2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

# Test Coverage in Python Programs

Hongyu Zhai
*UC Davis Computer Science*
Davis, CA, USA
hyzhai@ucdavis.edu

Casey Casalnuovo
*UC Davis Computer Science*
Davis, CA, USA
ccasal@ucdavis.edu

Prem Devanbu
*UC Davis Computer Science*
Davis, CA, USA
ptdevanbu@ucdavis.edu

***Abstract*—We study code coverage in several popular Python projects: *flask, matplotlib, pandas, scikit-learn*, and *scrapy*. Coverage data on these projects is gathered and hosted on the *Codecov* website, from where this data can be mined. Using this data, and a syntactic parse of the code, we examine the effect of control flow structure, statement type (*e.g.*, *if*, *for*) and code age on test coverage. We find that coverage depends on control flow structure, with more deeply nested statements being significantly less likely to be covered. This is a clear effect, which holds up in every project, even when controlling for the age of the line (as determined by *git blame*). We find that the age of a line *per se* has a small (but statistically significant) positive effect on coverage. Finally, we find that the kind of statement (*try, if, except, raise, etc*) has varying effects on coverage, with exception-handling statements being covered much less often. These results suggest that developers in Python projects have difficulty writing test sets that cover deeply-nested and error-handling statements, and might need assistance covering such code.**

***Index Terms*—Test coverage, Mining, Python, Code age**

## I. Introduction

When have we tested enough? Since testing can never show the *absence* of defects, we'll never know for sure, but various approaches have been proposed. Test *coverage* is one: it measures how much of the system is exercised during testing. For example, one might measure the fraction of executable statements actually executed (covered) during testing. High coverage levels indicate more careful testing, and are sometimes even mandated by Government regulation[1]. In addition, popular open-source projects (which welcome contributions from anyone) need to ensure that contributed code is of high quality; OSS projects thus need rigorous and automated integration testing practices [1].

However, ensuring high levels of coverage is difficult, especially in settings with complex, nested condition flow. Consider the fairly simple code fragment in Figure 1, where `C` refers to conditions and `S` to executable statements.

```
if C1: #Depth 1
    S1 #Depth 2
elif C2: #Depth 2
    S2 #Depth 3
elif C3: #Depth 3
    S3 #Depth 4
```

Fig. 1. Example of control flow nesting depth.

Covering the deeply nested statement S3 requires developers to find inputs that satisfy a complex condition like `C3 && !C2 && !C1`. Deeply nested statements impute complex path conditions, and thus would be even harder to cover. In addition, it is also known that exception-handling code, although quite common [2], is difficult to test, and sometimes defect prone [3]. However, if such code is largely left untested, software developers run the risk of shipping untested, and therefore potentially defective code.

How are these issues dealt with in practice? Is such deeply nested or exception-handling code covered by tests? We thus have two main research questions:

**RQ1** *Are more deeply nested statements less often tested?*

**RQ2** *Does the type of statement (condition, loop, exception-handling) affect the likelihood of being tested?*

Fortunately, data to answer such questions has become available. Many OSS projects use continuous integration services like Travis; some continuously measure test coverage and make archived coverage results available for future study. This data has been used recently [4] to study the evolution of coverage over time in several projects. Our goal here is a fine-grained understanding of how coverage relates to the difficulty of executing statements. We gather coverage data for 5 widely-used Python projects, and measure how coverage relates to control-flow nesting, and to statement types. We report the following findings.

1) Each increase in the level of control flow nesting reduces the probability of being tested by about 19%.
2) The probability of coverage depends on statement type. In particular, exception handling is 82% less likely to be covered compared to normal assignment statements.

Our findings suggest that developers need help (perhaps from automated testing tools) to specially focus on testing deeply nested code, and also error handling code. In particular, the findings of Toy [3] about the significant role of exception handling code in field failures suggest that the relatively much lower rate of coverage of exception handling code in these OSS Python projects requires more careful attention.

## II. Related Work

Test coverage is a well-known method that is often claimed as a good indirect measure of the defect-detection ability of a test set. Various coverage criteria, including statement, edge, path, and data-flow criteria [5] have been proposed. Prior empirical work has evaluated this claim, with mixed results [6], [7]. Kocchar *et al* [8] report a surprising lack of effect of code coverage on post-release quality; however their coverage data is aggregated at the project level, and does not

[1] *e.g.,* See the U.S Federal Aviation Administration document DO-178C.

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00027

record exactly which code remained untested. Our goal in this paper is not to evaluate the effectiveness of coverage, but to determine *which statements are actually being executed* during testing, and to seek indicators suggestive of which statements are less likely to be tested. By doing this, we hope to help developers and tool builders direct their efforts towards the types of code that tend to remain untested.

