POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

The Effect of Pythonic Constructs on the Software Development Process

CYRINE ZID
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Avril 2024

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

The Effect of Pythonic Constructs on the Software Development Process

présentée par **Cyrine ZID**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Giovanni BELTRAME, président
Foutse KHOMH, membre et directeur de recherche
Giuliano ANTONIOL, membre et codirecteur de recherche
Heng LI, membre
Sonia HAIDUC, membre externe

DEDICATION

To my parents
To my sister
To my aunts
To my family
To my friends
For their endless love, support, and encouragement

ACKNOWLEDGEMENTS

I want to express my deepest gratitude to my mentor *Professor Giuliano Antoniol*, the one who has been my source of inspiration and who has stood by me throughout these years. Thank you immensely for your devoted time, unwavering understanding, and the invaluable advice you have given me at every step of this journey. Without your invaluable support, none of this would be possible. Your presence has been a guiding light in dark moments, and your constant encouragement has given me the strength to persevere. Thank you for everything you've done, for your generosity, and for your dedication to my growth. It is with profound gratitude that I acknowledge the significant impact you have had on my academic and personal life. Thank you again for everything, and I am honored to have had the opportunity to work alongside you.

I would like to express my gratitude to my supervisor, Professor *Foutse Khomh*, for his unwavering guidance and support throughout my academic journey. His exceptional leadership has been a constant source of inspiration for me. I also want to thanks for the numerous opportunities you have provided. Your generosity is truly appreciated, and I am grateful for the chance to learn and grow under your mentorship.

My sincere thanks also go to Professor *Massimiliano Di Penta* for his invaluable assistance. This work would not have been possible without him and his team in Italy. Thank you for welcoming me into your lab; working alongside you has been one of the most fulfilling and enriching experiences I've had.

I am thankful to all my co-authors; without you, this work would have never taken shape. With you, I have come to understand the importance and value of a team, and that if we succeed one day, it is thanks to the people around us. This work is not just mine but ours. Without you, I would not be here.

I would like also to thank my thesis committee, Professor Giovanni Beltrame, Professor Heng Li, Professor Soumaya Yacout, and Professor Sonia Haiduc for accepting my invitation to be jury members.

Last but not least, I'm so grateful and thankful to my family for their unwavering support throughout my life. Without you, I would have never found the strength or courage to accomplish all that I have. Your encouragement has been my driving force, and I am profoundly thankful for the love and support you have showered upon me. Ultimately, every success I've achieved is also yours, and I am immensely grateful to you.

RÉSUMÉ

Dans un paysage en constante évolution de la technologie de l'information, le langage de programmation Python a émergé comme un acteur central, façonnant divers secteurs tels que l'apprentissage automatique et la science des données. La montée rapide de la popularité de Python repose sur ses qualités intrinsèques, une combinaison de polyvalence, de lisibilité et de structure. En tant que langage de choix pour de nombreux développeurs, Python joue un rôle crucial dans la résolution de défis complexes et la création de solutions innovantes.

Cependant, cette utilisation répandue de Python soulève des questions critiques sur son efficacité, en particulier en ce qui concerne les caractéristiques fonctionnelles de Python. Les fonctions lambda, les compréhensions de liste et d'autres mécanismes fonctionnels offrent des avantages indéniables, tels que la clarté du code et une approche de programmation élégante. Néanmoins, l'adoption de ces constructions n'est pas sans conséquence, et des nuances liées à leur compréhensibilité, à leur impact sur les performances et à leur tendance à induire plus de corrections méritent une analyse approfondie.

C'est dans ce contexte que cette thèse se situe. Organisé en huit chapitres, ce travail comprend une introduction et une revue de littérature contextualisant cette thèse dans le domaine plus large de la compréhension des programmes, de l'évolution des logiciels et de la programmation fonctionnelle. Il explore également les constructions spécifiques à Python et les études connexes sur la compréhension des programmes, les modifications induisant des corrections et les performances de Python. Le chapitre sur les connaissances préalables fournit des détails utiles pour rendre ce travail autonome et facile à lire.

Le chapitre 3 offre un bref aperçu des constructions fonctionnelles de Python étudiées et de leur complexité, tandis que le chapitre 4 explore la compréhensibilité des constructions fonctionnelles de Python grâce à une expérience contrôlée, révélant des informations sur la facilité de modification et la lisibilité perçue par rapport aux alternatives procédurales. L'étude indique que les lambdas sont plus simples à modifier que leur équivalent non fonctionnel, mais les constructions de compréhension présentent des défis en termes de compréhensibilité.

Le chapitre 5 examine dans quelle mesure les modifications des constructions fonctionnelles induisent des corrections. En analysant l'historique des modifications dans 200 projets Python open source, l'étude fournit des preuves que les modifications des constructions fonctionnelles ont une probabilité plus élevée d'induire des corrections, avec des variations entre différentes constructions.

Dans le chapitre 6, les implications de performance des compréhensions de listes de Python sont étudiées, comparant les temps d'exécution avec une programmation impérative équivalente. Les résultats montrent des gains de performance variables en fonction de la nature du code et des objets manipulés, soulignant l'importance des considérations contextuelles.

