An Empirical Study on the Fault-Inducing Effect of Functional Constructs in Python

Fiorella Zampetti
University of Sannio, Italy

François Belias, Cyrine Zid, Giuliano Antoniol
Polytechnique Montréal, QC, Canada

Massimiliano Di Penta
University of Sannio, Italy

***Abstract*—Functional programming is expected to introduce several benefits to programs, including fewer side effects, easier parallelization, and even, in some circumstances, better comprehensibility. This paper investigates the extent to which the addition/modification of certain programming language constructs, *i.e.,* lambdas, comprehensions, and map/filter/reduce, have higher chances to induce fixes than other changes. To this extent, we analyze the change history of 200 popular open-source programs written in Python, accounting for $\simeq 630k$ commits and $6M$ changes. The study results show that changes to functional constructs have higher odds to induce fixes than other changes, and that some functional constructs, such as lambdas and comprehensions, have higher odds to induce fixes than others. Finally, a qualitative analysis revealed different scenarios in which functional constructs have been fixed. Results of this study may trigger better development support when using functional constructs during development, and prioritize code review and testing on certain areas of the source code.**

***Index Terms*—Empirical; Functional Programming; Python; Fix-Inducing Changes; Empirical Study**

I. Introduction

Programming languages usually conceived to be used with imperative or object-oriented paradigms may also introduce functional constructs. Generally, functional programming is intended to avoid functions having side effects on data structures. This not only eases the parallelization, especially when high performance is required, but also avoids undesired side effects and consequently bugs [23], [46].

While some programming languages are purely functional, *i.e.,* they do not contain imperative features (*e.g.,* Haskell), many other languages allow a mix-up of functional and non-functional constructs. A complete list of such languages is available online [51]. In such a context, developers could implement the same functionality with or without functional constructs. The latter leads to an open discussion about possible effects on performance and code understandability [24], [33], [45], [50].

In this paper, we focus on functional programming constructs used in Python programs. The reason why we chose Python is its increasing popularity, especially for data science/machine learning applications. In many such data-intensive applications, the use of functional constructs may become quite natural and intuitive. Specifically, Python offers the following functional programming features:

- **Lambda functions:** allowing the definition of anonymous functions on the fly.
- **List/Dictionary/Set comprehensions:** providing a more compact, and sometimes faster way of populating a data structure (*i.e.,* a list, dictionary, or set) by using iterations and conditionals directly within the structure.
- **Map/Reduce/Filter functions:** allowing the applicability of a function across several items in an iterable, in one fell swoop.
- **Function purity and immutability:** functions with a deterministic behavior based on their inputs (purity) that do not modify the parameters (immutability).

While functional constructs have some promised, expected advantages, developers may or may not be able to use them properly [53]. This may be especially the case for "impure" languages, where developers are mostly used to working with imperative constructs.

This paper investigates the extent to which changes affecting functional constructs have higher chances of inducing fixes than other changes. While function purity/immutability is a very important component of functional programming, we leave it for future work, as it requires deeper (data-flow based) source code analysis, not feasible with the available Python analysis tools, and especially for performing a large-scale study.

The study has been conducted on a set of 200 Python projects, obtained starting from a list of "engineered" projects [35], filtering out projects forked from others, inactive, and with fewer than 100 commits (to have enough change history), and then ranking the remaining projects by decreasing number of forks and manually removing tutorials, paper repositories, notebooks, and non-English repositories. For the selected 200 projects, we analyzed their change history using the Python `AST` library, and an implementation of the SZZ algorithm [43] relying on *PyDriller* [44] and `git blame`. In total, we analyzed 633,803 commits (affecting Python files) accounting for 6,159,194 changes.

This study analyzes (i) whether changes to functional constructs have higher chances to induce fixes, (ii) whether the addition of new functional constructs induces more fixes than their changes, and (iii) whether the chances to induce fixes vary among different kinds of functional constructs. Additionally, we complement the quantitative analysis with a qualitative one, performed for validation purposes, but also for understanding the scenarios where functional constructs induce a fix.

Results of the study show that changes to functional constructs have 1.15 higher odds than other changes to directly induce fixes, and 2.13 higher odds to indirectly induce fixes, *i.e.,* the churn where the functional construct was changed induced a fix. Moreover, lambdas and comprehensions have higher odds to induce fixes than map/reduce/filter functions. The qualitative analysis confirmed the quantitative results, but also pointed out cases where functional constructs were factored out in non-functional alternatives.

Our study paves the way toward better Integrated Development Environments (IDEs) or program analysis tools aimed at helping developers in writing source code when making use of functional constructs. Second, it serves as a guide for developers to prioritize code review and testing activities, *i.e.,* functional constructs may require particular attention.

II. The Studied Python Functional Constructs

This section briefly describes the functional constructs we have analyzed in our study, referring to Python 3 syntax, whereas in Section II-D we explain the main differences between Python 2 and 3.

```
is_greater_than = lambda x, y: x>y
print(is_greater_than(1, 2))
```

Listing 1. Lambda expression example.

```
half_values_dict = {k:v / 2 for (k, v) in
    all_dict.items()}
even_list = [x for x in num_list if x % 2 ==
    0 ]
double_values = {2 * item  for item  in
    my_collection}
```

Listing 2. Dictionary, list, and set comprehension examples.

```
def check_odd(number):
    return number % 2 != 0

numbers = [0, 1, 2, 3, 4, 5, 6, 7]
odd_numbers = filter(check_odd, numbers)
```

Listing 3. Filter function example.

```
numbers = [1, 2, 3, 4, 5, 6, 7]
squares = map(lambda x: x ^ 2, numbers)
product = reduce(lambda x, y: x * y, numbers)
```

Listing 4. Map and Reduce functions examples.

A. Lambda Functions

A lambda function is an anonymous function that can be defined on the fly in the code (as in Listing 1). It can also be used as a parameter of a different function. Note that a lambda function can take any number of arguments, but can only have one expression. The lambda definition in Python has the following syntax:

`lambda arguments: expression`.

B. Dictionary/List/Set Comprehensions

Dictionary, list, and set comprehensions are "Pythonic" (although available in other languages such as Javascript) ways to create and manipulate dictionaries, lists, and sets. Specifically, a comprehension (i) iterates over a collection of objects, (ii) checks whether the condition (if any) is satisfied, and (iii) evaluates an expression, which becomes the element of the collection being created (as shown in Listing 2). A list comprehension can be defined with the following syntax (similar syntax applies to the other comprehensions):

`[expression` **`for`** `item` **`in`** `iterables` **`if`** `condition]`

where the condition is optional and may be composed of multiple conditions relying on Boolean operators.

C. Map/Reduce/Filter Functions

`map()`, `reduce()` and `filter()` are three high-order Python functions. They take a function as a parameter and return a data object as output. Their syntax is similar:

`FUN (function, iterable)`

where FUN is either `map`, `filter`, or `reduce`.

