GitPython Documentation

Release 3.1.18

Michael Trier

Jun 18, 2021

Contents

CHAPTER 1

Overview / Install

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

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

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

1.1 Requirements

- Python >= 3.6
- **Git 1.7.0 or newer** It should also work with older versions, but it may be that some operations involving remotes will not work as expected.
- GitDB - a pure python git database implementation

1.2 Installing GitPython

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

```
# pip install GitPython
```

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

- install pip
- pypi

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

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

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

1.3 Limitations

1.3.1 Leakage of System Resources

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

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

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

1.4 Getting Started

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

1.5 API Reference

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

1.6 Source Code

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

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

and cloned using:

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

Initialize all submodules to obtain the required dependencies with:

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

Finally verify the installation by running unit tests:

```
$ python -m unittest
```

1.7 Questions and Answers

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

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

1.8 Issue Tracker

The issue tracker is hosted by GitHub:

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

1.9 License Information

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

CHAPTER 2

GitPython Tutorial

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

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

2.1 Meet the Repo type

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

```
from git import Repo

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

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

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

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

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

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

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

Clone from existing repositories or initialize new empty ones.

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

Archive the repository contents to a tar file.

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

2.1.1 Advanced Repo Usage

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

Query relevant repository paths ...

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

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

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

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

You can also create new heads ...

```
new_branch = cloned_repo.create_head('feature')               # create a new branch ..
↪.
assert cloned_repo.active_branch != new_branch                # which wasn't checked␣
↪out yet ...
self.assertEqual(new_branch.commit, cloned_repo.active_branch.commit)  # pointing to␣
↪the checked-out commit
```

(continues on next page)

(continued from previous page)

```
# It's easy to let a branch point to the previous commit, without affecting anything␣
↪else
# Each reference provides access to the git object it points to, usually commits
assert new_branch.set_commit('HEAD~1').commit == cloned_repo.active_branch.commit.
↪parents[0]
```

... and tags ...

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

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

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

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

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

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

```
from git import RemoteProgress

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

self.assertEqual(len(cloned_repo.remotes), 1)                    # we have been␣
↪cloned, so should be one remote
self.assertEqual(len(bare_repo.remotes), 0)                      # this one was just␣
↪initialized
origin = bare_repo.create_remote('origin', url=cloned_repo.working_tree_dir)
```

(continues on next page)

(continued from previous page)

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

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

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

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

# prepare a merge
master = cloned_repo.heads.master                                      # right-hand side is ahead of us, in the future
merge_base = cloned_repo.merge_base(new_branch, master)                # allows for a three-way merge
cloned_repo.index.merge_tree(master, base=merge_base)                  # write the merge result into index
cloned_repo.index.commit("Merged past and now into future ;)",
                         parent_commits=(new_branch.commit, master.commit))

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

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

```
# create a new submodule and check it out on the spot, setup to track master branch of `bare_repo`
# As our GitPython repository has submodules already that point to GitHub, make sure we don't
# interact with them
```

(continues on next page)

(continued from previous page)

```
for sm in cloned_repo.submodules:
    assert not sm.remove().exists()                    # after removal, the sm doesn't␣
↪exist anymore
sm = cloned_repo.create_submodule('mysubrepo', 'path/to/subrepo', url=bare_repo.git_
↪dir, branch='master')

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

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

2.2 Examining References

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

```
import git
repo = git.Repo.clone_from(self._small_repo_url(), os.path.join(rw_dir, 'repo'),␣
↪branch='master')

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

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

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

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

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

Access the `reflog` easily.

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

2.3 Modifying References

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

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

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

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

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

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

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 binary SHA1 hash, being 20 bytes in size, or 40 bytes in hexadecimal notation.

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

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

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

Common fields are ...

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

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

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

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

```
hct.blobs[0].data_stream.read()         # stream object to read data from
hct.blobs[0].stream_data(open(os.path.join(rw_dir, 'blob_data'), 'wb'))  # write data␣
↪to 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('v0.8.1')
repo.commit('HEAD~10')
```

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

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

A commit object carries all sorts of meta-data

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

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

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

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

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

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

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
assert len(tree.hexsha) == 40
```

Once you have a tree, you can get its contents

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

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

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

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

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

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

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

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

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

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

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. Modify the index with ease

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

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

from git import Actor
author = Actor("An author", "author@example.com")
committer = Actor("A committer", "committer@example.com")
# commit by commit message and author and committer
index.commit("my commit message", author=author, committer=committer)
```

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

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

2.8 Handling Remotes

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