Le chapitre 7 examine l'impact des différents mécanismes de passage d'arguments de Python par rapport aux modifications induisant des corrections. En analysant l'historique d'évolution de 200 projets Python et un grand nombre de fonctions et d'appels, l'étude fournit des preuves que le passage d'arguments basé sur des mots-clés est moins susceptible de présenter des défauts que le passage d'arguments positionnels.

Le chapitre 8 conclut la thèse et explore l'évolution des définitions de fonctions et des appels de fonctions, mettant en lumière des opportunités d'amélioration dans l'éducation, la qualité du code et le developpement d'outils de support pour le développement en Python. Cette exploration approfondie permettra nous l'espérons aux développeurs de mieux comprendre les choix et considérations liés aux constructions fonctionnelles dans la programmation moderne en Python.

Chaque chapitre représente une facette de cette exploration approfondie, fournissant des réponses informées aux questions soulevées par l'utilisation de constructions fonctionnelles de Python. En synthétisant les résultats dans la conclusion, cette thèse vise à enrichir la pratique du développement Python en identifiant des opportunités d'amélioration dans l'éducation, la qualité du code ou le support d'outils. Ainsi, elle contribue à l'évolution continue du rôle de Python dans le paysage contemporain de la technologie de l'information.

ABSTRACT

In the ever-evolving landscape of information technology, Python has firmly established itself as a central programming language, wielding significant influence in areas such as machine learning and data science. Its widespread adoption is attributed to its versatility, readability, and structural qualities. As the language of choice for many developers, Python plays a central role in solving complex challenges and creating innovative solutions. However, questions persist regarding Python's effectiveness, particularly concerning its functional features, such as lambda functions and list comprehensions.

The focal points include Python functional constructs such as list comprehensions, dictionary comprehensions, set comprehensions, lambdas, and higher-order functions. Within this exploration, we introduce a research hypothesis: To what extent do Pythonic functional constructs impact code understandability and fault proneness, and how do these aspects vary across the different constructs under study? The research includes a detailed exploration of Pythonic concepts, considering factors such as bug-proneness and performance impact.

By focusing on the comprehensibility of Python's functional constructs through a controlled experiment we report evidence that functional constructs are not always the best choice. Indeed, our results provide insights into ease of modification and perceived readability compared to procedural alternatives.

Following that, this thesis investigates the effect of complexity on Python's functional constructs. The reported empirical study analyzing the revision history of 200 open-source Python projects reveals that modifications to functional constructs often lead to bug fixes and that complexity plays a role.

Constructs such as list comprehensions, which are more difficult to understand, are sometimes promulgated as a performance solution. To validate such a hypothesis we investigated the performance of Python list comprehension, comparing execution times with equivalent imperative programming counterparts in real-world applications. In contrast with previous findings, we found that in real projects variable performance gains are observed, underscoring the need for contextual considerations.

Finally, the research examines the impact of different Python argument-passing mechanisms on changes inducing bug fixes, providing evidence that keyword-based argument passing is less prone to defects than positional argument passing.

Overall, reported evidence shows that functional constructs are far from a silver bullet and

that the adoption of functional constructs, as well as uncommon parameter-passing mechanisms, should be carefully pondered. The synthesis of these results leads to a comprehensive understanding of the evolution of function definitions and function calls in Python. The findings highlight opportunities for improvement in education, code quality, and software support within the Python development landscape.

In conclusion, this in-depth exploration contributes to developers' understanding of the choices and considerations related to functional constructs in modern Python programming. Each thesis chapter addresses specific facets of this research, providing informed answers to questions raised by the use of functional constructs. By identifying opportunities for improvement in education, code quality, and software support, this thesis aims to enhance the practice of Python development, contributing to the ongoing evolution of Python's role in contemporary information technology.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

AST	Abstract Syntax Trees
GLMR	Generalized Linear Mixed Models
MRF	Map/Filter/Reduce
OR	Odds Ratio

CHAPTER 1 INTRODUCTION

1.1 Motivation

The landscape of software development comprises a diverse array of programming languages, each offering distinct paradigms for designing, building, and maintaining software systems. Among these, Python[1] stands out as a predominant choice and has emerged as a natural language selection across various domains and applications. Its meteoric rise has positioned it as the "go-to" language for developing libraries and applications in artificial intelligence. Python's popularity stems from its rich ecosystem, flexibility, and versatility as a multi-paradigm language. This thesis delves into understanding the implications of employing Pythonic constructs such as list, dictionary, or set comprehension in developing and evolving software applications. Python, as a multi-paradigm language, provides constructs that support different programming approaches, mixing elements of functional programming with Python's intrinsic imperative and object-oriented paradigms. A comprehensive review of existing research reveals contrasting conclusions regarding the usage of Pythonic constructs. Some studies have highlighted an increased tendency towards defects induced by these functional constructs compared to procedural alternatives [1]. Conversely, other works suggest performance improvements associated with the adoption of certain functional constructs, particularly in terms of code execution speed [2]. Despite these findings, questions linger regarding the true understanding and impact of these constructs in real-world software development contexts. Recent research has also raised questions about the practical benefits of certain transformations, such as converting "for" loops into list comprehensions [2]. This thesis aims to extend beyond established observations to thoroughly explore the impact of Pythonic constructs on software development processes. It aims to study the effect of Pythonic constructs on performance, code comprehensibility, and defect proneness within various real-world software development contexts. To achieve this objective, this research adopts a multidimensional approach, combining controlled experiments, in-depth empirical analyses of real-world Python projects, and specific case studies. The results of these diverse studies are presented, discussed, and contextualized in the subsequent chapters of this thesis.

