
PyDriller Documentation

Release 1.0

Davide Spadini

Jun 07, 2018

Contents

1	Overview / Install	1
1.1	Requirements	1
1.2	Installing PyDriller	1
1.3	Source Code	2
2	Getting Started	3
3	Configuration	5
3.1	Selecting the Commit Range	5
3.2	Filtering commits	6
3.3	Threads	6
4	Commit Object	9
5	Modifications	11
6	Git Repository	13
7	API Reference	15
7.1	GitRepository	15
7.2	RepositoryMining	17
7.3	Commit	18
7.4	Developer	20
8	Indices and tables	21
	Python Module Index	23

PyDriller is a Python framework that helps developers on mining software repositories. Born as a fork of [RepoDriller](#), with PyDriller you can easily extract information from any Git repository, such as commits, developers, modifications, diffs, and source codes, and quickly export CSV files.

1.1 Requirements

- [Python](#) 3.4 or newer
- [Git](#)
- [GitPython](#) - python library for Git
- [pytest](#) - used for running the tests
- [dateutil](#) used for handling dates
- [psutil](#) used for tests

1.2 Installing PyDriller

Installing PyDriller is easily done using [pip](#). Assuming it is installed, just run the following from the command-line:

```
# pip install pydriller
```

This command will download the latest version of GitPython from the [Python Package Index](#) and install it to your system. This will also install the necessary dependencies.

1.3 Source Code

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

- <https://github.com/ishepard/pydriller>

and cloned using:

```
$ git clone https://github.com/ishepard/pydriller
$ cd pydriller
```

Optionally (but suggested), make use of virtualenv:

```
$ virtualenv -p python3 venv
$ source venv/bin/activate
```

Install the requirements:

```
$ pip install -r requirements
```

and run the tests using pytest:

```
$ pytest
```

CHAPTER 2

Getting Started

Using PyDriller is very simple. You only need to create *RepositoryMining*: this class will receive in input the path to the repository and return a generator that iterates over the commits. For example:

```
for commit in RepositoryMining('path/to/the/repo').traverse_commits():
    print('Hash {}, author {}'.format(commit.hash, commit.author.name))
```

will print the name of the developers for each commit. It's simple, isn't it?

Inside *RepositoryMining*, you will have to configure which projects to analyze, for which commits, for which dates etc. For all the possible configurations, have a look at [Configuration](#).

Let's make another example: print all the modified files for every commit. This does the magic:

```
for commit in RepositoryMining('path/to/the/repo').traverse_commits():
    for modification in commit.modifications:
        print('Author {} modified {} in commit {}'.format(commit.author.name,
↪modification.filename, commit.hash))
```

That's it! It's simple, isn't it?

Behind the scenes, PyDriller opens the Git repository and extracts all the necessary information. Then, the framework returns a generator that can iterate over the commits.

Ok, let's try a more difficult problem: let's count how many commits contain the word “bug” in the message. We can do it like this:

```
count = 0
for commit in RepositoryMining('path/to/the/repo').traverse_commits():
    if 'bug' in commit.msg:
        count += 1
```

I bet you weren't expecting it to be that simple, right?

One of the main advantage of using PyDriller to mine software repositories, is that is highly configurable. Let's start with selecting which commit to analyze.

3.1 Selecting the Commit Range

By default, PyDriller executes the visitor for all the commits in the repository. However, filters can be applied to *RepositoryMining* to visit *only specific* commits.

- *single: str*: single hash of the commit. The visitor will be called only on this commit

FROM:

- *since: datetime*: only commits after this date will be analyzed
- *from_commit: str*: only commits after this commit hash will be analyzed
- *from_tag: str*: only commits after this commit tag will be analyzed

TO:

- *to: datetime*: only commits up to this date will be analyzed
- *to_commit: str*: only commits up to this commit hash will be analyzed
- *to_tag: str*: only commits up to this commit tag will be analyzed

Examples:

```
# Analyze single commit
RepositoryMining('path/to/the/repo', single='6411e3096dd2070438a17b225f44475136e54e3a
↪').traverse_commits()

# Since 8/10/2016
RepositoryMining('path/to/the/repo', since=datetime(2016, 10, 8, 17, 0, 0)).traverse_
↪commits()
```

(continues on next page)