```
empty_repo = git.Repo.init(os.path.join(rw_dir, 'empty'))
origin = empty_repo.create_remote('origin', repo.remotes.origin.url)
assert origin.exists()
assert origin == empty_repo.remotes.origin == empty_repo.remotes['origin']
origin.fetch()                  # assure we actually have data. fetch() returns␣
↪useful information
# Setup a local tracking branch of a remote branch
empty_repo.create_head('master', origin.refs.master)  # create local branch "master"␣
↪from remote "master"
empty_repo.heads.master.set_tracking_branch(origin.refs.master)  # set local "master"␣
↪to track remote "master
empty_repo.heads.master.checkout()  # checkout local "master" to working tree
# Three above commands in one:
empty_repo.create_head('master', origin.refs.master).set_tracking_branch(origin.refs.
↪master).checkout()
# rename remotes
origin.rename('new_origin')
# push and pull behaves similarly to `git push|pull`
origin.pull()
origin.push()
# assert not empty_repo.delete_remote(origin).exists()     # create and delete remotes
```

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

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

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

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

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

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

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

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

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

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

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

2.9 Submodule Handling

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

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

assert len(sms) == 1
sm = sms[0]
self.assertEqual(sm.name, 'gitdb')                          # git-python has gitdb as␣
↪single submodule ...
self.assertEqual(sm.children()[0].name, 'smmap')            # ... which has smmap as␣
↪single submodule

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

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

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

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

Additionally, GitPython adds functionality to track a specific branch, instead of just a commit. Supported by customized update methods, you are able to automatically update submodules to the latest revision available in the remote repository, as well as to keep track of changes and movements of these submodules. To use it, set the name of the branch you want to track to the `submodule.$name.branch` option of the *.gitmodules* file, and use GitPython update methods on the resulting repository with the `to_latest_revision` parameter turned on. In the latter case, the sha of your submodule will be ignored, instead a local tracking branch will be updated to the respective remote branch automatically, provided there are no local changes. The resulting behaviour is much like the one of svn::externals, which can be useful in times.

2.10 Obtaining Diff Information

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

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

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

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

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

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

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

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

2.11 Switching Branches

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

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

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

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

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

2.12 Initializing a repository

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

```
import git

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

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

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

2.13 Using git directly

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

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

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

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

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

2.14 Object Databases

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

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

2.14.1 GitDB

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

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

2.14.2 GitCmdObjectDB

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

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

2.15 Git Command Debugging and Customization

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

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

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

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

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

2.16 And even more . . .

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

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

CHAPTER 3

API Reference

3.1 Version

git.**__version__**
Current GitPython version.

CHAPTER 4

Roadmap

The full list of milestones including associated tasks can be found on GitHub: https://github.com/gitpython-developers/GitPython/issues

Select the respective milestone to filter the list of issues accordingly.

CHAPTER 5

Changelog

5.1 3.1.18

- drop support for python 3.5 to reduce maintenance burden on typing. Lower patch levels of python 3.5 would break, too.

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/50?closed=1

5.2 3.1.17

- Fix issues from 3.1.16 (see https://github.com/gitpython-developers/GitPython/issues/1238)
- Fix issues from 3.1.15 (see https://github.com/gitpython-developers/GitPython/issues/1223)
- Add more static typing information

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/49?closed=1

5.3 3.1.16 (YANKED)

- Fix issues from 3.1.15 (see https://github.com/gitpython-developers/GitPython/issues/1223)
- Add more static typing information

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/48?closed=1

5.4 3.1.15 (YANKED)

- add deprectation warning for python 3.5

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/47?closed=1

5.5 3.1.14

- git.Commit objects now have a `replace` method that will return a copy of the commit with modified attributes.
- Add python 3.9 support
- Drop python 3.4 support

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/46?closed=1

5.6 3.1.13

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/45?closed=1

5.7 3.1.12

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/44?closed=1

5.8 3.1.11

Fixes regression of 3.1.10.

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/43?closed=1

5.9 3.1.10

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/42?closed=1

5.10 3.1.9

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/41?closed=1

5.11 3.1.8

- support for 'includeIf' in git configuration files
- tests are now excluded from the package, making it conisderably smaller

See the following for more details: https://github.com/gitpython-developers/gitpython/milestone/40?closed=1

5.12 3.1.7

- Fix tutorial examples, which disappeared in 3.1.6 due to a missed path change.

5.13 3.1.6

- Greatly reduced package size, see https://github.com/gitpython-developers/GitPython/pull/1031

5.14 3.1.5

- rollback: package size was reduced significantly not placing tests into the package anymore. See https://github.com/gitpython-developers/GitPython/issues/1030

5.15 3.1.4

- all exceptions now keep track of their cause
- package size was reduced significantly not placing tests into the package anymore.

