
PyDriller Documentation

Release 1.0

Davide Spadini

Jul 11, 2021

1	Overview / Install	1
1.1	Requirements	1
1.2	Installing PyDriller	1
1.3	Source Code	1
1.4	How to cite PyDriller	2
2	Getting Started	3
3	Repository	5
3.1	Simple Scenario	5
3.2	Selecting projects to analyze	5
3.3	Selecting the Commit Range	6
3.4	Filtering commits	7
3.5	Other Configurations	8
3.6	Git Diff Algorithms	8
4	Commit	9
5	ModifiedFile	11
6	Git	13
7	Delta Maintainability	15
7.1	Background	15
7.2	Definition	15
7.3	Properties	16
7.4	Example usage	16
7.5	Under the hood	16
7.6	Relation to SIG DMM	17
7.7	References	17
8	Process Metrics	19
8.1	Change Set	19
8.2	Code Churn	20
8.3	Commits Count	21
8.4	Contributors Count	21
8.5	Contributors Experience	21

8.6	Hunks Count	22
8.7	Lines Count	22
9	API Reference	25
9.1	Version	25
9.2	Git	25
9.3	Repository	27
9.4	Commit	29
9.5	Developer	33
9.6	Process Metrics	33
10	Indices and tables	35
	Bibliography	37
	Python Module Index	39
	Index	41

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.txt
$ pip install -r test-requirements.txt
$ 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 *Repository*: 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 Repository('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 *Repository*, 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 *Repository*.

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 Repository(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 Repository('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 Repository('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)))
```

Repository is the main class of Pydriller, responsible of returning the list of commits you want. 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 *Repository*.

3.1 Simple Scenario

This is the “Hello World” of Pydriller:

```
for commit in Repository("/Users/dspadini/myrepo").traverse_commits():
    print(commit.hash)
```

The function *traverse_commits()* of *Repository* will return the selected commits, in this simple case *all of them*. Now let's see how we can customize *Repository*.

3.2 Selecting projects to analyze

The only required parameter of *Repository* 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 *Repository*:

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

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

(continues on next page)

(continued from previous page)

```
# 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**.

3.3 Selecting the Commit Range

By default, PyDriller analyzes all the commits in the repository. However, filters can be applied to *Repository* 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). **NOTE**: By default, PyDriller returns the commits from the oldest to the newest. If you need viceversa instead (from the newest to the oldest), use "order='reverse'".

Examples:

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

# Since 8/10/2016
Repository('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)
Repository('path/to/the/repo', since=dt1, to=dt2).traverse_commits()

# Between tags
from_tag = 'tag1'
to_tag = 'tag2'
Repository('path/to/the/repo', from_tag=from_tag, to_tag=to_tag).traverse_commits()
```

(continues on next page)

(continued from previous page)

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

# !!!!! ERROR !!!!! THIS IS NOT POSSIBLE
Repository('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!

3.4 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
Repository('path/to/the/repo', only_in_branch='branch1').traverse_commits()

# Only commits in branch1 and no merges
Repository('path/to/the/repo', only_in_branch='branch1', only_no_merge=True).traverse_
↳commits()

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

# Only these 3 commits
Repository('path/to/the/repo', only_commits=['hash1', 'hash2', 'hash3']).traverse_
↳commits()

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

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

3.5 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*.
- **include_refs** (*bool*): whether to include refs and HEAD in commit analysis (equivalent of adding the flag `--all`).
- **include_remotes** (*bool*): whether to include remote commits in analysis (equivalent of adding the flag `--remotes`).
- **clone_repo_to** (*str*): if the repository is a URL, `Pydriller` will clone it in this directory.

3.6 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.

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)
- **committer** (*Developer*): commit committer (name, email)
- **author_date** (*datetime*): authored date
- **author_timezone** (*int*): author timezone (expressed in seconds from epoch)
- **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_toplevel`)
- **parents** (*List[str]*): list of the commit parents
- **project_name** (*str*): project name
- **project_path** (*str*): project path
- **deletions** (*int*): number of deleted lines in the commit (as shown from `-shortstat`).
- **insertions** (*int*): number of added lines in the commit (as shown from `-shortstat`).
- **lines** (*int*): total number of added + deleted lines in the commit (as shown from `-shortstat`).
- **files** (*int*): number of files changed in the commit (as shown from `-shortstat`).
- **dmm_unit_size** (*float*): DMM metric value for the unit size property.
- **dmm_unit_complexity** (*float*): DMM metric value for the unit complexity property.

- **dmn_unit_interfacing** (*float*): DMM metric value for the unit interfacing property.

Example:

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

ModifiedFile

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)
- **filename**: return only the filename (e.g., given a path-like-string such as `"/Users/dspadini/pydriller/myfile.py"` returns `"myfile.py"`)
- **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 @@`).
- **diff_parsed**: diff parsed in a dictionary containing the added and deleted lines. The dictionary has 2 keys: `"added"` and `"deleted"`, each containing a list of `Tuple (int, str)` corresponding to (number of line in the file, actual line).
- **added**: number of lines added
- **removed**: number of lines removed
- **source_code**: source code of the file (can be `_None_` if the file is deleted or only renamed)
- **source_code_before**: source code of the file before the change (can be `_None_` if the file is added or only renamed)
- **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. These are the methods **after** the change.
- **methods_before**: list of methods of the file **before** the change (e.g., before the commit.)
- **changed_methods**: subset of `_methods_` containing **only** the changed methods.
- **nloc**: Lines Of Code (LOC) of the file
- **complexity**: Cyclomatic Complexity of the file
- **token_count**: Number of Tokens of the file

NOTE: the list of modifications might be empty if the commit is a merge commit. For more info on this, check out [this post](#).

For example:

```
for commit in Repository('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)
        )
```

Git is a wrapper for the most common utilities of Git, such as checkout and reset. For example, to checkout a specific commit or branch:

```
gr = Git('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.

Moreover, Git can be used to obtain different information from the repository:

```
gr = Git('test-repos/test1')
gr.get_list_commits()           # get the list of all commits
gr.get_commit('cc5b002')       # get the specific commit
gr.files()                      # get the list of files present in the repo at
↳ the current commit
gr.total_commits()             # get total number of commits
gr.get_commit_from_tag('v1.15') # get the commit with tag v1.15
```

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 = Git('test-repos/test5')
```

(continues on next page)

(continued from previous page)

```
commit = gr.get_commit('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.

7.1 Background

To assess the maintainability implications of commits, PyDriller offers an implementation of the *Open Source Delta Maintainability Model* (OS-DMM). The underlying Delta Maintainability Model was originally described in a paper that appeared at TechDebt 2019 [DiBiase2019]. A commercially available implementation supporting over 100 different languages with fine-grained analysis is offered by the [Software Improvement Group](#) (SIG).

The Open Source implementation included in PyDriller offers a partial implementation suitable for research experiments and measurements for systems written in common programming languages already supported by PyDriller. While the git-functionality of PyDriller is language agnostic, the *metrics* (such as method size and cyclomatic complexity) it supports require language-specific implementations – for which PyDriller relies on [Lizard](#).

The OS-DMM implementation extends the PyDriller metrics with three commit-level metrics related to risk in size, complexity, and interfacing.

7.2 Definition

In one sentence, the delta-maintainability metric is the proportion of *low-risk change* in a commit. The resulting value ranges from 0.0 (all changes are risky) to 1.0 (all changes are low risk). It rewards making methods better, and penalizes making things worse.

The starting point for the DMM is a *risk profile* [Heitlager2007]. Traditionally, risk profiles categorize methods (or, more generally, also referred to as *units*) into four bins: low, medium, high, and very high risk methods. The risk profile of a class then is a 4-tuple (l, m, h, v) representing the amount of code (number of lines) in each of the four categories.

For simplicity, in the context of the DMM, only two bins are used: low risk, and non-low (medium, high, or very high) risk. To transfer risk profiles from file (or system) level to commit level, we consider *delta risk profiles*. These are pairs (dl, dh) , with dl being the increase (or decrease) of low risk code, and dh the increase (or decrease) in high risk code.

The delta risk profile can then be used to determine good and bad change:

- Increases in low risk code are good, but increases in high risk code are bad.
- Decreases in high risk code are good, and decreases in low risk code are good only if no high risk code is added instead, and bad otherwise.

The dmm value is then computed as: $good\ change / (good\ change + bad\ change)$. It is undefined (None) if the denominator is zero.

7.3 Properties

The DMM can be used on arbitrary properties that can be determined at method (unit) level. The PyDriller OS-DMM implementation supports three properties:

- Unit **size**: Method length in lines of code; low risk threshold 15 lines.
- Unit **complexity**: Method cyclomatic complexity; low risk threshold 5.
- Unit **interfacing**: Method number of parameters; low risk threshold 2.

