PyNose: A Test Smell Detector For Python

Conference Paper · November 2021

DOI: 10.1109/ASE51524.2021.9678615

CITATIONS
13

READS
86

6 authors, including:

Yaroslav Golubev
JetBrains Research
54 PUBLICATIONS **146** CITATIONS
SEE PROFILE

Oleg Smirnov
8 PUBLICATIONS **23** CITATIONS
SEE PROFILE

Timofey Bryksin
JetBrains Research
97 PUBLICATIONS **320** CITATIONS
SEE PROFILE

PyNose: A Test Smell Detector For Python

Tongjie Wang*
University of California, Irvine
Irvine, CA, United States
tongjiew@uci.edu

Yaroslav Golubev*
JetBrains Research
Saint Petersburg, Russia
yaroslav.golubev@jetbrains.com

Oleg Smirnov
JetBrains Research
Saint Petersburg State University
Saint Petersburg, Russia
oleg.smirnov@jetbrains.com

Jiawei Li
University of California, Irvine
Irvine, CA, United States
jiawl28@uci.edu

Timofey Bryksin
JetBrains Research
Saint Petersburg State University
Saint Petersburg, Russia
timofey.bryksin@jetbrains.com

Iftekhar Ahmed
University of California, Irvine
Irvine, CA, United States
iftekha@uci.edu

***Abstract*—Similarly to production code, code smells also occur in test code, where they are called *test smells*. Test smells have a detrimental effect not only on test code but also on the production code that is being tested. To date, the majority of the research on test smells has been focusing on programming languages such as Java and Scala. However, there are no available automated tools to support the identification of test smells for Python, despite its rapid growth in popularity in recent years. In this paper, we strive to extend the research to Python, build a tool for detecting test smells in this language, and conduct an empirical analysis of test smells in Python projects.**

We started by gathering a list of test smells from existing research and selecting test smells that can be considered language-agnostic or have similar functionality in Python's standard *Unittest* framework. In total, we identified 17 diverse test smells. Additionally, we searched for Python-specific test smells by mining frequent code change patterns that can be considered as either fixing or introducing test smells. Based on these changes, we proposed our own test smell called *Suboptimal Assert*. To detect all these test smells, we developed a tool called PyNose in the form of a plugin to PyCharm, a popular Python IDE. Finally, we conducted a large-scale empirical investigation aimed at analyzing the prevalence of test smells in Python code. Our results show that 98% of the projects and 84% of the test suites in the studied dataset contain at least one test smell. Our proposed *Suboptimal Assert* smell was detected in as much as 70.6% of the projects, making it a valuable addition to the list.

***Index Terms*—Test smells, code smells, Python, empirical studies, code change patterns, mining software repositories**

I. Introduction

Code smells were introduced to identify potential maintainability issues in software systems [1], however, later they have been used as a measure of design quality of software projects [2], [3], [4]. Researchers found that code smells are associated with bugs [3], [5], fault-proneness [6], [7], and maintainability issues in the code base [1]. While investigating the underlying reasons for introducing code smells, researchers attributed various factors to this, including developers struggling with deadlines [8] or not caring about the impact of the applied design choices [1].

Similarly to production code, test code can also have code smells, in which case they are called *test smells*. Van Deursen et al. [9] defined test smells as being caused by poor design choices (similarly to regular code smells) when developing test cases.[1] Just like the code smells, test smells make the impacted test code harder to maintain and comprehend [10]. Moreover, recent studies have shown that test smells also impact the quality of production code [11].

Since test smells have a negative impact on the quality of production code, it is of great interest and importance to study and detect them. To date, the majority of the research on test smells has been focusing on statically typed languages like Java and Scala [10], [11], [12], [13], [14], [15]. However, in recent years, Python has been growing in popularity due to being the primary language used in Data Science and Machine Learning in particular [16], [17]. Furthermore, despite the empirical evidence against test smells, developers tend not to be aware of the smells that exist in their tests [13], and the lack of efficient tools can be one of the reasons for it. To the best of our knowledge, there are no works that study the existence and prevalence of test smells in Python code, and no tools exist that specifically aim at identifying test smells in this language.

In this paper, we aim to fill these gaps by curating a list of possible test smells for Python, a tool for their detection, and an empirical study of their pervasiveness in Python code. We started by conducting a small-scale mapping study to find different test smells studied in the literature and selecting test smells that can be considered language-agnostic or have analogous functionality in Python's standard *Unittest* framework. In total, we identified 17 diverse test smells. These test smells were all adopted from other papers dedicated to different programming languages, but it is natural to assume that Python has its own specific test smells. To discover them, we used a tool called PythonChangeMiner [18] to

*The first two authors contributed equally to this work.

[1]To avoid the ambiguity that exists in testing terminology between languages and frameworks, in this paper, we will always refer to individual tests or test methods as *test cases* and to classes that group them as *test suites*.

search for frequent change patterns in test suites. We manually evaluated 159 patterns that occur in at least three different projects and identified 32 possible changes that are related to *assert* functions in *Unittest* and are aimed at making the tests more specific and simplify the understanding of the testing logic. We bundled the less specific versions of these assertions together into a single *Suboptimal Assert* test smell. Thus, a total of 18 smells were identified for Python.

We developed PyNose, a plugin for PyCharm [19] that is able to detect these smells in the Python code. Using the tool, we performed an empirical study on the prevalence of test smells in 248 Python projects. Our results indicate that test smells are indeed common in Python test code, with 98% of projects and 84% of test suites having at least one test smell.

Overall, our contributions are as follows:

- We conducted a small-scale mapping study and compiled a list of test smells that are applicable to Python.
- We identified a new Python-specific test smell by analyzing Python test code changes.
- We developed a tool called PyNose as a plugin for PyCharm that can detect test smells from Python projects that use the standard *Unittest* framework. PyNose is available for researchers and practitioners on GitHub: https://github.com/JetBrains-Research/PyNose.
- We report the findings pertaining to the pervasiveness of test smells from an empirical study conducted on 248 Python projects.

The rest of the paper is organized as follows. In Section II, we discuss the existing works in the field of test smells detection and analysis. Section III describes the choice of test smells for Python and the search for Python-specific test smells. In Section IV, we describe the development of PyNose and its evaluation, and in Section V, we describe the empirical study that we conducted using the tool, as well as its results. In Section VI, we discuss threats to the validity of our study, and, finally, in Section VII, we conclude our paper and discuss possible future work.

II. Related Work

Similarly to production code, test code should be designed following proper established programming practices [20]. Van Deursen et al. [9] defined the term *test smells* as code smells that are caused by poor design choices when developing test cases and also defined a catalog of 11 test smells. Later, several researchers extended this catalog [14], [21], [22], [23]. While the majority of the research focused on test smells occurring in Java, several researchers investigated other languages and domains. For example, Bleser et al. investigated test smells in Scala [15], [24], while Peruma et al. [25] explored unit tests in mobile applications and identified several new test smells.

Researchers have also been investigating the negative impacts of test smells on software development [10], [11], [12], [13], [26]. By conducting two empirical studies, Bavota et al. [10], [12] showed that test smells are widely spread throughout software systems, and most test smells have a strong negative impact on the comprehensibility of test suites and production code. Spadini et al. [11] investigated the relationship between the presence of test smells and the change- and defect-proneness of test code, as well as the defect-proneness of the tested production code. They found that some test smells are more change-prone than others, and they also found that production code tested by smelly tests is comparatively more defect-prone. Tufano et al. [13] found that test smells are usually introduced when the corresponding test code is committed to the repository for the first time, and they tend to remain in a system for a long time. Virgínio et al. [26] investigated correlations between test coverage and test smells, and found that test smells influence code coverage.