[1] `https://www.tiobe.com/tiobe-index/`

1.2 Research question

In this thesis, we endeavor to address the following high-level research question:

> To what extent do Pythonic functional constructs impact code understandability, fault proneness, and performance, and how do these aspects vary across the different constructs under study?

We propose a structured approach for our multifaceted exploration, combining controlled experiments, empirical analyses of real Python projects, and specific case studies to unravel the multifaceted impact of Pythonic constructs on the software development process. Hence, in the following, we refine our high-level research question by providing more details, incorporating four sub-questions :

1. **RQa:** *How does the presence of functional constructs impact the difficulty of code modification and source code understandability?* To address this question, Chapter 4 presents the results of a controlled experiment involving 209 developers. The primary objective of the experiment was to assess the understandability of key Pythonic functional constructs—namely lambdas, comprehensions, and map/reduce/filter functions—in comparison to their procedural counterparts.

2. **RQb:** *To what extent do modifications involving functional constructs influence the frequency of fixes in the source code?* To answer this question, We analyzed the temporal evolution of 200 open-source Python projects, encompassing approximately 630,000 commits. The primary objective is to comprehend the relationship between alterations involving functional constructs—specifically lambdas, comprehensions, and map/reduce/filter functions—and the likelihood of inducing fixes. Moreover, the study explores how the complexity of these constructs and the survival time of resulting bugs contribute to the overall bug-proneness, details are presented in Chapter 5.

3. **RQc:** *Does the transformation of "for" "loops" into list comprehensions significantly contribute to performance improvements in real-world projects?* To address the question, Chapter 7 presents the findings of an empirical investigation aimed at understanding the nuanced impact of list comprehensions on program execution time, especially when compared with semantically equivalent procedural code.

The study can be conceptually divided into two parts: the first part is in vitro, and the second part is in corpore vili, implemented in the real project code. In vitro experiments involve replacing small procedural code snippets, often enclosed in a single loop, with list comprehensions. The results align with previous findings, demonstrating that list comprehensions exhibit notable advantages in execution speed, particularly when subjected to iterative amplification. However, when the actual code is transformed and executed, the situation becomes much more complex, and the transformed code may or may not run faster.

4. **RQd:** *In what ways do changes related to parameter-passing in function definitions and call sites affect fault proneness?* To answer the question of how changes related to parameter-passing in function definitions and call sites affect fault proneness, the study presented in Chapter 6 focuses on two distinct locations: the function definition level and the call sites.

   The research involved a curated set of 200 Python projects, carefully selected from an initial list of "engineered" projects [3]. The selection process included filtering based on criteria such as forking, activity level, and commit count. This resulted in a final set ranked by fork count. The change history of these projects was thoroughly analyzed using PyDriller [4] and the Python AST library [5]. To identify fix-inducing changes, the study utilized an implementation of the SZZ algorithm [6]. In total, the analysis covered 633,803 commits affecting Python files, encompassing 6,159,194 changes, 2,804,230 function definitions, and 12,579,843 call sites.

1.3 Contributions

This research contributes to the field of software development by providing in-depth insights into the impact of Pythonic constructs on various dimensions of the development process. The key contributions of this study include:

Contribution 1: *Comprehensive Analysis of Code Modification Difficulty.*

This research investigates the impact of functional constructs, including lambdas, comprehensions, and map/reduce/filter (MRF), on code modification difficulty and source code understandability in Python. The study addresses three research questions:

- **RQa$_1$:** *How difficult is it to modify code snippets relying on functional constructs compared to their procedural alternatives?*This question assesses code understandability by

comparing functional constructs and procedural alternatives in change tasks. We utilize a mixed-effect logistic model, analyzing factors like construct type, complexity, prior approvals, usage frequency, and student status.

- **RQa$_2$:** *To what extent do functional constructs influence the perceived source code understandability?* Investigates perceived code snippet understandability using functional constructs versus procedural alternatives. Developers diverging stacked bar charts and ordinal logistic regression, exploring relationships with usage frequency and complexity for each functional construct.

- **RQa$_3$:** *Why do developers use functional constructs, and in which scenarios?* This question quantitatively explores reasons for using functional constructs and their perceived disadvantages, providing insights into developers' motivations and concerns. This research contributes to understanding developers' experiences with Pythonic constructs.

Contribution 2: *Empirical Understanding of Fault-Inducing Changes.*

The study aims to explore the fault-proneness of modifications involving functional constructs compared to procedural alternatives, shedding light on scenarios within software development where these constructs may introduce defects. The research questions guiding the investigation are as follows:

- **RQb$_1$:** *To what extent do changes involving functional constructs induce more fixes than other changes?* Are changes involving functional constructs more likely to induce fixes than other changes? This question is addressed by comparing the proportion of fix-inducing changes involving functional constructs against those not involving functional constructs. The analysis employs Fisher's exact test and Odds Ratio (ORs) effect size. Additionally, a mixed-effect generalized linear model is constructed, considering factors such as changed churn size.