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/39?closed=1

5.16 3.1.3

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/38?closed=1

5.17 3.1.2

- Re-release of 3.1.1, with known signature

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/37?closed=1

5.18 3.1.1

- support for PyOxidizer, which previously failed due to usage of *__file__*.

See the following for details: https://github.com/gitpython-developers/gitpython/milestone/36?closed=1

5.19 3.1.0

- Switched back to using gitdb package as requirement (gitdb#59)

5.20 3.0.9

- Restricted GitDB (gitdb2) version requirement to < 4
- Removed old nose library from test requirements

5.20.1 Bugfixes

- Changed to use UTF-8 instead of default encoding when getting information about a symbolic reference (#774)
- Fixed decoding of tag object message so as to replace invalid bytes (#943)

5.21 3.0.8

- Added support for Python 3.8
- Bumped GitDB (gitdb2) version requirement to > 3

5.21.1 Bugfixes

- Fixed Repo.__repr__ when subclassed (#968)
- Removed compatibility shims for Python < 3.4 and old mock library
- Replaced usage of deprecated unittest aliases and Logger.warn
- Removed old, no longer used assert methods
- Replaced usage of nose assert methods with unittest

5.22 3.0.7

Properly signed re-release of v3.0.6 with new signature (See #980)

5.23 3.0.6

Note: There was an issue that caused this version to be released to PyPI without a signature
See the changelog for v3.0.7 and #980

5.23.1 Bugfixes

- Fixed warning for usage of environment variables for paths containing `$` or `%` (#832, #961)
- Added support for parsing Git internal date format (@<unix timestamp> <timezone offset>) (#965)
- Removed Python 2 and < 3.3 compatibility shims (#979)
- Fixed GitDB (gitdb2) requirement version specifier formatting in requirements.txt (#979)

5.24 3.0.5 - Bugfixes

see the following for details: https://github.com/gitpython-developers/gitpython/milestone/32?closed=1

5.25 3.0.4 - Bugfixes

see the following for details: https://github.com/gitpython-developers/gitpython/milestone/31?closed=1

5.26 3.0.3 - Bugfixes

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/30?closed=1

5.27 3.0.2 - Bugfixes

- fixes an issue with installation

5.28 3.0.1 - Bugfixes and performance improvements

- Fix a performance regression which could make certain workloads 50% slower
- Add *currently_rebasing_on* method on *Repo*, see the PR
- Fix incorrect *requirements.txt* which could lead to broken installations, see this issue for details.

5.29 3.0.0 - Remove Python 2 support

Motivation for this is a patch which improves unicode handling when dealing with filesystem paths. Python 2 compatibility was introduced to deal with differences, and I thought it would be a good idea to 'just' drop support right now, mere 5 months away from the official maintenance stop of python 2.7.

The underlying motivation clearly is my anger when thinking python and unicode, which was a hassle from the start, at least in a codebase as old as GitPython, which totally doesn't handle encodings correctly in many cases.

Having migrated to using *Rust* exclusively for tooling, I still see that correct handling of encodings isn't entirely trivial, but at least *Rust* makes clear what has to be done at compile time, allowing to write software that is pretty much guaranteed to work once it compiles.

Again, my apologies if removing Python 2 support caused inconveniences, please see release 2.1.13 which returns it.

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/27?closed=1

or run have a look at the difference between tags v2.1.12 and v3.0.0: https://github.com/gitpython-developers/GitPython/compare/2.1.12...3.0.0.

5.30 2.1.15

- Fixed GitDB (gitdb2) requirement version specifier formatting in requirements.txt (Backported from #979)
- Restricted GitDB (gitdb2) version requirement to < 3 (#897)

5.31 2.1.14

- Fixed handling of 0 when transforming kwargs into Git command arguments (Backported from #899)

5.32 2.1.13 - Bring back Python 2.7 support

My apologies for any inconvenience this may have caused. Following semver, backward incompatible changes will be introduced in a minor version.

5.33 2.1.12 - Bugfixes and Features

- Multi-value support and interface improvements for Git configuration. Thanks to A. Jesse Jiryu Davis.

or run have a look at the difference between tags v2.1.11 and v2.1.12: https://github.com/gitpython-developers/GitPython/compare/2.1.11...2.1.12

5.34 2.1.11 - Bugfixes

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/26?closed=1

or run have a look at the difference between tags v2.1.10 and v2.1.11: https://github.com/gitpython-developers/GitPython/compare/2.1.10...2.1.11

5.35 2.1.10 - Bugfixes

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/25?closed=1

or run have a look at the difference between tags v2.1.9 and v2.1.10: https://github.com/gitpython-developers/GitPython/compare/2.1.9...2.1.10

