision

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

Returns `git.Commit`

config_level = ('system', '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.

NOTE: On windows, system configuration cannot currently be read as the path is unknown, instead the global path will be used.

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

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.

`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*, ***kwargs*)
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_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

`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`

`head`

Return 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*, ***kwargs*)
Initialize a git repository at the given path if specified

`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

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

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

Examples:

```
git.Repo.init('/var/git/myrepo.git',bare=True)
```

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

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

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

rev

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

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.

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

Returns `git.Commit[]`

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

Returns Iterator yielding Tree objects

Note: Takes all arguments known to iter_commits method

re_author_committer_start = <_sre.SRE_Pattern object at 0x7fe754822270>

re_hexsha_only = <_sre.SRE_Pattern object at 0x7fe75486a8c8>

re_tab_full_line = <_sre.SRE_Pattern object at 0x7fe754852a80>

re_whitespace = <_sre.SRE_Pattern object at 0x7fe75487e4f0>

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'*)

Return Remote with the specified name

Raise ValueError if no remote with such a name exists

remotes

A list of Remote objects allowing to access and manipulate remotes

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

tag(*path*)

Return TagReference Object, reference pointing to a Commit or Tag

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

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

tree(*rev=None*)

The Tree object for the given treeish revision

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

Examples:

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

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

Raises AssertionError If we are a bare repository

`git.repo.`**is_git_dir**(*d*)

This is taken from the git setup.c:is_git_directory function.

`git.repo.`**touch**(*filename*)

3.16 Stats

class `git.stats.`**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
```

`files`

`total`

3.17 Utils

class `git.utils.`**`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

class `git.utils.`**`ConcurrentWriteOperation`**(*file_path*)

This class facilitates a safe write operation to a file on disk such that we:

- lock the original file
- write to a temporary file
- rename temporary file back to the original one on close
- unlock the original file

This type handles error correctly in that it will assure a consistent state on destruction

class `git.utils.`**`Iterable`**

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

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

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

class `git.utils.`**`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.

class `git.utils.`**`LazyMixin`**

Base class providing an interface to lazily retrieve attribute values upon first access. If slots are used, memory will only be reserved once the attribute is actually accessed and retrieved the first time. All future accesses will return the cached value as stored in the Instance's dict or slot.

class `git.utils.`**`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

class `git.utils.`**`SHA1Writer`**(*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.

Note: Based on the dulwich project

`close`()

`f`

`sha1`

`tell`()

`write`(*data*)

`write_sha`()

`git.utils.`**`join_path`**(*a*, **p*)

Join path tokens together similar to os.path.join, but always use '/' instead of possibly '' on windows.

`git.utils.`**`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 ''

`git.utils.`**`make_sha`**(*source=''*)

A python2.4 workaround for the sha/hashlib module fiasco

Note From the dulwich project

`git.utils.`**`to_native_path`**(*path*)

`git.utils.`**`to_native_path_linux`**(*path*)

`git.utils.`**`to_native_path_windows`**(*path*)

CHAPTER 4

Roadmap

The full list of milestones including associated tasks can be found on lighthouse: http://byronimo.lighthouseapp.com/projects/51787-gitpython/milestones

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index