`filter()` applies a function to each element of the iterable. It returns a Boolean so that only elements where the return value is `True` are retained. For instance, the filter function in Listing 3 retains only odd numbers from the list.

Similarly to `filter()`, `map()` applies the function (that can also be a lambda, as in the example of Listing 4) to each element of the iterable. In the example, it returns a list of squares for each element.

`reduce()` (which, differently from the previous functions, is not built-in, but part of the `functools` package) applies a function (that takes two arguments) to the items of an iterable cumulatively, to generate a single value. The first invocation is applied to the first two elements of the iterable, then to the partial results and the next element, and so on. For instance, in Listing 4, `reduce()` computes the product of the values contained in `numbers`.

D. Functional constructs: Python 2 versus Python 3

The studied functional constructs underwent major and breaking changes when moving from Python 2 to Python 3. While in Python 2 `map()` and `filter()` return a `list` type, in Python 3 they return an iterator [37]. To guarantee compatibility, developers must explicitly convert `map()` and `filter()` returned values into a list, *e.g.,*`list(map(...))` or else use a list comprehension. A further change was to move the previously built-in `reduce()` function into the `functools` package.

Despite long discussions and several attempts[1] to get rid of lambdas in Python, to date they are still there. However, Python 3 introduced a change with respect to Python 2, *i.e.,* a lambda accepting multiple parameters is no longer able to accept a tuple and unpack it over the parameters instead. Specifically, a Python 2 `lambda (x,y) : x+2*y` needs to be transformed into `lambda a : a[0]+2*a[1]`, *i.e.,* the unpacking is explicitly performed on the right-hand side. Then, both can be invoked by passing a tuple of two elements.

III. Study Design

The *goal* of this study is to investigate whether changes to Python functional constructs—and specifically lambdas, comprehensions, and map/reduce/filter functions—have more chances to induce fixes than other changes. The *quality focus*

[1]https://mail.python.org/pipermail/python-dev/2006-February/060415.html

is the software fault-proneness, which may be influenced by the use of such programming constructs. The *perspective* is that of researchers and practitioners aiming to understand the downside of using functional constructs. The *context* accounts for 200 open-source Python projects hosted on GitHub.

The study addresses the following three research questions:

- **RQ$_1$:** *To what extent do changes involving functional constructs induce more fixes than other changes?* We aim to provide the "overall" answer to our study goal, by looking at whether changes involving functional constructs have higher odds of inducing fixes than other changes.
- **RQ$_2$:** *How do the odds to induce fixes vary between additions of functional constructs and their update?* We investigate whether changes adding functional constructs are more likely to induce fixes than changes simply altering them. The conjecture we would like to verify is whether functional constructs tend to be "born defect-prone" and also whether changes made to them still induce further fixes.
- **RQ$_3$:** *How does the fault-inducing proneness vary among different types of functional constructs?* We are interested in verifying whether developers may experience different levels of difficulty when using different types of functional constructs, namely lambda functions, dictionary/list/set comprehensions, and filter/map/reduce functions.

A. Context Selection

We answer our research questions by mining the change history of 200 open-source Python projects hosted on GitHub and selected through the following procedure.

We relied on an existing dataset of engineered software projects made available by Munaiah *et al.* [35] by selecting only the ones mainly written in Python and classified as "engineered" by their Random Forest approach (which achieves 82% precision and 86% recall). For each engineered project, we used the GitHub API [1] to extract meta-data, and filter out projects that do not exist/are not accessible anymore. We also filtered out forks and projects with less than 100 commits, as we wanted to analyze projects with enough evolution history. To avoid biasing our results, we did not look at the presence of functional constructs in the chosen projects beforehand.

We ended up with a set of 6,509 projects, which we ranked by forks and manually selected the top 200 projects satisfying the following three criteria: (i) the project is not a tutorial, book code, or code examples, (ii) the project has its commit history and issues descriptions mostly written in English, and (iii) the project has at least one commit in the last year (from January to December 2021). The selected projects have a size (in NLOC) between 1k and 987k (median 18k). The complete list of projects is available in our online appendix [52].

B. Data Extraction

Once repositories of the projects described in Section III-A have been cloned, we analyzed their evolution history to identify (i) changes to functional constructs, and (ii) bug fixes, and fix-inducing changes. Finally, we combined the two pieces of information to perform the analysis of the results described in Section III-C. The analysis has been performed by considering all commits of all branches, excluding merge commits.

1) Analysis of Changes to Functional Constructs: One option to analyze Python changes would have been to leverage *Gumtree* [13]. Although this is the state-of-the-art for AST-based, fine-grained differencing analysis, for Python it leverages the *pythonparser* module [12], which suffers from two limitations: (i) it cannot parse Python 2 source code, and (ii) its AST nodes fail to properly distinguish comprehensions from regular lists/dictionaries. A much better parsing option for our purposes is the Python 3.10 Abstract Syntax Tree (AST) module [38], which was used in our work.

We parsed each Python file and then traversed its parse tree extracting relevant nodes. Each AST node is decorated with information including the beginning and ending line/column. Furthermore, each call node, of type `ast.Call`, contains an `ast.Name` node to store the identifier of the called object, *i.e.,*`id`, and, if needed, the passed arguments. Specifically, dictionary/list/set comprehension and lambda constructs are explicitly represented as AST nodes, *i.e.,* the AST features the `ast.DictComp, ast.ListComp, ast.SetComp` and `ast.Lambda` nodes. `Filter, map, and reduce` functions, instead, are call nodes (`ast.Call`) whose `id` is equal to `filter`, `map`, or `reduce`.

The extraction process has been performed on all Python files being modified in each commit and, for each of them, we stored the file name, the hash of the commit together with its date, as well as the list of functional constructs identified together with their location, *i.e.,* beginning and ending lines.

For each commit, Python files have been parsed in their version before and after the commit. If a file was added, all functional constructs identified are considered as added. Similarly, if a file was removed, its functional constructs were considered as removed.

For file changes, we relied on the git context diff to identify churns of code being modified with no surrounding context (*i.e.,* we set the context parameter to zero), as well as to trace source code lines between the two different versions of the file. Knowing the starting and ending location for each functional construct, as well as the starting and ending location of each code churn being modified in a commit, whenever a functional construct is involved in a change, given the two file versions, we consider the following three cases:

- if the construct appears in both versions and its lines have been changed, we conclude that the functional construct is *modified*;
- if the construct appears only in the first version, then we consider it as *deleted*; and
- similarly, if the construct appears only in the second version, we consider it as *added.*

The approach does not distinguish semantic-altering changes from semantic preserving changes (*e.g.,* refactoring). Therefore, we may detect irrelevant changes [30]. In our qualitative analysis (Section V), we discuss a sample of such cases.

TABLE I
SUMMARY OF THE TYPE OF CHANGES TO THE FUNCTIONAL CONSTRUCTS OBJECT OF THE STUDY.