5.36 2.1.9 - Dropping support for Python 2.6

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/24?closed=1

or run have a look at the difference between tags v2.1.8 and v2.1.9: https://github.com/gitpython-developers/GitPython/compare/2.1.8...2.1.9

5.37 2.1.8 - bugfixes

see the following for (most) details: https://github.com/gitpython-developers/gitpython/milestone/23?closed=1

or run have a look at the difference between tags v2.1.7 and v2.1.8: https://github.com/gitpython-developers/GitPython/compare/2.1.7...2.1.8

5.38 2.1.6 - bugfixes

- support for worktrees

5.39 2.1.3 - Bugfixes

All issues and PRs can be viewed in all detail when following this URL: https://github.com/gitpython-developers/GitPython/milestone/21?closed=1

5.40 2.1.1 - Bugfixes

All issues and PRs can be viewed in all detail when following this URL: https://github.com/gitpython-developers/GitPython/issues?q=is%3Aclosed+milestone%3A%22v2.1.1+-+Bugfixes%22

5.41 2.1.0 - Much better windows support!

Special thanks to @ankostis, who made this release possible (nearly) single-handedly. GitPython is run by its users, and their PRs make all the difference, they keep GitPython relevant. Thank you all so much for contributing !

5.41.1 Notable fixes

- The *GIT_DIR* environment variable does not override the *path* argument when initializing a *Repo* object anymore. However, if said *path* unset, *GIT_DIR* will be used to fill the void.

All issues and PRs can be viewed in all detail when following this URL: https://github.com/gitpython-developers/GitPython/issues?q=is%3Aclosed+milestone%3A%22v2.1.0+-+proper+windows+support%22

5.42 2.0.9 - Bugfixes

- *tag.commit* will now resolve commits deeply.
- *Repo* objects can now be pickled, which helps with multi-processing.
- *Head.checkout()* now deals with detached heads, which is when it will return the *HEAD* reference instead.
- *DiffIndex.iter_change_type(...)* produces better results when diffing

5.43 2.0.8 - Features and Bugfixes

- *DiffIndex.iter_change_type(...)* produces better results when diffing an index against the working tree.
- *Repo().is_dirty(...)* now supports the *path* parameter, to specify a single path by which to filter the output. Similar to *git status <path>*
- Symbolic refs created by this library will now be written with a newline character, which was previously missing.
- *blame()* now properly preserves multi-line commit messages.

- No longer corrupt ref-logs by writing multi-line comments into them.

5.44 2.0.7 - New Features

- ***IndexFile.commit(...,skip_hooks=False)* added. This parameter emulates the** behaviour of *–no-verify* on the command-line.

5.45 2.0.6 - Fixes and Features

- Fix: remote output parser now correctly matches refs with non-ASCII chars in them
- API: Diffs now have *a_rawpath*, *b_rawpath*, *raw_rename_from*, *raw_rename_to* properties, which are the raw-bytes equivalents of their unicode path counterparts.
- Fix: TypeError about passing keyword argument to string decode() on Python 2.6.
- Feature: setUrl API on Remotes

5.46 2.0.5 - Fixes

- Fix: parser of fetch info lines choked on some legitimate lines

5.47 2.0.4 - Fixes

- Fix: parser of commit object data is now robust against cases where commit object contains invalid bytes. The invalid characters are now replaced rather than choked on.
- Fix: non-ASCII paths are now properly decoded and returned in `.diff()` output
- Fix: *RemoteProgress* will now strip the ‘, ‘ prefix or suffix from messages.
- API: Remote.[fetch|push|pull](...) methods now allow the `progress` argument to be a callable. This saves you from creating a custom type with usually just one implemented method.

5.48 2.0.3 - Fixes

- Fix: bug in `git-blame --incremental` output parser that broken when commit messages contained `\r` characters
- Fix: progress handler exceptions are not caught anymore, which would usually just hide bugs previously.
- Fix: The *Git.execute* method will now redirect *stdout* to *devnull* if *with_stdout* is false, which is the intended behaviour based on the parameter’s documentation.

5.49 2.0.2 - Fixes

- Fix: source package does not include *.pyc files
- Fix: source package does include doc sources

5.50 2.0.1 - Fixes

- Fix: remote output parser now correctly matches refs with "@" in them

5.51 2.0.0 - Features

Please note that due to breaking changes, we have to increase the major version.