Several recent empirical studies made use of *Github* and examined open-source software(OSS) projects. Trautsch and Grabowski studied 10 open source Python projects (including scikit-learn) to investigate whether unit testing is widely used. Their findings suggested that developers believe that they are developing more unit tests than they actually do [9]. Another study of Hilton *et al.* [4] leveraged *Coveralls*[2] to gather coverage data of OSS projects. By doing so, they were able to include projects written in any language supported by *Coveralls*. Their goal was to study the impact of software evolution on coverage. While we analyze the effect of code age on coverage, our primary goal is to study the effects of control-flow nesting and statement type on coverage.

## III. Methodology

TABLE I
SUMMARY STATISTICS FOR THE SIZE OF THE PROJECTS IN PYTHON FILES, TRACKED NON-TEST PYTHON FILES, NUMBER OF TRACKED STATEMENTS, AND THE CODE COVERAGE FOR OUR SELECTED VERSIONS.

| Project | Files | Tracked Files | Tracked Stmts. | Coverage | Date |
|---|---|---|---|---|---|
| flask | 68 | 17 | 1965 | 90.59% | 2018.07.24 |
| matplotlib | 845 | 171 | 53050 | 75.43% | 2018.08.22 |
| pandas | 662 | 169 | 50268 | 91.98% | 2018.08.22 |
| scikit-learn | 714 | 264 | 36523 | 86.68% | 2018.08.22 |
| scrapy | 276 | 167 | 9346 | 87.18% | 2018.08.17 |

### A. Coverage Data

We choose 5 Python projects for study: *flask, matplotlib, pandas, scikit-learn*, and *scrapy*. All are under active development, include detailed guidelines for contributing, and have more than 5000 stars on Github. Table I displays information on the size and version of each project, ranging from smaller projects like *flask* to larger ones like *matplotlib*.

We gather the coverage reports using the API provided by *Codecov*[3], a popular service in the open-source community. We picked a recent version for each project (see Table I), and used the API to obtain all tracked coverage information. We measure coverage out of statements, the basic unit of programming syntax (see III-B for more detail). The files chosen to be tracked for coverage depend on how the developers configure their test set, so not all files contain coverage information. A detailed explanation of what files and statements are excluded from tracking and why is provided in Section III-C. We see in Table I that among these tracked statements, coverage is fairly high, ranging from 76% in *matplotlib* at the lowest to 91-92% for *flask* and *pandas* at the highest.

[2]https://coveralls.io/
[3]https://codecov.io

For each project, we use the `git ls-tree` command[4] to get the list of files tracked by `git`, and request coverage data for each file from *Codecov* using the `requests`[5] library. *Codecov* returns a JSON file with statement and branch coverage data for each tested line as applicable. Statement coverage information is binary, covered or not. For each branching statement, (*e.g., if*), *Codecov* uses static analysis to determine the number of statements potentially reachable from it (denominator), and dynamic analysis to determine the number reached during execution of the tests (numerator). Branch coverage is reported as a pair, numerator and denominator.

For each line, we also gathered the name of the author, the commit in which it was last modified, and the timestamp of that commit (found by `git blame`). We then calculate the age of the line in seconds by computing the time delta between the time when the coverage data is gathered and the time when the line was last modified.

### B. AST Data

We measure the degree of control flow nesting from the Python abstract syntax tree (*AST*). For each Python file in the 5 projects, we use the *ast* module from Python 3.7[6] to generate an *AST* for the file. Using a visitor pattern on the *AST*, we visit each statement in the tree, where statements are the node types defined in the Python 3 *ast* module. We also include one additional type (for 26 total) not explicitly listed in this module, *viz.,* `ExceptionHandler`, to more accurately capture *AST* information for *except* clauses.

From this tree, we calculate the depth in the *AST* for each statement node by the number of other statement nodes above it in the tree, with a statement at the top level having a depth of 1. We refer back to Figure 1 to demonstrate this idea for an *if* and *elif* statement. The first *if* is at depth 1, but each *elif* increases the depth by one, and all statements below the conditions are nested one level deeper in the tree. This matches with the intuition of how difficult a line is to cover, as each parent *elif* above a statement adds another condition to satisfy to reach it. If a single line contains 2 statements (such as a function definition and return on the same line *e.g. def double(x): return 2 * x*), we record 2 entries for that line. No line contains more than 2 statements and the vast majority contain 1. Additionally, we also record the previously discussed *AST* type of the statement node and indentation information for each node.

