# PyDriller Documentation

*Release 1.0*

**Davide Spadini**

**May 28, 2020**

# Contents

CHAPTER 1

# Overview / Install

PyDriller is a Python framework that helps developers on mining software repositories. 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

## 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
$ unzip test-repos.zip
```

and run the tests using pytest:

```
$ pytest
```

## 1.4 How to cite PyDriller

```
@inbook{PyDriller,
    title = "PyDriller: Python Framework for Mining Software Repositories",
    abstract = "Software repositories contain historical and valuable information␣
↪about the overall development of software systems. Mining software repositories␣
↪(MSR) is nowadays considered one of the most interesting growing fields within␣
↪software engineering. MSR focuses on extracting and analyzing data available in␣
↪software repositories to uncover interesting, useful, and actionable information␣
↪about the system. Even though MSR plays an important role in software engineering␣
↪research, few tools have been created and made public to support developers in␣
↪extracting information from Git repository. In this paper, we present PyDriller, a␣
↪Python Framework that eases the process of mining Git. We compare our tool against␣
↪the state-of-the-art Python Framework GitPython, demonstrating that PyDriller can␣
↪achieve the same results with, on average, 50% less LOC and significantly lower␣
↪complexity.URL: https://github.com/ishepard/pydrillerMaterials: https://doi.org/10.
↪5281/zenodo.1327363Pre-print: https://doi.org/10.5281/zenodo.1327411",
    author = "Spadini, Davide and Aniche, Maurício and Bacchelli, Alberto",
    year = "2018",
    doi = "10.1145/3236024.3264598",
    booktitle = "The 26th ACM Joint European Software Engineering Conference and␣
↪Symposium on the Foundations of Software Engineering (ESEC/FSE)",
}
```

# 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 will 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.

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*.

We can also pass a list of repositories (both local and remote), and PyDriller will analyze sequentially. In case of a remote repository, PyDriller will clone it in a temporary folder, and delete it afterwards. For example:

```
urls = ["repos/repo1", "repos/repo2", "https://github.com/ishepard/pydriller.git",
↪"repos/repo3", "https://github.com/apache/hadoop.git"]
for commit in RepositoryMining(path_to_repo=urls).traverse_commits():
    print("Project {}, commit {}, date {}".format(
            commit.project_path, commit.hash, commit.author_date))
```

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!

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.

Furthermore, PyDriller can calculate structural metrics of every file changed in a commit. To calculate these metrics, Pydriller relies on Lizard, a powerful tool that can analyze source code of many different programming languages, both at class and method level!

```
for commit in RepositoryMining('path/to/the/repo').traverse_commits():
    for mod in commit.modifications:
        print('{} has complexity of {}, and it contains {} methods'.format(
              mod.filename, mod.complexity, len(mod.methods)))
```

CHAPTER 3

# Configuration

One of the main advantage of using PyDriller to mine software repositories, is that it is highly configurable. We will now see all the options that once can pass to RepositoryMining.

## 3.1 Selecting projects to analyze

The only required parameter of *RepositoryMining* is **path_to_repo**, which specifies the repo(s) to analyze. It must be of type *str* or *List[str]*, meaning analyze only one repository or more than one.

Furthermore, PyDriller supports both local and remote repositories: if you pass an URL, PyDriller will automatically create a temporary folder, clone the repository, run the study, and finally delete the temporary folder.

For example, the following are all possible inputs for *RepositoryMining*:

```
# analyze only 1 local repository
url = "repos/pydriller/"

# analyze 2 local repositories
url = ["repos/pydriller/", "repos/anotherrepo/"]

# analyze both local and remote
url = ["repos/pydriller/", "https://github.com/apache/hadoop.git", "repos/anotherrepo
↪"]