Metric	Value	Q1	Median	Q3
map	All	3	20	78
	Addition	2	9	31
	Update	0	4.5	21
	Removal	0	2.5	9
reduce	All	0	0	2
	Addition	0	0	0
	Update	0	0	0
	Removal	0	0	0
filter	All	0	4	13.5
	Addition	0	2	5
	Update	0	0	3
	Removal	0	0	1
List comp.	All	74	237	745
	Addition	34	99	281.5
	Update	18	66	241
	Removal	12	32.5	90
Set comp.	All	0	0	2
	Addition	0	0	0
	Update	0	0	0
	Removal	0	0	0
Dict comp.	All	0	6	28
	Addition	0	2	10
	Update	0	0	3.5
	Removal	0	0	2
Lambdas	All	37.5	168	468
	Addition	18	63	171
	Update	8	27	135
	Removal	5	18	52

Table I reports descriptive statistics (first, second and third quartile) of changes involving functional constructs affecting the studied projects. Note that the "All" line does not discriminate between the type of change, *i.e.,* addition, update, and removal. As it can be seen, changes related to some constructs such as `reduce`, `filter`, set and dictionary comprehensions are rare. They are less used especially in recent Python versions (where `reduce` is even in a separate package), while comprehensions are more common for lists than for dictionaries and sets.

2) Analysis of Fix-Inducing Changes: The fix-inducing analysis has been performed by leveraging a lightweight version of the SZZ algorithm [43], similarly to what has been done in previous work analyzing the relationship between code naturalness and defects [42]. This is because, differently from Java—where other authors have identified curated projects properly handling defects through an issue tracker [11]—we noticed that Python projects make an inconsistent usage of issue trackers, and often contain fixes not even linked to the issue tracker. At the same time, we could have limited our attention to the subset of fixes explicitly linked, but this could have possibly biased the dataset [8].

For this reason, we decided to rely on a simpler, lightweight identification of fix commits, *i.e.,* we limited our attention to commits whose message matches the following regular expression: "$\backslash b(fix|fixed|fixing|fixes|bugs?)\backslash b''$ ($\backslash b$ stands for matching word boundaries).

Once fix commits have been identified, our analyzer determines fix-inducing commits by leveraging *PyDriller* features and the Unix `git blame` (applied to the lines affected by the fix). In doing so, we discard changes only involving white spaces, as well as changes to commented lines. Through the `git blame -p` (porcelain option), we also handle file renaming and we map line numbers between inducing and fixing changes. For each changed line belonging to fixed Python files, we obtain a candidate introduction location, *i.e.,* commit, file name, and line number.

While some studies on SZZ suggest not considering the first commit of a file [10], in our study we intentionally decided to not do so, as we are also interested to investigate the extent to which functional constructs created in the first file instance are subject to fixes.

3) Combining Functional Changes and Fix-Inducing Changes: As the last step of our analysis, we combine the two aforementioned analyses, *i.e.,* changes to functional constructs and fix-inducing changes. Given a change in a commit, we identify (i) whether it induces a fix, and if yes, which lines of the churn induce the fix, (ii) whether the churn contains a change to functional constructs, and if yes which ones, and whether the construct was added or modified, and (iii) whether the functional construct change and the fix-inducing change occur on the same line.

C. Analysis Methodology

In the following, we describe the methodology adopted to address the research questions of the study.

To address RQ_1, we compare the proportion of fix-inducing changes involving functional constructs with those not involving functional constructs. The latter has been done following a methodology similar to a previous work analyzing the extent to which refactoring induces a fix [11], except that we operate at a finer-grained level of detail. Specifically, the analysis has been performed at the granularity of changes, *i.e.,* consecutive lines changed in a source code file. Therefore, we divide the changes into the following groups:

1) The churn affects a functional construct and induces a fix;
2) The churn affects a functional construct and does not induce a fix;
3) The churn does not affect a functional construct, but induces a fix; and
4) The churn does not affect a functional construct nor it induces a fix.

Then, we compare the proportions of (1) over (2) and (3) over (4) using Fisher's exact test [15], and determine the Odds Ratio (ORs) effect size. An odd is the ratio between the probability of an event to occur (inducing a fix in our case), and its probability of not occurring (not inducing a fix). The OR, instead, is the ratio between the odd of the experimental group (changes to functional constructs) and the control group (other changes).

The first part of the analysis considers the two events (addition/change of a functional construct, and inducing a fix) co-occurring if they happen in the same change. This means that one could change a functional construct in a line, although a neighboring-changed line has induced a fix. The rationale is that we would like to analyze the fix-inducing effect of the whole changed fragment, *i.e.,* while one was changing a functional construct, the fix was induced.

Since the two events could still be uncorrelated, we restrict the analysis by looking at whether the change to the functional construct and the fix-inducing change occur on the same line, and recompute the Fisher's test results and the ORs. In this case, (1) becomes "the churn has affected functional construct(s) and induced a fix on the same line where the functional construct was/were changed", while (3) becomes "the churn has induced a fix, but either it has not affected a functional construct or if so, this happened on a different line than the fix".

While the aforementioned analysis could be enough for addressing RQ$_1$, there could be confounding factors influencing the odds a change has to induce a fix. Among others, we consider the change size, since previous literature [31] points out that a large change has higher odds to induce a fix. Of course, there could be many further metrics, and in general factors, that we do not consider as part of this study, as discussed in Section VI.

To consider this effect, we build a binominal, mixed-effect generalized linear model. This model relates a dichotomous dependent variable (whether a change induces a fix), with independent variables, *i.e.,* the churn size (in LOC) and whether the change affected a functional construct, along with the interaction between the two independent variables. Also, the model considers the project as a random effect, to separate the peculiar characteristics of a project from the effect produced by the investigated factor. The model has been built using the *glmer* function from the *R* [39] *lme4* package [4]. The model reports: (i) fitting indicators, *i.e.,* Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC), log likelihood, deviance and degree of freedom of residuals; (ii) statistics for scaled residuals, (iii) variance and standard deviation for the random effects, and (iv) estimate, standard error, *z*-value and *p*-value for the fixed effects. The mixed-effect model involves multiple comparisons, therefore we adjusted *p*-values using the Benjamini-Hochberg correction [6].

To address RQ$_2$, we look at whether a change has involved (i) the addition of new functional constructs, and/or (ii) the revisions to existing functional constructs. Similarly to RQ$_1$, we leverage a mixed-effect model to determine whether a fix-inducing change is correlated with additions, changes, as well as their interaction with the change size. In this context, controlling the effect of size is particularly important, as some additions may tend to be larger than modifications.

For RQ$_3$, we analyze how the fix-inducing-proneness varies among different types of functional constructs. Also in this case, we use a mixed effect model that considers whether a change is fix-inducing as the dependent variable, and as independent variables whether the change involved the three types of functional constructs (lambdas, comprehensions, or map/reduce/filter functions).

TABLE II
RQ$_1$: FISHER'S EXACT TEST AND ODDS RATIO BETWEEN CHANGES AND FIX-INDUCING CHANGES.