(continued from previous page)

```
# Between 2 dates
dt1 = datetime(2016, 10, 8, 17, 0, 0)
dt2 = datetime(2016, 10, 8, 17, 59, 0)
RepositoryMining('path/to/the/repo', since=dt1, to=dt2).traverse_commits()

# Between tags
from_tag = 'tag1'
to_tag = 'tag2'
RepositoryMining('path/to/the/repo', from_tag=from_tag, to_tag=to_tag).traverse_
↳ commits()

# Up to a date
dt1 = datetime(2016, 10, 8, 17, 0, 0, tzinfo=to_zone)
RepositoryMining('path/to/the/repo', to=dt1).traverse_commits()

# !!!!! ERROR !!!!! THIS IS NOT POSSIBLE
RepositoryMining('path/to/the/repo', from_tag=from_tag, from_commit=from_commit).
↳ traverse_commits()
```

IMPORTANT: it is **not** possible to configure more than one filter of the same category (for example, more than one *from*). It is also **not** possible to have the *single* filter together with other filters!

3.2 Filtering commits

PyDriller comes with a set of common commit filters that you can apply:

- *only_in_branches*: *List[str]*: only visits commits that belong to certain branches.
- *only_in_main_branch*: *bool*: only visits commits that belong to the main branch of the repository.
- *only_no_merge*: *bool*: only visits commits that are not merge commits.
- *only_modifications_with_file_types*: *List[str]*: only visits commits in which at least one modification was done in that file type, e.g., if you pass “.java”, then, the it will visit only commits in which at least one Java file was modified; clearly, it will skip other commits.

Examples:

```
# Only commits in main branch
RepositoryMining('path/to/the/repo', only_in_main_branch=True).traverse_commits()

# Only commits in main branch and no merges
RepositoryMining('path/to/the/repo', only_in_main_branch=True, only_no_merge=True).
↳ traverse_commits()

# Only commits that modified a java file
RepositoryMining('path/to/the/repo', only_modifications_with_file_types=['.java']).
↳ traverse_commits()
```

3.3 Threads

TODO

PyDriller can divide the work of analyzing a repository among multiple threads. If your machine has several cores, this can significantly improve performance. However, your *CommitVisitors must be thread-safe*, and your analysis must tolerate visiting commits in a relatively arbitrary order. By default, PyDriller uses only 1 thread.

```
RepositoryMining('path/to/repo/', mv, num_threads=5)
```


CHAPTER 4

Commit Object

A Commit contains a hash, a committer (name and email), an author (name, and email), a message, the authored date, committed date, a list of its parent hashes (if it's a merge commit, the commit has two parents), and the list of modification.

For example:

```
for commit in RepositoryMining('path/to/the/repo').traverse_commits():
    print(
        'Hash: {}'.format(commit.hash),
        'Author: {}'.format(commit.author.name),
        'Committer: {}'.format(commit.committer.name),
        'Author date: {}'.format(commit.author_date.strftime("%Y-%m-%d %H:%M:%S")),
        'Message: {}'.format(commit.msg),
        'Merge: {}'.format(commit.merge),
        'In main branch: {}'.format(commit.in_main_branch)
    )
```

Modifications

You can get the list of modified files, as well as their diffs and current source code. To that, all you have to do is to get the list of *Modifications* that exists inside *Commit*. A modification object has the following fields:

- *old_path*: old path of the file (can be *_None_* if the file is added)
- *new_path*: new path of the file (can be null if the file is deleted)
- *change_type*: type of the change
- *diff*: diff of the change
- *source_code*: source code of the file (can be null if the file is deleted)
- *added*: number of lines added
- *removed*: number of lines removed
- *filename*: filename

For example:

```
for commit in RepositoryMining('path/to/the/repo').traverse_commits():
    for m in commit.modifications:
        print(
            "Author {}".format(commit.author.name),
            " modified {}".format(m.filename),
            " with a change type of {}".format(m.change_type.name)
        )
```


CHAPTER 6

Git Repository

GitRepository is a wrapper for the most common utilities of Git. It receives in input the path to repository, and it takes care of the rest. For example, with GitRepository you can checkout a specific commit:

```
gr = GitRepository('test-repos/git-1/')
gr.checkout('a7053a4dcd627f5f4f213dc9aa002eb1caf926f8')
```

However, **be careful!** Git checkout changes the state of the repository on the hard disk, hence you should not use this command if other processes (maybe threads? or multiple repository mining?) read from the same repository.

