Test Polarity: Detecting Positive and Negative Tests

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

ABSTRACT

Positive tests (aka, happy path tests) cover the expected behavior of the program, while negative tests (aka, unhappy path tests) check the unexpected behavior. Ideally, test suites should have both positive and negative tests to better protect against regressions. In practice, unfortunately, we cannot easily identify whether a test is positive or negative. A better understanding of whether a test suite is more positive or negative is fundamental to assessing the overall test suite capability in testing expected and unexpected behaviors. In this paper, we propose *test polarity*, an automated approach to detect positive and negative tests. Our approach runs/monitors the test suite and collects runtime data about the application execution to classify the test methods as positive or negative. In a first evaluation, test polarity correctly classified 117 tests as as positive or negative. Finally, we provide a preliminary empirical study to analyze the test polarity of 2,054 test methods from 12 real-world test suites of the Python Standard Library. We find that most of the analyzed test methods are negative (88%) and a minority is positive (12%). However, there is a large variation per project: while some libraries have an equivalent number of positive and negative tests, others have mostly negative ones.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Runtime environments**.

KEYWORDS

software testing, test quality, runtime monitoring, dynamic analysis, python

ACM Reference Format:
Andre Hora. 2024. Test Polarity: Detecting Positive and Negative Tests. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663793

```python
class MonthRangeTestCase(unittest.TestCase):
    def test_january(self):
        # Tests valid lower boundary case.
        self.assertEqual(calendar.monthrange(2004,1), (3,31))

    def test_february_leap(self):
        # Tests February during leap year.
        self.assertEqual(calendar.monthrange(2004,2), (6,29))

    def test_february_nonleap(self):
        # Tests February in non-leap year.
        self.assertEqual(calendar.monthrange(2010,2), (0,28))

    def test_december(self):
        # Tests valid upper boundary case.
        self.assertEqual(calendar.monthrange(2004,12), (2,31))

    def test_zeroth_month(self):
        # Tests low invalid boundary case.
        with self.assertRaises(calendar.IllegalMonthError):
            calendar.monthrange(2004, 0)

    def test_thirteenth_month(self):
        # Tests high invalid boundary case.
        with self.assertRaises(calendar.IllegalMonthError):
            calendar.monthrange(2004, 13)

    def test_illegal_month_reported(self):
        with self.assertRaisesRegex(calendar.IllegalMonthError, '65'):
            calendar.monthrange(2004, 65)
```

Figure 1: Test class `MonthRangeTestCase` of the calendar Python Standard Library. Positive test: ⊕. Negative test: ⊖.

1 INTRODUCTION

Positive tests (also known as happy path, sunny day, and good weather tests) cover expected behaviors of the program, that is, the normal execution, in which nothing goes wrong [20]. In contrast, negative tests (also known as unhappy path and bad weather tests) check unexpected behaviors, that is, the abnormal execution, like exceptional cases. Ideally, test suites should have both positive and negative tests to catch more bugs, protect against regressions, and ensure sustainable software evolution [1, 4, 9, 15, 16, 27].

For example, consider the test class `MonthRangeTestCase`[1] presented in Figure 1, which tests method `monthrange`[2] of the calendar Python Standard Library. This method computes the number of days in a given month and throws an exception when the month is invalid. Fortunately, the tests have comments with hints on what cases they are verifying. For instance, `test_january` and `test_december` are testing valid cases, that is, the expected months 1 (for January) and 12 (for December). Similarly, `test_february_leap` and `test_february_nonleap` also test valid cases, *i.e.,* February in leap and non-leap years. In contrast, `test_zeroth_month` and `test_thirteenth_month` are testing invalid cases, that is, the unexpected months 0 and 13. Lastly, `test_illegal_month_reported` checks another invalid input, as its name properly suggests. Based on the test names, comments, and code, if we could flag those

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663793

[1] https://github.com/python/cpython/blob/92ed7e4/Lib/test/test_calendar.py#L877
[2] https://github.com/python/cpython/blob/92ed7e4/Lib/calendar.py#L161

tests as positive or negative, we would possibly classify the first four tests as positive and the last three tests as negative. The fact that the class `MonthRangeTestCase` has both positive and negative tests is an indication that it has good protection against regressions [1, 4, 9, 15, 16, 27].

