pytest-inline: An Inline Testing Tool for Python

Yu Liu[1], Zachary Thurston[2], Alan Han[2], Pengyu Nie[1], Milos Gligoric[1], Owolabi Legunsen[2]
yuki.liu@utexas.edu, {zwt3, ayh9}@cornell.edu, {pynie, gligoric}@utexas.edu, legunsen@cornell.edu
[1] UT Austin, USA, [2] Cornell University, USA

***Abstract*—We present pytest-inline, the first inline testing framework for Python. We recently proposed inline tests to make it easier to test individual program statements, but there is currently no framework-level support available for developers to write inline tests in Python. To fill this gap, we design and implement pytest-inline as a pytest plugin, which is the most popular Python testing framework. In pytest-inline, a developer can write inline tests by assigning test inputs to variables in the target statement and specifying the expected outputs. Then, pytest-inline runs each inline test and fails if the target statement's output does not match the expected result. In this paper, we describe the design of pytest-inline, the testing features that it provides, and the intended use cases. Our evaluation of pytest-inline on the inline tests we wrote for 80 target statements from 31 open-source Python projects shows that using it to run inline tests incurs negligible overhead, at 0.012x. pytest-inline is open-sourced, and a video demo of pytest-inline can be found at https://www.youtube.com/watch?v=pZgiAxR_uJg.**

***Index Terms*—inline tests, software testing, Python, pytest**

I. Introduction

Software testing is the main way of checking code quality, but there is a gap in today's testing frameworks: they do not support testing *individual statements*. That is, the unit tests [1], integration tests [2], and system tests [3] supported by current frameworks can be too coarse-grained or ill-suited for developer testing needs that exist at the statement level. Yet, developers may want to test statements because:

1) Single-statement bugs occur frequently [4], but unit tests often do not catch single-statement bugs [5].
2) Some statements are hard to understand or are error-prone, e.g., regular expressions (regexes) [6], bit manipulation [7], string manipulation [8], or collection handling [9].
3) Complex logic can be in a statement instead of a method or function, e.g., Python one-liners [10] or Java streams [11].
4) The statement that developers want to check, i.e., the *target statement*, may be buried deeply in complicated logic that is hard to check with unit tests.

Without framework-level support for testing statements, developers use *ad hoc* approaches, like (1) "printf debugging"—printing values of variables to the console to gain visibility [12], or (2) using websites or in-IDE pop-ups to test regexes [13]. These are not ideal: developers wastefully add and then remove print statements, and lose mental focus and productivity to copy code to and from websites and pop-ups. Also, developers cannot easily reuse the outcomes of these methods. Lastly, if a target statement is in privately accessible code, some developers violate core software engineering principles to enable unit testing.

We proposed *inline tests* to meet developer needs below the unit-testing level and to provide framework-level support for testing statements [14]. An inline test is a statement that allows providing arbitrary inputs and test oracles for checking the immediately preceding statement that is not an inline test. Inline tests can bring the power of unit tests to the statement level, but they should not replace unit tests or debuggers [14].

This paper presents *pytest-inline*, a framework for writing inline tests in Python. Using *pytest-inline*, users can assign test inputs to variables in a target statement and use a provided API to write oracles that specify the expected outputs. Also, *pytest-inline* runs inline tests in an isolated context and does not require interpreting the whole project. To facilitate ease of installation and use, we develop *pytest-inline* as a plugin for pytest [15]. We choose pytest because it is the most popular Python testing framework and is extensible using plugins.

We build *pytest-inline* by extending the prototype in our original paper [14]. The original prototype has features of three kinds of test oracles, setting test display names, parameterized tests, disabled tests, grouping tests by tags, and repeated tests. In *pytest-inline*, we implement all JUnit [16] (a mature testing framework for Java) features that are applicable to inline tests:

1) Five other kinds of test oracles;
2) Timeout;
3) Specifying test order;
4) Running inline tests in parallel;
5) Specifying assumptions.

We are also submitting *pytest-inline* as an officially-supported pytest plugin [17].

We evaluate *pytest-inline* on the 31 open-source Python projects and their 80 statements with 87 inline tests in our original paper. We find that the overhead to run inline tests is negligible, at 0.012x of the time to run unit tests. Our user study on the original prototype showed that all nine participants find inline tests easy to write and say that most inline tests are beneficial. Our *pytest-inline* tool will enable further research on inline testing.