### C. Data Cleaning

Recall from Table I that coverage is tracked only in a fraction of the files of the project. Thus, we manually compared these files against the testing configuration files in each project. In most cases, untracked files were explicitly excluded by the developers in testing configuration files. The untracked files mostly related to testing, documentation, and examples/walkthroughs, which are arguably reasonable to exclude. We did

[4]https://git-scm.com/docs/git-ls-tree
[5]https://github.com/requests/requests
[6]https://docs.python.org/3/library/ast.html

find 20 files (3 in *flask* and 17 in *pandas*) whose exclusion we were unable to explain after manual inspection; this set was small enough to not be of serious concern given the significance of our primary results (see below).

Additionally, we noticed that the developers of some projects included test files in their coverage tracking while other projects did not. We believe that including the test files themselves in the coverage results would manually inflate the coverage, and thus exclude them from our models. We call files test files if they appear in testing directories - specifically, if they have the pattern *tests/* or *testing/* in their file path. We manually examined the files in each category to ensure that the directories were clustered correctly.

Next, we looked for inconsistencies between our *AST* data and tracked lines within files that had coverage data. After excluding "doc string" lines (which are essentially comments; ignored by `coverage.py` but counted by the *ast* module), a few error cases remained. 1236 lines of the coverage information cannot be matched with the *AST* data, and 1460 lines of the *AST* files cannot be matched with the coverage data. These arise primarily from an known error in the Python *AST* that does not correctly label the line information of multiline strings[7], but also from a difference in how the *AST* and `coverage.py` label the start of functions with annotations. There are a few cases where *CodeCov* reported coverage data on lines that are not statements, or had files with missing coverage information [8]. However, as these issues only affect about 1.3% of the data from tracked files, we again argue their exclusion unlikely to significantly impact our results. After excluding the test files and these mismatched lines, we obtain coverage data on around 151K Python statements.

As mentioned previously, *CodeCov* also provides information on branch coverage for statements with multiple paths (*if, for, etc*). These branch coverage lines accounted for slightly more than 11K statements. However, *pandas* and *scikit-learn* did not enable branch tracking in their tests, making this information incomplete in the *Codecov* data[9]. For instance, *if* statements had associated branch coverage information only 36.6% of the time in non test files. Fortunately, branch coverage is readily converted to statement coverage: the associated numerator is non-zero exactly when the branching statement is covered, since one of its "followers" is hit. Therefore, we analyzed the statement coverage data both with and without this converted branch coverage information. These results are very similar, so we present the analysis including the converted branching data for completeness.

### D. Regression Modeling

We model these results using standard logistic regression, with the response being whether or not the line was covered. We use the *AST* depth as the independent variable, and use the *AST* node type, and the age of the line as covariates, with the project id as a fixed effect, and line-number in the file as a control. The distributions of line number and line age were highly long tailed, and were log transformed to improve model fit [10]. We remove outliers with high standardized residuals (>3), check model diagnostic plots, and limit the effects of multi-colinearity by ensuring VIF scores are less than 5 [10]. We report the McFadden pseudo $R^2$ to estimate how much of the variance in coverage is explained by our model [11].

## IV. Results

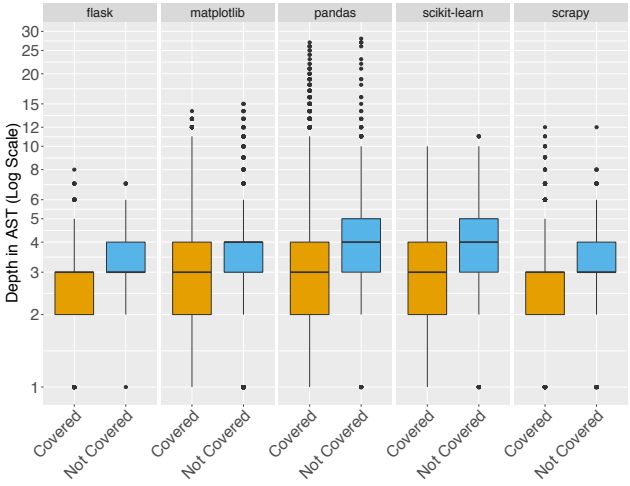


Fig. 2. Boxplots of control-flow nesting (*AST* depth) for all projects, broken by covered and uncovered statements. Depth is log-scaled. Differences are statistically significant ($p < .001$) from Wilcox tests after Benjamini-Hochberg family-wise adjustment for multiple hypothesis testing

First, consider **RQ1**: are more deeply nested statements less likely to be tested? Figure 2 shows the distributions of *AST* nesting depths, separated by project, plotted against whether or not the line is covered. Despite the differences between projects, it is clear that untested statements are usually nested more deeply in control flow than the tested statements.