	$p-$value	OR	Conf. Int.
CHURN-LEVEL	**<0.001**	2.23	[2.18, 2.28]
LINE-LEVEL	**<0.001**	1.15	[1.12, 1.18]

TABLE III
RQ$_1$: MIXED-EFFECT LOGISTIC REGRESSION RELATING CHANGES TO FUNCTIONAL CONSTRUCTS, CHANGE SIZE, AND THEIR INTERACTION WITH FIX-INDUCING CHANGES.

AIC	BIC	logLik	deviance	df.residuals
1,188,360.7	1,188,428.4	-594,175.3	1,188,350.7	5,659,731
SCALED RESIDUALS:				
Min	**1Q**	**Median**	**3Q**	**Max**
-0.695	-0.182	-0.124	-0.071	40.513
RANDOM EFFECTS:				
Groups			**Variance**	**Std.Dev.**
Project (Intercept)			1.37	1.171
Number of obs: 5,659,736, Groups: Project: 200				
FIXED EFFECTS:				
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-4.39	0.08	-52.63	**<0.001**
Func.	0.59	0.03	22.26	**<0.001**
Churn Size	0.12	0.001	81.10	**<0.001**
Func:Churn Size	-0.078	0.008	-9.93	**<0.001**

IV. STUDY RESULTS

This section discusses the results for the defined RQs.

A. RQ$_1$: To what extent do changes involving functional constructs induce more fixes than other changes?

To address RQ$_1$, we compare the proportion of fix-inducing changes occurring in commits adding/modifying functional constructs versus other changes, discriminating between whether the overlap occurs at churn(s) or line(s) level.

Table II reports the results of Fisher's exact test, as well as the ORs and their confidence interval, considering data from all the studied Python projects. An OR greater than one indicates that a commit where a functional construct is added or modified has higher odds than other commits to induce a bug. As shown in the first line of Table II, commits introducing or changing a functional construct have significantly higher odds ($OR = 2.23$) to induce a bug than other changes.

If we consider that a functional construct induces a bug only if a line where it has been introduced/modified is also modified in a bug-fixing commit, the OR is reduced to 1.15. This means that changes to functional constructs still have higher odds to be directly responsible for fixes than other changes, yet the odds substantially decreases. The higher OR for the churn-level overlap may indicate a high proneness to the code surrounding functional constructs (regardless of the specific line) to be subject to induce fixes.

Despite what observed above, it is still possible that higher odds could be due to other reasons. Above all, larger changes may be more likely to induce fixes than smaller ones. For this reason, we have evaluated whether, even in the presence of the "size" effect, *i.e.,* the number of lines of code being modified in the commit, commits dealing with functional constructs

TABLE IV

RQ_2: MIXED-EFFECT LOGISTIC REGRESSION RELATING THE TYPE OF CHANGE TO FUNCTIONAL CONSTRUCTS (ADDITION/CHANGE) WITH FIX-INDUCING CHANGES.

	AIC	**BIC**	**logLik**	**deviance**	**df.residuals**
	1,487,823.4	1,487,878.0	-743,907.7	1,487,815.4	6,158,991
SCALED RESIDUALS:					
	Min	**1Q**	**Median**	**3Q**	**Max**
	-1.010	-0.202	-0.132	-0.084	32.618
RANDOM EFFECTS:					
Groups				**Variance**	**Std.Dev.**
Project (Intercept)				1.253	1.119
Number of obs: 6,158,995, Groups: Project: 200					
FIXED EFFECTS:					
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**	
(Intercept)	-3.90	0.08	-49.077	**<0.001**	
Added Func.	1.15	0.01	80.43	**<0.001**	
Changed Func.	0.24	0.02	13.73	**<0.001**	

TABLE V

RQ_2: MIXED-EFFECT LOGISTIC REGRESSION RELATING THE TYPE OF CHANGE TO FUNCTIONAL CONSTRUCTS (ADDITION/CHANGE), SIZE OF THE CHANGED CHURN, AND THEIR INTERACTION WITH FIX-INDUCING CHANGES.

	AIC	**BIC**	**logLik**	**deviance**	**df.residuals**
	1,188,319.7	1,188,414.5	-594,152.8	1,188,305.7	5,659,729
SCALED RESIDUALS:					
	Min	**1Q**	**Median**	**3Q**	**Max**
	-0.733	-0.182	-0.124	-0.071	40.512
RANDOM EFFECTS:					
Groups				**Variance**	**Std.Dev.**
Project (Intercept)				1.371	1.171
Number of obs: 5,659,736, Groups: Project: 200					
FIXED EFFECTS:					
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**	
(Intercept)	-4.39	0.08	-52.61	**<0.001**	
Added Func.	-0.06	0.06	-1.04	0.2997	
Churn Size	0.12	0.001	81.32	**<0.001**	
Changed Func.	0.70	0.03	22.73	**<0.001**	
Added Func:Churn Size	0.02	0.01	1.54	0.1510	
Churn Size:Changed Func.	-0.11	0.01	-9.79	**<0.001**	

correlate with fix-inducing changes. Table III reports the results of the binomial mixed-effect logistic regression, considering the project as a random effect and considering functional changes, change sizes, and their interaction as independent variables.

Results point out that the change to functional construct, the change size, and their interaction have a statistically significant effect on the probability that the change induces a fix. Specifically, by observing the estimates, the presence of a change impacting a functional construct increases by $e^{0.59} = 1.80$ times the odds that a commit induces a fix, while a unit increment of the change size increases the odds by $e^{0.12} = 1.13$. In other words, a change involving $\simeq 6$ lines would have the same odds to induce a fix as a single change to a functional construct. Finally, the interaction between changes to functional constructs and change size has a very small estimate.

RQ_1 Summary: Changes dealing with functional constructs have higher odds of inducing a fix than other changes, even if at line-level granularity the effect size is reduced. When controlling for "size", changes to functional constructs still play a significant role in inducing fixing activities.

B. RQ_2: How do the odds to induce fixes vary between additions of functional constructs and their update?

Table IV reports the results of the mixed-effect logistic model used to evaluate whether changes adding functional constructs are more likely to induce a fix compared to changes simply modifying them. By looking at the estimates shown in the table, we can state that functional constructs have higher chances of being "born defect-prone". Specifically, a commit introducing a new functional construct increases by $e^{1.15} = 3.16$ times the odds that the change introduces a bug, while a commit modifying existing functional construct increments by $e^{0.24} = 1.27$ times the odds that the change induces a fix.

These results point out that developers tend to modify existing functional constructs to fix bugs introduced when adding functional constructs into the code base. However, it is still possible that when fixing a bug related to a functional construct, or when naturally evolving them, developers introduce new bugs.

Similar to what has been done for answering RQ_1, we have evaluated the effect of the change size. Note that, before that, we still preferred to show the model without the size variable, to highlight the differences.