- **IMPORTANT**: This release drops support for python 2.6, which is officially deprecated by the python maintainers.
- **CRITICAL**: *Diff* objects created with patch output will now not carry the — and +++ header lines anymore. All diffs now start with the @@ header line directly. Users that rely on the old behaviour can now (reliably) read this information from the a_path and b_path properties without having to parse these lines manually.
- *Commit* now has extra properties *authored_datetime* and *committer_datetime* (to get Python datetime instances rather than timestamps)
- *Commit.diff()* now supports diffing the root commit via *Commit.diff(NULL_TREE)*.
- *Repo.blame()* now respects *incremental=True*, supporting incremental blames. Incremental blames are slightly faster since they don't include the file's contents in them.
- Fix: *Diff* objects created with patch output will now have their *a_path* and *b_path* properties parsed out correctly. Previously, some values may have been populated incorrectly when a file was added or deleted.
- Fix: diff parsing issues with paths that contain "unsafe" chars, like spaces, tabs, backslashes, etc.

5.52 1.0.2 - Fixes

- IMPORTANT: Changed default object database of *Repo* objects to *GitCmdObjectDB*. The pure-python implementation used previously usually fails to release its resources (i.e. file handles), which can lead to problems when working with large repositories.
- CRITICAL: fixed incorrect *Commit* object serialization when authored or commit date had timezones which were not divisiblej by 3600 seconds. This would happen if the timezone was something like *+0530* for instance.
- A list of all additional fixes can be found on GitHub
- CRITICAL: *Tree.cache* was removed without replacement. It is technically impossible to change individual trees and expect their serialization results to be consistent with what *git* expects. Instead, use the *IndexFile* facilities to adjust the content of the staging area, and write it out to the respective tree objects using *IndexFile.write_tree()* instead.

5.53 1.0.1 - Fixes

- A list of all issues can be found on GitHub

5.54 1.0.0 - Notes

This version is equivalent to v0.3.7, but finally acknowledges that GitPython is stable and production ready.

It follows the semantic version scheme, and thus will not break its existing API unless it goes 2.0.

5.55 0.3.7 - Fixes

- *IndexFile.add()* will now write the index without any extension data by default. However, you may override this behaviour with the new *write_extension_data* keyword argument.
  - Renamed *ignore_tree_extension_data* keyword argument in *IndexFile.write(...)* to *ignore_extension_data*
- If the git command executed during *Remote.push(...)|fetch(...)* returns with an non-zero exit code and GitPython didn't obtain any head-information, the corresponding *GitCommandError* will be raised. This may break previous code which expected these operations to never raise. However, that behavious is undesirable as it would effectively hide the fact that there was an error. See this issue for more information.
- If the git executable can't be found in the PATH or at the path provided by *GIT_PYTHON_GIT_EXECUTABLE*, this is made obvious by throwing *GitCommandNotFound*, both on unix and on windows.
  - Those who support **GUI on windows** will now have to set *git.Git.USE_SHELL = True* to get the previous behaviour.
- A list of all issues can be found on GitHub

5.56 0.3.6 - Features

- **DOCS**
  - special members like *__init__* are now listed in the API documentation
  - tutorial section was revised entirely, more advanced examples were added.
- **POSSIBLY BREAKING CHANGES**
  - As *rev_parse* will now throw *BadName* as well as *BadObject*, client code will have to catch both exception types.
  - Repo.working_tree_dir now returns None if it is bare. Previously it raised AssertionError.
  - IndexFile.add() previously raised AssertionError when paths where used with bare repository, now it raises InvalidGitRepositoryError
- Added *Repo.merge_base()* implementation. See the respective issue on GitHub
- *[include]* sections in git configuration files are now respected
- Added *GitConfigParser.rename_section()*
- Added *Submodule.rename()*
- A list of all issues can be found on GitHub

5.57 0.3.5 - Bugfixes

- push/pull/fetch operations will not block anymore
- diff() can now properly detect renames, both in patch and raw format. Previously it only worked when create_patch was True.

- repo.odb.update_cache() is now called automatically after fetch and pull operations. In case you did that in your own code, you might want to remove your line to prevent a double-update that causes unnecessary IO.
- *Repo(path)* will not automatically search upstream anymore and find any git directory on its way up. If you need that behaviour, you can turn it back on using the new *search_parent_directories=True* flag when constructing a *Repo* object.
- IndexFile.commit() now runs the *pre-commit* and *post-commit* hooks. Verified to be working on posix systems only.
- A list of all fixed issues can be found here: https://github.com/gitpython-developers/GitPython/issues?q=milestone%3A%22v0.3.5+-+bugfixes%22+

5.58 0.3.4 - Python 3 Support

- Internally, hexadecimal SHA1 are treated as ascii encoded strings. Binary SHA1 are treated as bytes.
- Id attribute of Commit objects is now *hexsha*, instead of *binsha*. The latter makes no sense in python 3 and I see no application of it anyway besides its artificial usage in test cases.
- **IMPORTANT**: If you were using the config_writer(), you implicitly relied on __del__ to work as expected to flush changes. To be sure changes are flushed under PY3, you will have to call the new *release()* method to trigger a flush. For some reason, __del__ is not called necessarily anymore when a symbol goes out of scope.
- The *Tree* now has a *.join('name')* method which is equivalent to *tree / 'name'*