We make *pytest-inline* publicly available on GitHub: https://github.com/EngineeringSoftware/pytest-inline.

II. Example

Fig. 1 shows an inline test for code that we simplified from google-research/bert [18]. Line 5 checks if the variable `name` matches a regex for a pattern that ends in a colon and has at least one digit. Directly checking that regex is not easy without statement-level testing: it is buried in a `for` loop, and the matched result is not the return value of the function.

```
1  def get_assignment_map_from_checkpoint(tvars, init_c):
2      ...
3      for var in tvars:
4          name = var.name
5          m = re.match("^(.*):\\d+$", name)
6          Here().given(name, "a:0").check_eq(m, "a")
7          if m is not None:
8              name = m.group(1)
9      ...
```

Fig. 1: Example Python code with an inline test in blue.

The inline test that we write for Line 5 is on Line 6. Every inline test has three parts. The "Declare" (`Here()`) part tells *pytest-inline* that the statement is an inline test. The "Assign" (`given(name, "a:0")`) part allows providing test inputs for the variables in the target statement. In this case, `"a:0"` is input for `name`. Finally, the "Assert" (`check_eq(m, "a")`) part allows specifying a test oracle. In this case, given the test input for `name`, the `m.group(1)` that the target statement computes should be `"a"` for this inline test to pass.

III. The *pytest-inline* Framework

A. API

The *pytest-inline* API provides functions for:

1) **Declare**. This API component, `Here()`, signals *pytest-inline* to process a statement as an inline test and allows users to optionally specify (1) a customized test name, (2) if it is a parameterized inline test, (3) a number of times to repeat the inline test, (4) a list of tags for filtering tests, (5) if the inline test is disabled, or (6) a timeout.
2) **Assign**. This API component, `given()`, allows providing test inputs for inline tests; it takes two arguments: a variable that is used in the target statement and the value that is assigned to that variable.
3) **Assert**. This API component, `check_*`, allows specifying inline test oracles. The `check_eq`, `check_neq`, `check_same`, and `check_not_same` functions take the expected value and the actual value. The `check_true`, `check_false`, `check_none`, and `check_not_none` functions take only the actual value.

B. Features

Table I lists the features that we implement in *pytest-inline* with examples. The top five rows show features that were in our original prototype [14], and the bottom five rows show new features that we add in *pytest-inline*. To build upon our original prototype, we analyze features that JUnit [16] provides and extend the *pytest-inline* API to support those that are applicable to inline tests.

Parameterized inline tests allow testing the same target statement on multiple pairs of inputs and outputs. Timeout can be provided so that *pytest-inline* terminates after a specified duration (which can be useful if there is a deadlock). Running inline tests in parallel can save time. Developers can specify names, tags, and test orders to organize their inline tests. Tags can be used for marking and filtering tests. One test can have multiple tags while only one display name. We also add features to allow developers to specify how many times to repeat an inline test and whether to temporarily disable inline tests. Repeating tests is useful for detecting flaky tests [19]. Disabling tests can be used to skip failed tests until the bug is fixed. *pytest-inline* supports the use of assumptions, with which tests only execute when a certain pre-condition is satisfied. Finally, we have added five new kinds of test oracles to increase the readability of tests.

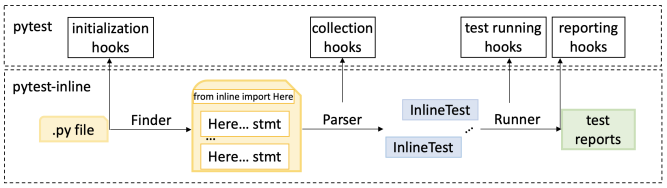

Fig. 2: Architecture of *pytest-inline*.

```
(inline-dev) liuyu@luzhou:~/bert$ pytest modeling.py
============================= test session starts ==============================
platform linux -- Python 3.9.15, pytest-7.2.0, pluggy-1.0.0
rootdir: /home/liuyu/bert
plugins: inline-0.1.0
collected 1 item

modeling.py .                                                            [100%]

============================== 1 passed in 0.02s ===============================
```

Fig. 3: Sample *pytest-inline* output when an inline test passes.

C. Implementation

Fig. 2 shows the architecture of *pytest-inline*; it has three components: (1) Finder, (2) Parser, and (3) Runner.