As shown in Table V, only the change to existing functional constructs, the change size, and their interaction have a statistically significant effect on the likelihood that the change is inducing a fix. In particular, the update of a functional construct has $e^{0.70} = 2.01$ higher odds of inducing a fix, while a unitary increment in the number of changed lines increases the odds by $e^{0.12} = 1.13$. As it can be seen, when accounting for the "size", changes adding new functional constructs are not statistically significant, likely because additions often occur in the context of larger changes, and in that case, the "size" variable already captures the effect.

RQ_2 Summary: The addition of new functional constructs has higher odds of inducing fixes than changes to those constructs. However, such a higher effect is highly captured by the change size for changes adding new functional constructs.

C. RQ_3: How does the fault-inducing proneness vary among different types of functional constructs?

Table VI shows the results of the mixed-effect logistic regression model performed to check the extent to which the fault inducing proneness of a change varies among different types of functional constructs, *i.e.,* lambda functions, list/set/dictionary comprehensions, and map/reduce/filter functions. In this case, we do not control for size, as we are simply interested in comparing the effect (in isolation) of the three types of changes, regardless of their interaction with other factors.

As the table shows, all three different categories of functional constructs have a statistically significant correlation with fix-inducing changes. Changes dealing with lambda functions have the highest odds of inducing a fix (*i.e.,* $OR = e^{0.98} = 2.66$),

TABLE VI
RQ$_3$: MIXED-EFFECT LOGISTIC REGRESSION RELATING THE TYPE OF FUNCTIONAL CONSTRUCT WITH FIX-INDUCING CHANGES.

AIC	BIC	logLik	deviance	df.residuals
1,488,093.6	1,488,161.7	-744,041.8	1,488,083.6	6,158,990
SCALED RESIDUALS:				
Min	**1Q**	**Median**	**3Q**	**Max**
-1.111	-0.202	-0.132	-0.084	32.637
RANDOM EFFECTS:				
Groups			**Variance**	**Std.Dev.**
Project (Intercept)			1.248	1.117
Number of obs: 6,158,995, Groups: Project: 200				
FIXED EFFECTS:				
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-3.90	0.08	-49.17	**<0.001**
Comprehensions	0.66	0.02	43.75	**<0.001**
Lambdas	0.98	0.02	58.40	**<0.001**
Map/Reduce/Filter	0.28	0.04	7.19	**<0.001**

followed by changes handling list, set, or dictionary comprehensions, *i.e.,* $OR = e^{0.66} = 1.93$, and finally, changes involving map/reduce/filter functions with an $OR = e^{0.28} = 1.32$. This result is somewhat expected, considering that Python developers mainly rely on lambda functions and list comprehensions during their normal development activities, while only rarely introducing map/reduce/filter functions, as well as set and dictionary comprehensions (see Table I). Also, although this needs to be further investigated, it may be that lambdas and comprehensions have a more complex syntax that makes bug introduction easier.

RQ$_3$ Summary: The fix-inducing proneness of changes involving functional constructs varies based on the type of functional construct being introduced and/or modified, with changes handling lambda functions having the highest odds to induce a fix, followed by comprehensions.

V. QUALITATIVE ANALYSIS AND IMPLICATIONS

In the following, we report the results of a qualitative analysis of a subset of the studied fixes, as well as the study implications.

A. Qualitative Analysis

This qualitative analysis complements the quantitative results reported in Section IV and has a two-fold goal: (i) validate our methodology, and (ii) identify and discuss different scenarios in which functional constructs are changed or removed.

We extracted a statistically significant, randomly stratified sample (using projects as strata), consisting of 340 out of 2,442 fixes, in which there is a line-level overlap between the change to the functional construct and the bug fix. The chosen sample size ensures a confidence interval of $\pm 5\%$ for a confidence level of 95%.

Each sample was manually analyzed by two independent annotators (two authors) in two validation steps, to determine whether the fix is a semantic-altering change involving a functional construct, and added some notes about the change rationale. For the latter, our aim was not to perform a categorization, but mainly to identify (i) cases of bulk removal/replacement, (ii) refactoring or major reformatting not filtered out by the diff, and (iii) changes in which a functional construct was translated into a non-functional alternative. During this validation, the annotators achieved a Cohen's kappa [9] inter-rater agreement of 0.63, which can be considered a strong agreement. Finally, all validation discrepancies have been solved by a third annotator.

In the validated sample, we found 265 cases (78%) in which the fixes changed, or removed the functional constructs and altered the program semantics. In 52 such cases, there was a major code block removal and replacement, *i.e.,* the functional construct was incidentally removed along with other changes. This is unsurprising and consistent with previous studies on software quality, which found that code smells are mainly removed because the code containing them disappears [49].

Among the cases of specific fixes to functional constructs, we found several instances in which the construct was changed either because of various evolutionary needs, developers' misunderstanding, or (3 cases in our sample) for compatibility reasons between Python 2 and 3, as discussed in Section II-D. In this context, we found cases where developers admit their fix to functional construct in the commit message. Table VII reports some examples of documented changes to functional constructs, highlighting that the reasons for such changes are many-fold. Many of the shown cases concern fixes within the construct, which could be caused by a misinterpretation/misuse of the construct itself. For instance, #6 (see Listing 5) shows a list comprehension misuse being fixed, where the commit message clearly states that: *"The `isdir()` test must be on the outer (`for d in dirs`) scope rather than the inner: otherwise `listdir()` may be called with a non-existent directory"*. For comprehensions, we found a change (#4) related to Python 2 compatibility (*i.e.,* dictionary comprehension unsupported), and a case (#5) in which the fix concerned a `filter` function, that has been replaced by a comprehension. Finally, for `map`, we found a case (#9) of incompatibility of the used syntax with Python 3. As shown in Listing 6, because of that, developers decided to use list comprehensions instead.

Although our sample features 75 (22%) commits in which we could not confirm a semantics-varying change to functional constructs, many such cases are very interesting to discuss, because they might have relevant implications. In particular, 64 out of the semantic-preserving changes fall in the following cases:

- Various forms of semantically-equivalent changes (38 cases). These include formatting over multiple lines, often highlighted by tools like *flake8* that the space-insensitive `diff -w` missed. Those often aim at improving the understandability of complex lambda or comprehensions. Also, these include other performance-preserving changes, *e.g.,* moving a line without altering the program's semantics, but also changes aimed at improving the program from a non-functional perspective. A very interesting one is shown in Listing 7, where a function invocation is extracted out of a list comprehension, because such a construct is not optimized for function calls, and this could cause a performance degradation [45], [50].

TABLE VII
EXAMPLES OF DOCUMENTED FIXES TO FUNCTIONAL CONSTRUCTS.