Investigating ways for an automated detection of test smells has also received attention from the research community. Van Rompaey et al. [27] proposed a set of metrics defined in terms of unit test concepts and compared the proposed detection techniques effectiveness with human review. Greiler et al. [14] analyzed the relationship between the development of a test fixture and possible test smells within it. They also designed a static analysis tool to identify fixture-related test smells and evaluated them by discovering test smells in three industrial projects. Palomba et al. [28] developed an automated textual-based approach for detecting several types of test smells. Compared with the *code metrics*-based techniques proposed by Greiler et al. and Van Rompaey et al., the textual-based technique proved to be more effective in detecting certain test smells. Peruma et al. [25], [29] recently developed a tool called tsDetect capable of detecting 19 test smells in Java.

More recently, researchers have been investigating ways to help testers refactor test smells. Lambiase et al. [30] presented an IntelliJ-based plugin that enables an automated identification and refactoring of test smells using IntelliJ Platform's APIs. Santana et al. [31] proposed another tool that can be used in an IDE, providing testers with an environment for automated detection of lines of code affected by test smells, as well as a semi-automated refactoring for Java projects. Virgínio et al. [32] presented a tool designed to analyze test suite quality in terms of test smells. Their tool is the first one that relies on both code coverage and the presence of test smells to measure the quality of tests.

Overall, the majority of the mentioned research has been focusing on Java. However, in recent years, Python has been growing more popular because of its important role in Data Science and Machine Learning in particular [16], [17]. To the best of our knowledge, no tools exist that specifically aim at identifying Python test smells. PyNose addresses this gap. Furthermore, there is no large-scale analysis regarding the prevalence of test smells in Python code, and our study is the first towards filling this gap in research.

III. Selecting Test Smells

The goal of our study is to build a tool that can identify test smells in Python code as well as to assess to what extent test smells are prevalent in Python test suites. The general pipeline of our study is demonstrated in Figure 1. In Section III, we curate the list of appropriate test smells by

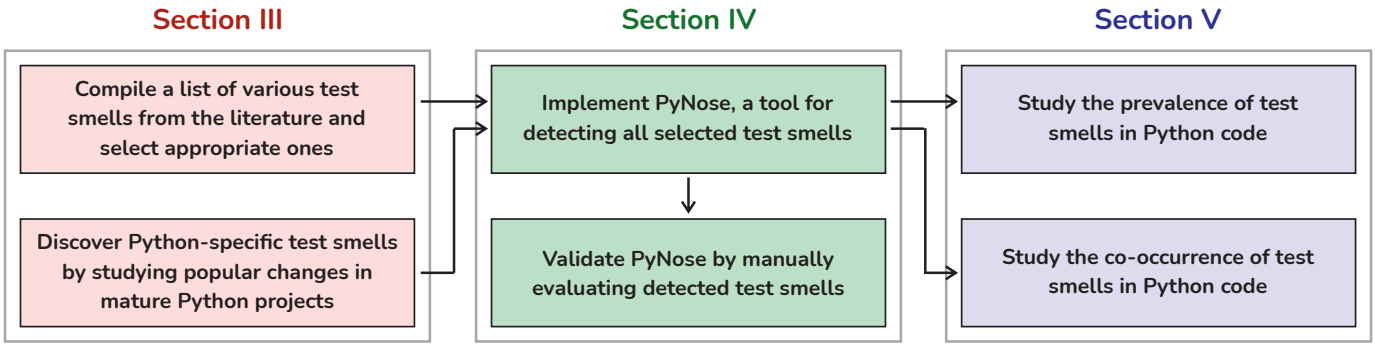

Fig. 1. The overall pipeline of the study.

conducting a systematic mapping study (Section III-A) and then augmenting the list by identifying Python-specific test smells (Section III-B).

A. Systematic mapping study of test smells

As a first step, we conducted a small-scale systematic mapping study on test smells to curate a list of test smells discussed in the literature. According to Kitchenham et al. [33], the goal of the mapping study is to survey the available knowledge about a topic.

Search Question. Our search question was phrased as follows: *What test smells have been studied in literature to date?*

Search Keywords. To determine the optimal set of search keywords, we conducted a pilot search on two well-known digital libraries, IEEE and ACM. This process was intended to identify relevant words utilized in test smell publications. We conducted our query only on the title and abstract of the publication to avoid false positives. The finalized search string is presented below.

> ***Title**: ("test smell" OR "test smells") AND **Abstract**: ("test smell" OR "test smells").*

Data Source. To discover relevant publications, we used three of the most popular online paper search engines: ACM Digital Library, IEEE Xplore, and Scopus.

Search Period. To obtain as many related works as possible, we queried all related studies before 2020. This resulted in a list of papers that were published between 2006 to 2020.

Initial Results. Our initial search of the three digital libraries resulted in 54 publications. To narrow down the search results, next, we filtered out publications that were not part of our inclusion criteria. A summary of the inclusion and exclusion criteria used to filter the retrieved literature is shown in Table I. The filtering process helped us to reduce the number of studies significantly, however, this may have resulted in leaving out some relevant studies. Thus, we conducted backward snowballing [34] (*i.e.,* looking for additional studies in the reference lists of the selected studies, as suggested by Keele et al. [35]). In our work, we implemented a single iteration of backward snowballing.

To ensure the reliability of the selected studies, each study was evaluated by three authors of this paper. Each selected study underwent an agreement process, and in case of uncertainty and disagreement, we discussed it until we reached consensus. We finally ended up with with a set of 29 studies. Next, we merged the lists of test smells mentioned in these papers, which resulted in a list of 33 different test smells encountered in Java, Scala, and Android systems. The full list of papers and test smells is available online in the supplementary materials [36].

TABLE I
INCLUSION AND EXCLUSION CRITERIA.

Inclusion Criteria
1. Publications that implement software engineering methodologies, approaches, and practices in test smell detection and refactoring.
2. Available in digital format.
Exclusion Criteria
1. Publications that are not written in English.
2. Websites, leaflets, and grey literature.
3. Published in 2021.
4. Full-text not available online.
5. Tools not associated with peer-reviewed papers.
6. Duplicated publications.

Next, we considered the possibility of implementing each test smell for Python. There were several reasons why some of the test smells could not be implemented:

The test smell is not applicable to Python. For example, the *Resource Optimism* [9] test smell in Java occurs if a `File` object is used without checking for its existence. However, in Python, files always associate with resources, because, according to the Python official documentation, "`open()` is the standard way to open files for reading and writing with Python" [37].

The test smell detection relies on the production code that is being tested. For example, to identify *Eager Test* [9] and *Lazy Test* [9], we need to know what the corresponding production files and production classes are. A lot of recent works study test-to-code traceability [38], [39], [40] and a lot of different approaches have been suggested. However, reliably making a strict one-to-one connection between a test method and a production method in the static analysis environment is difficult [39], which is why leave the support of such test smells for future work.

The test smell detection is possible only when the test is executed. For example, for the *Test Run War* [9], it is necessary to actually run the test case, which is not possible in a static analysis environment. Even after running, identifying such test smells is non-trivial, and for practical purposes we had to exclude them.

Finally, we selected 17 test smells for implementing. We list them below.

Assertion Roulette occurs when a test case has multiple non-documented assertions. Multiple assertion statements without a descriptive message impact the readability, understandability, and maintainability, as it becomes more difficult to understand the reason why this test fails [9].

Conditional Test Logic runs against the rule that test cases need to be simple and execute all statements in the production code. Conditions within the test case alter the behavior of the test and lead to situations where the test fails to detect defects in the production code under some conditions [25].

Constructor Initialization is made by developers who are unaware of the purpose of the `setUp()` method that contains the preparation needed to perform test cases. As a result, they would define a constructor for the test suite, which is not ideal in practice [25].