# analyze 1 remote repository
url = "https://github.com/apache/hadoop.git"
```

To keep track of what project PyDriller is analyzing, the *Commit* object has a property called **project_name**.

## 3.2 Selecting the Commit Range

By default, PyDriller analyzes 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

*ORDER*:

- **order** *(str)*: one between 'date-order', 'author-date-order', 'topo-order', and 'reverse' (see this for more information). By default, PyDriller uses the flag "–reverse", and it returns the commits in reversed chronological order (from the oldest to the newest). If you need viceversa instead (from the newest to the oldest), use "order='reverse'".

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()

# 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!

## 3.3 Filtering commits

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

- **only_in_branch** *(str)*: only analyses commits that belong to this branch.
- **only_no_merge** *(bool)*: only analyses commits that are not merge commits.
- **only_authors** *(List[str])*: only analyses commits that are made by these authors. The check is made on the username, NOT the email.
- **only_commits** *(List[str])*: only these commits will be analyzed.
- **only_releases** *(bool)*: only commits that are tagged ("release" is a term of GitHub, does not actually exist in Git)
- **filepath** *(str)*: only commits that modified this file will be analyzed.
- **only_modifications_with_file_types** *(List[str])*: only analyses commits in which **at least** one modification was done in that file type, e.g., if you pass ".java", it will visit only commits in which at least one Java file was modified; clearly, it will skip other commits (e.g., commits that did not modify Java files).

Examples:

```
# Only commits in branch1
RepositoryMining('path/to/the/repo', only_in_branch='branch1').traverse_commits()

# Only commits in branch1 and no merges
RepositoryMining('path/to/the/repo', only_in_branch='branch1', only_no_merge=True).
↪traverse_commits()

# Only commits of author "ishepard" (yeah, that's me)
RepositoryMining('path/to/the/repo', only_authors=['ishepard']).traverse_commits()

# Only these 3 commits
RepositoryMining('path/to/the/repo', only_commits=['hash1', 'hash2', 'hash3']).
↪traverse_commits()

# Only commit that modified "Matricula.javax"
RepositoryMining('path/to/the/repo', filepath='Matricula.javax').traverse_commits()

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

## 3.4 Other Configurations

Some `git` commands, such as `git diff`, can be customized by the user. In this section, we report some of the customization that can be used within pydriller.

- **histogram** *(bool)*: uses `git diff --histogram` instead of the normal git. See *Git Diff Algorithms*.

## 3.5 Git Diff Algorithms

Git offers four different algorithms in `git diff`:

- Myers (default)
- Minimal (improved Myers)
- Patience (try to give contextual diff)
- Histogram (kind of enhanced patience)

Differences between four diff algorithms

Based on the comparison between Myers and Histogram in a study by Nugroho, et al (2019), various `diff` algorithms in the `git diff` command produced unequal *diff* outputs. From the result of patches analysis, they found that Histogram is better than Myers to show the changes of code that can be expected to recover the changing operations. Thus, in this tool, we implement histogram `diff` algorithm to consider differences in source code.

# CHAPTER 4

## Commit Object

A Commit object has all the information of a Git commit, and much more. More specifically:

- **hash** *(str)*: hash of the commit
- **msg** *(str)*: commit message
- **author** *(Developer)*: commit author (name, email)
- **author_date** *(datetime)*: authored date
- **author_timezone** *(int)*: author timezone (expressed in seconds from epoch)
- **committer** *(Developer)*: commit committer (name, email)
- **committer_date** *(datetime)*: commit date
- **committer_timezone** *(int)*: commit timezone (expressed in seconds from epoch)
- **branches** *(List[str])*: List of branches that contain this commit
- **in_main_branch** *(Bool)*: True if the commit is in the main branch
- **merge** *(Bool)*: True if the commit is a merge commit
- **modifications** *(List[Modifications])*: list of modified files in the commit (see *Modifications*)
- **parents** *(Set[str])*: list of the commit parents
- **project_name** *(str)*: project name
- **project_path** *(str)*: project path

Example:

```
for commit in RepositoryMining('path/to/the/repo').traverse_commits():
    print(
        'The commit {} has been modified by {}, '
        'committed by {} in date {}'.format(
            commit.hash,
            commit.author.name,
```

(continues on next page)

(continued from previous page)

```
            commit.committer.name,
            commit.committer_date
        )
    )
```