#	URL	Commit Message
1	https://github.com/tgalal/yowsup/commit/f167600	…Fixes #2778 **<lambda>**() takes 0 positional arguments but 2 were given
2	https://github.com/pyca/cryptography/commit/0d0d70b	…remove **lambda** not necessary for dismiss
3	https://github.com/davidhalter/jedi/commit/cd7774f	…**lambda** can be used as a default param in function which means there have been slight changes to the parser to allow that
4	https://github.com/numba/numba/commit/747ac0d	fix: python26 compatibility python26 doesn't like **dictionary comprehensions**
5	https://github.com/kliment/Printrun/commit/be7a16b	Python 3: Replace **filter** with **comprehensions** …
6	https://github.com/sosreport/sos/commit/4aae08a	[fibrechannel] fix **list comprehension filter** scope …
7	https://github.com/getsentry/sentry/commit/d4ee549	Issue #51: chart displaying dates into the future …This is also handled with this changeset by padding the points for every hour as calculated from max_days parameter to the template **filter** chart_data.
8	https://github.com/pyqtgraph/pyqtgraph/commit/03c01d3	Fixes related to CSV exporter: …removed call to **reduce**() from exporter
9	https://github.com/HIPS/autograd/commit/9224718	Fix Python 3 incompatibility in table outputter '**map(...)**.index()' doesn't work on Py3, just use a list comprehension instead.

```
devs = [join(d, dev) for d in dirs for dev in listdir(d) if isdir(d)]
```

```
devs = [join(d, dev) for d in dirs if isdir(d) for dev in listdir(d)]
```

Listing 5. List comprehension misuse example [22]

```
match = lambda key: error_type == key[0] and len(re.findall(key[1], error_message)) != 0
matches = map(match, keys)
```

```
matches = [error_type == key[0] and len(re.findall(key[1], error_message)) != 0 for key in keys]
```

Listing 6. Compatibility of `map()` in Python 3 example [19]

- Refactoring, including for example variable renaming, but also extracting methods containing a comprehension (16 cases).
- Replacement of functional constructs with non-functional alternatives or with different ones (10 cases). Besides those due to Python compatibility (hence semantic changing for a given Python version), others were performed for different reasons, including making the code clear. This includes cases where developers decided to avoid lambdas (as it is debated a lot among Python developers), and replace them with explicit functions. For instance, this happens in Listing 8. Also, there are cases where comprehensions were translated in other functional constructs, *e.g.,* `map()` as in Listing 9, `reduce` in list comprehension with aggregation functions (see Listing 10), or list comprehensions into loops.

B. Implications

Our study results trigger several implications for developers, educators, and researchers.

Developers should carefully ponder the use of functional constructs, balancing syntactic elegance (and shorter code) with understandability, also knowing that performance may or may not improve [45], [50]. In some cases, the awareness of given patterns and antipatterns (*e.g.,* calls within comprehensions are not optimal) would be desirable. Moreover, developers

```
return [x for x in expr if x in self._pki_minions()]
```

```
minions = self._pki_minions()
return [x for x in expr if x in minions]
```

Listing 7. Avoiding function calls in comprehensions [20]

```
rq = lambda s: s.strip("\"'")
```

```
def rq(s):
    return s.strip("\"'")
```

Listing 8. Lambda removal example [18]

should give high priority to code review (*e.g.,* by using specific checklists) and testing of functional constructs. For example, coverage criteria should be applied when testing comprehensions, other than conventional loops and conditionals.

Educators should, on the one hand, give proper emphasis to the syntax and usage examples of functional constructs, showing typical misuse cases. On the other hand, they should also discuss the pros and cons of using functional constructs during development, as to the need for properly reviewing/testing such code.

Researchers could develop, similar to what was done for lambdas in Java [47], [16] refactoring tools aimed at supporting the conversion. Also, automated bug fixing approaches should focus on typical mistakes occurring when using functional constructs. Last, but not least, it would be useful to conduct experiments to assess the understandability of functional constructs vs. non-functional alternatives.

VI. THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation. These threats concern sources of imprecision in our measurements. As explained in Section III-B2, given the heterogeneity of Python projects in terms of how issues are managed, our analysis of fix-inducing commits is based on fixing information originating from commits only. Upon interpreting our results, this simply means that we identify changes inducing "any type of fixes" and not only those traced in issue trackers and classified as fixed and closed bugs there. We do not necessarily assume that fixes are related to defect

```
password = [ord(x) for x in list(password)]
```

```
password = list(map(ord, password))
```

Listing 9. Replace comprehension with `map()` example [21]

```
stroke_y = reduce(lambda total, pt: total +
    pt.y, stroke.points, 0.0)
stroke_x = reduce(lambda total, pt: total +
    pt.x, stroke.points, 0.0)
```

```
stroke_y = sum([pt.y for pt in stroke.
   points])
stroke_x = sum([pt.x for pt in stroke.
   points])
```

Listing 10. Replace `reduce()` with comprehensions and aggregation functions example [17]

fixes, as properly pointed out in related work about issue misclassification [3], [25].

On the same line, the approach to identifying changes to functional constructs may be prone to possible imprecision, and in general, the scripts on which this work is based may be error-prone. The qualitative analysis described in Section V-A mitigates this threat, and we have discussed the cases where the approach fails (*e.g.,* refactoring, or other semantically-equivalent changes).

Threats to *internal validity* concern factors internal to our study that can influence the observed results. First, the study aims to find a correlation between changes occurring to functional constructs and fix-inducing changes. We have complemented our statistical evidence with some qualitative analysis, as explained in Section V-A. However, through the achieved results, we cannot claim causation. Also, in some cases the cause of a fix may be different from what it appears. For example the migration from one version of Python to another may generate fixes that seem to be unrelated to that. Another confounding effect is the change size, and we have used appropriate models to evaluate its effect. However, proper care should be taken if using the considered factors (*i.e.,* change to functional constructs) in predictive models, as they could correlate with other factors.

Threats to *conclusion validity* concern the relationship between experimentation and outcome. We have used appropriate statistical tests (Fisher's exact test and mixed-effect models) and effect size measures (odds ratio). Also, we made sure that p-values (that are, in any case, very small) are still significant after adjusting them through Benjamini-Hochberg [6] correction.

Threats to *external validity* concern the generalizability of our findings. Our study is intentionally scoped in terms of programming language (Python) and functional constructs studied (lambdas, comprehensions, and map/reduce/filter). We do not know whether the obtained results would generalize to other programming languages, and above all to other functional programming constructs, such as immutable functions. Moreover, although the considered sample is relatively large, results may not generalize to other projects, hence replications are desirable.

Threats to *reliability validity* concern the extent to which the obtained findings can be reproduced. We are making available the analysis scripts and working datasets, other than providing full details about the data extraction and analysis procedures.

VII. Related Work

This section discusses related work about (i) studies on functional programming, and approaches to support functional programming, and (ii) studies about fix-inducing features in programs.