- **RQb$_2$:** *How do the odds to induce fixes vary between additions of functional constructs and their modifications?* Do changes adding functional constructs have a higher likelihood of inducing fixes than changes simply modifying them? This question examines whether functional constructs tend to be "born fault-prone" and whether changes made to them still induce further bugs. A mixed-effect model is employed to determine correlations with fix-inducing changes, considering additions, modifications, and their interaction with changed churn size.

- **RQb$_3$:** *How does the fault-proneness vary among different types of functional constructs?* Do developers experience different levels of difficulty when using different types of functional constructs—lambdas, comprehensions, and map/reduce/filter functions? This research question involves analyzing fault-proneness variation among these constructs. A mixed-effect model is utilized, considering whether a change is fix-inducing as the dependent variable and whether the changed churn involves any of the three types of functional constructs as independent variables.

- **RQb$_4$:** *Is there a relation between the cyclomatic complexity of functional constructs and their fault-proneness?* Does the fault-proneness of different functional constructs vary concerning their complexity? This question explores the variability in fault-proneness across various complexity levels for each functional construct. Separate mixed-effect models are employed for each functional construct, and Fisher's exact test is used to compare the fault-proneness of constructs with low and high complexity.

- **RQb$_5$:** *What is the survival time of bugs involving different functional constructs?* How long do bugs involving functional constructs remain in the code? This question is addressed through survival analysis using the Cox proportional hazard model on fixes involving functional constructs. Kaplan-Meier curves and forest plots visualize hazard ratios and confidence intervals of covariate hazard ratios. Median survival times are extracted using appropriate *R* packages.

Together, these research questions aim to provide comprehensive insights into the fault-proneness and survivability of functional constructs in Python projects, comparing them to procedural alternatives and considering various factors influencing these phenomena.

Contribution 3: *Real-World Performance Evaluation.*

The study contributes practical knowledge about the actual performance implications of transforming "for" "loops" into "list comprehensions" in real-world Python projects. Focusing on two research questions:

- **RQc$_1$:** *Is list comprehension code running faster than its equivalent for code?* aims to establish a theoretical baseline by analyzing the performance of synthetic yet realistic code, comparing the execution speed of for loop code with semantically equivalent list comprehensions.

- **RQc$_2$:** *Does transforming for loops into list comprehensions result in faster real-world projects?* investigates the real-world impact of this transformation, examining the number of for loops worth transforming, the success rate of transformation, and whether the performance is significantly affected.

The data analysis utilizes a performance index, denoted as ρ, representing the ratio of execution time before and after the transformation. This index is applied to various projects, and the distribution of ρ is visually depicted using box plots to highlight performance differences. Tukey five-number summaries are reported for each project, providing additional insights. To identify statistically significant differences in performance, the study employs the Wilcoxon [7] signed-rank test, a non-parametric, paired test suitable for randomized experiments. Complementing these statistical tests, the analysis incorporates Cliff's delta effect size measure [8], offering a comprehensive evaluation of the overall impact of transforming for loops into list comprehensions.

Contribution 4: *Insights into fix-induced changes due to Python parameter passing mechanisms.*

Through a comprehensive methodology that incorporates quantitative and qualitative analyses, our research investigates the impact of changes related to parameter-passing in both function definitions and call sites on fix-inducing changes in Python projects. Four research questions are addressed:

- **RQd$_1$:** *To what extent do changes to parameter-passing in function definitions induce more fixes than others?* Examines whether changes involving different parameter-passing mechanisms in function signatures have higher odds of inducing fixes compared to other changes. Fisher's exact test and Odds Ratio (OR) effect size are employed for the analysis.

- **RQd$_2$:** *How does the fault-inducing proneness in function definitions vary among different types of parameter passing mechanisms?* Follows up on RQ$_1$, exploring whether specific parameter-passing mechanisms are more likely to induce fixes. A mixed-effect generalized linear model is developed, considering confounding factors such as churn size, file size, and complexity.

- **RQd$_3$:** *To what extent do changes to function argument-passing in call sites induce more fixes than others?* Explores whether changes to arguments in function call sites

induce more fixes than other changes. Fisher's exact test, similar to RQ_1, is used for comparison.

- **RQd_4:** *How does the fault-inducing proneness in call sites vary among different argument passing mechanisms?* Deepens the analysis of RQ_3, investigating specific argument passing mechanisms that induce more fixes. A mixed-effect model is applied, considering factors such as the number of arguments and features of call site argument construction.

Overall, this study provides a holistic approach to understanding the ramifications of different parameter-passing mechanisms on code maintainability, offering both quantitative and qualitative perspectives in the analysis of Python projects.

The contributions described above collectively advance our understanding of the nuanced effects of Pythonic constructs on software development practices and we hope will help guide developers, researchers, and educators in navigating the use of these constructs.

1.4 Organization of the Thesis

This thesis is organized into eight chapters to provide a cohesive and comprehensive exploration of the impact of pythonic constructs on code quality. Each chapter contributes to an understanding of the topics under investigation.

Chapter 2 is a literature review that contextualizes the research within the broader landscape of functional programming and Pythonic constructs. By examining prior studies, this chapter sheds light on existing gaps in the literature.