GitRepository also contains a function to parse the a *diff*, very useful to obtain the list of lines added or deleted for future analysis. For example, if we run this:

```
diff = '@@ -2,6 +2,7 @@ aa'+\
      ' bb'+\
      '-cc'+\
      ' log.info(\"aa\")'+\
      '+log.debug(\"b\")'+\
      ' dd'+\
      ' ee'+\
      ' ff'
gr = GitRepository('test-repos/test1')
parsed_lines = gr.parse_diff(diff)

added = parsed_lines['added']
deleted = parsed_lines['deleted']

print('Added: {}'.format(added))      # result: Added: [(4, 'log.debug("b")')]
print('Deleted: {}'.format(deleted))  # result: Deleted: [(3, 'cc')]
```

the result is:

```
Added: [(4, 'log.debug("b")')]
Deleted: [(3, 'cc')]
```

Another very useful API (especially for researchers ;)) is the one that, given a commit, allows you to retrieve all the commits that last “touched” the modified lines of the file. Let’s see an example:

```
# commit abc modified line 1 of file A
# commit def modified line 2 of file A
# commit ghi modified line 3 of file A
# commit lmn deleted lines 1 and 2 of file A

gr = GitRepository('test-repos/test5')

commit = gr.getcommit('lmn')
buggy_commits = gr.get_commits_last_modified_lines(commit)
print(buggy_commits)          # result: (abc, def)
```

Since in commit **lmn** 2 lines were deleted (line 1 and 2), PyDriller can retrieve the commits in which those lines were last modified (in our example, commit **abc** and **def**).

Isn’t it cool? :)

Checkout the API reference of this class for the complete list of the available functions.

7.1 GitRepository

class pydriller.git_repository.**GitRepository** (*path: str*)

__init__ (*path: str*)

Init the Git Repository.

Parameters **path** (*str*) – path to the repository

__module__ = 'pydriller.git_repository'

checkout (*_hash: str*) → None

Checkout the repo at the specified commit. BE CAREFUL: this will change the state of the repo, hence it should *not* be used with more than 1 thread.

Parameters **_hash** – commit hash to checkout

files () → List[str]

Obtain the list of the files (excluding .git directory).

Returns List[str], the list of the files

get_change_sets () → List[pydriller.domain.commit.ChangeSet]

Return the list of all the commits in the repo.

Returns List[ChangeSet], the list of all the commits in the repo

get_commit (*commit_id: str*) → pydriller.domain.commit.Commit

Get the specified commit.

Parameters **commit_id** (*str*) – hash of the commit to analyze

Returns Commit

get_commit_from_tag (*tag: str*) → pydriller.domain.commit.Commit

Obtain the tagged commit.

Parameters `tag` (*str*) – the tag

Returns Commit commit: the commit the tag referred to

get_commits_last_modified_lines (*commit*: *pydriller.domain.commit.Commit*, *modification*: *pydriller.domain.commit.Modification = None*) → *Set[str]*

Given the Commit object, returns the set of commits that last “touched” the lines that are modified in the files included in the commit. It applies SZZ. The algorithm works as follow: (for every file in the commit)

1- obtain the diff

2- obtain the list of deleted lines

3- blame the file and obtain the commits were those lines were added

Can also be passed as parameter a single Modification, in this case only this file will be analyzed.

Parameters

- **commit** (*Commit*) – the commit to analyze
- **modification** (*Modification*) – single modification to analyze

Returns the set containing all the bug inducing commits

get_head () → *pydriller.domain.commit.ChangeSet*
Get the head commit.

Returns *ChangeSet* of the head commit

parse_diff (*diff*: *str*) → *Dict[str, List[Tuple[int, str]]]*

Given a diff, returns a dictionary with the added and deleted lines. The dictionary has 2 keys: “added” and “deleted”, each containing the corresponding added or deleted lines. For both keys, the value is a list of *Tuple (int, str)*, corresponding to (number of line in the file, actual line).

Parameters **diff** (*str*) – diff of the commit

Returns Dictionary

reset () → *None*

Reset the state of the repo, checking out the main branch and discarding local changes (-f option).

total_commits () → *int*

Calculate total number of commits.

Returns the total number of commits

7.2 RepositoryMining

```
class pydriller.repository_mining.RepositoryMining (path_to_repo: str, single: str =
                                                    None, since: datetime.datetime
                                                    = None, to: datetime.datetime
                                                    = None, from_commit: str =
                                                    None, to_commit: str = None,
                                                    from_tag: str = None, to_tag: str
                                                    = None, reversed_order: bool =
                                                    False, only_in_main_branch: bool
                                                    = False, only_in_branches:
                                                    List[str] = None,
                                                    only_modifications_with_file_types:
                                                    List[str] = None, only_no_merge:
                                                    bool = False)
```

```
__init__(path_to_repo: str, single: str = None, since: datetime.datetime = None, to: date-
time.datetime = None, from_commit: str = None, to_commit: str = None, from_tag: str
= None, to_tag: str = None, reversed_order: bool = False, only_in_main_branch: bool =
False, only_in_branches: List[str] = None, only_modifications_with_file_types: List[str] =
None, only_no_merge: bool = False)
```

Init a repository mining.