A. Studies on Functional Programming

After Java introduced lambdas in its release 8, researchers started investigating how developers rely on lambda expressions and functional operations (*e.g.,* map and filter), given their ability to enable parallelism and make the code more succinct and readable. Tanaka *et al.* [46] show that lambda is the most accepted function idiom in Java (16%). A related study was performed on Python by Rao and Chimalakonda [41], who found that 78.57% of open-source Python projects have at least one lambda expression in their code, mainly introduced for replacing functions, anonymous classes, iterators in built-in libraries, and for using the same function for multiple purposes. Alexandru *et al.* [2] conducted, through interviews, a study on the usage of Pythonic idioms, including functional constructs. The study points out how developers with different experience have a different perception of Pythonic idioms in terms of support for improving source code understandability and performance. Their study also analyzes, on a sample of 1,000 projects, the usage of Pythonic constructs, and indicate that constructs also considered in our study (in particular comprehensions and lambdas) are largely used in Python programs.

Lucas *et al.* [33] conducted an in-depth study to evaluate the effect of introducing lambda expressions on program comprehension. By looking at 66 pairs of real code snippets before and after the introduction of lambda expressions, and measuring code readability, they did not find evidence that the introduction of lambdas improves software readability. However, by surveying software practitioners, they found that the introduction of lambda expressions improves program comprehension. Using a different approach, Hanenberg and Mehlhorn [24] evaluated the readability of lambdas in comparison to anonymous inner classes (AICs) in Java, showing that lambdas without type annotations are more readable than AICs.

Zheng *et al.* [53], investigated the extent to which the usage of lambdas in Java programs can be significantly compromised by collateral side effects, *e.g.,* memory leaks or efficiency issues. First of all, they pointed out that lambda deletion is increasing year by year, with lambdas built on top of customized functional interfaces having more chances of being removed. Furthermore, the introduction of performance degradation, together with poor readability of the code (*e.g.,* long body or complex logic) are the more predominant reasons why developers tend to remove lambdas and replace them with imperative constructs, method references, or anonymous objects for better extensibility.

While the aforementioned studies focus on lambdas/functional constructs usage and effects on comprehension, we focus on their fault-proneness, although the two phenomena (*i.e.,* low understandability and fix-inducing changes) may be interrelated.

From a different perspective, Gyori *et al.* [23], proposed an approach, `LambdaFicator` [16], integrated with the NetBeans IDE, aimed at converting imperative code to functional code using lambdas, and evaluated it on nine open-source Java projects. `LambdaFicator` automates two refactoring types, *i.e.,* converting (i) anonymous inner classes to lambda expressions, and (ii) for loops iterating over collections to functional operations relying on lambdas. Finally, Tsantalis *et al.* [47] showed how lambdas can be used to refactor code clones having behavioral differences.

The qualitative analysis we have performed shows that functional constructs are indeed subject to refactoring, although we observed several cases of functional construct removal, *e.g.,* translating lambdas to functions or comprehensions to loops.

B. Studies on Fix-Inducing Changes

Among other applications, the SZZ algorithm has been used to conceive just-in-time defect prediction approaches, *i.e.,* approaches aimed at leveraging features in fix-inducing changes (as opposed to features in other changes) to build defect prediction models able to indicate whether a change is likely to induce a fix. A seminal work in this area is the one by Kim *et al.* [31], which achieved 78% accuracy and 65% recall. Further work in this area has been proposed by several authors [26], [28], [29], [34]. While our aim is not to directly create defect prediction models, we at least use one major prediction of bugginess (*i.e.,* the change size) as a co-factor to be controlled.

A general study on fix-inducing constructs has been performed by Ferzund *et al.* [14]. They analyzed 8 projects relying on three different programming languages, *i.e.,* C, C++ and Java, and identified different kinds of fix-inducing constructs (*e.g.,* function calls, pointer references, or null constants) which do not include, however, a specific analysis on functional constructs as we did.

Ray *et al.* [42] have studied the correlation between fix-inducing changes and code naturalness. They found that fix-inducing changes are more likely to be introduced by unnatural code than by natural code. While our goal is different, we use an approach similar to what they used for identifying fix-inducing changes. Bavota *et al.* [5] and then Di Penta *et al.* [11] have studied, at different levels of granularity, the relationship between refactoring and fix-inducing changes. We use an analysis procedure very similar to what they have used, although we analyze different changes, *i.e.,* to function constructs instead of refactoring. Palomba *et al.* [36] leveraged the SZZ algorithm to perform a fine-grained analysis on the relationship between faults and code smells, determining whether fixes are actually induced when the source code was already smelly. They found that while there is a significant correlation between the presence of code smells and classes' fault proneness, the analysis performed was not able to identify a clear causation relationship. On the same line, Lenarduzzi *et al.* [32] studied whether SonarQube warnings induce fixes, finding that the "harmfulness" of many such warnings is low.

Finally, researchers also looked at developer-related factors. These include ownership (*i.e.,* single-author code appears to be riskier than code shared across multiple authors [40]), the (lack of) developers' communication [7], developers' experience [48], and sentiment [27]. It is possible that developer-related factors could interact with the use of specific programming constructs. However, a deeper investigation of this would be part of our future work agenda.

VIII. Conclusion

This paper discusses the results of an empirical study aimed at analyzing the extent to which changes to three types of Python functional constructs—*i.e.,* lambda functions, dictionary/list/set comprehensions, and map/reduce/filter functions—have higher odds to induce fixes than other changes.

To this aim, we analyzed the evolutionary history of 200 engineered, highly-forked open-source projects hosted on GitHub. Results of the study highlight that:

1) Changes affecting functional constructs have more than twice higher odds to induce fixes than other changes, while the odds increase is reduced to 15% if we only consider fixes impacting the same line(s) of the functional construct. Also, a change involving functional constructs has the same odds of inducing fixes as the addition/change of $\simeq 6$ lines of code.
2) When a new functional construct is added, this induces fixes with higher odds than when it is modified. However, this highly depends on the change size.
3) Lambdas have higher odds to induce fixes, followed by comprehensions, whereas the effect of map/reduce/filter functions is more limited.
4) A qualitative analysis of the results revealed different scenarios in which functional constructs have been fixed. These include not only cases in which the construct syntax and content (*e.g.,* a comprehension loop/condition) have been changed because of a previous misunderstanding or in general in the context of a bug fix, but also changes due to Python version compatibility. Moreover, we also found cases in which developers decide to move from one functional construct toward an alternative one, or to get rid of it, writing a non-functional alternative solution.

Future work aims at exploring the effect of other functional programming constructs (*e.g.,* immutability). This would require suitable program analysis tools. Also, we plan to investigate the effect of functional constructs on program understanding. Last, but not least, we would like to conceive developers' support to ease the usage of functional constructs, avoid mistakes, and automatically fix recurring bugs.

Acknowledgments

Di Penta and Zampetti acknowledge the Italian "PRIN 2021" project EMELIOT "Engineered MachinE Learning-intensive IoT systems".

References

[1] "GitHub REST API https://docs.github.com/en/rest (last access: 03/22/2022."