Default Test occurs when an IDE creates default test suites when the project is created and developers keep the default name. For example, PyCharm by default names the test suites `MyTestCase`. These suites are meant to serve as an example for developers when writing unit tests and should be renamed. Not renaming them upfront causes developers to start adding test cases into these files, making the default test suite a container of all test cases. This can also cause problems when the suites need to be renamed in the future [25].

Duplicate Assert occurs when a test case tests for the same condition multiple times [25].

Empty Test occurs when a test case does not contain executable statements. Such tests are possibly created for debugging purposes and then forgotten about or contain commented out code [25].

Exception Handling occurs when passing or failing of a test case is dependent on the production method explicitly throwing an exception. Instead, developers should utilize special functionality of testing frameworks for that, such as an `assertRaises()` function [25].

General Fixture occurs when a test suite fixture is too general and some test cases only access a part of it. The fixture of a test suite is a special method that is executed before the test cases in the suite and serves as a setup step. A drawback of it being too general is that unnecessary work is being done when a test suite is run [9].

Ignored Test is caused by ignored test cases when it is possible to suppress some test cases from running. These ignored test cases add unnecessary overhead by increasing code complexity and making comprehension more difficult [25].

Lack of Cohesion of Test Cases occurs if test cases are grouped together in one test suite but are not cohesive. Cohesion of a class is a metric that indicates how well various parts and responsibilities of a class are tied together. If test cases in a suite are not cohesive, this can cause comprehensibility and maintainability issues [14].

Magic Number Test occurs when assert statements in a test case contain numeric literals (*i.e.*, magic numbers) as parameters instead of more descriptive constants or variables [25].

Obscure In-Line Setup occurs when the test case contains too many setup steps. This can hinder inferring the actual purpose of the assertion in the test. Ideally, such preparation should be moved to a fixture or a separate method [14].

Redundant Assertion occurs when a test case contains assertion statements that are either always true or always false, and are therefore unnecessary [25].

Redundant Print occurs when there is a print statement within the test. Print statements are considered to be redundant in unit tests as unit tests are usually executed as a part of an automated process with little to no human intervention [25].

Sleepy Test occurs when developers need to pause the execution of statements in a test case for a certain duration (*i.e.*, simulate an external event) and then continue with the execution. Explicitly causing a thread to sleep can lead to unexpected results as the processing time for a task can vary on different devices [25].

Test Maverick was derived from the *General Fixture* described above. If the test suite has a fixture with setup, but a test case in this suite does not use this setup, this test case is a maverick (outlier). The setup procedure will be executed before the test case is executed, but it is not needed [14].

Unknown Test occurs when the test case has no assertion in it. It is possible to create a test case that does not use assertions, however, such a test is more difficult to understand and interpret [25].

During this selection, we also decided to focus specifically on the *Unittest* testing framework [41] that is included into the Python Standard Library. Python also has a lot of popular third-party testing frameworks like *PyTest* [42] and *Robot* [43], however, certain test smells would look differently in different frameworks, and it is out of the scope of this paper to support them all. There are two reasons for choosing specifically *Unittest*. Firstly, it remains one of the most popular testing frameworks in Python while also being the default one [44]. Secondly, according to its documentation, *Unittest was originally inspired by JUnit* [41], which allows us to detect some test smells from the literature that were originally proposed for *JUnit*, for example, fixture-related test smells. Additionally, several other frameworks support launching test suites from *Unittest*, and can therefore also be detected in this case.

B. Identifying Python-specific test smells

In addition to the test smells identified above, our goal was to include Python-specific test smells. To discover Python-specific test smells, we used a tool called PYTHONCHANGEMINER [18] to search for frequent change patterns in the histories of test suites. We explain the steps of this process in detail in this section.

1) Project selection: To carry out this research, we needed to collect a dataset of mature open-source Python projects. As a starting point, we took GHTorrent [45], a large collection of GitHub data, more specifically, their latest dump at the time of the compilation, compiled in July 2020 [46]. To process it, we used a tool called PGA-create [47] that had been previously used to create Public Git Archive (PGA) [48]. This tool processes the SQL dump to create a CSV file with a list of projects that facilitates their convenient filtering. Next, we selected all projects with at least 50 stars, which allowed us to filter out toy projects. We also only considered projects with Python as the main language that are not forks. This resulted in identifying 26,072 projects. Of them, we randomly selected 10,000. The reason for not simply picking top projects by stars is that testing might be organized very differently in projects of different scale, and simply picking the largest or the most popular repositories could skew our data towards a specific type of projects.

Next, we analyzed the history of the projects to find all commits where at least one Python test file was changed. We defined a Python test file as any file with the `.py` extension that has the word `test` in its filename, since Unittest has a naming convention of having the word `test` in the name of the test file [41]. We have conducted a small manual analysis by selecting 100 random Python files with the word `test` in their name and checking whether they are actually related to tests. In this random sample, all 100 files were related to testing, with 96 explicitly containing test suites and test cases, and another 4 containing auxiliary methods and testing utils.

4,580 of the projects had at least one commit that *changed* such files. As we were looking for code changes, we selected these 4,580 projects. Since our goal was to analyze the changes themselves, for practical purposes, we decided to select a smaller set of projects using the criteria recommended in literature [49]. We selected projects with at least 1,000 commits, 10 contributors, 2 years since the first commit and no more than 1 year since the last push. This resulted in 450 projects. For the purposes of this paper, we will call this the *Primary* dataset; the list is available online [36].

2) Change pattern mining: To identify Python-specific test smells, we started by mining the histories of the collected projects and finding patterns in the changes made to test files that might be considered as either fixing or introducing a test smell. We extracted all changes made to Python test files from the identified 450 projects and processed these files using PYTHONCHANGEMINER [18].

PYTHONCHANGEMINER is a tool that we developed for mining code change patterns in Python code. The tool is based on the algorithm developed by Nguyen et al. [50] for Java. The parser in their tool is written specifically for the syntax of the Java language, and their tool stores graphs and works with them as Java objects, so we could not directly reuse the tool. At the same time, the algorithm itself is not language-specific, because it relies only on the abstract syntax trees (AST) of code before and after the change, which is why we implemented it for Python. The operation process of PYTHONCHANGEMINER is similar to that of the tool by Nguyen et al. Here, we briefly explain the procedure.

PYTHONCHANGEMINER works in two stages: building change graphs and mining patterns. In the first stage, the versions of code before and after the change are parsed into a special representation introduced by Nguyen et al. called *fine-grained Program Dependence Graphs* (fgPDGs). fgPDGs are graphs with three types of nodes: *data* nodes (variables, literals, constants, etc.), *operation* nodes (arithmetic, bit-wise operations, etc.), and *control* nodes (control sequences like `if`, `while`, `for`, etc.). These nodes are connected using two types of edges: *control* edges represent a connection between a *control* node and a node that it controls and *data* edges show the flow of the data in the program, such edges also have labels specifying the flow of data.

Then, unchanged nodes in the two fgPDGs of code before and after the change are connected together by special *map* edges, resulting in new graphs called *change graphs*. We used GumTree [51] to detect corresponding unchanged nodes in the versions before and after the change and connect them with a *map* edge. This is carried out on a function level and therefore, this way, we obtain a special *change graph* that represents each change to each testing function from the history of projects in our dataset. You can find an example of fgPDGs and a change graph in the supplementary materials [36].

The second stage of PYTHONCHANGEMINER involves searching these change graphs for patterns. This part is also done similarly to the work of Nguyen et al. [50]. First, all pairs of nodes representing function calls that are also connected with the *map* edge are considered to be the initial patterns that are then recursively expanded to contain new nodes. The pattern is defined by two thresholds: *minimum size*, indicating the minimum number of graph nodes in the pattern, and *minimum frequency*, indicating the minimum number of repetitions of the pattern in the corpus. Changing these parameters influences what is considered to be a pattern and, therefore, how many patterns are detected. This way, the patterns are expanding to detect isomorphic subgraphs within our corpus of graphs.