The original DMM paper also used coupling and cloned code as properties, but these are not easily computed per commit with the Lizard infrastructure. The thresholds are language-independent by design, and have been determined empirically following the procedure described in [Alves2010], using industrial benchmark data collected by SIG [SIG2019].

7.4 Example usage

Collecting DMM values from a git repository is straightforward:

```
from pydriller import Repository

rm = Repository("https://github.com/avandeursen/dmm-test-repo")
for commit in rm.traverse_commits():
    print("{} | {} | {} | {} | {}".format(
        commit.msg,
        commit.dmm_unit_size,
        commit.dmm_unit_complexity,
        commit.dmm_unit_interfacing
    ))
```

The resulting dmm values are proportions with values between 0.0 and 1.0. Files that are changed in a commit, but which are written in languages not supported by PyDriller (Lizard) are ignored – these are often configuration (.xml, .yaml) or documentation (.txt, .md) files. If none of the files changed in a commit are in languages supported by Pydriller, the dmm value is None.

7.5 Under the hood

The main public API consists of the three `dmm_unit_size`, `dmm_unit_complexity`, and `dmm_unit_interfacing` properties on the `Commit` class, as illustrated above. Under the hood, the DMM implementation can be easily configured or accessed:

- The thresholds are set as separate constants in the `Method` class;
- The main methods implementing the DMM are parameterized with an enum characterizing the DMM property of interest.

- There are separate (protected) methods to compute risk profiles and delta-risk profiles at `Commit` and `Modification` level, which can be used to collect more detailed information for selected (e.g., lowly rated) commits.

7.6 Relation to SIG DMM

PyDriller's OS-DMM and SIG's DMM differ in the following ways:

- OS-DMM offers only support for the approximately 15 languages supported by [Lizard](#).
- OS-DMM relies on [Lizard](#) for the identification of *methods* (units) in source files. While for simple cases SIG and [Lizard](#) tooling will agree, this may not be the case for more intricate cases involving e.g., lambdas, inner classes, nested functions, etc.
- OS-DMM relies on [Lizard](#) for simple line counting, which also counts white space. SIG's DMM on the other hand ignores lines that contain no statements (such as blank lines or lines with just `{ . . . }` brackets)
- OS-DMM uses the thresholds as empirically determined by SIG, based on SIG's measurement methodology [[Alves2010](#)]. OS-DMM's [Lizard](#)-based metric values may be different, and hence may classify methods in different risk bins for methods close to the thresholds.

Consequently, individual DMM values are likely to differ a few percentage points between the SIG DMM and OS-DMM implementations. However, in terms of trends and statistical analysis, the outcomes will likely be very similar. Therefore:

- For research purposes, we recommend the fully open PyDriller implementation ensuring reproducible results.
- For commercial purposes including day to day monitoring of maintainability at commit, code, file, component, project, and portfolio level, we recommend the more robust SIG implementation.

7.7 References

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.

8.1 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.

8.2 Code Churn

This metric measures the code churns of a file.

Depending on the parametrization, a code churn is the sum of either

- (a) (added lines - removed lines) or
- (b) (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].

The calculation variant (a) or (b) can be configured by setting the CodeChurn init parameter:

- `add_deleted_lines_to_churn`

8.3 Commits Count

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

The class `CommitsCount` 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]`.

8.4 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]`.

8.5 Contributors Experience

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

The class `ContributorExperience` 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].

8.6 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].

8.7 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()
```

(continues on next page)

(continued from previous page)

```
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].

9.1 Version

`pydriller.__version__`
Current Pydriller version.

9.2 Git

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

class `pydriller.git.Git` (*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 Repository.

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

`__module__ = 'pydriller.git'`

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

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.ModifiedFile = 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 Dict commits: a dictionary having as keys the files of the commit, and as values the commits that last touched those files.

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)

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

9.3 Repository

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

exception pydriller.repository.MalformedUrl (message)

__init__ (message)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'pydriller.repository'

class pydriller.repository.Repository (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, include_refs: bool = False, include_remotes: bool = False, num_workers: int = 1, 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, include_refs: bool = False, include_remotes: bool = False, num_workers: int = 1, 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. 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)
- **include_refs** (*bool*) – whether to include refs and HEAD in commit analysis
- **include_remotes** (*bool*) – whether to include remote commits in analysis
- **num_workers** (*int*) – number of workers (i.e., threads). Please note, if *num_workers* > 1 the commits order is not maintained.
- **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
- **only_releases** (*bool*) – analyze only tagged commits
- **histogram_diff** (*bool*) – add the “-histogram” option when asking for the diff
- **skip_whitespaces** (*bool*) – add the “-w” option when asking for the diff
- **clone_repo_to** (*bool*) – if the repo under analysis is remote, clone the repo to the specified directory
- **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'`

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

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

9.4 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

deletions

Return the number of deleted lines in the commit (as shown from `-shortstat`).

