GitPython Documentation

Release 0.1.7

Michael Trier

November 19, 2014

Contents

Contents:

CHAPTER 1

Overview / Install

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

GitPython is a port of the grit library in Ruby created by Tom Preston-Werner and Chris Wanstrath.

1.1 Requirements

- Git tested with 1.5.3.7
- Python Nose - used for running the tests
- Mock by Michael Foord used for tests. Requires 0.5 or higher

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, which can be browsed at:

http://gitorious.org/git-python

and cloned from:

git://gitorious.org/git-python/mainline.git

1.6 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. Once you have created a repository object, you can traverse it to find parent commit(s), trees, blobs, etc.

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")
```

2.2 Getting a list of commits

From the `Repo` object, you can get a list of `Commit` objects.

```
>>> repo.commits()
[<git.Commit "207c0c4418115df0d30820ab1a9acd2ea4bf4431">,
 <git.Commit "a91c45eee0b41bf3cdaad3418ca3850664c4a4b4">,
 <git.Commit "e17c7e11aed9e94d2159e549a99b966912ce1091">,
 <git.Commit "bd795df2d0e07d10e0298670005c0e9d9a5ed867">]
```

Called without arguments, `Repo.commits` returns a list of up to ten commits reachable by the master branch (starting at the latest commit). You can ask for commits beginning at a different branch, commit, tag, etc.

```
>>> repo.commits('mybranch')
>>> repo.commits('40d3057d09a7a4d61059bca9dca5ae698de58cbe')
>>> repo.commits('v0.1')
```

You can specify the maximum number of commits to return.

```
>>> repo.commits('master', max_count=100)
```

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

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

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

2.3 The Commit object

Commit objects contain information about a specific commit.

```
>>> head = repo.commits()[0]

>>> head.id
'207c0c4418115df0d30820ab1a9acd2ea4bf4431'

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

>>> head.tree
<git.Tree "563413aedbeda425d8d9dcbb744247d0c3e8a0ac">

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

>>> head.authored_date
(2008, 5, 7, 5, 0, 56, 2, 128, 0)

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

>>> head.committed_date
(2008, 5, 7, 5, 0, 56, 2, 128, 0)

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

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

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

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

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

```
>>> repo.commits()[0].parents[0].parents[0].parents[0]
```

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

2.4 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.commits()[0].tree
<git.Tree "a006b5b1a8115185a228b7514cdcd46fed90dc92">

>>> tree.id
'a006b5b1a8115185a228b7514cdcd46fed90dc92'
```

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

```
>>> contents = tree.values()
[<git.Blob "6a91a439ea968bf2f5ce8bb1cd8ddf5bf2cad6c7">,
 <git.Blob "e69de29bb2d1d6434b8b29ae775ad8c2e48c5391">,
 <git.Tree "eaa0090ec96b054e425603480519e7cf587adfc3">,
 <git.Blob "980e72ae16b5378009ba5dfd6772b59fe7ccd2df">]
```

The tree is implements a dictionary protocol so it can be used and acts just like a dictionary with some additional properties.

```
>>> tree.items()
[('lib', <git.Tree "310ebc9a0904531438bdde831fd6a27c6b6be58e">),
 ('LICENSE', <git.Blob "6797c1421052efe2ded9efdbb498b37aeae16415">),
 ('doc', <git.Tree "a58386dd101f6eb7f33499317e5508726dfd5e4f">),
 ('MANIFEST.in', <git.Blob "7da4e346bb0a682e99312c48a1f452796d3fb988">),
 ('.gitignore', <git.Blob "6870991011cc8d9853a7a8a6f02061512c6a8190">),
 ('test', <git.Tree "c6f6ee37d328987bc6fb47a33fed16c7886df857">),
 ('VERSION', <git.Blob "9faa1b7a7339db85692f91ad4b922554624a3ef7">),
 ('AUTHORS', <git.Blob "9f649ef5448f9666d78356a2f66ba07c5fb27229">),
 ('README', <git.Blob "9643dcf549f34fbd09503d4c941a5d04157570fe">),
 ('ez_setup.py', <git.Blob "3031ad0d119bd5010648cf8c038e2bbe21969ecb">),
 ('setup.py', <git.Blob "271074302aee04eb0394a4706c74f0c2eb504746">),
 ('CHANGES', <git.Blob "0d236f3d9f20d5e5db86daefe1e3ba1ce68e3a97">)]
```

This tree contains three `Blob` objects and one `Tree` object. The trees are subdirectories and the blobs are files. Trees below the root have additional attributes.