In practice, unfortunately, we cannot easily identify whether a test is positive or negative. One solution would be to rely on the test names and comments (like in the presented example). However, test methods face the same issues of application method names, that is, they are often poorly named, with generic, meaningless, and obsolete names [16, 18, 27]. Another solution would be to inspect the test code itself. That would work for simple tests (like in the presented example), however, real-world test suites may be complex and large, with thousands of test methods. In addition, that would require developers with expertise on the subject. Therefore, it is not feasible to confidently rely on test names, comments, or code to properly detect positive or negative tests.

To the best of our knowledge, there is no approach to detect positive and negative tests in a simple and explainable way. Better understanding whether a test suite is more positive or negative is fundamental to (i) *assess the overall test suite capability in testing expected and unexpected behaviors* and (ii) *have actionable data to improve the tests*. If a test suite is over-concentrated on positive tests, while neglecting the negative ones, this may suggest that unexpected behaviors are not being properly tested. On the other hand, if a test suite only focuses on negative tests, this may suggest that the expected behaviors are not covered.

In this paper, we propose *test polarity*, an automated approach to detect positive and negative tests. Our approach runs/monitors the test suite and collects runtime data about the application execution to classify the test methods as positive or negative. We evaluate test polarity by assessing its precision in correctly classifying the tests as positive or negative. For this purpose, we manually classified 117 test methods and our automated approach correctly classified all 117 test methods.

Finally, we provide an initial empirical study to analyze the test polarity of 2,054 tests from 12 real-world test suites of the Python Standard Library. We find that most of the analyzed test methods are negative (88%) and a minority is positive (12%). However, there is a large variation per project: while some libraries have an equivalent number of positive and negative tests, others have mostly negative ones. Our results are publicly available at: https://doi.org/10.5281/zenodo.10149477.

Contributions. The contributions of this paper are threefold. First, we propose *test polarity*, an automated approach to detect positive and negative tests (Section 2). Second, we provide an evaluation of *test polarity* (Section 3). Third, we provide a preliminary empirical study of test polarity on real-world test suites (Section 4).

2 TEST POLARITY

Test polarity is an automated approach to detect positive and negative tests. It runs the test suite and collects runtime data about the application execution to classify the test methods as positive or negative. Specifically, it has three major steps, as summarized in Figure 2. First, it identifies the tested paths of each application method and ranks them according to their execution frequency, creating a ranking of tested paths. Second, for each test method, it detects which tested paths are executed and their respective position in the ranking. Finally, it classifies each test method as positive or negative based on the ranking position of their tested paths.

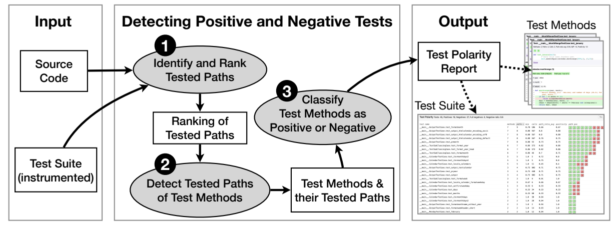

Figure 2: Overview of the test polarity approach.