Returns int deletion lines

dmm_unit_complexity

Return the Delta Maintainability Model (DMM) metric value for the unit complexity property.

It represents the proportion (between 0.0 and 1.0) of maintainability improving change, when considering the cyclomatic complexity of the modified methods.

It rewards (value close to 1.0) modifications to low-risk (low complexity) methods, or splitting risky (highly complex) ones. It penalizes (value close to 0.0) working on methods that remain complex or get more complex.

Returns The DMM value (between 0.0 and 1.0) for method complexity in this commit. or None if none of the programming languages in the commit are supported.

dmm_unit_interfaceing

Return the Delta Maintainability Model (DMM) metric value for the unit interfacing property.

It represents the proportion (between 0.0 and 1.0) of maintainability improving change, when considering the interface (number of parameters) of the modified methods.

It rewards (value close to 1.0) modifications to low-risk (with few parameters) methods, or splitting risky (with many parameters) ones. It penalizes (value close to 0.0) working on methods that continue to have or are extended with too many parameters.

Returns The dmm value (between 0.0 and 1.0) for method interfacing in this commit. or None if none of the programming languages in the commit are supported.

dmm_unit_size

Return the Delta Maintainability Model (DMM) metric value for the unit size property.

It represents the proportion (between 0.0 and 1.0) of maintainability improving change, when considering the lengths of the modified methods.

It rewards (value close to 1.0) modifications to low-risk (small) methods, or splitting risky (large) ones. It penalizes (value close to 0.0) working on methods that remain large or get larger.

Returns The DMM value (between 0.0 and 1.0) for method size in this commit, or None if none of the programming languages in the commit are supported.

files

Return the number of modified files of the commit (as shown from `-shortstat`).

Returns int modified files number

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

insertions

Return the number of added lines in the commit (as shown from `-shortstat`).

Returns int insertion lines

lines

Return the number of modified lines in the commit (as shown from `-shortstat`).

Returns int insertion + deletion lines

merge

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

Returns bool merge

modified_files

Return a list of modified files. The list is empty if the commit is a merge commit. For more info

on this, see <https://haacked.com/archive/2014/02/21/reviewing-merge-commits/> or <https://github.com/ishepard/pydriller/issues/89#issuecomment-590243707>

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

project_path

Return the absolute path of the project.

Returns project path

class pydriller.domain.commit.DMMProperty
Maintainability properties of the Delta Maintainability Model.

UNIT_COMPLEXITY = 2

UNIT_INTERFACING = 3

UNIT_SIZE = 1

__module__ = 'pydriller.domain.commit'

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

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

UNIT_COMPLEXITY_LOW_RISK_THRESHOLD = 5

Threshold used in the Delta Maintainability Model to establish whether a method is low risk in terms of its cyclomatic complexity. The procedure to obtain the threshold is described in the *PyDriller documentation*.

UNIT_INTERFACING_LOW_RISK_THRESHOLD = 2

Threshold used in the Delta Maintainability Model to establish whether a method is low risk in terms of its interface. The procedure to obtain the threshold is described in the *PyDriller documentation*.

UNIT_SIZE_LOW_RISK_THRESHOLD = 15

Threshold used in the Delta Maintainability Model to establish whether a method is low risk in terms of its size. The procedure to obtain the threshold is described in the *PyDriller documentation*.

__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'

is_low_risk(*dmm_prop*: pydriller.domain.commit.DMMProperty) → bool

Predicate indicating whether this method is low risk in terms of the given property.

Parameters *dmm_prop* – Property according to which this method is considered risky.

Returns True if and only if the method is considered low-risk w.r.t. this property.

```
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'
```

```
class pydriller.domain.commit.ModifiedFile (old_path: Optional[str], new_path: Optional[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: Optional[str], new_path: Optional[str], change_type: pydriller.domain.commit.ModificationType, diff_and_sc: Dict[str, str])
```

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

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

```
added_lines
```

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 Lizard 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

```
deleted_lines
```

Return the total number of deleted lines in the file.