Parameters

- **path_to_repo** (*str*) – absolute path to the repository you have to analyze
- **single** (*str*) – hash of a single commit to analyze
- **since** (*datetime*) – starting date
- **to** (*datetime*) – ending date
- **from_commit** (*str*) – starting commit (only if *since* is None)
- **to_commit** (*str*) – ending commit (only if *to* is None)
- **from_tag** (*str*) – starting the analysis from specified tag (only if *since* and *from_commit* are None)
- **to_tag** (*str*) – ending the analysis from specified tag (only if *to* and *to_commit* are None)
- **reversed_order** (*bool*) – whether the commits should be analyzed in reversed order
- **only_in_main_branch** (*bool*) – whether only commits in main branch should be analyzed
- **only_in_branches** (*List[str]*) – only commits in these branches will be analyzed
- **only_modifications_with_file_types** (*List[str]*) – only modifications with that file types will be analyzed
- **only_no_merge** (*bool*) – if True, merges will not be analyzed

```
__module__ = 'pydriller.repository_mining'
```

```
traverse_commits() → Generator[[pydriller.domain.commit.Commit, NoneType], NoneType]
Analyze all the specified commits (all of them by default), returning a generator of commits.
```

7.3 Commit

```
class pydriller.domain.commit.ChangeSet (id: str, date: datetime.datetime)
```

```
    __init__ (id: str, date: datetime.datetime)
```

Light-weight version of the commit, storing only the hash and the date. Used for filter out commits before asking for more complex information (like diff and source code).

Parameters

- **id** (*str*) – hash of the commit
- **date** – date of the commit

```
    __module__ = 'pydriller.domain.commit'
```

```
class pydriller.domain.commit.Commit (commit: git.objects.commit.Commit, path: str,  
                                     main_branch: str)  $\rightarrow$  None
```

```
    __init__ (commit: git.objects.commit.Commit, path: str, main_branch: str)  $\rightarrow$  None
```

Create a commit object.

```
    __module__ = 'pydriller.domain.commit'
```

author

Return the author of the commit as a Developer object.

Returns author

author_date

Return the authored datetime.

Returns datetime author_datetime

author_timezone

Author timezone expressed in seconds from epoch.

Returns int timezone

branches

Return the set of branches that contain the commit.

Returns set(str) branches

committer

Return the committer of the commit as a Developer object.

Returns committer

committer_date

Return the committed datetime.

Returns datetime committer_datetime

committer_timezone

Author timezone expressed in seconds from epoch.

Returns int timezone

hash

Return the SHA of the commit.

Returns str hash

in_main_branch

Return True if the commit is in the main branch, False otherwise.

Returns bool in_main_branch

merge

Return True if the commit is a merge, False otherwise.

Returns bool merge

modifications

Return a list of modified files.

Returns List[Modification] modifications

msg

Return commit message.

Returns str commit_message

parents

Return the list of parents SHAs.

Returns List[str] parents

```
class pydriller.domain.commit.Modification(old_path: str, new_path: str, change_type: py-
driller.domain.commit.ModificationType, par-
ents: List[str], hash: str, path: str = None,
modifications_list=None)
```

```
__init__(old_path: str, new_path: str, change_type: pydriller.domain.commit.ModificationType, par-
ents: List[str], hash: str, path: str = None, modifications_list=None)
```

Initialize a modification. A modification carries on information regarding the changed file. Normally, you shouldn't initialize a new one.

```
__module__ = 'pydriller.domain.commit'
```

added

Return the total number of added lines in the file.

Returns int lines_added

diff

Return the diff of the file of the current commit.

Returns str diff

filename

Return the filename. Given a path-like-string (e.g. "/Users/dspadini/pydriller/myfile.py") returns only the filename (e.g. "myfile.py")

Returns str filename

removed

Return the total number of deleted lines in the file.

Returns int lines_deleted

source_code

Return the source code of the file on the current commit.

Returns str source_code

```
class pydriller.domain.commit.ModificationType
    An enumeration.
```

```
ADD = (1,)
COPY = (2,)
DELETE = (4,)
MODIFY = 5
RENAME = (3,)
__module__ = 'pydriller.domain.commit'
```

7.4 Developer

```
class pydriller.domain.developer.Developer(name: str, email: str)
```

```
    __init__(name: str, email: str)
```

Class to identify a developer.