2.1 Identify and Rank Tested Paths

First, we run and monitor the test suite of the target application. In this process, we collect application methods executed by the test suite as well as their respective tested paths. A *tested path* represents a set of input values that will make the method behave in the same way, that is, execute the same lines of code. We define a *tested path* of an executed method m as a sorted set of executed lines of code of m. This definition makes the paths computable and comparable. For example, after running the test suite of `test_calendar`[3] of the Python Standard Library, two tested paths are identified in method `monthrange` (Figure 3): Tested Path 1 with lines {164, 166, 167, 168}, which represents the "normal" execution, and Tested Path 2 with lines {164, 165}, which represents the "abnormal" execution, when an exception is thrown.

```
161 def monthrange(year, month):
162     """Return weekday (0-6 ~ Mon-Sun) and number of days (28-31) for
163        year, month."""
164     if not 1 <= month <= 12:
165         raise IllegalMonthError(month)
166     day1 = weekday(year, month, 1)
167     ndays = mdays[month] + (month == FEBRUARY and isleap(year))
168     return day1, ndays
```

Figure 3: Method `monthrange` of the Python Standard Library.

Next, for each application method, we rank their tested paths according to their call frequency, creating a *ranking of tested paths*. In this ranking, the most called paths are top-ranked, while the least called ones are bottom-ranked. For example, the ranking of tested paths for method `monthrange` is: (1st) Tested Path 1 with 218 calls and (2nd) Tested Path 2 with 3 calls, as summarized in Table 1.

Table 1: Ranking of tested paths (`monthrange`).

Pos	Path	Lines of Code	Calls
1	Tested Path 1	164, 166, 167, 168	218
2	Tested Path 2	164, 165	3

[3] https://github.com/python/cpython/blob/92ed7e4/Lib/test/test_calendar.py

2.2 Detect Tested Paths of Test Methods

Next, for each test method, we identify which application methods are executed. Then, we detect which tested paths are called and their respective position in the *ranking of tested paths*.

For example, the test method `test_january` of `MonthRangeTestCase` (see Figure 1) execute two application methods: `monthrange` and `weekday`. Method `monthrange` has two tested paths (as we have seen in the previous step). Method `weekday` also has two tested paths (Tested Path 1 with 193 calls and Tested Path 2 with 25 calls). Specifically, the test method `test_january` calls the Tested Path 1 of `monthrange` and the Tested Path 1 of `weekday`. On the other hand, the test method `test_illegal_month_reported` executes only one application method: `monthrange`. However, differently from `test_january`, the test method `test_illegal_month_reported` calls the Tested Path 2 of `monthrange`. Table 2 summarizes the tested paths of the test methods in `MonthRangeTestCase`.

Table 2: Test methods and tested paths (`MonthRangeTestCase`).

Test Methods	Tested Paths	
	`monthrange`	`weekday`
`test_january`	TP1	TP1
`test_february_leap`	TP1	TP1
`test_february_nonleap`	TP1	TP1
`test_december`	TP1	TP1
`test_zeroth_month`	TP2	-
`test_thirteenth_month`	TP2	-
`test_illegal_month_reported`	TP2	-

2.3 Classify Tests as Positive or Negative

The final step is to compute the polarity of the test methods, that is, classify the test methods as positive or negative. For this purpose, we rely on the ranking position of their tested paths. Test methods that always execute the top-ranked tested paths are classified as positive, otherwise, they are classified as negative. The rationale is that the top-ranked tested path represents the "happy path", *i.e.,* the normal behavior of the method [20]. This way, if a test method always stays on the happy path, it can be seen as a positive test. In contrast, if a test method deviates from the happy path, it can be seen as a negative test. Section 3 provides a detailed evaluation of the proposed approach.

For example, `test_january` calls two top-ranked tested paths (*i.e.,* TP1 of `monthrange` and TP1 of `weekday`), thus, it is classified as positive (%TP1 is 100%). In contrast, `test_illegal_month_reported` calls a non-top-ranked tested path (*i.e.,* TP2 of `monthrange`), thus, it is classified as negative (%TP1 is 0%). Table 3 summarizes the polarity in `MonthRangeTestCase`.

2.4 Implementation Notes

We developed a tool that implements the proposed test polarity approach. This tool is implemented in Python and targets Python systems. Our tool is implemented with the support of the standard trace function [22, 24], which is the basis for performing runtime analysis in Python [12, 13]. Our solution relies on SpotFlow [14], a tool to ease runtime analysis in Python.[4]

[4] https://github.com/andrehora/spotflow

Table 3: Summary of the test polarity (`MonthRangeTestCase`).