Our logistic regression in Table II quantifies this relationship. Note, that with logistic regression, the coefficient of a predictor is interpreted as a log-odds ratio. So, the depth of the *AST* has odds ratio $exp(-0.21) = 0.81$. For each additional level of control-flow nesting a statement about 19% *less* likely to be tested[10]. Considering the ANOVA values, we see this coefficient is both statistically significant and explains about 3.3% of the variance in coverage in our data.

Next, we consider **RQ2**: how likely different statement types are to be covered. In our regression model, we use the *assign* (ordinary assignment statements) as the baseline for comparison against the other types. The coefficients show *assign* statements are more likely to be covered than many other types[11]. However, function and class definitions (which

[7]https://bugs.python.org/issue16806

[8]See https://codecov.io/gh/scikit-learn/scikit-learn/src/a8cd4f4c80357bf124e9c30f8488a406d06db21c/setup.py for an example.

[9]Even in the projects with the branch flag enabled, some conditionals still only had statement coverage information.

[10]Thus 35% $(1-(0.81)^2)$ less at 2 levels deep, and 47% at 3 levels deep, 57% at 4 and so on.

[11]Of the 26 statement types defined in the grammar, only 20 appear in the tracked code.

are covered when invoked) are almost 11 and 7 times as likely to be covered as *assignment*s; *Import*s, which *can* be conditionally invoked (but typically not), are thrice as likely. We also see that *ifs* are only about 33% more likely tested than assignment statements.

Exception raising (*raise*) and handling (*ExceptHandler*) are the **least** likely of all statements to be tested, both roughly around *81% less likely* to be tested than *assign* statements. This is notable, in light of widely-cited results that suggest that defects often occur in error-handling code [3].

The rest of the statements are all mostly somewhat less likely to be tested than *Assignment* (but statistically significantly so). These include *Continue* (52% less) *break* (44% less), *return* (37% less), and *Expr* statements (mostly function calls), about 43% less. *Try*, *With* and *Delete* are all about 33% less likely. *For* loops are about 22% less likely to be covered, and *while* loops are *not* statistically significantly different from assignments. In future work, we hope to examine whether these differences cause the defects in all these relatively less tested types of statements tend to persist, undetected.

We note that *line_age* (as recorded by `git blame`) is statistically significant, but with a small effect: the likelihood of coverage increases by just 13.3% when the age is *doubled*. This is consistent with the findings of Hilton *et al* [4] that coverage increases only very slightly, if at all, over time.

Finally our controls: we see that the line number (position in the file, while statistically significant, explains far less of the variance in coverage (both $< 1\%$). The amount of coverage per project varies; with the baseline flask and the pandas projects having proportionally more coverage than the other projects. All together, the project explains about 5.3% of the variance.

TABLE II
LOGISTIC REGRESSION MODEL FOR STATEMENT COVERAGE *without test files*. THE BASELINE *AST* NODE TYPE IS ASSIGNMENT, AND THE BASELINE PROJECT IS FLASK. THE MCFADDEN PSEUDO R-SQUARED IS 0.147.

| | *Dependent variable:* |
|---|---|
| | factor(numerator >0) |
| ast_depth | −0.210*** (0.005) |
| factor(node_type)Assert | −0.121 (0.170) |
| factor(node_type)AugAssign | 0.063 (0.068) |
| factor(node_type)Break | −0.574*** (0.159) |
| factor(node_type)ClassDef | 1.931*** (0.165) |
| factor(node_type)Continue | −0.737*** (0.150) |
| factor(node_type)Delete | −0.402** (0.197) |
| factor(node_type)ExceptHandler | −1.680*** (0.055) |
| factor(node_type)Expr | −0.570*** (0.023) |
| factor(node_type)For | −0.243*** (0.054) |
| factor(node_type)FunctionDef | 2.417*** (0.075) |
| factor(node_type)If | 0.286*** (0.024) |
| factor(node_type)Import | 1.157*** (0.099) |
| factor(node_type)ImportFrom | 1.069*** (0.072) |
| factor(node_type)Pass | −1.364*** (0.084) |
| factor(node_type)Raise | −1.640*** (0.036) |
| factor(node_type)Return | −0.457*** (0.025) |
| factor(node_type)Try | −0.414*** (0.067) |
| factor(node_type)While | −0.288 (0.206) |
| factor(node_type)With | −0.414*** (0.126) |
| factor(project)matplotlib | −1.377*** (0.082) |
| factor(project)pandas | 0.342*** (0.084) |
| factor(project)scikit-learn | −0.382*** (0.083) |
| factor(project)scrapy | −0.409*** (0.088) |
| log(line_number) | 0.119*** (0.007) |
| log(line_age) | 0.125*** (0.005) |
| Constant | 0.150 (0.134) |
| Observations | 151,048 |
| Log Likelihood | −55,240.990 |
| Akaike Inf. Crit. | 110,536.000 |