Building upon this groundwork, Chapter 3 explores fundamental background knowledge to help the reader understand the contribution and approaches presented in the following chapters.

Chapter 4, presents a first empirical study, where we focus on the understandability of Pythonic functional constructs. This empirical study is followed by Chapter 5, where an in-depth exploration is undertaken to understand the fault-inducing effects of functional constructs in Python. Both chapters present empirical findings that shed light on the practical implications of utilizing these constructs in real-world scenarios.

In Chapter 6, the thesis adopts a performance-oriented perspective, scrutinizing the comparative aspects of list comprehensions versus "for" "loops" in real Python projects. This

analysis seeks to clarify the practical considerations associated with choosing one construct over the other.

Chapter 7 extends the empirical exploration, shifting focus to the relationship between various Python argument-passing mechanisms and code fixes. This investigation contributes insights into the intricacies of code maintenance and the impact of different parameter-passing strategies.

Finally, Chapter 8 formulates conclusions drawn from this dissertation, discusses the limitations inherent to the work, and outlines potential avenues for future research.

CHAPTER 2 LITERATURE REVIEW

This chapter presents a review of existing research in the domains of functional programming, Pythonic constructs, and related studies on program comprehension, fix-inducing changes, and Python's performance. By synthesizing findings from various research strands, this chapter aims to contextualize the current work within the broader field of software engineering and programming languages.

2.1 Functional Programming and Pythonic Constructs

A limited number of studies exist on Python's functional constructs, in recent years research mostly focused on Java functional programming. The subsequent paragraphs discuss various studies related to Java before transitioning to an examination of Python-specific functional constructs.

Python supports multiple programming paradigms and in open-source projects is mostly adopted for its object-oriented nature (Dyer*et al.* [9]). The introduction of lambdas in Java version 8 sparked research into how developers leverage lambda expressions and functional operations for parallelism, conciseness, and readability.

Tanaka*et al.* [10] identified lambda as the most accepted functional idiom in Java, while Rao and Chimalakonda [11] reported the prevalence of lambda expressions in Python projects, commonly used to replace functions, anonymous classes, and iterators. Alexandru*et al.* [12] conducted an interview-based study on the usage of "Pythonic" idioms, revealing varied perceptions among developers based on experience levels, emphasizing improved understandability and performance.

Lucas*et al.* [13] focused on the impact of lambdas on code readability, finding that while quantitative results did not consistently show improvements, developers perceived enhanced program comprehension. Hanenberg and Mehlhorn [14] compared the readability of lambdas to anonymous inner classes in Java, suggesting that lambdas without type annotations are generally more readable.

Zheng *et al.* [15] delved into potential issues with lambdas in Java, revealing that lambdas built on customized functional interfaces are more likely to be removed, with performance degradation and diminished code readability emerging as predominant reasons for developers opting to remove or replace lambdas. Gyori *et al.* [16] proposed LambdaFicator, a tool integrated with the NetBeans IDE, to automate the conversion of imperative code to func-

tional code using lambdas. Their evaluation indicated that lambdas do not entirely replace anonymous inner classes, but they exhibit a high level of applicability (55%). Additionally, Tsantalis *et al.* [17] demonstrated how lambdas can be used to refactor code clones with behavioral differences, with 75% of refactored clone pairs requiring fewer than two lambdas.

While this thesis generally confirms the perceived higher understandability of procedural alternatives over functional constructs, the outcomes of change tasks varied depending on the construct and its complexity.

Following the widespread adoption of lambdas in Java and their subsequent integration into Python projects, researchers explored understanding developers' reliance on lambda expressions and functional operations (e.g., map and filter) for enabling parallelism, reducing code size, and enhancing readability.

As shown in Chapter 4 and the related publication [18], which involved over 200 participants to investigate the understandability of functional constructs, including lambdas, comprehensions, and map/reduce/filter functions. It appears that procedural code is generally easier to modify than complex lambdas, and comprehensions are more manageable than their procedural counterparts. Overall, functional constructs were perceived as more challenging to understand than their procedural equivalents, with varying levels of difficulty based on experience and construct usage frequency.

In a nutshell, the aforementioned studies focus on the usage and effects of lambdas and other functional constructs on comprehension, our study delves into their fault-proneness, acknowledging the potential interdependence of low understandability and fix-inducing changes.

2.2 Code Comprehension Studies with Humans

Developers spend a substantial portion of their time comprehending source code while conducting both development and maintenance tasks [19]. This has led several researchers to further investigate developers' behavior during program comprehension tasks. For instance, Hofmeister *et al.* [20] studied the effect of different identifier naming styles on program comprehension during bug localization. Their results show that source code is more complex to comprehend when using only letters and abbreviations as identifier names. Johnson *et al.* [21] conducted a controlled experiment involving 275 participants aimed at assessing the readability of "nesting" and "looping", highlighting that decreasing the level of nesting will decrease the developer's time in properly understanding a code snippet. On the same line, McCauley *et al.* [22] conducted a study involving students to compare their ability to properly comprehend recursive and iterative programs.

Yu *et al.* [23] looked at how developers comprehend test code. By experimenting with 44 developers, they found that the level of experience with automated tests is an influential factor in properly understanding and extending an existing test suite.