Test Methods	Tested Paths	%TP1	Polarity
`test_january`	TP1, TP1	100%	
`test_february_leap`	TP1, TP1	100%	
`test_february_nonleap`	TP1, TP1	100%	
`test_december`	TP1, TP1	100%	
`test_zeroth_month`	TP2	0%	
`test_thirteenth_month`	TP2	0%	
`test_illegal_month_reported`	TP2	0%	

3 PRELIMINARY EVALUATION

3.1 Design

To evaluate test polarity, we assess its precision in correctly classifying the tests as positive or negative. For this purpose, we analyze real-world test suites of 12 libraries provided by the Python Standard Library: gzip, email, calendar, ftplib, collections, os, tarfile, pathlib, logging, smtplib, argparse, and configparser. Those libraries are fundamental to building any Python application, allowing it to handle emails, logging, operating systems, collections, etc.

In total, test polarity classified 2,054 test methods and we randomly selected 324 (*i.e.,* 95% confidence level and 5% confidence interval) to perform a manual analysis. For each selected test method, we manually inspected their source code and manually classified it as positive (when it only verified valid and expected cases) or negative (when it verified invalid, unexpected, and exceptional cases). In both cases, we considered test names, test comments, variable names, exception raising, references to issues, and any available resources to manually classify the test method. In case of unclear polarity, we did not classify the test method. Finally, we verified whether test polarity correctly classified the test methods as positive or negative taking the manual classification as an oracle.

3.2 Results

Among the 324 randomly selected test methods, we could manually classify 117 (112 negatives and 5 positives), while 207 test methods had unclear polarity (the high rate of tests with unclear polarity reinforces the need for an automated approach, like test polarity). Our automated approach correctly classified all 117 test methods.

Negative tests. Overall, the negative tests checked exceptional or edge cases. For example, the gzip test `test_bad_gzip_file`[5] (Figure 4) was manually classified as negative because it verifies the creation of an invalid gzip file. Our approach classified this test as negative because it executed five bottom-ranked tested paths. As another example of correctly classified negative test, the email test `test_long_lines`[6] verifies multiple edge cases of the `parse` function, like long strings with carriage return separator, newline separator, and special characters.

Positive tests. Positive tests were harder to manually classify than negative ones because exceptional and edge cases are easier to spot in the test. Figure 5 presents a positive test in the collections

[5] https://github.com/python/cpython/blob/97ce15c5/Lib/test/test_gzip.py#L428

[6] https://github.com/python/cpython/blob/97ce15c5/Lib/test/test_email/test_email.py#L3625

```
def test_bad_gzip_file(self):
    with open(self.filename, 'wb') as file:
        file.write(data1 * 50)
    with gzip.GzipFile(self.filename, 'r') as file:
        self.assertRaises(gzip.BadGzipFile, file.readlines)
```

Figure 4: Negative test in gzip (`test_bad_gzip_file`).

library. The test `test_match_args`[7] checks if the `__match_args__` attribute of `Point` contains the correct field names `x` and `y`. This test executes the top-1 tested path of `namedtuple`, thus, it is classified as positive. As another example of a correctly classified positive test, `test_april`[8] of calendar checks the number of days in April given different starting days of the week. In this case, the test executes the top-1 tested paths of `monthrange` and `weekday`.

```
def test_match_args(self):
    Point = namedtuple('Point', 'x y')
    self.assertEqual(Point.__match_args__, ('x', 'y'))
```

Figure 5: Positive test in collections (`test_match_args`).

4 PRELIMINARY EMPIRICAL STUDY

RQ: What is the test polarity of real-world test suites? We compute the test polarity of the 12 Python libraries presented in Section 3 and the results are summarized in Figure 6. Overall, we find more negative tests than positive ones: considering the 2,054 analyzed test methods, 88% are negative, while 12% are positive. However, there is a large variation per library. The libraries with the most positive tests are ftplib (51%), os (47%), and calendar (40%). In contrast, gzip and smtplib have only negative tests. Indeed, well-established and critical projects (like the Python Standard Library) are likely to be formed by experienced developers [7, 11, 26], who are more likely to test unexpected/unhappy cases [2, 17, 21, 25].