Parameters

- **name** (*str*) – name and surname of the developer
- **email** (*str*) – email of the developer

```
__module__ = 'pydriller.domain.developer'
```

CHAPTER 8

Indices and tables

- `genindex`
- `modindex`
- `search`

p

- `pydriller.domain.commit`, [18](#)
- `pydriller.domain.developer`, [20](#)
- `pydriller.git_repository`, [15](#)
- `pydriller.repository_mining`, [17](#)

Symbols

__init__() (pydriller.domain.commit.ChangeSet method), 18
 __init__() (pydriller.domain.commit.Commit method), 18
 __init__() (pydriller.domain.commit.Modification method), 19
 __init__() (pydriller.domain.developer.Developer method), 20
 __init__() (pydriller.git_repository.GitRepository method), 15
 __init__() (pydriller.repository_mining.RepositoryMining method), 17
 __module__ (pydriller.domain.commit.ChangeSet attribute), 18
 __module__ (pydriller.domain.commit.Commit attribute), 18
 __module__ (pydriller.domain.commit.Modification attribute), 19
 __module__ (pydriller.domain.commit.ModificationType attribute), 20
 __module__ (pydriller.domain.developer.Developer attribute), 20
 __module__ (pydriller.git_repository.GitRepository attribute), 15
 __module__ (pydriller.repository_mining.RepositoryMining attribute), 17

A

ADD (pydriller.domain.commit.ModificationType attribute), 19
 added (pydriller.domain.commit.Modification attribute), 19
 author (pydriller.domain.commit.Commit attribute), 18
 author_date (pydriller.domain.commit.Commit attribute), 18
 author_timezone (pydriller.domain.commit.Commit attribute), 18

B

branches (pydriller.domain.commit.Commit attribute), 18

C

ChangeSet (class in pydriller.domain.commit), 18
 checkout() (pydriller.git_repository.GitRepository method), 15
 Commit (class in pydriller.domain.commit), 18
 committer (pydriller.domain.commit.Commit attribute), 18
 committer_date (pydriller.domain.commit.Commit attribute), 18
 committer_timezone (pydriller.domain.commit.Commit attribute), 18
 COPY (pydriller.domain.commit.ModificationType attribute), 20

D

DELETE (pydriller.domain.commit.ModificationType attribute), 20
 Developer (class in pydriller.domain.developer), 20
 diff (pydriller.domain.commit.Modification attribute), 19

F

filename (pydriller.domain.commit.Modification attribute), 19
 files() (pydriller.git_repository.GitRepository method), 15

G

get_change_sets() (pydriller.git_repository.GitRepository method), 15
 get_commit() (pydriller.git_repository.GitRepository method), 15
 get_commit_from_tag() (pydriller.git_repository.GitRepository method), 15
 get_commits_last_modified_lines() (pydriller.git_repository.GitRepository method), 16

`get_head()` (pydriller.git_repository.GitRepository method), 16
`GitRepository` (class in pydriller.git_repository), 15

H

`hash` (pydriller.domain.commit.Commit attribute), 18

I

`in_main_branch` (pydriller.domain.commit.Commit attribute), 18

M

`merge` (pydriller.domain.commit.Commit attribute), 19
`Modification` (class in pydriller.domain.commit), 19
`modifications` (pydriller.domain.commit.Commit attribute), 19
`ModificationType` (class in pydriller.domain.commit), 19
`MODIFY` (pydriller.domain.commit.ModificationType attribute), 20
`msg` (pydriller.domain.commit.Commit attribute), 19

P

`parents` (pydriller.domain.commit.Commit attribute), 19
`parse_diff()` (pydriller.git_repository.GitRepository method), 16
`pydriller.domain.commit` (module), 18
`pydriller.domain.developer` (module), 20
`pydriller.git_repository` (module), 15
`pydriller.repository_mining` (module), 17

R

`removed` (pydriller.domain.commit.Modification attribute), 19
`RENAME` (pydriller.domain.commit.ModificationType attribute), 20
`RepositoryMining` (class in pydriller.repository_mining), 17
`reset()` (pydriller.git_repository.GitRepository method), 16

S

`source_code` (pydriller.domain.commit.Modification attribute), 19

T

`total_commits()` (pydriller.git_repository.GitRepository method), 16
`traverse_commits()` (pydriller.repository_mining.RepositoryMining method), 17