In our work, we use the same thresholds as Nguyen et al.: *minimum size* of 3 and *minimum frequency* of 3. It is possible that studying specifically the testing code requires different thresholds, we leave such analysis for future work. We additionally add a *maximum size* threshold of 20. This is done to make the process faster by stopping the patterns from growing too large. Our own preliminary experiments and our analysis of the results of Nguyen et al. demonstrated that the majority of discovered patterns are small. More specifically, the *Depth* pattern corpus provided by Nguyen et al. [50] contains a total of 9,289 patterns, of which 8,697 (93.6%) patterns are 20 nodes or smaller. Since smaller patterns are much more frequent and are easier to analyze, we decided to focus on them. An example of a discovered pattern is presented in Figure 2.

3) Test smells detection: In total, PYTHONCHANGEMINER was able to discover 8,239 different patterns in the *Primary* dataset. Of them, 652 patterns were *cross-project*, meaning

```
self.assertTrue('GR...' in results)
self.assertIn('GR...', results)
```

(a) Commit in the Obspy project [52].

```
self.assertTrue("foo" in llvm_bc)
self.assertIn("foo", llvm_bc)
```

(b) Commit in the Numba project [53].

```
self.assertTrue('password' in account.data)
self.assertIn('password', account.data)
```

(c) Commit in the Reviewboard project [54].

Fig. 2. An example of a change pattern identified in several projects on GitHub.

they were encountered in at least two different projects, and 159 appeared in at least three different projects. Three authors of the paper independently manually labeled all 159 of such changes to discover changes that either fix or introduce possible test smells. The reason for focusing on these changes is that they are inherently more universal among different developers. Along with analyzing the code changes themselves, the authors also looked at the corresponding commit messages, since commit messages may contain the rationale for a change. After individual labeling, the authors discussed their labels and reached a perfect agreement.

Of the studied 159 patterns, 70 (44%) constituted various changes to assertion functionality, similar to the example shown in Figure 2. Three authors of the paper independently came to a conclusion that the candidates for possible Python-specific test smells can be found only within this group, because other common changes in testing code correlate to various other aspects of software engineering: data structures, data processing, etc., that are not directly related to testing itself. For example, popular changes include changing the level of the logger (error, info, debug, etc.) or changing the shape of a `numpy` array. Such patterns are important, but are not directly related to testing or test smells.

We categorized assert-related change patterns into three categories, which we describe below with specific examples.

i. **Assertion changes that alter the logic.** Often, when developers change an assertion in a test case, they do it to update the logic behind the test. For example, a pattern that occurred in six different projects is changing from `assertEqual` to `assertRegex`. This way, instead of checking for an exact equality between an object and a string, a regular expression is passed that can support variations in strings. One commit message reads: *Use a more permissive comparison for jsonschema.ValidationError messages* [55].

Another common pattern involved changing from `assertEqual` to `assertIn`, where instead of one correct result, there is a list of values. Conversely, another common example is changing from `assertIsNone` to another function `assertIsInstance`. This makes the check more specific: the object is not compared to `None` but rather needs to be an object of a new specific class. One commit message conveys a similar idea: *NullSort instead of None. A more descriptive placeholder for "don't sort"* [56].

ii. **Assertion changes that do not alter the logic and use *more* appropriate functions.**

A large portion of the patterns involved keeping the assertion logic the same, but replacing the assertion function with a more appropriate one to make the code succinct. In total, eight such patterns were identified. These changes are Python-specific in the sense that they rely heavily on a wide range of assertion functions that *Unittest* supports.

The most popular pattern is shown in Figure 2. It occurs in seven different projects and moves from using `assertTrue(X in Y)` to `assertIn(X, Y)`. One commit message describes this change in great detail: *Use more specific assertions for 'in' checks. A lot of old code used 'assertTrue(blah in blah)', or variants on that, which didn't tell you much if there was a failure. Nowadays, we have assertIn and assertNotIn, which we can use instead. This switches our tests to use these* [54]. This commit message indicates that the original code (before the change) can be considered a test smell since using general assertions can make it difficult to infer the reason of failure by "hiding" the actual assertion in its body, whereas using specific assertions can make it easier.

Another change that strives to remove the ambiguity of a general assertion occurs in four repositories, and it moves from using `assertFalse(X == Y)` to `assertNotEqual(X, Y)`. Sometimes `assertTrue` is changed to another specific assertion. For example, in three different projects `assertTrue(X <= Y)` is changed to *Unittest*'s `assertLessEqual(X, Y)`. One commit message expectedly comments this: *Use more specific asserts in unit tests* [57].

In Python, it is considered bad practice to check the equality of a boolean value when you can check the value itself, so in this case a boolean assertion is more correct and more interpretable, which is reflected in a common change pattern where `assertEqual(X, False)` is changed to `assertFalse(X)`.

In this section, we have given examples of some commit messages that describe the changes along with the change pattern. We believe that these commit messages justify considering the wrong choice of an assertion function in *Unittest* as a test smell. We called this smell *Suboptimal Assert*.

iii. **Assertion changes that do not alter the logic and use *less* appropriate functions.**

Interestingly, we also discovered seven change patterns that move from an appropriate assertion function to a more general one. Following the logic of the previous section, these can be treated as introducing a test smell.

The most popular such change is moving from a more specific `assertIsNotNone(X)` to a more general `assertNotEqual(X, None)`. One commit message describes this change as a *Fix in test for Python 2.6 compatibility* [58]. However, the changes in this pattern were made in 2014–2015, and since Python 2 is deprecated from 2020, this is no longer a problem.

A similar message described commits in two different projects that moved from `assertNotIn(X, Y)`

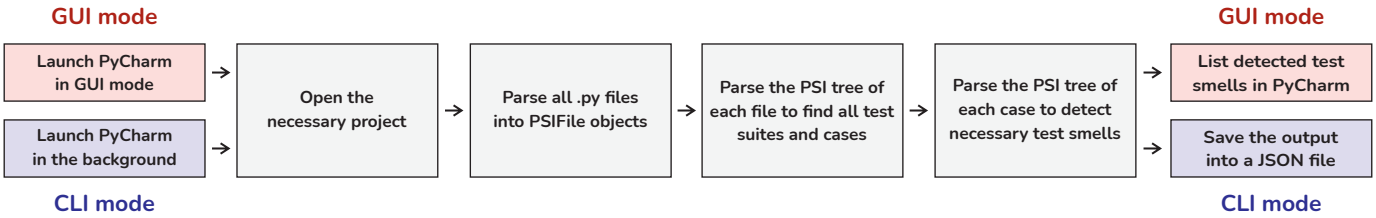

Fig. 3. The pipeline of PyNose operation in two regimes: GUI mode and CLI mode.

to `assertTrue(X not in Y)` and two more different projects that moved from `assertLess(X, Y)` to `assertTrue(X < Y)`. The same changes can be found for functions like `assertGreater(X, Y)` and `assertIsNone(X)`, with one commit message saying: *remove fancy test assertions that are unavailable on 2.6* [59].

In total, we encountered 12 different suboptimal asserts (either fixed, introduced, or both). We extrapolated them to similar functions and opposite cases where necessary: for example, if there is a suboptimal assert that contains `assertLess`, it can also be formulated for `assertGreater`, etc. This resulted in a total of 32 different assertions that can be considered a part of the *Suboptimal Assert* test smell, the full list is available online [36].

IV. PyNose

Once we curated the list of test smells (explained in Section III), our next goal was to implement a tool to identify them in actual Python code. We developed a tool called PyNose that currently identifies 18 test smells (17 language-agnostic from the existing literature and one Python-specific elicited by us as described in Section III-B), and can be run from both the graphical user interface and the command line. Figure 3 shows the operating pipeline of PyNose. In this section, we explain it in greater details.

