mutmut Documentation

Anders Hovmöller

Sep 16, 2023

Contents

Mutmut is a mutation testing system for Python, with a strong focus on ease of use. If you don't know what mutation testing is try starting with this article.

Some highlight features:

- Found mutants can be applied on disk with a simple command making it very easy to work with the results
- Remembers work that has been done, so you can work incrementally
- Supports all test runners (because mutmut only needs an exit code from the test command)
- If you use the hammett test runner you can go extremely fast! There's special handling for this runner that has some pretty dramatic results.
- Can use coverage data to only do mutation testing on covered lines
- Battle tested on real libraries by multiple companies

If you need to run mutmut on a python 2 code base use mutmut `1.5.0`. Mutmut `1.9.0` is the last version to support python `3.4`, `3.5` and `3.6`.

CHAPTER 1

Install and run

You can get started with a simple:

```
pip install mutmut
mutmut run
```

This will by default run pytest (or unittest if pytest is unavailable) on tests in the "tests" or "test" folder and it will try to figure out where the code to mutate lies. Run

```
mutmut --help
```

for the available flags, to use other runners, etc. The recommended way to use mutmut if the defaults aren't working for you is to add a block in `setup.cfg`. Then when you come back to mutmut weeks later you don't have to figure out the flags again, just run `mutmut run` and it works. Like this:

```
[mutmut]
paths_to_mutate=src/
backup=False
runner=python -m hammett -x
tests_dir=tests/
dict_synonyms=Struct, NamedStruct
```

To use multiple paths either in the `paths_to_mutate` or `tests_dir` option use a comma or colon separated list. For example:

```
[mutmut]
paths_to_mutate=src/,src2/
tests_dir=tests/:tests2/
```

You can stop the mutation run at any time and mutmut will restart where you left off. It's also smart enough to retest only the surviving mutants when the test suite changes.

To print the results run `mutmut show`. It will give you a list of the mutants grouped by file. You can now look at a specific mutant diff with `mutmut show 3`, all mutants for a specific file with `mutmut show path/to/file.py` or all mutants with `mutmut show all`.

You can also write a mutant to disk with `mutmut apply 3`. You should **REALLY** have the file you mutate under source code control and committed before you apply a mutant!

CHAPTER 2

Whitelisting

You can mark lines like this:

```
some_code_here()  # pragma: no mutate
```

to stop mutation on those lines. Some cases we've found where you need to whitelist lines are:

- The version string on your library. You really shouldn't have a test for this :P
- Optimizing break instead of continue. The code runs fine when mutating break to continue, but it's slower.

See also *Advanced whitelisting and configuration*

CHAPTER 3

Example mutations

- Integer literals are changed by adding 1. So 0 becomes 1, 5 becomes 6, etc.
- < is changed to <=
- break is changed to continue and vice versa

In general the idea is that the mutations should be as subtle as possible. See `__init__.py` for the full list.

CHAPTER 4

Workflow

This section describes how to work with mutmut to enhance your test suite.

1. Run mutmut with `mutmut run`. A full run is preferred but if you're just getting started you can exit in the middle and start working with what you have found so far.
2. Show the mutants with `mutmut results`
3. Apply a surviving mutant to disk running `mutmut apply 3` (replace 3 with the relevant mutant ID from `mutmut results`)
4. Write a new test that fails
5. Revert the mutant on disk
6. Rerun the new test to see that it now passes
7. Go back to point 2.

Mutmut keeps a result cache in `.mutmut-cache` so if you want to make sure you run a full mutmut run just delete this file.

If you want to re-run all survivors after changing a lot of code or even the configuration, you can use *for ID in $(mutmut result-ids survived); do mutmut run $ID; done* (for bash).

You can also tell mutmut to just check a single mutant:

```
mutmut run 3
```

CHAPTER 5

Advanced whitelisting and configuration

mutmut has an advanced configuration system. You create a file called `mutmut_config.py`. You can define two functions there: `init()` and `pre_mutation(context)`. `init` gets called when mutmut starts and `pre_mutation` gets called before each mutant is applied and tested. You can mutate the `context` object as you need. You can modify the test command like this:

```
def pre_mutation(context):
    context.config.test_command = 'python -m pytest -x ' + something_else
```

or skip a mutant:

```
def pre_mutation(context):
    if context.filename == 'foo.py':
        context.skip = True
```

or skip logging:

```
def pre_mutation(context):
    line = context.current_source_line.strip()
    if line.startswith('log.'):
        context.skip = True
```

look at the code for the `Context` class for what you can modify. Please open a github issue if you need help.

It is also possible to disable mutation of specific node types by passing the `--disable-mutation-types` option. Multiple types can be specified by separating them by comma:

```
mutmut run --disable-mutation-types=string,decorator
```

Inversely, you can also only specify to only run specific mutations with `--enable-mutation-types`. Note that `--disable-mutation-types` and `--enable-mutation-types` are exclusive and cannot be combined.

CHAPTER 6

Selecting tests to run

If you have a large test suite or long running tests, it can be beneficial to narrow the set of tests to run for each mutant down to the tests that have a chance of killing it. Determining the relevant subset of tests depends on your project, its structure, and the metadata that you know about your tests. `mutmut` provides information like the file to mutate and coverage contexts (if used with the `--use-coverage` switch). You can set the `context.config.test_command` in the `pre_mutation(context)` hook of `mutmut_config.py`. The `test_command` is reset after each mutant, so you don't have to explicitly (re)set it for each mutant.