Finder. Given a source file, Finder obtains its abstract syntax tree (AST) using the Python AST library [21]. Within the AST, Finder locates the import of `Here` and the statements that start with `Here()`.

Parser. Given the output of Finder, Parser first traverses the AST using the visitor pattern to discover each inline test and its target statement—the first statement that precedes the inline test but is not itself an inline test. Then Parser (1) extracts values assigned to the arguments in the `Here()` constructor, (2) extracts the assumption in `assume()` if it exists, (3) constructs an assignment statement from each `given()`, (4) constructs an assertion statement from each `check_*()`. Parser throws a `MalformedException` if *pytest-inline*'s API is being misused. Lastly, Parser constructs a program encapsulating the inline test with the parsed assignment statements, target statement, and assertion statements. If there is an assumption, Parser wraps the program in an if statement with the assumption as the condition.

Runner. Given the program encapsulating each inline test from Parser, Runner executes the program in an isolated context containing only the local variables it needs and then produces a test outcome (pass/fail). Runner automatically imports the libraries required by the program (e.g., `re`) so that developers do not have to write import statements in inline tests. The test outcome is formatted as standard pytest output, as shown in Fig. 3.

Integration with pytest. We use pytest's hook functions to implement *pytest-inline* as a pytest plugin, namely by extending and customizing pytest's configuration, collection,

TABLE I: The features in *pytest-inline*. The top five rows are in our original prototype [14]; the bottom five are new.

	Feature	Description	Example
Prototype	display name	Customize test name	Here(test_name="check_match_name")...
	parameterized tests	Pass different sets of inputs to tests	Here(parameterized=True).given(name, ["a:0", "a:1:1"]) .check_eq(m.group(1), ["a", "a:1"])
	repeated tests	Repeat a test a specified number of times	Here(repeated=2)...
	tagged tests	Tag tests for filtering	Here(tag=["regex"])... `$ pytest --inlinetest-group="tag-name"`
	disabled tests	Disable a test	Here(disabled=True)...
New	timeout	Fail a test if execution time exceeds given duration	Here(timeout=5)...
	assumptions	Execute a test when the assumption is satisfied	Here().assume(platform.system() == "Linux")...
	run tests in order	Run some tests first	`$ pytest --inlinetest-order="tag-name"`
	run tests in parallel	Run tests in parallel with plugin pytest-xdist [20]	`$ pytest -n auto`
	more kinds of oracles	check_neq, check_none, check_not_none, check_same, check_not_same	Here().given(name, "a:a").check_none(m)

```
(inline-dev) liuyu@luzhou:~/bert$ pytest modeling.py
============================= test session starts ==============================
platform linux -- Python 3.9.15, pytest-7.2.0, pluggy-1.0.0
rootdir: /home/liuyu/bert
plugins: inline-0.1.0
collected 1 item

modeling.py F                                                            [100%]

================================== FAILURES ===================================
_____________________ [inlinetest] line326 _____________________________________
<ast>:7: AssertionError
E   AssertionError: m.group(1) == 'aa'
E   Actual: a
E   Expected: aa
=========================== short test summary info ============================
FAILED modeling.py::line326 - AssertionError: m.group(1) == 'aa'
============================== 1 failed in 0.06s ===============================
```

Fig. 4: Sample *pytest-inline* output when an inline test fails.

running, and reporting phases. For example, we hook the `pytest_exception_interact` function to customize error reporting to pretty-print the failing assertion, expected output, and the observed output (instead of a long stack trace). Fig. 4 shows the output of an example failing inline test (changing `check_eq(m.group(1), "a")` to `check_eq(m.group(1), "aa")` in Fig. 1). Fig. 2 shows the other hooks that *pytest-inline* uses.

IV. Installation and Usage

Installation. Conda [22] is the recommended package manager for installing pytest and *pytest-inline*. A Conda environment with Python 3.9 can be created like so (pytest requires Python 3.7 or higher):

```
$ conda create --name inlinetest python=3.9 pip -y
$ conda activate inlinetest
```

The next step is to install pytest and *pytest-inline* in the Conda environment like so:

```
$ pip install pytest-inline
```

Usage. By default, pytest recursively discovers and runs all "test_*.py" or "*_test.py" files in the current directory. Our *pytest-inline* plugin also recursively processes all ".py" files in the current directory. Users can use patterns to specify what files to process. For example, this command runs all tests in files that start with "a" and end with ".py":

```
$ pytest  a*.py
```