A. Tool internals

PyNose is implemented as a plugin for PyCharm [19], a popular IDE for Python developed by JetBrains. The plugin supports two different modes of operation: *Graphical User Interface (GUI) mode* and *Command Line Interface (CLI) mode*. Internally, PyNose uses Program Structure Interface (PSI) [60] from JetBrains' IntelliJ Platform (that PyCharm is built upon) to parse Python source code and build syntactic and semantic code models for analysis. When the project is opened and the interpreter is set up, the tool uses PSI and other related PyCharm API to gather all `.py` files in the project in the form of `PSIFile` objects.

Next, the tool extracts all Python classes that are sub-classes of `unittest.TestCase`. With the help of PSI, PyNose can deal with importing `unittest` or `unittest.TestCase` under alias or test cases that are not direct sub-classes of `unittest.TestCase`. After collecting individual test suites, each detector class (corresponding to each test smell) invokes `PsiElementVisitor` to create a custom visitor for the necessary `PsiElement`, which allows PyNose to identify test smells. For example, for the *Magic Number Test*, we use a custom visitor of `PyCallExpression` to find all assertions, and then check if one of the provided arguments is a `PyNumericLiteralExpression`. If there is a match, the *Magic Number Test* smell is declared to be found.

For the test smells from the literature, we implemented their detection in the same way as they are described in the original papers, using the mentioned thresholds. For example, we detect *Obscure In-Line Setup* the same way as Greiler et al. [14], by counting the number of local variables in a test case and flagging the case as smelly if this number is larger than a threshold of 10, and detect *Lack of Cohesion of Test Cases* the same way as Palomba et al. [28], by calculating pairwise cosine similarities between test cases. Detection rules for all the supported test smells are presented in Table II, the citations mark the works, from where the detection rules were adapted from. Where necessary, we used code entities analogous to their counterparts in Java, for example, `@unittest.skip()` decorator in the place of the `@Ignore` annotation. If there were several different heuristics to detect the same smell in different papers, we selected one based on its recency and its convenience to implement using the PSI and the IntelliJ platfrom.

When the analysis is done, PyNose can show the detected test smells inside the IDE or save them to a JSON file for further analysis.

B. Evaluation

We conducted an experimental evaluation of the effectiveness of PyNose in correctly detecting test smells. As there are no existing datasets containing information for all the supported smells, we decided to construct our own validation set. We randomly selected eight projects that did not make it into the *Primary* dataset. We then used the definitions of test smells to identify and tag test files with the information regarding the types of smells they exhibit. This process resulted in a total of 37 annotated files. The list of projects, together with some statistics about their testing files, is shown in Table III. To ensure an unbiased annotation process, three authors of the paper individually did the labelling and discussed their results afterwards to reach a consensus. All the three authors have experience with Python development ranging from two to five years, which includes exposure to developing unit tests.

Next, we ran PyNose on the same set of projects and compared our results against the oracle. We calculated precision, recall, and F1 score for each test smell. We also calculated

TABLE II
THE DETECTION RULES FOR SUPPORTED TEST SMELLS. CITATIONS INDICATE WORKS WHERE THE RULES WERE ADOPTED FROM.

Assertion Roulette	A test case contains more than one assertion statement without an explanation/message. [29]
Conditional Test Logic	A test case contains one or more control statements (*i.e.*, `if`, `for`, `while`). [29]
Constructor Initialization	A test suite contains a constructor declaration (an `__init__` method). [29]
Default Test	A test suite is called `MyTestCase`. [29]
Duplicate Assert	A test case contains more than one assertion statement with the same parameters. [29]
Empty Test	A test case does not contain a single executable statement. [29]
Exception Handling	A test case contains either the `try/except` statement or the `raise` statement. [29]
General Fixture	Not all fields instantiated within the `setUp()` method of a test suite are utilized by all test cases in this test suite. [29]
Ignored Test	A test case contains the `@unittest.skip` decorator. [29]
Lack of Cohesion of Test Cases	The mean of the pairwise cosine similarities between test cases in a test suite ≤ 0.4. [28]
Magic Number Test	A test case contains an assertion statement that contains a numeric literal as an argument. [29]
Obscure In-Line Setup	A test case contains ten or more local variables declarations. [14]
Redundant Assertion	A test case contains an assertion statement in which (1) the expected and actual parameters of equality are the same, *e.g.*, `assertEqual(X, X)` or (2) the assertion of truth is carried out on the unchangeable object, *e.g.*, `assertTrue(True)`. [29]
Redundant Print	A test case invokes the `print()` function. [29]
Sleepy Test	A test case invokes the `time.sleep()` function with no comment. [29]
Suboptimal Assert	A test case contains at least one of the suboptimal asserts.
Test Maverick	A test suite contains at least one test case that does not use a single field from the `SetUp()` method. [14]
Unknown Test	A test case does not contain a single assertion statement. [29]

TABLE III
EIGHT PROJECTS SELECTED FOR THE EVALUATION OF PYNOSE. THE COLUMNS INDICATE THE NUMBER OF TESTING FILES, SUITES, AND CASES WITH *Unittest*.

Project	T. Files	T. Suites	T. Cases
ali1234/vhs-teletext	12	23	56
cea/sec_ivre	1	1	13
davidhalter/jedi	3	4	17
demisto/content	9	10	203
justiniso/polling	1	1	4
Lagg/steamodd	6	13	45
plamere/spotipy	3	17	114
pygridtools/drmaa-python	2	4	16
Total	37	73	468

the weighted average of these three metrics for all test smells with the weights being the number of instances of each test smell in the projects. The results of the conducted evaluation are presented in Table IV.

Several test smells were encountered very rarely in the validation projects, with three of them having only a single example. This has to do with the fact that these test smells are just rare in Python in general (see Section V-C2). However, these test smells have very robust definitions that are easy to detect: *Default Test* requires the tool to simply check the name of the test suite, *Constructor Initialization* requires the tool to simply check the presence of an `__init__` method, and *Sleepy Test* simply looks for the `sleep()` function in the body of the test case.

As shown in Table IV, PYNOSE achieves a high level of correctness with F1 scores ranging from 81.5% to 100% for different test smells. For the cases where the tool did not achieve 100%, we investigated the mismatch.

In one instance, *Assertion Roulette* was not detected because of a non-conventional name of the test case, where the name started with a _ symbol instead of the word `test*` as is the convention. A human rater could tag such a test case as

TABLE IV
THE RESULTS OF THE EVALUATION. **INST.** STANDS FOR INSTANCES AND INDICATES A TRUE NUMBER OF TEST SUITES WITH A GIVEN SMELL IN THE VALIDATION DATASET.