Some studies leveraged eye tracking to better understand developers' behavior during program comprehension tasks. For instance, Jbara and Feitelson [24] conducted an experiment to measure the time and effort spent by developers reading and understanding regular code, *i.e.,* repetitions of the same basic pattern. Peitek *et al.* [25] used eye-tracking to record the eye movements of 31 developers with different levels of expertise (novices and intermediate developers), to confirm or confute existing knowledge about the effects of linearity of source code on reading order. Bauer *et al.* [26] used an eye-tracker device to shed light on the impact of indentation styles on program comprehension. Specifically, they asked 22 participants to provide the output of code snippets with different indentation styles written in Java. Their results show that indentation does not significantly influence code comprehension.

Other studies leverage an even more complex setting using medical devices. For instance, Peitek *et al.* [27] conducted a study with functional magnetic resonance imaging (fMRI), involving 19 participants to observe the comprehension level of code snippets with different complexity levels. Fakhoury *et al.* [28] used a new methodology consisting of functional Near Infrared Spectroscopy (fNIRS) and eye tracking devices to properly measure program comprehension, *i.e.,* the impact of lexical, structural, and readability issues on developers' cognitive load during bug localization tasks.

We share with all the aforementioned work the goal to study the impact of certain code constructs on program comprehension. To the best of our knowledge, ours is the first study with human participants concerning Pythonic functional constructs.

2.3 Studies on Atoms of Confusion

This is also a relevant issue for Python though it was mostly investigated concerning other programming languages. In the context of this thesis, we extend this understanding to Python's functional constructs, specifically investigating potential "atoms of confusion" related to argument-passing in the Python language.

Program comprehension has been recognized as a major concern in software development, and since the early 90s, researchers have addressed this problem from a different perspective.

Focusing on source code comprehension, previous work has recognized how the code organization, combined with specific programming features, influences program comprehension [29–31]. More recently, Gopstein *et al.* [32] identified, through a study with human

participants, 15 atoms of confusion in source code, related to different features such as type conversion, conditional operators, omitted curly braces, or precedence. In a follow-up study on C/C++ programs, it has been found that such atoms of confusion tend to be prevalent, more commented, and more buggy than other code [33], while a study by Langhout and Aniche [34] identified, through an experiment with 132 human participants, which are the most prevalent atoms of confusion in the case of Java source code.

Through a qualitative study, it has been pointed out that developers' confusion may originate from multiple sources, and this has to be considered in program comprehension studies [35]. Finally, de Oliveira *et al.* [36] studied, using eye-tracking how the presence of atoms of confusion increases the amount of time developers "fixate" source code.

Despite the effort put into defining source code atoms of confusion, to the best of our knowledge, ours is the first study related to possible "atoms of confusion" related to argument-passing, and, in our case specific to the Python language.

Langhout and Aniche have identified 14 "atoms of confusion" for Java programs [34], for example, related to postfix increment/decrement, absence of indentation, conditional operator, use of arithmetic as logic, or logic as control flow. In an experiment with 132 students, they showed how seven of such atoms of confusion represent a real comprehension issue, with indentation or curly brace removals being the most critical one. Their study, also because it is Java-related, did not consider parameter-related atoms of confusion.

Bad practices and poor design choices have also been codified into catalogs of *anti-pattern* [37–40] Spaghetti code, too long list of parameters and misleading variable method names are known to hinder program comprehension, ultimately increasing effort and defects.

2.4 Fix-Inducing Changes

As mentioned earlier, our interest extends beyond the comprehension of Pythonic constructs, encompassing a comprehensive exploration of fix-inducing changes and their correlation with Pythonic constructs. In the subsequent sections, we discuss existing studies that aim to correlate various types of software changes, as well as specific program properties, to those changes that induce fixes.

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 [41], which achieved 78% accuracy and 65% recall. Further work in this area has been proposed

by several authors [42–45]. While our aim is not to directly create defect prediction models, we at least use one major prediction of bugginess (*i.e.,* the changed churn size) as a co-factor to be controlled.

Ferzund *et al.* [46] 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, use of null, pointers) which do not include, however, a specific analysis on functional constructs as we did.

Ray *et al.* [47] 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.* [48] and then Di Penta *et al.* [49] 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 functional constructs instead of refactoring.

Palomba *et al.* [50] leveraged the SZZ algorithm to perform a fine-grained analysis on the relationship between faults and code smells, determining whether fixes are induced when the source code is 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, [51] 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 [52]), the (lack of) developers' communication [53], developers' experience [54], and sentiment [55]. 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.

2.5 Studies on the Impact of Atoms Of Confusion on Program Comprehension

Researchers in [19] highlighted that developers spend a substantial portion of their time comprehending source code while conducting both development and maintenance tasks. Based on this, researchers have looked at "atoms of confusion" in source code, *i.e.,* code elements that tend to negatively impact program comprehension. Specifically, [56] conducted a study involving 222 professional programmers to measure how quickly and accurately they can interpret code snippets having similar functionality but implemented with different source

code structures. Their results highlight that (i) code snippets with `for` are considered much harder to understand compared to snippets containing `if` statements, (ii) snippets with negated predicates are harder to understand, and (iii) `for` loops counting down are harder to understand than loops that count up. [32] conducted a study involving humans to identify 15 atoms of confusion in source code. To show the real-world relevance of these atoms of confusion, in a follow-up study [57] they found that atoms of confusion occur in practice once per 23 lines. On the same line, [58] looked at the relevance of misunderstanding patterns by performing repository mining and a survey with developers. Their results highlight that all but one atom occurs in the analyzed projects, as well as, according to developers, only some atoms of confusion may cause misunderstandings. Furthermore, [59] provided insights into misinterpretations caused by code that has side-effects, highlighting that *pre-increment decrement, post-increment decrement* and *logic as control flow* atoms make use of expressions with side-effects.