This section gives examples to show how this could be done for some concrete use cases. All examples use the default test runner (`python -m pytest -x --assert=plain`).

6.1 Selection based on source and test layout

If the location of the test module has a strict correlation with your source code layout, you can simply construct the path to the corresponding test file from `context.filename`. Suppose your layout follows the following structure where the test file is always located right beside the production code:

```
mypackage
├── production_module.py
├── test_production_module.py
└── subpackage
    ├── submodule.py
    └── test_submodule.py
```

Your `mutmut_config.py` in this case would look like this:

```
import os.path

def pre_mutation(context):
    dirname, filename = os.path.split(context.filename)
    testfile = "test_" + filename
    context.config.test_command += ' ' + os.path.join(dirname, testfile)
```

6.2 Selection based on imports

If you can't rely on the directory structure or naming of the test files, you may assume that the tests most likely to kill the mutant are located in test files that directly import the module that is affected by the mutant. Using the `ast` module of the Python standard library, you can use the `init()` hook to build a map which test file imports which module, and then lookup all test files importing the mutated module and only run those:

```
import ast
from pathlib import Path

test_imports = {}


class ImportVisitor(ast.NodeVisitor):
    """Visitor which records which modules are imported."""
    def __init__(self) -> None:
        super().__init__()
        self.imports = []

    def visit_Import(self, node: ast.Import) -> None:
        for alias in node.names:
            self.imports.append(alias.name)

    def visit_ImportFrom(self, node: ast.ImportFrom) -> None:
        self.imports.append(node.module)


def init():
    """Find all test files located under the 'tests' directory and create an abstract␣
↪syntax tree for each.
    Let the ``ImportVisitor`` find out what modules they import and store the␣
↪information in a global dictionary
    which can be accessed by ``pre_mutation(context)``."""
    test_files = (Path(__file__).parent / "tests").rglob("test*.py")
    for fpath in test_files:
        visitor = ImportVisitor()
        visitor.visit(ast.parse(fpath.read_bytes()))
        test_imports[str(fpath)] = visitor.imports


def pre_mutation(context):
    """Construct the module name from the filename and run all test files which␣
↪import that module."""
    tests_to_run = []
    for testfile, imports in test_imports.items():
        module_name = context.filename.rstrip(".py").replace("/", ".")
        if module_name in imports:
            tests_to_run.append(testfile)
    context.config.test_command += f"{' '.join(tests_to_run)}"
```

6.3 Selection based on coverage contexts

If you recorded coverage contexts and use the `--use-coverage` switch, you can access this coverage data inside the `pre_mutation(context)` hook via the `context.config.coverage_data` attribute. This attribute is a dictionary in the form `{filename: {lineno: [contexts]}}`.

Let's say you have used the built-in dynamic context option of `Coverage.py` by adding the following to your `.coveragerc` file:

```
[run]
dynamic_context = test_function
```

`coverage` will create a new context for each test function that you run in the form `module_name.function_name`. With `pytest`, we can use the `-k` switch to filter tests that match a given expression.

```
import os.path

def pre_mutation(context):
    """Extract the coverage contexts if possible and only run the tests matching this␣
↪data."""
    if not context.config.coverage_data:
        # mutmut was run without ``--use-coverage``
        return
    fname = os.path.abspath(context.filename)
    contexts_for_file = context.config.coverage_data.get(fname, {})
    contexts_for_line = contexts_for_file.get(context.current_line_index, [])
    test_names = [
        ctx.rsplit(".", 1)[-1]  # extract only the final part after the last dot,␣
↪which is the test function name
        for ctx in contexts_for_line
        if ctx  # skip empty strings
    ]
    if not test_names:
        return
    context.config.test_command += f' -k "{" or ".join(test_names)}"'
```

Pay attention that the format of the context name varies depending on the tool you use for creating the contexts. For example, the `pytest-cov` plugin uses `::` as separator between module and test function. Furthermore, not all tools are able to correctly pick up the correct contexts. `coverage.py` for instance is (at the time of writing) unable to pick up tests that are inside a class when using `pytest`. You will have to inspect your `.coverage` database using the Coverage.py API first to determine how you can extract the correct information to use with your test runner.

6.4 Making things more robust

Despite your best efforts in picking the right subset of tests, it may happen that the mutant survives because the test which is able to kill it was not included in the test set. You can tell `mutmut` to re-run the full test suite in that case, to verify that this mutant indeed survives. You can do so by passing the `--rerun-all` option to `mutmut run`. This option is disabled by default.

CHAPTER 7

JUnit XML support

In order to better integrate with CI/CD systems, `mutmut` supports the generation of a JUnit XML report (using https://pypi.org/project/junit-xml/). This option is available by calling `mutmut junitxml`. In order to define how to deal with suspicious and untested mutants, you can use

```
mutmut junitxml --suspicious-policy=ignore --untested-policy=ignore
```

The possible values for these policies are:

- `ignore`: Do not include the results on the report at all
- `skipped`: Include the mutant on the report as "skipped"
- `error`: Include the mutant on the report as "error"
- `failure`: Include the mutant on the report as "failure"

If a failed mutant is included in the report, then the unified diff of the mutant will also be included for debugging purposes.

CHAPTER 8

Resources

- Source Code on Github
- Travis Testing
- Python Package Index