Test Smell	Inst.	Precision	Recall	F1
Assertion Roulette	42	100%	97.6%	98.8%
Conditional Test Logic	20	80%	100%	88.9%
Constructor Initialization	1	100%	100%	100%
Default Test	2	100%	100%	100%
Duplicate Assertion	6	100%	100%	100%
Empty Test	1	100%	100%	100%
Exception Handling	10	100%	100%	100%
General Fixture	11	100%	100%	100%
Ignored Test	3	100%	100%	100%
Lack of Cohesion	13	78.6%	84.6%	81.5%
Magic Number Test	23	100%	82.6%	90.5%
Obscure Inline Setup	3	100%	100%	100%
Redundant Assertion	2	100%	100%	100%
Redundant Print	2	100%	100%	100%
Sleepy Test	1	100%	100%	100%
Suboptimal Assert	10	100%	100%	100%
Test Maverick	5	100%	100%	100%
Unknown Test	10	83.3%	100%	90.1%
Weighted average	—	**94.0%**	**95.8%**	**94.9%**

having the *Assertion Roulette* test smell, however, PYNOSE failed to do so. PYNOSE also incorrectly identified several *Conditional Test Logic* test smells. *Conditional Test Logic* is detected by the presence of control statements (*i.e.*, `if`, `for`, etc.) irrespective of their impact on the assertion. For example, the `for` statement can be used simply to assign a variable and such cases are incorrectly tagged as *Conditional Test Logic* by PYNOSE. *Lack of Cohesion* relies on the cohesiveness of test cases in a test suite. PYNOSE measures cohesiveness using cosine similarity, whereas human raters used their subjective judgement, which resulted in a mismatch between the output of PYNOSE and the opinion of the human raters in several cases. Several *Magic Number Tests* were not detected because the comparison to a literal occurred in

assertions with complex parameters that are not yet supported. For example, `assertEqual(df.shape, (1, ))` was tagged as a *Magic Number Test* test smell by a human rater, however, PYNOSE failed to do so, because the literal is located in a tuple. Finally, two cases of *Unknown Test* turned out to be false positives. The tool considered the test case to not have assertions, when in reality an assertion was present, but it was from the unsupported `pytest` framework.

For all test smells together, PYNOSE achieves the precision of 94% and the recall of 95.8%. Table V shows the comparison between the obtained values and the reported numbers of TSDETECT [29], a similar tool for Java. It can be seen that the values are similar, however, we plan to conduct a more thorough and direct comparison of tools in the future.

TABLE V
THE COMPARISON OF PERFORMANCE BETWEEN PYNOSE AND TSDETECT.

Detector	Language	Precision	Recall	F1
TSDETECT [29]	Java	96.0%	97.1%	96.5%
PYNOSE	Python	94.0%	95.8%	94.9%

V. PREVALENCE OF TEST SMELLS

After developing and validating PYNOSE, we conducted an empirical study on test smell prevalence in open-source Python projects. In this section, we present the details and the results of this study.

A. Selecting projects to analyze

The goal of our study was to analyze test smell prevalence in Python projects using PYNOSE. This was done to increase the subject diversity among the existing empirical studies on test smells, as well as to gain an understanding of how test smells are diffused in Python code. We decided to study the presence of test smells in the same *Primary* dataset that was used for mining code change patterns. We decided to do so because the *Primary* dataset represents mature open-source Python projects that use testing within them.

However, to make sure that the results of the study are robust and do not depend on the results from Section III-B3, we decided to also run the tool on an additional dataset. To gather it, we used the same procedure as described in Section III-B1, but with one condition being slightly relaxed: we gathered projects with the number of commits between 500 and 1,000, instead of at least 1,000 commits. This resulted in 239 additional projects; the full list is available online [36]. We will refer to this dataset as the *Secondary* dataset. While we draw our general conclusions from the *Primary* dataset, since it contains more projects with larger histories, the purpose of the *Secondary* dataset is to make sure that the reported results are unbiased.

B. Methodology

We ran PYNOSE on all the projects in the *Primary* and *Secondary* datasets separately. We dropped the results where not a single test suite was found, and only considered test suites with at least one test case and test files with at least one test suite. Test smells can occur on various levels of granularity: *Constructor Initialization*, *Default Test*, *General Fixture*, and *Lack of Cohesion* manifest at the level of a test suite as a whole, while other test smells such as *Conditional Test Logic* are formulated at the test case level.

We analyzed the test smells using their appropriate granularity. A test suite is considered smelly if it contains at least one test case with a given smell. A test file can also be considered a valid object for comparison, however, even though in Python and in *Unittest* it is possible to have several test suites in one test file, this granularity is still largely similar to a test suite, and often a test file contains just one or two test suites. We also calculated the distribution of test smells among projects to get a more coarse-grained picture of the test smells prevalence.

We studied the most common and the least common test smells, as well as the prevalence of the newly proposed *Suboptimal Assert*. Additionally, we studied the co-occurrence of different test smells in individual test suites and discussed the correlations between test smells.

C. Results

In this section, we discuss the results of the empirical study of the test smells prevalence in Python code.

1) General information: In total, at least one *Unittest* test case was found in 248 projects out of the 450 in the *Primary* dataset (55.1%). From here on out, all percentages are calculated based on these 248 projects. In total, in these 248 projects, PYNOSE detected 9,158 test files, 16,681 test suites, and 96,736 test cases. More detailed statistics are presented in Table VI. It can be seen from the table that even mature projects vary greatly by the amount of testing within them. In our dataset, one test file on average had 1.8 test suites, and one test suite on average had 5.8 test cases.

TABLE VI
THE SUMMARY OF THE AMOUNT OF TESTING ENTITIES PER PROJECT.

	Test files	Test suites	Test cases
Minimum	1	1	1
Mean	36.9	67.3	390.1
Maximum	323	870	5,121

2) Test smells distribution: The distribution of 18 detected test smells is presented in Figure 4. In general, it can be seen that the studied test smells are prevalent in Python code. There are only 5 projects (2%) that have no smells, however, all of them are very small projects, with the largest having only 13 test cases. All the other projects (98%) have tests smells in one way or another. Test smells such as *Assertion Roulette* and *Conditional Test Logic* are among the most common test smells and occur in almost 90% of projects that use *Unittest* in the *Primary* dataset. Also among the most popular test smells are *Magic Number Test*, *General Fixture*, and *Unknown Test*.

On the other end of the spectrum, we can see test smells that rarely occur in Python code. *Empty Test* occurs in just 0.7% of the test suites, although, interestingly, even these instances

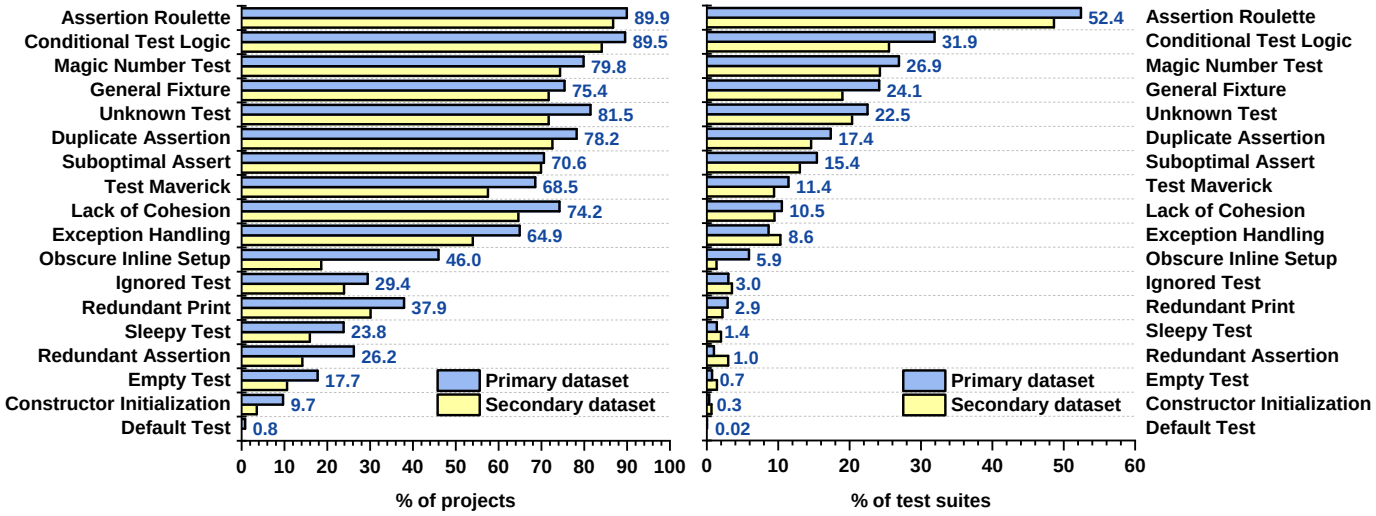

Fig. 4. The prevalence of different test smells among all projects and test suites in *Primary* and *Secondary* datasets. The percentages relate to projects that use *Unittest*, the numbers near bars are shown for the *Primary* dataset.

are spread out among as much as 17.7% of the projects. *Constructor Initialization* occurs in 9.7% of the projects and 0.3% of the test suites. Finally, the rarest of all test smells that occurs in only two projects and only three test suites within them, is *Default Test*. Figure 4 also shows that our introduced *Suboptimal Assert* smell constitutes an important addition to Python test smells: it occurs at least once in 70.6% of the projects and 15.4% of the test suites.