Users of *pytest-inline* can use the `inlinetest-group` option to run inline tests with the specified tag:

```
$ # run the inline test with the tag "add"
$ pytest --inlinetest-group="add"
```

The `-k` option allows specifying an inline test to run by name:

```
$ pytest -k "add" # run the inline test named add
```

Tests can be run in three modes in *pytest-inline*: default, inlinetest-only, and inlinetest-disable. The default mode runs both inline tests and unit tests. The inlinetest-only mode runs only inline tests, and the inlinetest-disable mode skips inline tests and runs only unit tests:

```
$ pytest # run all tests
$ pytest --inlinetest-only  # run only inline tests
$ pytest --inlinetest-disable  # skip inline tests
```

When collecting tests, *pytest-inline* tries to import dependencies of the module and throws an error if those dependencies are not installed. Users can use `inlinetest-ignore-import-errors` to ignore such errors and skip the collection of the affected files (but also skip the inline tests in those files):

```
$ pytest --inlinetest-ignore-import-errors
```

The default order of running inline tests in each file is by line number order. Users can use tags and `inlinetest-order` to override the test-run order:

```
# run test tagged "str", then "bit", and then the rest
$ pytest --inlinetest-order="str" --inlinetest-order="bit"
```

Users can filter out inline tests that should not be run by using `inlinetest-group` with a tag name. Doing so only runs tests with the given tag(s):

```
# run only the tests with tags "str" and "bit"
$ pytest --inlinetest-group="str" --inlinetest-group="bit"
```

In *pytest-inline*, users can run inline tests in parallel after installing the plugin, pytest-xdist [20], and using the `-n` option to specify the number of processes to use.

```
$ pip install pytest-xdist
$ pytest -n 4 # run tests in parallel with 4 processes
$ pytest -n auto # run tests in parallel with all CPU cores
```

Lastly, to generate test reports in HTML format, users can use the pytest-html plugin and the `html` option:

```
$ pip install pytest-html
$ pytest --html=report.html
```

V. Evaluation

We present the results of our evaluation of *pytest-inline*'s performance when running inline tests. We use the same environment to run these experiments as in our original paper [14].

TABLE II: Results of standalone experiments. **Dup** = duplication count, **#IT**= total no. of inline tests, $T_{\mathbf{IT}}$**[s]**= total inline tests run time, $t_{\mathbf{IT}}$**[s]**= average run time per inline test

Dup	#IT	$T_{\mathbf{IT}}$[s]	$t_{\mathbf{IT}}$[s]
x1	87	8.21	0.094
x10	870	8.84	0.010
x100	8,700	15.21	0.002
x1000	87,000	120.17	0.001

TABLE III: Results of integrated experiments. **Dup** = duplication times, **#UT**= total no. of unit tests, **#IT**= total no. of inline tests, $T_{\mathbf{UT}}$ **[s]**= total time to run unit tests, $T_{\mathbf{ITE}}$ **[s]**= total time to run unit tests with inline tests enabled, $O_{\mathbf{ITE}}$= overhead of running unit tests with inline tests enabled, $T_{\mathbf{ITD}}$**[s]**= total time to run unit tests with inline tests disabled, $O_{\mathbf{ITD}}$= overhead of running unit tests with inline tests disabled.

Dup	#UT	#IT	$T_{\mathbf{UT}}$ [s]	$T_{\mathbf{ITE}}$ [s]	$O_{\mathbf{ITE}}$	$T_{\mathbf{ITD}}$[s]	$O_{\mathbf{ITD}}$
x1	160,111	27	599.09	606.19	0.012	601.22	0.004
x10	160,111	270	603.29	607.20	0.006	601.94	-0.002
x100	160,112	2,700	593.93	638.02	0.074	630.42	0.061
x1000	160,113	27,000	649.53	689.50	0.062	640.93	-0.013

Standalone experiments. To measure the cost of running inline tests, we run the same inline tests as in our original paper [14]. These are 87 inline tests that we manually wrote for 80 statements in 50 examples from 31 Python projects. Since inline tests are still new and not abundant on open-source projects, it is hard to assess *pytest-inline* costs as the number of inline tests grows. In the interim, we follow the same approach as in our original paper to simulate such costs: duplicating each inline test 10, 100, and 1000 times.