Langhout and Aniche [34], conducted a controlled experiment to identify the most prevalent atoms of confusion in Java source code. Their results show that developers are 2.7 up to 56 times more likely to make mistakes in snippets affected by atoms of confusion. Furthermore, when faced with both versions of the code snippets, developers perceived the version affected by the atom as more confusing and less readable. Oliveira *et al.* [36] conducted an eye-tracking study to examine how the presence of atoms of confusion impacts the time developers spend fixating on source code. Specifically, by measuring the tasks' completion time, the accuracy of the answers, and by analyzing the distribution of visual attention, the authors show an increase of 43.02% in time and 36.8% in gaze transitions in code snippets affected by atoms. Furthermore, their results confirm that the regions that receive most of the eye attention are those with atoms of confusion.

In a more recent study Da Costa *et al.* [60] conducted a controlled experiment with 32 novice Python programmers measuring time, number of attempts, and visual effort with eye tracking through fixation duration, fixation count, and regressions count. Their results highlight that the clarified version of the code with *Operator Precedence* reduced the time spent in the region that contains the atom and the number of answer attempts.

From a different perspective, [61] looked at atoms of confusion on code reviews. By manually analyzing code review comments, results did not show any relationship between atoms of confusion and the presence of confusion comments in code reviews. However, atoms of confusion are mostly not removed in pull requests.

2.6 Testing and Argument Passing

Integration testing [62] plays a vital role in software development. Our study, while static, aligns with integration testing principles. Mutant operators [63] addressing parameter passing ordering resonate with our findings. Our work complements testing efforts, offering insights for prioritizing integration testing. While our study focuses on argument-passing from a static perspective, we acknowledge its significance as a key element in integration testing [62]. Jin and Offutt's coupling-based criteria for integration testing [64], advocate the exercise of call sites, paths from variable definitions in the caller (callee), and usage in callee (caller), similarity can be found with our study observations. Notably, our findings indicate that parameter-related issues are not confined to function signatures but also extend to the callee's body. Furthermore, mutant operators, as defined by Kim *et al.* [63], emphasize the importance of addressing mistakes in parameter passing ordering as representative mutants.

Our study reveals that argument passing, particularly in a keyword-based way that doesn't rely on ordering, can mitigate such errors. We posit that studies like ours offer valuable insights for prioritizing areas in need of enhanced integration testing coverage.

2.7 Python Performance and Usage

The Python language dominates various areas of software development, especially in machine learning, data science, and research and development applications. The widespread popularity of Python can be attributed to several factors, as highlighted by Hamidi et al. [65]. In their study, Hamidi et al. focused on six programming languages, namely C#, C++, Java, R, Matlab, and Python. Their results reveal that Python's popularity stems from the availability of frameworks like TensorFlow, which simplify the learning and development of machine learning applications. They also emphasized the robust support provided by the Python community, significantly aiding newcomers in their development endeavors.

Similarly, Nagpal et al. [66] enumerate the reasons why Python is an apt programming language for data analysis and scientific/technical applications. Firstly, Python's popularity in these domains arises from its simpler syntax compared to other languages, making coding in Python akin to writing pseudo-code. Python requires fewer lines of code for various tasks, enhancing code readability. Moreover, Python's independence from the operating system contributes to its exceptional portability and flexibility [66]. However, it's noteworthy that none of these reasons explicitly endorse Python for its performance.

In their examination of Python's performance, Nagpal et al. [66] consider two main aspects. Firstly, speed is evaluated concerning compiled languages. Although some performance tests

suggest that Python can be faster than certain compiled languages, general benchmarks indicate that compiled languages like C/C++ and more modern ones like Go tend to outperform Python. The second aspect involves memory consumption. Python's flexibility in data types can lead to substantial memory usage, making it less favorable for memory-intensive tasks. In summary, developers face a trade-off between Python's simplicity and ease of development and the absolute performance offered by compiled languages.

To sum up, while recognizing certain commonalities, it is important to emphasize that a considerable part of this thesis is dedicated to investigating novel and rarely explored research questions.

CHAPTER 3 PRIOR KNOWLEDGE

As we delve into the intricacies of Python's functional constructs and argument-passing mechanisms, it is important to establish a good understanding of Python's key concepts and our collected metrics. Within this chapter, we navigate through the realm of Python's functional constructs, unraveling the intricacies of lambda functions, dictionary/list/set comprehensions, and map/reduce/filter functions. Additionally, we dissect the nuances of argument-passing mechanisms, deciphering the various ways Python manages information flow within a program. With a focused examination of Python List Comprehensions, we dissect their internal workings employing tools such as the Python Abstract Syntax Tree. This chapter serves as the cornerstone for our empirical studies, providing a comprehensive exploration of the functional paradigm of Python.