In comparison with previous works that study Java and Android code [25], it can be said that the lists of the most popular test smells generally look similar. It seems that Python code has larger percentages by projects, however, direct comparison here should be carried out in future work. One specific test smell that seems to be less prevalent in Python is *Exception Handling* that occurs in 64.9% of the projects and only in 8.6% of the test suites. *Unittest* supports a convenient list of assertions like `assertRaises`, `assertRaisesRegex` and others that may prevent the users from using `try/except` keywords in tests.

It can also be seen that the results for the *Primary* dataset and the *Secondary* dataset are similar to each other, with the only noticeable exception being the *Obscure In-Line Setup*, which is rarer in the *Secondary* dataset. The values for *Suboptimal Assert* are also similar. This demonstrates that our results obtained for the *Primary* dataset are unbiased.

Overall, our results show that various test smells are prevalent in Python code, even some of the rarer ones still occur in more than a quarter of all projects. While some of them can be considered more subjective, others make it significantly harder to maintain the code base and to interpret the results of testing in case of failure. We hope that in the future PyNose can be used to help developers and researchers to combat the spread of test smells in their repositories.

3) Co-occurrence of test smells: In the previous section, we discussed how prevalent different test smells are. However, such an approach considers test smells independently from each other and does not fully describe the actual "smelliness" of code. To get a better understanding, we also analyzed the co-occurrence of test smells.

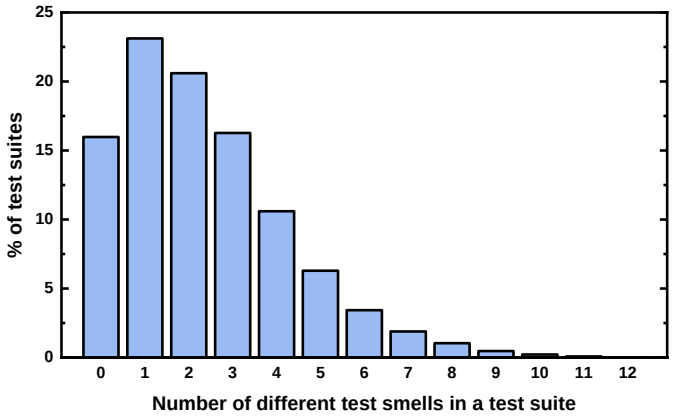

Fig. 5. The distribution of the number of different test smells among individual test suites.

Figure 5 shows the distribution of how many different smells co-exist within individual test suites. It can be seen that only 16% of all test suites are free from smells. The remaining 84% of the test suites have at least one smell: 23.1% have exactly one smell, 20.6% have two smells, 16.3% have three smells, and this number gradually decreases with the amount of co-occurring test smells. The highest occurring number in the *Primary* dataset appears in a single test suite with 12 distinct test smells. This large test suite with 25 test cases, in addition to all the most popular test smells, contains commented out empty test cases, catching errors with `try/except` instead of using specific assertions of error messages, and sleepy tests. It also uses `assertEqual(X, True)` instead of `assertTrue(X)`. We believe that helping developers find such suites might be useful for the maintenance of the project.

Figure 5 also sheds a new light on the prevalence of test smells in Python code. With more than half of all test suites

having two different test smells or more, their effect on the maintainability of code can become more complex.

We also additionally studied the co-occurrence of specific pairs of test smells. For all pairs of test smells, we calculated the following value: what percentage of test suites that have test smell X also have test smell Y. Two pairs of test smells are completely connected. Firstly, if the test is *Empty* (*i.e.*, contains no executable statements), it is automatically *Unknown* (*i.e.*, has no direct assertions). Secondly, if there is a *Test Maverick* in a test suite, this test suite automatically has a *General Fixture*. *Test Maverick* occurs when the test suite has a `setUp()` method with fields and the given test case does not use any of the fields in it. Of course, this automatically means that there is at least one method that does not use all of the fields, which is the definition of a *General Fixture*. Other strongly connected pairs are all associated with *Assertion Roulette* due to its popularity. If a test suite has a *Duplicate Assert*, it has an *Assertion Roulette* in 93.1% of the cases. It might not be the case if the duplication has explicit messages (because *Assertion Roulette* is only considered if assertions have no messages), but since it is very common to not write error messages, duplicated assertions can become a roulette. The same goes for *Redundant Assertion*, 83.5% of the test suites with which also have an *Assertion Roulette*. This also makes sense, because if there is a redundant assertion, there probably should be some other assertion that is more meaningful.

This co-occurrence of test smells demonstrates that test smells have relationship with one another that should be explored in greater detail in the future.

VI. Threats to Validity

While we structured our study to avoid introducing bias and worked to eliminate the effects of random noise, it is possible that our mitigation strategies may not have been effective. This section reviews the threats to validity to our study.

It is possible that during the systematic mapping study of test smells we missed some test smells that are applicable to Python. Also, Python grammar is rather large, and is being actively updated, so PythonChangeMiner does not support all Python language constructs, and it is possible that we may have missed potential test smell changes because of this. We also relied on pattern detection thresholds from the original paper by Nguyen et al. [50], while it is possible that they could be different for Python and for testing code. However, the tool supports all the main features of Python and still produced a large number of code change patterns. In addition, PyNose is built in such a way that it is simple to add new test smells in the future.

The results of both parts of our study—searching for Python-specific test smells and analyzing the prevalence of test smells in Python code—rely on a specific set of open-source projects that we selected and might not generalize to all projects, including proprietary ones. However, we analyzed two moderately large datasets (*Primary* and *Secondary*) for our tasks that were curated using various conditions suggested in the literature. We believe that the similarity of results from both datasets demonstrates the reproducibility of the results of the empirical study.

It is possible for PyNose to have some unnoticed errors in its implementation. However, we tested the tool rigorously on synthetic data and performed manual evaluation on real-world data to minimize the risk as much as possible.

One threat to validity is related to the detection of specific test smells. Some of the implementations of test smells rely on specific thresholds that were picked from the literature. It is possible that these thresholds are different for Python, and this requires further study.

VII. Conclusions and Future Work

Test smells are prevalent in commonly used programming languages such as Java and have a detrimental effect not only on the quality of test code but also on the production code [11].

In this work, we presented PyNose, the first tool for test smell detection in Python code that is capable of identifying 18 test smells. 17 out of these 18 test smells were adapted from test smells for other programming languages described in the literature, and we added one test smell called *Suboptimal Assert* by analyzing the most frequent changes made to test files in 450 open-source Python projects. Experiments on a set of eight real-world projects showed that PyNose is capable of detecting test smells with 94% precision and 95.8% recall, which is on par with other publicly available tools for test smell detection. Our empirical analysis shows that test smells are prevalent in Python code, with 98% of the projects and 84% of the test suites having at least one test smell in them. The most frequent detected test smells were *Assertion Roulette*, *Conditional Test Logic*, and *Magic Number Test*. We also observed that the proposed Python-specific *Suboptimal Assert* smell occurs in the code rather often, being present in as much as 70.6% of the projects.