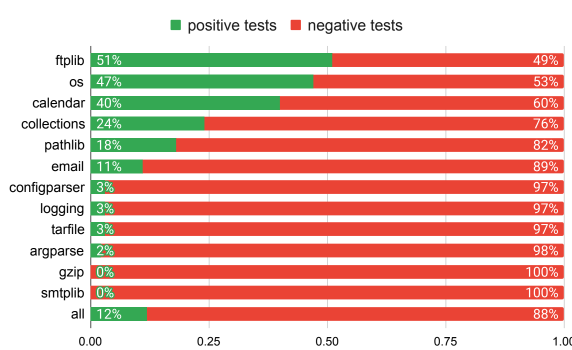

Figure 6: Test polarity of Python libraries.

Table 4 presents examples of negative and positive tests on each library. Notice that some negative tests contain words like *bad*, *illegal*, *error*, *empty*, *null*, *issue*, *debug*, and *limits*, which is an indication that they are indeed checking exceptional cases. In contrast, some positive test names are neutral or have words like *valid*, *safe*, and *basic*. Further studies need to be performed to better explore not only the negative tests but also the positive ones. For example, we can apply sentiment analysis [19] on the test names and compare them with our test polarity classification.

[7]https://github.com/python/cpython/blob/97ce15c5/Lib/test/test_collections.py#L695
[8]https://github.com/python/cpython/blob/97ce15c5/Lib/test/test_calendar.py#L743

Table 4: Examples of negative and positive tests.

Library	Negative Test	Positive Test
argparse	test_bad_type	test_open_args
calendar	test_illegal_weekday_reported	test_april
collections	test_new_builtins_issue_43102	test_field_doc
configparser	test_parsing_error	test_safeconfigparser_deprecation
email	test_decode_null_word	test_get_msg_id_valid
ftplib	test_parse257	test_login
gzip	test_bad_params	-
logging	test_with_valueerror_in_close	test_filename
os	test_bad_pathlike	test_items
pathlib	test_glob_empty_pattern	test_match
smtplib	test_debuglevel	-
tarfile	test_number_field_limits	test_basic

> **Summary**: Most of the analyzed test methods are negative (88%) and a minority is positive (12%). However, there is a large variation per project: while some libraries have an equivalent number of positive and negative tests, others have mostly negative ones. Contrary to prior findings suggesting that developers are likely to focus on positive tests, we find no test suite that is monopolized by positive tests.

5 RELATED WORK

Overall, the literature shows that developers are more likely to test the expected behaviors and avoid the unexpected ones [2, 3, 5, 6, 8, 10, 17, 21, 23, 25]. For example, in an experiment with developers, Teasley *et al.* [25] found evidence of using a positive test strategy (*i.e.,* testing the expected behavior), which was partially mitigated by increasing the expertise of the developers. Indeed, studies show that only experienced developers are more prone to test unexpected/unhappy cases [2, 17, 21, 25]. Moreover, the literature shows that "happy path testing" is considered a test smell that should be avoided [10], but there is no approach to automatically detect this smell. Our study sheds light on the polarity of real-world test suites with an automated solution to detect positive and negative tests.

6 CONCLUSION AND FURTHER STUDIES

This paper proposed and evaluated *test polarity*, an automated approach to detect positive and negative tests. We also provided a preliminary empirical study, which analyzed 2,054 test methods of 12 Python libraries. We found that most of the analyzed test methods are negative (88%) and a minority is positive (12%), however, there was a large variation per project. Our initial research opens room for novel empirical studies in software testing to better understand the polarity of real-world systems. Moreover, more research is needed to manually classify the tests with unclear polarity.

