GitPython Documentation

Release 0.1.6

Michael Trier

February 01, 2009

CONTENTS

Contents:

CHAPTER

ONE

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 0.4

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/projects/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

TWO

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

THREE

API Reference

3.1 Actor

class **`Actor`** (*name, email*)

`from_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 **`Blob`** (*repo, id, mode=None, name=None*)

`basename`

`blame`
The blame information for the given file at the given commit

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

`data`
The binary contents of this blob.

Returns str

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

Returns str

`size`
The size of this blob in bytes

Returns int

3.3 Git

class **`Git`** (*git_dir*)
The Git class manages communication with the Git binary

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

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(output)) # extended_output = True

get_dir

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

3.4 Commit

class **Commit** (*repo, id, tree=None, author=None, authored_date=None, committer=None, committed_date=None, message=None, parents=None*)

actor
Parse out the actor (author or committer) info

Returns [str (actor name and email), time (acted at time)]

count
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 optinal path

Returns int

diff
Show diffs 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.

Returns git.Diff[]

diffs

find_all
Find all commits matching the given criteria. `repo`

Unexpected indentation.

is the Repo

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

path is an optinal path

options is a Hash of optional 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`

`list_from_string`
Parse out commit information into a list of Commit objects

`repo` is the Repo

`text` is the text output from the git command (raw format)

Returns git.Commit[]

`stats`

`summary`

3.5 Diff

class `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.

`list_from_string`

3.6 Errors

exception `GitCommandError`

exception `InvalidGitRepositoryError`

exception `NoSuchPathError`

3.7 Head

class `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'
```

`find_all`
Find all Heads

repo is the Repo

kwargs is a dict of options

Returns git.Head[]

`from_string`
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

`list_from_string`
Parse out head information into an array of baked head objects

`repo` is the Repo

`text` is the text output from the git command

Returns git.Head[]

3.8 Lazy

class **`LazyMixin`**()

3.9 Repo

class **`Repo`**(*path=None*)

`active_branch`
The name of the currently active branch.

Returns str (the branch name)

`alternates`
The list of alternates for this repo

Returns list[str] (pathnames of alternates)

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

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

`prefix` is the optional prefix

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 archive and prefixed with 'myproject/'>
```

Returns str (containing 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

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 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 optinal 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 optinal path

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**

`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

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

`skip` is the number of commits to skip (default 0)

Returns `git.Commit[]`

`commits_between`(*frm, to, path=''*)
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

`path` is an optional path

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 optinal path

since is a string represeting a date/time

Returns `git.Commit[]`

create
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 is any 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
git-daemon export of 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

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)

options is any additional options 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[]`

init_bare
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 is any 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 working directory.

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

log (*commit='master', path=None, **kwargs*)

The commit log for a treeish

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 **Stats** (*repo, total, files*)

list_from_string

3.11 Tag

class **Tag** (*name, commit*)

find_all

Find all Tags

repo is the Repo

kwargs is a dict of options

Returns `git.Tag[]`

from_string

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`

list_from_string

Parse out tag information into an array of baked Tag objects

repo is the Repo

text is the text output from the git command

Returns `git.Tag[]`

3.12 Tree

class **`Tree`**(*repo, id, mode=None, name=None*)

`basename`

`content_from_string`

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

`dashify`(*string*)

`is_git_dir`(*d*)

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

`touch`(*filename*)

CHAPTER

FOUR

Indices and tables

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

INDEX