Future research directions for this work include:

- Supporting more test smells, including those that rely on production code.
- Discovering more Python-specific smells, which requires a specific analysis of the optimal pattern searching parameters for Python.
- Conducting a more thorough comparison of PyNose to other tools, for example, to tsDetect that works with Java. It would also be of interest to employ the tools together to carry out a comparison of large Python and Java datasets from the standpoint of test smell distribution.
- Analyzing test smell prevalence in Python on a larger dataset of projects and in other dimensions, for example, it would be of great interest to see how test smells correlate with test coverage [26].

PyNose is available on GitHub for use in the IDE and for research: https://github.com/JetBrains-Research/PyNose, all the research artifacts of this study are also publicly available: https://zenodo.org/record/5156098.

REFERENCES

[1] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[2] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos, "An empirical investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, no. 2, pp. 127–139, 2003.

[3] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[4] G. A. Oliva, I. Steinmacher, I. Wiese, and M. A. Gerosa, "What can commit metadata tell us about design degradation?" in *Proceedings of the 2013 International Workshop on Principles of Software Evolution*. ACM, 2013, pp. 18–27.

[5] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*. IEEE Computer Society, 2009, pp. 390–400.

[6] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 33, 2014.

[7] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.

[8] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[9] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 2001, pp. 92–95.

[10] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 56–65.

[11] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 1–12.

[12] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[13] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 4–15.

[14] M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 322–331.

[15] J. De Bleser, D. Di Nucci, and C. De Roover, "Assessing diffusion and perception of test smells in scala projects," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 457–467.

[16] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *Information*, vol. 11, no. 4, p. 193, 2020.

[17] D. Sarkar, R. Bali, and T. Sharma, "Practical machine learning with python," *A Problem-Solvers Guide To Building Real-World Intelligent Systems. Berkely: Apress*, 2018.

[18] Y. Golubev, J. Li, V. Bushev, T. Bryksin, and I. Ahmed, "Changes from the trenches: Should we automate them?" *arXiv preprint arXiv:2105.10157*, 2021.

[19] PyCharm. (accessed: 01.08.2021) The python ide for professional developers. [Online]. Available: https://www.jetbrains.com/pycharm/

[20] K. Reitz and T. Schlusser, *The Hitchhiker's guide to Python: best practices for development*. O'Reilly Media, Inc., 2016.

[21] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[22] B. Van Rompaey, B. Du Bois, and S. Demeyer, "Characterizing the relative significance of a test smell," in *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, 2006, pp. 391–400.

[23] M. Breugelmans and B. Van Rompaey, "Testq: Exploring structural and maintenance characteristics of unit test suites," in *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. Citeseer, 2008.

[24] J. De Bleser, D. Di Nucci, and C. De Roover, "Socrates: Scala radar for test smells," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, 2019, pp. 22–26.

[25] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, 2019, pp. 193–202.

[26] T. Virgínio, R. Santana, L. A. Martins, L. R. Soares, H. Costa, and I. Machado, "On the influence of test smells on test coverage," in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, 2019, pp. 467–471.

[27] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[28] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 311–322.

[29] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "tsdetect: an open source test smells detection tool," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1650–1654.

[30] S. Lambiase, A. Cupito, F. Pecorelli, A. De Lucia, and F. Palomba, "Just-in-time test smell detection and refactoring: The darts project," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 441–445.

[31] R. Santana, L. Martins, L. Rocha, T. Virgínio, A. Cruz, H. Costa, and I. Machado, "Raide: a tool for assertion roulette and duplicate assert identification and refactoring," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020, pp. 374–379.

[32] T. Virgínio, L. Martins, L. Rocha, R. Santana, A. Cruz, H. Costa, and I. Machado, "Jnose: Java test smell detector," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020, pp. 564–569.

[33] B. A. Kitchenham, D. Budgen, and P. Brereton, *Evidence-based software engineering and systematic reviews*. CRC press, 2015, vol. 4.

[34] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 2014, pp. 1–10.

[35] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," Technical report, Ver. 2.3 EBSE Technical Report. EBSE, Tech. Rep., 2007.

[36] PyNose. (accessed: 01.08.2021) Supplementary materials for this paper. [Online]. Available: https://zenodo.org/record/5156098

[37] P. Documentation. (accessed: 01.08.2021) File and directory access. [Online]. Available: https://docs.python.org/3/library/filesys.html

[38] A. Kicsi, L. Tóth, and L. Vidács, "Exploring the benefits of utilizing conceptual information in test-to-code traceability," in *2018 IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE, 2018, pp. 8–14.

[39] A. Kicsi, L. Vidács, and T. Gyimóthy, "Testroutes: A manually curated method level dataset for test-to-code traceability," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 593–597.

[40] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 61–70.

[41] Unittest. (accessed: 01.08.2021) Unit testing framework for python. [Online]. Available: https://docs.python.org/3/library/unittest.html

[42] pytest. (accessed: 01.08.2021) Testing framework for python. [Online]. Available: https://docs.pytest.org/en/stable/

[43] R. Framework. (accessed: 01.08.2021) Testing framework for python. [Online]. Available: https://robotframework.org/

[44] G. L. Turnquist, *Python Testing Cookbook*. Packt Publishing Ltd, 2011.
[45] G. Gousios, "The ghtorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132
[46] G. dumps. (accessed: 01.08.2021) Ghtorrent archive sql dumps. [Online]. Available: http://ghtorrent-downloads.ewi.tudelft.nl/mysql/
[47] P. Create. (accessed: 01.08.2021) A tool to create the pga dataset. [Online]. Available: https://github.com/src-d/datasets/tree/master/PublicGitArchive/pga-create
[48] V. Markovtsev and W. Long, "Public git archive: A big code dataset for all," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 34–37.
[49] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," pp. 92–101, 2014.
[50] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 819–830.
[51] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324.
[52] C. in Obspy. (accessed: 01.08.2021) A diff in the obspy project. [Online]. Available: https://github.com/obspy/obspy/commit/7719290b8dd6940dd195a195120c075a4f94cf42
[53] C. in Numba. (accessed: 01.08.2021) A diff in the numba project. [Online]. Available: https://github.com/numba/numba/commit/edbb26caad50cd0cc6352e6fe5b84fbd6edaaf9b
[54] C. in Reviewboard. (accessed: 01.08.2021) A commit message in the reviewboard project. [Online]. Available: https://github.com/reviewboard/reviewboard/commit/1758bf53057ee7b648ace1c557031d9460c88c00
[55] C. in Girder. (accessed: 01.08.2021) A commit message in the girder project. [Online]. Available: https://github.com/girder/girder/commit/f4b6040618dbe9a7edca99d8f6344a316b5b1f10
[56] C. in Beets. (accessed: 01.08.2021) A commit message in the beets project. [Online]. Available: https://github.com/beetbox/beets/commit/2b921b19fd5a23bc2e86060c2c12137d63dfe1db
[57] C. in Python-Chess. (accessed: 01.08.2021) A commit message in the python-chess project. [Online]. Available: https://github.com/niklasf/python-chess/commit/944a0e682174ff32a6f9689176aa9016bab44a31
[58] C. in Gensim. (accessed: 01.08.2021) A commit message in the gensim project. [Online]. Available: https://github.com/RaRe-Technologies/gensim/commit/342f10a2472fb22d811a398f5a6d49d1b6a88ab0
[59] C. in Requests. (accessed: 01.08.2021) A commit message in the requests project. [Online]. Available: https://github.com/psf/requests/commit/a8555d811df0a4aaf2dd1f083ba0bc71679101ca
[60] P. S. Interface. (accessed: 01.08.2021) Program structure interface in intellij platform. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/psi.html