ACKNOWLEDGMENT

This research is supported by CNPq, CAPES, and FAPEMIG.

REFERENCES

[1] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide.* Simon and Schuster.

[2] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2021. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4925–4946.

[3] Gina R Bai, Justin Smith, and Kathryn T Stolee. 2021. How students unit test: Perceptions, practices, and pitfalls. In *ACM Conference on Innovation and Technology in Computer Science Education*. 248–254.

[4] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.

[5] Lex Bijlsma, Niels Doorn, Harrie Passier, Harold Pootjes, and Sylvia Stuurman. 2021. How do students test software units?. In *International Conference on Software Engineering: Software Engineering Education and Training*. IEEE, 189–198.

[6] Adnan Causevic, Rakesh Shukla, Sasikumar Punnekkat, and Daniel Sundmark. 2013. Effects of negative testing on TDD: An industrial experiment. In *International Conference on Agile Processes in Software Engineering and Extreme Programming*. Springer, 91–105.

[7] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. 2017. A systematic mapping study of software development with GitHub. *IEEE Access* 5 (2017), 7173–7192.

[8] Stephen H Edwards and Zalia Shams. 2014. Do student programmers all tend to write the same software tests?. In *Conference on Innovation & technology in Computer Science Education*. 171–176.

[9] Michael Feathers. 2004. *Working Effectively with Legacy Code.* Prentice Hall Professional.

[10] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software* 138 (2018), 52–81.

[11] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. 2015. Will they like this? evaluating code contributions with language models. In *Working Conference on Mining Software Repositories*. IEEE, 157–167.

[12] Andre Hora. 2021. What Code Is Deliberately Excluded from Test Coverage and Why?. In *International Conference on Mining Software Repositories*. 392–402.

[13] Andre Hora. 2023. Excluding code from test coverage: practices, motivations, and impact. *Empirical Software Engineering* 28, 1 (2023), 1–33.

[14] Andre Hora. 2024. SpotFlow: Tracking Method Calls and States at Runtime. In *International Conference on Software Engineering*. 1–5.

[15] Cem Kaner, Sowmya Padmanabhan, and Douglas Hoffman. 2013. *The Domain Testing Workbook.* Context Driven Press.

[16] Vladimir Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns.* Simon and Schuster.

[17] Laura Marie Leventhal, Barbee M Teasley, Diane S Rohlman, and Keith Instone. 1993. Positive test bias in software testing among professionals: A review. In *International Conference on Human-Computer Interaction*. Springer, 210–218.

[18] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship.* Pearson Education.

[19] Walaa Medhat, Ahmed Hassan, and Hoda Korashy. 2014. Sentiment analysis algorithms and applications: A survey. *Ain Shams engineering journal* 5, 4 (2014), 1093–1113.

[20] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code.* Pearson Education.

[21] Rahul Mohanani, Iflaah Salman, Burak Turhan, Pilar Rodríguez, and Paul Ralph. 2018. Cognitive biases in software engineering: a systematic mapping study. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1318–1339.

[22] Giles Reger and Klaus Havelund. 2016. What is a trace? A runtime verification perspective. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 339–355.

[23] Iflaah Salman, Burak Turhan, and Sira Vegas. 2019. A controlled experiment on time pressure and confirmation bias in functional software testing. *Empirical Software Engineering* 24, 4 (2019), 1727–1761.

[24] sys.settrace. January, 2024. https://docs.python.org/3/library/sys.htmlsys.settrace.

[25] Barbee E Teasley, Laura Marie Leventhal, Clifford R Mynatt, and Diane S Rohlman. 1994. Why software testing is sometimes ineffective: Two applied studies of positive test strategy. *Journal of Applied Psychology* 79, 1 (1994), 142.

[26] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *International Conference on Software Engineering*. 356–366.

[27] Titus Winters, Hyrum Wright, and Tom Manshreck. 2020. Software Engineering at Google: Lessons Learned from Programming over Time.

Received 28-JAN-2024; accepted 2024-04-09