CHAPTER 5

# 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 _None_ if the file is deleted)
- **change_type**: type of the change: can be Added, Deleted, Modified, or Renamed.
- **diff**: diff of the file as Git presents it (e.g., starting with @@ xx,xx @@).
- **source_code**: source code of the file (can be _None_ if the file is deleted)
- **source_code_before**: source code of the file before the change (can be _None_ if the file is added)
- **added**: number of lines added
- **removed**: number of lines removed
- **nloc**: Lines Of Code (LOC) of the file
- **complexity**: Cyclomatic Complexity of the file
- **token_count**: Number of Tokens of the file
- **methods**: list of methods of the file. The list might be empty if the programming language is not supported or if the file is not a source code file.

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),
            " and the complexity is {}".format(m.complexity)
        )
```

# CHAPTER 6

## GitRepository

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 (if you pass a bug fixing commit, it will retrieve the bug inducing).

PS: Since PyDriller 1.9, this function can be customized to use "git hyper-blame" (check this for more info). Git hyper blame can be instructed to skip specific commits (like commits that refactor the code).

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**).

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

CHAPTER 7

# Process Metrics

Process metrics capture aspects of the development process rather than aspects about the code itself. From release 1.11 PyDriller can calculate `change_set`, `code churn`, `commits count`, `contributors count`, `contributors experience`, `history complexity`, `hunks count`, `lines count` and `minor contributors`. Everything in just one line!

The metrics can be run between two commits (setting up the parameters `from_commit` and `to_commit`) or between two dates (setting up the parameters `since` and `to`)

Below an example of how call the metrics.

## 7.1 Change Set

This metric measures the of files committed together.

The class `ChangeSet` has two methods:

- `max()` to count the *maximum* number of files committed together;
- `avg()` to count the *average* number of files committed together. **Note:** The average value is rounded off to the nearest integer.

For example:

```
from pydriller.metrics.process.change_set import ChangeSet
metric = ChangeSet(path_to_repo='path/to/the/repo',
                   from_commit='from commit hash',
                   to_commit='to commit hash')

maximum = metric.max()
average = metric.avg()
print('Maximum number of files committed together: {}'.format(maximum))
print('Average number of files committed together: {}'.format(average))
```

will print the maximum and average number of files committed together in the evolution period `[from_commit, to_commit]`.

**Note:** differently from the other metrics below, the scope of this metrics is the evolution period rather than the single files.

It is possible to specify the dates as follows:

```
from datetime import datetime
from pydriller.metrics.process.change_set import ChangeSet
metric = ChangeSet(path_to_repo='path/to/the/repo',
                   since=datetime(2019, 1, 1),
                   to=datetime(2019, 12, 31))

maximum = metric.max()
average = metric.avg()
print('Maximum number of files committed together: {}'.format(maximum))
print('Average number of files committed together: {}'.format(average))
```

The code above will print the maximum and average number of files committed together between the `1st January 2019` and `31st December 2019`.

## 7.2 Code Churn

This metric measures the code churns of a file. A code churn is the sum of (added lines - removed lines) across the analyzed commits.

The class `CodeChurn` has three methods:

- `count()` to count the *total* size of code churns of a file;
- `max()` to count the *maximum* size of a code churn of a file;
- `avg()` to count the *average* size of a code churn of a file. **Note:** The average value is rounded off to the nearest integer.

For example:

```
from pydriller.metrics.process.code_churn import CodeChurn
metric = CodeChurn(path_to_repo='path/to/the/repo',
                   from_commit='from commit hash',
                   to_commit='to commit hash')
files_count = metric.count()
files_max = metric.max()
files_avg = metric.avg()
print('Total code churn for each file: {}'.format(files_count))
print('Maximum code churn for each file: {}'.format(files_max))
print('Average code churn for each file: {}'.format(files_avg))
```

will print the total, maximum and average number of code churn for each modified file in the evolution period `[from_commit, to_commit]`.

## 7.3 Commits Count

This metric measures the number of commits made to a file.

The class `CommitCount` has one method:

- `count()` to count the number of commits to a file.

For example:

```
from pydriller.metrics.process.commits_count import CommitsCount
metric = CommitsCount(path_to_repo='path/to/the/repo',
                      from_commit='from commit hash',
                      to_commit='to commit hash')
files = metric.count()
print('Files: {}'.format(files))
```

will print the number of commits for each modified file in the evolution period `[from_commit, to_commit]`.

## 7.4 Contributors Count

This metric measures the number of developers that contributed to a file.

The class `ContributorsCount` has two methods:

- `count()` to count the number of contributors who modified a file;
- `count_minor()` to count the number of *minor* contributors who modified a file, i.e., those that contributed less than 5% to the file.

For example:

```
from pydriller.metrics.process.contributors_count import ContributorsCount
metric = ContributorsCount(path_to_repo='path/to/the/repo',
                           from_commit='from commit hash',
                           to_commit='to commit hash')
count = metric.count()
minor = metric.count_minor()
print('Number of contributors per file: {}'.format(count))
print('Number of "minor" contributors per file: {}'.format(minor))
```

will print the number of developers that contributed to each of the modified file in the evolution period `[from_commit, to_commit]` and the number of developers that contributed less than 5% to each of the modified file in the evolution period `[from_commit, to_commit]`.

## 7.5 Contributors Experience

This metric measures the percetage of the lines authored by the highest contributor of a file.

The class `ContriutorExperience` has one method:

- `count()` to count the number of lines authored by the highest contributor of a file;

For example:

```
from pydriller.metrics.process.contributors_experience import ContributorsExperience
metric = ContributorsExperience(path_to_repo='path/to/the/repo',
                                from_commit='from commit hash',
                                to_commit='to commit hash')
files = metric.count()
print('Files: {}'.format(files))
```

will print the percentage of the lines authored by the highest contributor for each of the modified file in the evolution period `[from_commit, to_commit]`.

## 7.6 Hunks Count

This metric measures the number of hunks made to a file. As a hunk is a continuous block of changes in a `diff`, this number assesses how fragmented the commit file is (i.e. lots of changes all over the file versus one big change).

The class `HunksCount` has one method:

- `count()` to count the median number of hunks of a file.

For example:

```
from pydriller.metrics.process.hunks_count import HunksCount
metric = HunksCount(path_to_repo='path/to/the/repo',
                    from_commit='from commit hash',
                    to_commit='to commit hash')
files = metric.count()
print('Files: {}'.format(files))
```

will print the median number of hunks for each of the modified file in the evolution period `[from_commit, to_commit]`.

## 7.7 Lines Count

This metric measures the number of added and removed lines in a file. The class `LinesCount` has seven methods:

- `count()` to count the total number of added and removed lines for each modified file;
- `count_added()`, `max_added()` and `avg_added()` to count the total, maximum and average number of added lines for each modified file;
- `count_removed()`, `max_removed()` and `avg_removed()` to count the total, maximum and average number of removed lines for each modified file.

**Note:** The average values are rounded off to the nearest integer.

For example:

```
from pydriller.metrics.process.lines_count import LinesCount
metric = LinesCount(path_to_repo='path/to/the/repo',
                    from_commit='from commit hash',
                    to_commit='to commit hash')

added_count = metric.count_added()
added_max = metric.max_added()
added_avg = metric.avg_added()
print('Total lines added per file: {}'.format(added_count))
print('Maximum lines added per file: {}'.format(added_max))
print('Average lines added per file: {}'.format(added_avg))
```

will print the total, maximum and average number of lines added for each modified file in the evolution period `[from_commit, to_commit]`.

While:

```
from pydriller.metrics.process.lines_count import LinesCount
metric = LinesCount(path_to_repo='path/to/the/repo',
                    from_commit='from commit hash',
                    to_commit='to commit hash')

removed_count = metric.count_removed()
removed_max = metric.max_removed()
removed_avg = metric.avg_removed()
print('Total lines removed per file: {}'.format(removed_count))
print('Maximum lines removed per file: {}'.format(removed_max))
print('Average lines removed per file: {}'.format(removed_avg))
```

will print the total, maximum and average number of lines removed for each modified file in the evolution period `[from_commit, to_commit]`.

CHAPTER 8

# API Reference

## 8.1 GitRepository

This module includes 1 class, GitRepository, representing a repository in Git.

**class** pydriller.git_repository.**GitRepository**(*path: str*, *conf=None*)

Class representing a repository in Git. It contains most of the logic of PyDriller: obtaining the list of commits, checkout, reset, etc.

**__del__**()

**__init__**(*path: str*, *conf=None*)

Init the Git RepositoryMining.

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

**__module__ = 'pydriller.git_repository'**

**checkout**(*_hash: str*) → None

Checkout the repo at the speficied 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

**clear**()

According to GitPython's documentation, sometimes it leaks resources. This holds especially for Windows users. Hence, we need to clear the cache manually.

**files**() → List[str]

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

**Returns** List[str], the list of the files

**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_gitpython**(*commit: git.objects.commit.Commit*) → pydriller.domain.commit.Commit

Build a PyDriller commit object from a GitPython commit object. This is internal of PyDriller, I don't think users generally will need it.

**Parameters** **commit** (*GitCommit*) – GitPython commit

**Returns** Commit commit: PyDriller 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*, *hashes_to_ignore_path: str = None*) → Dict[str, 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
- **hashes_to_ignore_path** (*str*) – path to a file containing hashes of commits to ignore.

**Returns** the set containing all the bug inducing commits

**get_commits_modified_file**(*filepath: str*) → List[str]

Given a filepath, returns all the commits that modified this file (following renames).

**Parameters** **filepath** (*str*) – path to the file

**Returns** the list of commits' hash

**get_head**() → pydriller.domain.commit.Commit

Get the head commit.

**Returns** Commit of the head commit

**get_list_commits**(*rev='HEAD'*, ***kwargs*) → Generator[[pydriller.domain.commit.Commit, None], None]

Return a generator of commits of all the commits in the repo.

**Returns** Generator[Commit], the generator of all the commits in the repo

**get_tagged_commits**()

Obtain the hash of all the tagged commits.

**Returns** list of tagged commits (can be empty if there are no tags)

**git**

GitPython object Git.

**Returns** Git

**repo**
GitPython object Repo.

**Returns** Repo

**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

## 8.2 RepositoryMining

This module includes 1 class, RepositoryMining, main class of PyDriller.

**class** pydriller.repository_mining.**RepositoryMining**(*path_to_repo: Union[str, List[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_branch: str = None, only_modifications_with_file_types: List[str] = None, only_no_merge: bool = False, only_authors: List[str] = None, only_commits: List[str] = None, only_releases: bool = False, filepath: str = None, histogram_diff: bool = False, skip_whitespaces: bool = False, clone_repo_to: str = None, order: str = None*)
This is the main class of PyDriller, responsible for running the study.

**__init__**(*path_to_repo: Union[str, List[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_branch: str = None, only_modifications_with_file_types: List[str] = None, only_no_merge: bool = False, only_authors: List[str] = None, only_commits: List[str] = None, only_releases: bool = False, filepath: str = None, histogram_diff: bool = False, skip_whitespaces: bool = False, clone_repo_to: str = None, order: str = None*)
Init a repository mining. The only required parameter is "path_to_repo": to analyze a single repo, pass the absolute path to the repo; if you need to analyze more repos, pass a list of absolute paths.

Furthermore, PyDriller supports local and remote repositories: if you pass a path to a repo, PyDriller will run the study on that repo; if you pass an URL, PyDriller will clone the repo in a temporary folder, run the study, and delete the temporary folder.

**Parameters**

- **path_to_repo** (*Union[str,List[str]]*) – absolute path (or list of absolute paths) to the repository(ies) 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 (**DEPRECATED**)
- **only_in_branch** (*str*) – only commits in this branch 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