```
>>> contents = tree["lib"]
<git.Tree "c1c7214dde86f76bc3e18806ac1f47c38b2b7a3">

>>> contents.name
'test'

>>> contents.mode
'040000'
```

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">
```

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">
```

2.5 The Blob object

A blob represents a file. Trees often contain blobs.

```
>>> blob = tree['urls.py']
<git.Blob "b19574431a073333ea09346eafd64e7b1908ef49">
```

A blob has certain attributes.

```
>>> blob.name
'urls.py'

>>> blob.mode
'100644'

>>> blob.mime_type
'text/x-python'

>>> blob.size
415
```

You can get the data of a blob as a string.

```
>>> blob.data
"from django.conf.urls.defaults import *\nfrom django.conf..."
```

You can also get a blob directly from the repo if you know its name.

```
>>> repo.blob("b19574431a073333ea09346eafd64e7b1908ef49")
<git.Blob "b19574431a073333ea09346eafd64e7b1908ef49">
```

2.6 What Else?

There is more stuff in there, like the ability to tar or gzip repos, stats, log, blame, and probably a few other things. Additionally calls to the git instance are handled through a `__getattr__` construct, which makes available any git commands directly, with a nice conversion of Python dicts to command line parameters.

Check the unit tests, they're pretty exhaustive.

CHAPTER 3

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.

classmethod from_string(*string*)

Create an Actor from a string.

str is the string, which is expected to be in regular git format

Format John Doe <jdoe@example.com>

Returns Actor

3.2 Blob

class git.blob.**Blob**(*repo*, *id*, *mode=None*, *name=None*)

A Blob encapsulates a git blob object

DEFAULT_MIME_TYPE = 'text/plain'

basename

Returns The basename of the Blobs file name

classmethod blame(*repo*, *commit*, *file*)

The blame information for the given file at the given commit

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.

data

The binary contents of this blob.

Returns str

NOTE The data will be cached after the first access.

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.

size
The size of this blob in bytes

Returns int

NOTE The size will be cached after the first access

3.3 Git

class git.cmd.**Git**(*git_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.

execute(*command*, *istream=None*, *with_keep_cwd=False*, *with_extended_output=False*, *with_exceptions=True*, *with_raw_output=False*)
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(). GitPython uses get_work_tree() as its working directory by default and get_git_dir() for bare repositories.

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.

with_raw_output Whether to avoid stripping off trailing whitespace.

Returns:

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

get_dir

Returns Git directory we are working on

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

3.4 Commit

class git.commit.**Commit**(*repo*, *id*, *tree=None*, *author=None*, *authored_date=None*, *committer=None*, *committed_date=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.

classmethod actor(*line*)

Parse out the actor (author or committer) info

Returns [Actor, gmtime(acted at time)]

classmethod count(*repo*, *ref*, *path=''*)

Count the number of commits reachable from this ref

repo is the Repo

ref is the ref from which to begin (SHA1 or name)

path is an optional path

Returns int

classmethod diff(*repo*, *a*, *b=None*, *paths=None*)

Creates diffs between a tree and the index or between two trees:

repo is the Repo

a is a named commit

b is an optional named commit. Passing a list assumes you wish to omit the second named commit and limit the diff to the given paths.

paths is a list of paths to limit the diff to.

Returns git.Diff[]:

```
between tree and the index if only a is given
between two trees if a and b are given and are commits
```

diffs

Returns git.Diff[] Diffs between this commit and its first parent or all changes if this commit is the first commit and has no parent.

classmethod find_all(*repo*, *ref*, *path=''*, ***kwargs*)

Find all commits matching the given criteria. repo

is the Repo

ref is the ref from which to begin (SHA1 or name)

path is an optinal path, if set only Commits that include the path will be considered

kwargs optional keyword arguments to git where max_count is the maximum number of commits to fetch skip is the number of commits to skip

Returns git.Commit[]

id_abbrev

Returns First 7 bytes of the commit's sha id as an abbreviation of the full string.

classmethod **list_from_string**(*repo*, *text*)
Parse out commit information into a list of Commit objects

repo is the Repo

text is the text output from the git-rev-list command (raw format)

Returns git.Commit[]

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.

3.5 Diff

class git.diff.**Diff**(*repo*, *a_path*, *b_path*, *a_commit*, *b_commit*, *a_mode*, *b_mode*, *new_file*, *deleted_file*, *rename_from*, *rename_to*, *diff*)
A Diff contains diff information between two commits.

classmethod **list_from_string**(*repo*, *text*)

3.6 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.

3.7 Head

class git.head.**Head**(*name*, *commit*)
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.id
'1c09f116cbc2cb4100fb6935bb162daa4723f455'
```

classmethod **find_all**(*repo*, ***kwargs*)
Find all Heads in the repository

repo is the Repo

kwargs Additional options given as keyword arguments, will be passed to git-for-each-ref

Returns git.Head[]

List is sorted by committerdate

classmethod **from_string**(*repo*, *line*)
Create a new Head instance from the given string.

repo is the Repo

line is the formatted head information

Format:

```
name: [a-zA-Z_/]+
<null byte>
id: [0-9A-Fa-f]{40}
```

Returns git.Head

classmethod **list_from_string**(*repo*, *text*)
Parse out head information into a list of head objects

repo is the Repo

text is the text output from the git-for-each-ref command

Returns git.Head[]

3.8 Lazy

class git.lazy.**LazyMixin**

lazy_properties = []

3.9 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.

DAEMON_EXPORT_FILE = 'git-daemon-export-ok'

active_branch
The name of the currently active branch.

Returns str (the branch name)

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

archive_tar(*treeish='master'*, *prefix=None*)
Archive the given treeish

treeish is the treeish name/id (default 'master')

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

Examples:

```
>>> repo.archive_tar
<String containing tar archive>

>>> repo.archive_tar('a87ff14')
<String containing tar archive for commit a87ff14>

>>> repo.archive_tar('master', 'myproject/')
<String containing tar bytes archive, whose files are prefixed with 'myproject/'>
```

Returns str (containing bytes of tar archive)

archive_tar_gz(*treeish='master'*, *prefix=None*)
Archive and gzip the given treeish

treeish is the treeish name/id (default 'master')

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

Examples:

```
>>> repo.archive_tar_gz
<String containing tar.gz archive>

>>> repo.archive_tar_gz('a87ff14')
<String containing tar.gz archive for commit a87ff14>

>>> repo.archive_tar_gz('master', 'myproject/')
<String containing tar.gz archive and prefixed with 'myproject/'>
```

Returns str (containing the bytes of tar.gz archive)

blob(*id*)
The Blob object for the given id

id is the SHA1 id of the blob

Returns `git.Blob`

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

Returns `git.Head[]`

commit(*id*, *path=''*)
The Commit object for the specified id

id is the SHA1 identifier of the commit

path is an optional path, if set the returned commit must contain the path.

Returns `git.Commit`

commit_count (*start='master'*, *path=''*)
The number of commits reachable by the given branch/commit

start is the branch/commit name (default 'master')

path is an optional path Commits that do not contain the path will not contribute to the count.

Returns `int`

commit_deltas_from (*other_repo*, *ref='master'*, *other_ref='master'*)
Returns a list of commits that is in `other_repo` but not in self

Returns git.Commit[]

commit_diff (*commit*)

The commit diff for the given commit `commit` is the commit name/id

Returns `git.Diff[]`

commits (*start='master'*, *path=''*, *max_count=10*, *skip=0*)
A list of Commit objects representing the history of a given ref/commit

start

is the branch/commit name (default 'master')

path is an optional path to limit the returned commits to Commits that do not contain that path will not be returned.

max_count

is the maximum number of commits to return (default 10)

skip is the number of commits to skip (default 0) which will effectively move your commit-window by the given number.