5.59 0.3.3

- When fetching, pulling or pushing, and an error occurs, it will not be reported on stdout anymore. However, if there is a fatal error, it will still result in a GitCommandError to be thrown. This goes hand in hand with improved fetch result parsing.
- Code Cleanup (in preparation for python 3 support)
  - Applied autopep8 and cleaned up code
  - Using python logging module instead of print statements to signal certain kinds of errors

5.60 0.3.2.1

- Fix for #207

5.61 0.3.2

- Release of most recent version as non-RC build, just to allow pip to install the latest version right away.
- Have a look at the milestones (https://github.com/gitpython-developers/GitPython/milestones) to see what's next.

5.62 0.3.2 RC1

- **git** command wrapper
- Added `version_info` property which returns a tuple of integers representing the installed git version.
- Added GIT_PYTHON_GIT_EXECUTABLE environment variable, which can be used to set the desired git executable to be used. despite of what would be found in the path.
- **Blob** Type
- Added mode constants to ease the manual creation of blobs
- **IterableList**
- Added __contains__ and __delitem__ methods
- **More Changes**
- Configuration file parsing is more robust. It should now be able to handle everything that the git command can parse as well.
- The progress parsing was updated to support git 1.7.0.3 and newer. Previously progress was not enabled for the git command or only worked with ssh in case of older git versions.
- Parsing of tags was improved. Previously some parts of the name could not be parsed properly.
- The rev-parse pure python implementation now handles branches correctly if they look like hexadecimal sha's.
- GIT_PYTHON_TRACE is now set on class level of the Git type, previously it was a module level global variable.
- GIT_PYTHON_GIT_EXECUTABLE is a class level variable as well.

5.63 0.3.1 Beta 2

- Added **reflog support** (reading and writing)
  - New types: `RefLog` and `RefLogEntry`
  - Reflog is maintained automatically when creating references and deleting them
  - Non-intrusive changes to `SymbolicReference`, these don't require your code to change. They allow to append messages to the reflog.
    - `abspath` property added, similar to `abspath` of Object instances
    - `log()` method added
    - `log_append(...)` method added
    - `set_reference(...)` method added (reflog support)
    - `set_commit(...)` method added (reflog support)
    - `set_object(...)` method added (reflog support)
  - **Intrusive Changes** to `Head` type
  - `create(...)` method now supports the reflog, but will not raise `GitCommandError` anymore as it is a pure python implementation now. Instead, it raises `OSError`.
  - **Intrusive Changes** to `Repo` type

  - `create_head(...)` method does not support kwargs anymore, instead it supports a logmsg parameter

- Repo.rev_parse now supports the [ref]@{n} syntax, where *n* is the number of steps to look into the reference's past
- **BugFixes**
  - Removed incorrect ORIG_HEAD handling
- **Flattened directory** structure to make development more convenient.
- ---

  Note: This alters the way projects using git-python as a submodule have to adjust their sys.path to be able to import git-python successfully.

- Misc smaller changes and bugfixes

5.64 0.3.1 Beta 1

- Full Submodule-Support
- Added unicode support for author names. Commit.author.name is now unicode instead of string.
- Head Type changes
- config_reader() & config_writer() methods added for access to head specific options.
- tracking_branch() & set_tracking_branch() methods added for easy configuration of tracking branches.

5.65 0.3.0 Beta 2

- Added python 2.4 support

5.66 0.3.0 Beta 1

5.66.1 Renamed Modules

- For consistency with naming conventions used in sub-modules like gitdb, the following modules have been renamed
  - git.utils -> git.util
  - git.errors -> git.exc
  - git.objects.utils -> git.objects.util

5.66.2 General

- Object instances, and everything derived from it, now use binary sha's internally. The 'sha' member was removed, in favor of the 'binsha' member. An 'hexsha' property is available for convenient conversions. They may only be initialized using their binary shas, reference names or revision specs are not allowed anymore.