Table II shows the times to run *pytest-inline* with varying number of tests. Without duplication, the average time per inline test is 0.094s. With duplication, the average time per inline test gradually reduces to 0.001s, likely for two reasons. First, the cost of discovering inline tests is amortized with duplication, so the actual cost per inline test could be slightly higher. Second, repeatedly running an inline test benefits from reduced warm-up time. The total time to run all inline tests is almost constant when we duplicate each inline test 10 or 100 times, but that time grows dramatically when we duplicate 1000 times. This dramatic growth suggests that regression testing techniques [23] will be needed to reduce inline testing costs. *pytest-inline* can be the basis on which to build those regression testing techniques. Overall, we conclude that the time required for running each inline test is tiny.

Integrated experiments. We also measure the overhead of running inline tests and unit tests together in the runtime environment specified by each project. To do so, we run inline tests and unit tests four times. The first run is for warm-up, and we average the times for the last three runs. Among 31 Python projects in our original paper, we choose the ten whose unit testing environment can be successfully configured with Python 3.7 or greater (as required by pytest): `bokeh/bokeh`, `RaRe-Technologies/gensim`, `geekcomputers/Python`, `joke2k/faker`, `mitmproxy/mitmproxy`, `numpy/numpy`, `pandas-dev/pandas`, `psf/black`, `pypa/pipenv`, and `scrapy/scrapy`. Table III shows the results. There, $O_{\mathbf{ITE}}$ is the overhead when inline tests are enabled and run with unit tests. Without duplication, the overhead per inline test is negligible, at 0.012x. The overhead is similar with duplication. For example, when duplicating inline tests 1000 times, which brings the number of inline tests close to that of unit tests, the overhead is 0.062x.

On user perceptions. The user study that we performed using the original Python prototype [14] showed that participants found inline testing easy to use and beneficial. Now that we released *pytest-inline* and are submitting it as an official pytest plugin for developers and researchers to use, we will be able to continuously obtain feedback from users.

VI. Conclusion and Future Work

We presented *pytest-inline* for writing inline tests in Python. We implemented *pytest-inline* as a pytest plugin and we are submitting it to the pytest developers so that it becomes official and community-maintained. Our performance evaluation of *pytest-inline* showed that the cost of running inline tests is negligible, and our original prototype already helped find two accepted bugs in two projects. In the future, we will add more features to *pytest-inline* based on community feedback, and use it to advance research on inline testing.

References

[1] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *ISSRE*, 2014, pp. 201–211.
[2] A. Orso, "Integration testing of object-oriented software," p. 119, 1998.
[3] W. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-end integration testing design," in *COMPSAC*, 2001, pp. 166–171.
[4] A. V. Kamienski, L. Palechor, C.-P. Bezemer, and A. Hindle, "PySStuBs: Characterizing single-statement bugs in popular open-source python projects," in *MSR*, 2021, pp. 520–524.
[5] J. Latendresse, R. Abdalkareem, D. E. Costa, and E. Shihab, "How effective is continuous integration in indicating single-statement bugs?" in *MSR*, 2021, pp. 500–504.
[6] L. G. Michael, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions," in *ASE*, 2019, pp. 415–426.
[7] S. Bae, "Bit manipulation," in *JavaScript Data Structures and Algorithms*, 2019, pp. 339–349.
[8] A. Eghbali and M. Pradel, "No strings attached: An empirical study of string-related software bugs," in *ASE*, 2020, pp. 956–967.
[9] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in Python," in *ICST*, 2021, pp. 148–158.
[10] C. Mayer, *Python One-Liners: Write Concise, Eloquent Python Like a Professional*. No Starch Press, 2020.
[11] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.
[12] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *SQJ*, vol. 25, no. 1, pp. 83–110, 2017.
[13] https://regex101.com.
[14] Y. Liu, P. Nie, O. Legunsen, and M. Gligoric, "Inline tests," in *ASE*, 2022.
[15] https://docs.pytest.org/en/7.2.x.
[16] https://junit.org/junit5/.
[17] https://github.com/pytest-dev.
[18] https://github.com/google-research/bert.
[19] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019, pp. 312–322.
[20] https://github.com/pytest-dev/pytest-xdist.
[21] https://github.com/python/cpython/blob/main/Lib/ast.py.
[22] https://docs.conda.io/projects/conda/en/stable.
[23] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, "Evaluating regression test selection opportunities in a very large open-source ecosystem," in *ISSRE*, 2018, pp. 112–122.