Returns int lines_deleted

```
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

language_supported

Return whether the language used in the modification can be analyzed by Pydriller. Languages are derived from the file extension. Supported languages are those supported by Lizard.

Returns True iff language of this Modification can be analyzed.

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

token_count

Calculate the token count of functions.

Returns token count

9.5 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'`

9.6 Process Metrics

This module contains the abstract class to implement process metrics.

```
class pydriller.metrics.process.process_metric.ProcessMetric (path_to_repo:  
                                                    str, since: date-  
                                                    time.datetime =  
                                                    None, to: date-  
                                                    time.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 10

Indices and tables

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

Bibliography

- [DiBiase2019] Marco di Biase, Ayushi Rastogi, Magiel Bruntink, and Arie van Deursen. **The Delta Maintainability Model: measuring maintainability of fine-grained code changes**. IEEE/ACM International Conference on Technical Debt (TechDebt) at ICSE 2019, pp 113-122 ([preprint](#), [doi](#)).
- [Heitlager2007] Ilja Heitlager, Tobias Kuipers, and Joost Visser. **A Practical Model for Measuring Maintainability**. 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, IEEE, pp 30-39 ([preprint](#), [doi](#))
- [Alves2010] Tiaga Alves, Christiaan Ypma, and Joost Visser. **Deriving metric thresholds from benchmark data**. IEEE International Conference on Software Maintenance (ICSM), pages 1–10. IEEE, 2010 ([preprint](#), [doi](#)).
- [SIG2019] Reinier Vis, Dennis Bijslma, and Haiyun Xu. SIG/TÜViT Evaluation Criteria Trusted Product Maintainability: Guidance for producers. Version 11.0. Software Improvement Group, 2019 ([online](#)).

p

`pydriller.domain.commit`, [29](#)
`pydriller.domain.developer`, [33](#)
`pydriller.git`, [25](#)
`pydriller.metrics.process.process_metric`,
[33](#)
`pydriller.repository`, [27](#)

Symbols

- __del__()** (*pydriller.git.Git method*), 25
__init__() (*pydriller.domain.commit.Commit method*), 29
__init__() (*pydriller.domain.commit.Method method*), 31
__init__() (*pydriller.domain.commit.ModifiedFile method*), 32
__init__() (*pydriller.domain.developer.Developer method*), 33
__init__() (*pydriller.git.Git method*), 25
__init__() (*pydriller.metrics.process.process_metric.ProcessMetric method*), 34
__init__() (*pydriller.repository.MalformedUrl method*), 27
__init__() (*pydriller.repository.Repository method*), 27
__module__ (*pydriller.domain.commit.Commit attribute*), 29
__module__ (*pydriller.domain.commit.DMMPProperty attribute*), 31
__module__ (*pydriller.domain.commit.Method attribute*), 31
__module__ (*pydriller.domain.commit.ModificationType attribute*), 32
__module__ (*pydriller.domain.commit.ModifiedFile attribute*), 32
__module__ (*pydriller.domain.developer.Developer attribute*), 33
__module__ (*pydriller.git.Git attribute*), 25
__module__ (*pydriller.metrics.process.process_metric.ProcessMetric attribute*), 34
__module__ (*pydriller.repository.MalformedUrl attribute*), 27
__module__ (*pydriller.repository.Repository attribute*), 28
- A**
 ADD (*pydriller.domain.commit.ModificationType attribute*), 32
 tribute), 32
 added_lines (*pydriller.domain.commit.ModifiedFile attribute*), 32
 author (*pydriller.domain.commit.Commit attribute*), 29
 author_date (*pydriller.domain.commit.Commit attribute*), 29
 author_timezone (*pydriller.domain.commit.Commit attribute*), 29
- B**
 branches (*pydriller.domain.commit.Commit attribute*), 29
- C**
 changed_methods (*pydriller.domain.commit.ModifiedFile attribute*), 32
 checkout() (*pydriller.git.Git method*), 25
 clear() (*pydriller.git.Git method*), 25
 Commit (*class in pydriller.domain.commit*), 29
 committer (*pydriller.domain.commit.Commit attribute*), 29
 committer_date (*pydriller.domain.commit.Commit attribute*), 29
 committer_timezone (*pydriller.domain.commit.Commit attribute*), 29
 complexity (*pydriller.domain.commit.ModifiedFile attribute*), 32
 COPY (*pydriller.domain.commit.ModificationType attribute*), 32
 count() (*pydriller.metrics.process.process_metric.ProcessMetric method*), 34
- D**
 DELETE (*pydriller.domain.commit.ModificationType attribute*), 32
 deleted_lines (*pydriller.domain.commit.ModifiedFile attribute*), 32