Returns `git.Commit[]`

commits_between (*frm*, *to*)
The Commits objects that are reachable via `to` but not via `frm` Commits are returned in chronological order.

from is the branch/commit name of the younger item

to is the branch/commit name of the older item

Returns `git.Commit[]`

commits_since (*start='master'*, *path=''*, *since='1970-01-01'*)
The Commits objects that are newer than the specified date. Commits are returned in chronological order.

start is the branch/commit name (default 'master')

path is an optional path to limit the returned commits to.

since is a string representing a date/time

Returns `git.Commit[]`

classmethod create (*path*, *mkdir=True*, ***kwargs*)
Initialize a bare git repository at the given path

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

mkdir if specified will create the repository directory if it doesn't already exists. Creates the directory with a mode=0755.

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

Examples:

```
git.Repo.init_bare('/var/git/myrepo.git')
```

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

daemon_export
If True, git-daemon may export this repository

description
the project's description

diff(*a*, *b*, **paths*)
The diff from commit `a` to commit `b`, optionally restricted to the given file(s)

a is the base commit

b is the other commit

paths is an optional list of file paths on which to restrict the diff

Returns `str`

fork_bare(*path*, ***kwargs*)
Fork a bare git repository from this repo

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 forked repo)

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

Returns `git.Head[]`

classmethod init_bare(*path*, *mkdir=True*, ***kwargs*)
Initialize a bare git repository at the given path

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

mkdir if specified will create the repository directory if it doesn't already exists. Creates the directory with a mode=0755.

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

Examples:

```
git.Repo.init_bare('/var/git/myrepo.git')
```

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

is_dirty
Return the status of the index.

Returns `True`, if the index has any uncommitted changes, otherwise `False`

NOTE Working tree changes that have not been staged will not be detected !

log (*commit='master'*, *path=None*, ***kwargs*)
The Commit for a treeish, and all commits leading to it.

kwargs keyword arguments specifying flags to be used in git-log command, i.e.: max_count=1 to limit the amount of commits returned

Returns `git.Commit[]`

tags
A list of `Tag` objects that are available in this repo

Returns `git.Tag[]`

tree (*treeish='master'*)
The Tree object for the given treeish reference

treeish is the reference (default 'master')

Examples:

```
repo.tree('master')
```

Returns `git.Tree`

3.10 Stats

class `git.stats.`**Stats** (*repo*, *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
```

classmethod list_from_string (*repo*, *text*)
Create a Stat object from output retrieved by git-diff.

Returns git.Stat

3.11 Tag

class git.tag.**Tag**(*name*, *commit*)

classmethod find_all(*repo*, ***kwargs*)
Find all Tags in the repository

repo is the Repo

kwargs Additional options given as keyword arguments, will be passed to git-for-each-ref

Returns git.Tag[]

List is sorted by committerdate

classmethod from_string(*repo*, *line*)
Create a new Tag instance from the given string.

repo is the Repo

line is the formatted tag information

Format:

```
name: [a-zA-Z_/]+
<null byte>
id: [0-9A-Fa-f]{40}
```

Returns git.Tag

classmethod list_from_string(*repo*, *text*)
Parse out tag information into an array of Tag objects

repo is the Repo

text is the text output from the git-for-each command

Returns git.Tag[]

3.12 Tree

class git.tree.**Tree**(*repo*, *id*, *mode=None*, *name=None*)

basename

static content_from_string(*repo*, *text*)
Parse a content item and create the appropriate object

repo

is the Repo

text is the single line containing the items data in *git ls-tree* format

Returns git.Blob or git.Tree

get(*key*)

items()

keys()

values()

3.13 Utils

git.utils.**dashify**(*string*)

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

git.utils.**touch**(*filename*)

CHAPTER 4

Indices and tables

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

g