- **only_authors** (*List[str]*) – only commits of these authors will be analyzed (the check is done on the username, NOT the email)
- **only_commits** (*List[str]*) – only these commits will be analyzed
- **filepath** (*str*) – only commits that modified this file will be analyzed
- **order** (*str*) – order of commits. It can be one of: 'date-order', 'author-date-order', 'topo-order', or 'reverse'. Default is reverse.

**__module__ = 'pydriller.repository_mining'**

**traverse_commits**() → Generator[[pydriller.domain.commit.Commit, None], None]

Analyze all the specified commits (all of them by default), returning a generator of commits.

## 8.3 Commit

This module contains all the classes regarding a specific commit, such as Commit, Modification, ModificationType and Method.

**class** pydriller.domain.commit.**Commit**(*commit: git.objects.commit.Commit*, *conf*)

Class representing a Commit. Contains all the important information such as hash, author, dates, and modified files.

**__init__**(*commit: git.objects.commit.Commit*, *conf*) → None

Create a commit object.

**Parameters**

- **commit** – GitPython Commit object
- **conf** – Configuration class

**__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

**project_name**
Return the project name.

**Returns** project name

**class** pydriller.domain.commit.**Method**(*func*)

This class represents a method in a class. Contains various information extracted through Lizard.

**__init__**(*func*)

Initialize a method object. This is calculated using Lizard: it parses the source code of all the modifications in a commit, extracting information of the methods contained in the file (if the file is a source code written in one of the supported programming languages).

**__module__ = 'pydriller.domain.commit'**

**class** pydriller.domain.commit.**Modification**(*old_path: str, new_path: str, change_type: pydriller.domain.commit.ModificationType, diff_and_sc: Dict[str, str]*)

This class contains information regarding a modified file in a commit.

**__init__**(*old_path: str, new_path: str, change_type: pydriller.domain.commit.ModificationType, diff_and_sc: Dict[str, str]*)

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

**changed_methods**

Return the list of methods that were changed. This analysis is more complex because lizzard runs twice: for methods before and after the change

**Returns** list of methods

**complexity**

Calculate the Cyclomatic Complexity of the file.

**Returns** Cyclomatic Complexity of the file

**diff_parsed**

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).

**Returns** Dictionary

**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

**methods**

Return the list of methods in the file. Every method contains various information like complexity, loc, name, number of parameters, etc.

**Returns** list of methods

**methods_before**

Return the list of methods in the file before the change happened. Each method will have all specific info, e.g. complexity, loc, name, etc.

**Returns** list of methods

**new_path**
New path of the file. Can be None if the file is deleted.

**Returns** str new_path

**nloc**
Calculate the LOC of the file.

**Returns** LOC of the file

**old_path**
Old path of the file. Can be None if the file is added.

**Returns** str old_path

**removed**
Return the total number of deleted lines in the file.

**Returns** int lines_deleted

**token_count**
Calculate the token count of functions.

**Returns** token count

**class** pydriller.domain.commit.**ModificationType**
Type of Modification. Can be ADD, COPY, RENAME, DELETE, MODIFY or UNKNOWN.

**ADD = 1**

**COPY = 2**

**DELETE = 4**

**MODIFY = 5**

**RENAME = 3**

**UNKNOWN = 6**

**__module__ = 'pydriller.domain.commit'**

## 8.4 Developer

This module includes only 1 class, Developer, representing a developer.

**class** pydriller.domain.developer.**Developer**(*name: str*, *email: str*)
This class represents a developer. We save the email and the name.

**__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'**

## 8.5 Process Metrics

This module contains the abstract class to implement process metrics.

**class** pydriller.metrics.process.process_metric.**ProcessMetric**(*path_to_repo: str*, *since: datetime.datetime = None*, *to: datetime.datetime = None*, *from_commit: str = None*, *to_commit: str = None*)

Abstract class to implement process metrics

**__init__**(*path_to_repo: str*, *since: datetime.datetime = None*, *to: datetime.datetime = None*, *from_commit: str = None*, *to_commit: str = None*)

**Path_to_repo** path to a single repo

**Parameters**

- **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)

**__module__ = 'pydriller.metrics.process.process_metric'**

**count**()

Implement the main functionality of the metric

CHAPTER 9

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## p

# Index

## Symbols



## A



## B



## C



## D

## F



## G



## H



## I



## M



## N



## O



## P



## R



## T



## U