deletions (*pydriller.domain.commit.Commit attribute*), 29

Developer (*class in pydriller.domain.developer*), 33

diff_parsed (*pydriller.domain.commit.ModifiedFile attribute*), 32

dmm_unit_complexity (*pydriller.domain.commit.Commit attribute*), 29

dmm_unit_interfacing (*pydriller.domain.commit.Commit attribute*), 30

dmm_unit_size (*pydriller.domain.commit.Commit attribute*), 30

DMMPProperty (*class in pydriller.domain.commit*), 31

F

filename (*pydriller.domain.commit.ModifiedFile attribute*), 32

files (*pydriller.domain.commit.Commit attribute*), 30

files () (*pydriller.git.Git method*), 25

G

get_commit () (*pydriller.git.Git method*), 26

get_commit_from_gitpython () (*pydriller.git.Git method*), 26

get_commit_from_tag () (*pydriller.git.Git method*), 26

get_commits_last_modified_lines () (*pydriller.git.Git method*), 26

get_commits_modified_file () (*pydriller.git.Git method*), 26

get_head () (*pydriller.git.Git method*), 26

get_list_commits () (*pydriller.git.Git method*), 26

get_tagged_commits () (*pydriller.git.Git method*), 27

Git (*class in pydriller.git*), 25

H

hash (*pydriller.domain.commit.Commit attribute*), 30

I

in_main_branch (*pydriller.domain.commit.Commit attribute*), 30

insertions (*pydriller.domain.commit.Commit attribute*), 30

is_low_risk () (*pydriller.domain.commit.Method method*), 31

L

language_supported (*pydriller.domain.commit.ModifiedFile attribute*), 32

lines (*pydriller.domain.commit.Commit attribute*), 30

M

MalformedURLException, 27

merge (*pydriller.domain.commit.Commit attribute*), 30

Method (*class in pydriller.domain.commit*), 31

methods (*pydriller.domain.commit.ModifiedFile attribute*), 33

methods_before (*pydriller.domain.commit.ModifiedFile attribute*), 33

ModificationType (*class in pydriller.domain.commit*), 31

modified_files (*pydriller.domain.commit.Commit attribute*), 30

ModifiedFile (*class in pydriller.domain.commit*), 32

MODIFY (*pydriller.domain.commit.ModificationType attribute*), 32

msg (*pydriller.domain.commit.Commit attribute*), 31

N

new_path (*pydriller.domain.commit.ModifiedFile attribute*), 33

nloc (*pydriller.domain.commit.ModifiedFile attribute*), 33

O

old_path (*pydriller.domain.commit.ModifiedFile attribute*), 33

P

parents (*pydriller.domain.commit.Commit attribute*), 31

ProcessMetric (*class in pydriller.metrics.process.process_metric*), 33

project_name (*pydriller.domain.commit.Commit attribute*), 31

project_path (*pydriller.domain.commit.Commit attribute*), 31

pydriller.__version__ (*built-in variable*), 25

pydriller.domain.commit (*module*), 29

pydriller.domain.developer (*module*), 33

pydriller.git (*module*), 25

pydriller.metrics.process.process_metric (*module*), 33

pydriller.repository (*module*), 27

R

RENAME (*pydriller.domain.commit.ModificationType attribute*), 32

repo (*pydriller.git.Git attribute*), 27

Repository (*class in pydriller.repository*), 27

reset () (*pydriller.git.Git method*), 27

T

`token_count` (`pydriller.domain.commit.ModifiedFile attribute`), 33

`total_commits()` (`pydriller.git.Git method`), 27

`traverse_commits()` (`pydriller.repository.Repository method`), 28

U

`UNIT_COMPLEXITY` (`pydriller.domain.commit.DMMProperty attribute`), 31

`UNIT_COMPLEXITY_LOW_RISK_THRESHOLD` (`pydriller.domain.commit.Method attribute`), 31

`UNIT_INTERFACING` (`pydriller.domain.commit.DMMProperty attribute`), 31

`UNIT_INTERFACING_LOW_RISK_THRESHOLD` (`pydriller.domain.commit.Method attribute`), 31

`UNIT_SIZE` (`pydriller.domain.commit.DMMProperty attribute`), 31

`UNIT_SIZE_LOW_RISK_THRESHOLD` (`pydriller.domain.commit.Method attribute`), 31

`UNKNOWN` (`pydriller.domain.commit.ModificationType attribute`), 32