- IndexEntry instances contained in IndexFile.entries now use binary sha's. Use the .hexsha property to obtain the hexadecimal version. The .sha property was removed to make the use of the respective sha more explicit.
- If objects are instantiated explicitly, a binary sha is required to identify the object, where previously any rev-spec could be used. The ref-spec compatible version still exists as Object.new or Repo.commit|Repo.tree respectively.
- The .data attribute was removed from the Object type, to obtain plain data, use the data_stream property instead.
- ConcurrentWriteOperation was removed, and replaced by LockedFD
- IndexFile.get_entries_key was renamed to entry_key
- IndexFile.write_tree: removed missing_ok keyword, its always True now. Instead of raising GitCommandError it raises UnmergedEntriesError. This is required as the pure-python implementation doesn't support the missing_ok keyword yet.
- diff.Diff.null_hex_sha renamed to NULL_HEX_SHA, to be conforming with the naming in the Object base class

5.67 0.2 Beta 2

- Commit objects now carry the 'encoding' information of their message. It wasn't parsed previously, and defaults to UTF-8
- Commit.create_from_tree now uses a pure-python implementation, mimicking git-commit-tree

5.68 0.2

5.68.1 General

- file mode in Tree, Blob and Diff objects now is an int compatible to definitions in the stat module, allowing you to query whether individual user, group and other read, write and execute bits are set.
- Adjusted class hierarchy to generally allow comparison and hash for Objects and Refs
- Improved Tag object which now is a Ref that may contain a tag object with additional Information
- id_abbrev method has been removed as it could not assure the returned short SHA's where unique
- removed basename method from Objects with path's as it replicated features of os.path
- from_string and list_from_string methods are now private and were renamed to _from_string and _list_from_string respectively. As part of the private API, they may change without prior notice.
- Renamed all find_all methods to list_items - this method is part of the Iterable interface that also provides a more efficients and more responsive iter_items method
- All dates, like authored_date and committer_date, are stored as seconds since epoch to consume less memory - they can be converted using time.gmtime in a more suitable presentation format if needed.
- Named method parameters changed on a wide scale to unify their use. Now git specific terms are used everywhere, such as "Reference" (ref) and "Revision" (rev). Previously multiple terms where used making it harder to know which type was allowed or not.
- Unified diff interface to allow easy diffing between trees, trees and index, trees and working tree, index and working tree, trees and index. This closely follows the git-diff capabilities.
- Git.execute does not take the with_raw_output option anymore. It was not used by anyone within the project and False by default.

5.68.2 Item Iteration

- Previously one would return and process multiple items as list only which can hurt performance and memory consumption and reduce response times. iter_items method provide an iterator that will return items on demand as parsed from a stream. This way any amount of objects can be handled.
- list_items method returns IterableList allowing to access list members by name

5.68.3 objects Package

- blob, tree, tag and commit module have been moved to new objects package. This should not affect you though unless you explicitly imported individual objects. If you just used the git package, names did not change.

5.68.4 Blob

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

5.68.5 GitCommand

- git.subcommand call scheme now prunes out None from the argument list, allowing to be called more comfortably as None can never be a valid to the git command if converted to a string.
- Renamed 'git_dir' attribute to 'working_dir' which is exactly how it is used

5.68.6 Commit

- 'count' method is not an instance method to increase its ease of use
- 'name_rev' property returns a nice name for the commit's sha

5.68.7 Config

- The git configuration can now be read and manipulated directly from within python using the GitConfigParser
- Repo.config_reader() returns a read-only parser
- Repo.config_writer() returns a read-write parser

5.68.8 Diff

- Members a a_commit and b_commit renamed to a_blob and b_blob - they are populated with Blob objects if possible
- Members a_path and b_path removed as this information is kept in the blobs
- Diffs are now returned as DiffIndex allowing to more quickly find the kind of diffs you are interested in

5.68.9 Diffing

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

5.68.10 Index

- A new Index class allows to read and write index files directly, and to perform simple two and three way merges based on an arbitrary index.

5.68.11 References

- References are object that point to a Commit
- SymbolicReference are a pointer to a Reference Object, which itself points to a specific Commit
- They will dynamically retrieve their object at the time of query to assure the information is actual. Recently objects would be cached, hence ref object not be safely kept persistent.

5.68.12 Repo

- Moved blame method from Blob to repo as it appeared to belong there much more.
- active_branch method now returns a Head object instead of a string with the name of the active branch.
- tree method now requires a Ref instance as input and defaults to the active_branch instead of master
- is_dirty now takes additional arguments allowing fine-grained control about what is considered dirty
- Removed the following methods:
  - 'log' method as it as effectively the same as the 'commits' method
  - 'commits_since' as it is just a flag given to rev-list in Commit.iter_items
  - 'commit_count' as it was just a redirection to the respective commit method
  - 'commits_between', replaced by a note on the iter_commits method as it can achieve the same thing
  - 'commit_delta_from' as it was a very special case by comparing two different repjrelated repositories, i.e. clones, git-rev-list would be sufficient to find commits that would need to be transferred for example.
  - 'create' method which equals the 'init' method's functionality
  - 'diff' - it returned a mere string which still had to be parsed
  - 'commit_diff' - moved to Commit, Tree and Diff types respectively
- Renamed the following methods:
  - commits to iter_commits to improve the performance, adjusted signature
  - init_bare to init, implying less about the options to be used
  - fork_bare to clone, as it was to represent general clone functionality, but implied a bare clone to be more versatile
  - archive_tar_gz and archive_tar and replaced by archive method with different signature
- 'commits' method has no max-count of returned commits anymore, it now behaves like git-rev-list
- The following methods and properties were added
  - 'untracked_files' property, returning all currently untracked files
  - 'head', creates a head object
  - 'tag', creates a tag object
  - 'iter_trees' method