*Note:* *p<0.1; **p<0.05; ***p<0.01

| | Df | Deviance | Resid. Df | Resid. Dev |
|---|---|---|---|---|
| NULL | | | 151047 | 129587.09 |
| ast_depth | 1 | 4278.24 | 151046 | 125308.84 |
| factor(node_type) | 19 | 7308.36 | 151027 | 118000.49 |
| factor(project) | 4 | 6656.98 | 151023 | 111343.51 |
| log(line_number) | 1 | 343.64 | 151022 | 110999.86 |
| log(line_age) | 1 | 517.89 | 151021 | 110481.97 |

## V. THREATS

The main *external validity* threat arises from the representativeness and number of the projects chosen. Our projects are reasonably diverse, including web services (*flask*), web scraping clients (*scrapy*), data analysis (*pandas*), machine learning (*scikit-learn*) and plotting (*matplotlib*). All are in wide practical use, quite mature, and reasonably well tested. In addition, from the consistent appearance of the box plots, and the use of projects as a fixed effect in the regression, we can conclude that the effects we observe do hold, with strong statistical significance in the data overall. Finally, although we model only 150K statements, the effects are strong enough to be observed with statistical significance in our data, with vanishingly low p-values (*viz.,* low coefficient variances) that they can be expected to generalize.

The *internal validity* threats arise mainly from missing data and mis-measurement of coverage. We mitigate this by carefully analyzing and manually checking the data. As explained in Section III-C, there is only a small amount of missing data. Given the statistical significance of the main effects we report (on nesting depth and error handling) we believe that these findings are robust nevertheless. Regarding measurement of coverage, the tool used for measurement, `coverage.py` is quite mature and stable; furthermore, the data is collected and archived by *Codecov*, which is a stable and widely used site, so we believe these risks are minimal.

## VI. CONCLUSION

We analyze test coverage data on several widely-used, mature Python projects. We report two main findings: first, control-flow nesting has a strong, cumulative effect on coverage—more deeply nested code is significantly less likely to be tested. We also find that error-handling code is far less likely to be tested. Given widely-cited prior literature [3] suggesting that defects tend to occur quite often in this code, this is quite a troubling finding. In future work, we hope to examine the relationship of coverage, statement type, and control flow nesting to the actual detection of defective code in these projects: even with high-levels of coverage, if defect-prone code is not being tested, it might happen that coverage and post-release quality are not strongly related [8]! This material is based upon work supported by the National Science Foundation under Grant No. 1414172.

## REFERENCES


[1] Vasilescu, Bogdan and Van Schuylenburg, Stef and Wulms, Jules and Serebrenik, Alexander and van den Brand, Mark GJ, "Continuous integration in a social-coding world: Empirical evidence from GitHub," in *Software maintenance and evolution (icsme), 2014 ieee international conference on*. IEEE, 2014, pp. 401–405.

[2] Sinha, Saurabh and Harrold, Mary Jean, "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, 2000.

[3] Toy, Wing N, "Fault-tolerant design of local ESS processors," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1126–1145, 1978.

[4] Hilton, Michael and Bell, Jonathan and Marinov, Darko, "A Large-scale Study of Test Coverage Evolution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 53–63. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238183

[5] Rapps, Sandra and Weyuker, Elaine J., "Selecting software test data using data flow information," *IEEE transactions on software engineering*, no. 4, pp. 367–375, 1985.

[6] Hutchins, Monica and Foster, Herb and Goradia, Tarak and Ostrand, Thomas, "Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria," in *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 191–200.

[7] Horgan, Joseph R. and London, Saul and Lyu, Michael R, "Achieving software quality with testing coverage measures," *Computer*, vol. 27, no. 9, pp. 60–69, 1994.

[8] Kochhar, Pavneet Singh and Lo, David and Lawall, Julia and Nagappan, Nachiappan, "Code coverage and postrelease defects: A large-scale study on open source projects," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1213–1228, 2017.

[9] F. Trautsch and J. Grabowski, "Are There Any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 207–218.

[10] Cohen, Jacob and Cohen, Patricia and West, Stephen G and Aiken, Leona S and others, "Applied multiple regression/correlation analysis for the behavioral sciences," 2003.

[11] McFadden, Daniel and others, "Conditional logit analysis of qualitative choice behavior," 1973.