[2] C. V. Alexandru, J. J. Merchante, S. Panichella, S. Proksch, H. C. Gall, and G. Robles, "On the usage of pythonic idioms," in *Onward!* ACM, 2018, pp. 1–11.

[3] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research*. IBM, 2008, p. 23.

[4] D. Bates, M. Mächler, B. Bolker, and S. Walker, "Fitting linear mixed-effects models using lme4," *Journal of Statistical Software*, vol. 67, no. 1, pp. 1–48, 2015.

[5] G. Bavota, B. D. Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*, 2012, pp. 104–113.

[6] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[7] M. L. Bernardi, G. Canfora, G. A. Di Lucca, M. Di Penta, and D. Distante, "The relation between developers' communication and fix-inducing changes: An empirical study," *J. Syst. Softw.*, vol. 140, pp. 111–125, 2018. [Online]. Available: https://doi.org/10.1016/j.jss.2018.02.065

[8] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 121–130.

[9] J. Cohen, "A coefficient of agreement for nominal scales," *Educ Psychol Meas.*, 1960.

[10] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes," *IEEE Trans. Software Eng.*, vol. 43, no. 7, pp. 641–657, 2017.

[11] M. Di Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, 2020, pp. 556–567. [Online]. Available: https://doi.org/10.1145/3368089.3409695

[12] J. Falleri and H. Ham, "Gumtree pythonparser module https://github.com/GumTreeDiff/pythonparser (last access: 03/10/2022)."

[13] 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. [Online]. Available: https://doi.org/10.1145/2642937.2642982

[14] J. Ferzund, S. N. Ahsan, and F. Wotawa, "Bug-inducing language constructs," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, 2009, pp. 155–159. [Online]. Available: https://doi.org/10.1109/WCRE.2009.40

[15] R. A. Fisher, "On the interpretation of chi-square from contingency tables, and the calculation of p," *Journal of the Royal Statistical Society*, vol. 85, no. 1, pp. 87–94, 1922.

[16] L. Franklin, A. Gyori, J. Lahoda, and D. Dig, "LAMBDAFICATOR: from imperative to functional programming through automated refactoring," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 1287–1290.

[17] Github.com, "kivy/kivy commit: da2162a," https://github.com/kivy/kivy/commit/da2162a, 2011, (Last access: 30/03/2022).

[18] ——, "celery/django-celery commit: a45011," https://github.com/celery/django-celery/commit/a45011, 2016, (Last access: 30/03/2022).

[19] ——, "Hips/autograd commit: 9224718," https://github.com/HIPS/autograd/commit/9224718, 2016, (Last access: 30/03/2022).

[20] ——, "saltstack/salt commit: 1a9f03a," https://github.com/saltstack/salt/commit/1a9f03a, 2017, (Last access: 30/03/2022).

[21] ——, "saltstack/salt commit: 3df886d," https://github.com/saltstack/salt/commit/3df886d, 2017, (Last access: 30/03/2022).

[22] ——, "sosreport/sos commit: 4aae08a," https://github.com/sosreport/sos/commit/4aae08a, 2018, (Last access: 30/03/2022).

[23] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, pp. 543–553.

[24] S. Hanenberg and N. Mehlhorn, "Two n-of-1 self-trials on readability differences between anonymous inner classes (aics) and lambda expressions (les) on Java code snippets," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–39, 2022.

[25] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*, 2015, pp. 103–104.

[26] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, 2019, pp. 34–45. [Online]. Available: https://doi.org/10.1109/MSR.2019.00016

[27] S. F. Huq, A. Z. Sadiq, and K. Sakib, "Understanding the effect of developer sentiment on fix-inducing changes: An exploratory study on github pull requests," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, 2019, pp. 514–521.

[28] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 2072–2106, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9400-x

[29] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Software Eng.*, vol. 39, no. 6, pp. 757–773, 2013. [Online]. Available: https://doi.org/10.1109/TSE.2012.70

[30] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 351–360. [Online]. Available: https://doi.org/10.1145/1985793.1985842

[31] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Software Eng.*, vol. 34, no. 2, pp. 181–196, 2008.

[32] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?" in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, 2020, pp. 501–511. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054821

[33] W. Lucas, R. Bonifácio, E. D. Canedo, D. Marcílio, and F. Lima, "Does the introduction of lambda expressions improve the comprehension of Java programs?" in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, 2019, pp. 187–196.

[34] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," *IEEE Trans. Software Eng.*, vol. 44, no. 5, pp. 412–428, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2693980

[35] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[36] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9535-z

[37] Python.org, "Python 3.0 - what's new https://docs.python.org/3.0/whatsnew/3.0.html (last access: 03/10/2022)."

[38] ——, "Python AST module https://docs.python.org/3/library/ast.html (last access: 03/10/2022)."

[39] R Core Team, *R: A Language and Environment for Statistical Computing*, 2012, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[40] F. Rahman and P. T. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 491–500.

[41] A. E. Rao and S. Chimalakonda, "An exploratory study towards understanding lambda expressions in Python," in *Proceedings of the Evaluation and Assessment in Software Engineering*, 2020, pp. 318–323.

[42] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. T. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 428–439. [Online]. Available: https://doi.org/10.1145/2884781.2884848

[43] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*, 2005.

[44] D. Spadini, M. F. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 908–911. [Online]. Available: https://doi.org/10.1145/3236024.3264598

[45] Switowski.com, "For loop vs. list comprehension https://switowski.com/blog/for-loop-vs-list-comprehension (last access: 03/10/2022)."

[46] H. Tanaka, S. Matsumoto, and S. Kusumoto, "A study on the current status of functional idioms in Java," *IEICE Transactions on Information and Systems*, vol. 102, no. 12, pp. 2414–2422, 2019.

[47] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 60–70.

[48] M. Tufano, G. Bavota, D. Poshyvanyk, M. Di Penta, R. Oliveto, and A. De Lucia, "An empirical study on developer-related factors characterizing fix-inducing commits," *J. Softw. Evol. Process.*, vol. 29, no. 1, 2017. [Online]. Available: https://doi.org/10.1002/smr.1797

[49] 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 Trans. Software Eng.*, vol. 43, no. 11, pp. 1063–1088, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2017.2653105

[50] N. Vandeput, "List comprehensions vs. for loops: It is not what you think https://towardsdatascience.com/list-comprehensions-vs-for-loops-it-is-not-what-you-think-34071d4d8207 (last access: 03/10/2022)."

[51] Wikipedia, "List of programming languages by type https://en.wikipedia.org/wiki/List_of_programming_languages_by_type\#Impure (last access: 03/10/2022)."

[52] F. Zampetti, F. Bellas, C. Zid, G. Antoniol, and M. Di Penta, "Dataset of the paper "An Empirical Study on the Fault-Inducing Effect of Functional Constructs in Python"," Mar. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6396698

[53] M. Zheng, J. Yang, M. Wen, H. Zhu, Y. Liu, and H. Jin, "Why do developers remove lambda expressions in Java?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 67–78.