  - 'config_reader' method
  - 'config_writer' method
  - 'bare' property, previously it was a simple attribute that could be written
- Renamed the following attributes
  - 'path' is now 'git_dir'
  - 'wd' is now 'working_dir'
- Added attribute
  - 'working_tree_dir' which may be None in case of bare repositories

5.68.13 Remote

- Added Remote object allowing easy access to remotes
- Repo.remotes lists all remotes
- Repo.remote returns a remote of the specified name if it exists

5.68.14 Test Framework

- Added support for common TestCase base class that provides additional functionality to receive repositories tests can also write to. This way, more aspects can be tested under real-world (un-mocked) conditions.

5.68.15 Tree

- former 'name' member renamed to path as it suits the actual data better
- added traverse method allowing to recursively traverse tree items
- deleted blob method
- added blobs and trees properties allowing to query the respective items in the tree
- now mimics behaviour of a read-only list instead of a dict to maintain order.
- content_from_string method is now private and not part of the public API anymore

5.69 0.1.6

5.69.1 General

- Added in Sphinx documentation.
- Removed ambiguity between paths and treeishs. When calling commands that accept treeish and path arguments and there is a path with the same name as a treeish git cowardly refuses to pick one and asks for the command to use the unambiguous syntax where '–' separates the treeish from the paths.
- `Repo.commits`, `Repo.commits_between`, `Repo.commits_since`, `Repo.commit_count`, `Repo.commit`, `Commit.count` and `Commit.find_all` all now optionally take a path argument which constrains the lookup by path. This changes the order of the positional arguments in `Repo.commits` and `Repo.commits_since`.

5.69.2 Commit

- `Commit.message` now contains the full commit message (rather than just the first line) and a new property `Commit.summary` contains the first line of the commit message.
- Fixed a failure when trying to lookup the stats of a parentless commit from a bare repo.

5.69.3 Diff

- The diff parser is now far faster and also addresses a bug where sometimes b_mode was not set.
- Added support for parsing rename info to the diff parser. Addition of new properties `Diff.renamed`, `Diff.rename_from`, and `Diff.rename_to`.

5.69.4 Head

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

5.69.5 Repo

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

5.70 0.1.5

5.70.1 General

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

5.70.2 Blob

- No longer strip newlines from Blob data.

5.70.3 Commit

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

5.70.4 Repo

- Corrected problems with creating bare repositories.
- Repo.tree no longer accepts a path argument. Use:

```
>>> dict(k, o for k, o in tree.items() if k in paths)
```

- Made daemon export a property of Repo. Now you can do this:

```
>>> exported = repo.daemon_export
>>> repo.daemon_export = True
```

- Allows modifying the project description. Do this:

```
>>> repo.description = "Foo Bar"
>>> repo.description
'Foo Bar'
```

- Added a read-only property Repo.is_dirty which reflects the status of the working directory.
- Added a read-only Repo.active_branch property which returns the name of the currently active branch.

5.70.5 Tree

- Switched to using a dictionary for Tree contents since you will usually want to access them by name and order is unimportant.
- Implemented a dictionary protocol for Tree objects. The following:

  child = tree.contents['grit']

  becomes:

  child = tree['grit']

- Made Tree.content_from_string a static method.

5.71 0.1.4.1

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

5.72 0.1.4

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

5.72.1 Commit

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

5.72.2 Git

- Renamed module to cmd.
- Removed shell escaping completely.
- Added support for `stderr`, `stdin`, and `with_status`.
- `git_dir` is now optional in the constructor for `git.Git`. Git now falls back to `os.getcwd()` when git_dir is not specified.
- add a `with_exceptions` keyword argument to git commands. `GitCommandError` is raised when the exit status is non-zero.
- add support for a `GIT_PYTHON_TRACE` environment variable. `GIT_PYTHON_TRACE` allows us to debug GitPython's usage of git through the use of an environment variable.

5.72.3 Tree

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

5.72.4 Repo

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

5.73 0.1.2

5.73.1 Tree

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

5.74 0.1.1

Fixed up some urls because I'm a moron

5.75 0.1.0

initial release

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

G