3.1 The Studied Python Functional Constructs And Their Complexity

This section briefly describes the functional constructs we have analyzed in our study together with the procedure used when computing their complexity.

3.1.1 McCabe's cyclomatic complexity for functional constructs

The cyclomatic complexity is a metric quantifying the complexity of a piece of code by accounting for independent paths in it. It is mainly computed by relying on control flow graphs with respect to functions, modules, methods, or classes within a software program.

The cyclomatic complexity has been defined by Thomas J. McCabe in 1976 [67], and it is based on a control flow representation of the program. For what concerns the complexity of the Python functional constructs object of this study, we have summed up the number of conditionals (including if expressions), loops, and generator functions (we will provide a more detailed explanation in Section 3.3 on Python Abstract Syntax Tree) being used within the construct definition plus 1.

3.1.2 Lambda Functions

A lambda function is an anonymous function that can be defined on the fly in the code. It can take any number of arguments, but can only have one expression. The power of lambda functions is better shown when they are used as an anonymous function inside another

function, *i.e.,* passing them as a parameter of a different function. The lambda definition in Python has the following syntax:

```
lambda arguments:  expression
```

```
print((lambda n: 0 if n % 10 == 1 and n % 100 != 11 else 1 if n % 10
    >= 2 and (n % 100 < 10 or n % 100 >= 20) else 2)(10))
```

Listing 3.1 Lambda expression example.

Listing 3.1 shows a lambda function being defined on the fly. Its cyclomatic complexity is equal to three since there are only two conditionals with no loops and no generators.

3.1.3 Dictionary/List/Set Comprehensions

Dictionary, list, and set comprehensions are "Pythonic" (although available in other languages, *e.g.,* Javascript) ways to create and manipulate dictionaries, lists, and sets. Specifically, 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.

A list comprehension can be defined with the following syntax:

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

A set comprehension can be defined with the following syntax :

```
{expression for item in iterable if condition}
```

A dictionary comprehension can be defined with the following syntax :

```
{key_expression:  value_expression for item in iterable if condition }
```

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

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

Listing 3.2 Dictionary, list and set comprehension examples.

Listing 3.2 shows a list comprehension, whose complexity is equal to three due to the presence of one loop, and one conditional.

3.1.4 Map/Reduce/Filter Functions

`Map()`, `reduce()` and `filter()`, are three high-order Python functions. They take a function as a parameter and return a function as output. They can be used by relying on the following syntax:

`FUN (function, iterable)`

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

`filter()` is used to apply a decision function, *i.e.,* a predicate, to each value in a collection. A decision of `True` implies that the value is passed; otherwise, the value is rejected. For instance, the filter function in Listing 3.3 retains only the strings made up of at most three characters.

```
chunked=['soleil','vie','HIVER','IL','monde','ver','de','a']
output = filter(lambda x: len(x) < 3, chunked)
print(list(output))
```

Listing 3.3 Filter function example.

`map()` assures that a given function (that can also be a lambda, as in the example of Listing 3.4) is used to map each item from one collection to a different collection, *i.e.,* an ideal way to apply a built-in function to a collection of data. In the example, it returns a series of values from 10.000 up to 10.007.

```
print(list(map(lambda x: x + 10000, range(8))))
```

Listing 3.4 Map function example.

`reduce()` which, differently from the previous function, is not built-in, but part of the `functools` package, is a higher-order function that folds a function into each pair of items in an iterable. Specifically, it 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 3.5, `reduce()` creates a string by concatenating the elements in one of the two lists based on the number of elements within them.

```
subject_givenName = ['julien','dave']
username=['jay']
print(reduce(lambda a, b: '%s | %s' % (a, b), subject_givenName or
   username))
```

Listing 3.5 Reduce function example.

Differently from comprehensions and lambda functions, when computing the cyclomatic complexity of the `filter()`, `map()` and `reduce()` functions, if they take as an argument a lambda function, we have also considered the complexity of the lambda. Specifically, the complexity for the `filter()` function shown in Listing 3.3 is equal to 2, as for the `map()` function in Listing 3.4 due to the presence of a single generator function.

3.1.5 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 [68]. 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 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 Python 3:

```
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.

3.2 The different types of argument-passing mechanisms

This section provides a brief overview of the different types of argument-passing mechanisms supported by the Python programming language. As a terminological note, we use the term "parameter" to refer to the formal parameters in the function definitions, and as "arguments" the actual parameter values passed when invoking the function. However, the features considered in our analyses use the term *arg* in parameters too, as we refer to them as they are detected by the Python AST parser [5]. Python provides the following features for defining parameters and consequently passing arguments:

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

```
# numLines has a default value
def exampleFun(fileName,numLines=1):
    with open(fileName) as f:
        ...
# Arguments are passed by position
lines=exampleFun("foo.txt",3)
# Scrambled arguments are passed by keyword
moreLines=exampleFun(numLines=10,fileName="bar.txt")
# numLines is omitted, hence it is equal to 1
otherLines=exampleFun(fileName="bar.txt")
```

Listing 3.6 Positional vs. keyword based argument passing.

```
def funcWithVarArgs(*args):
    if len(args)>0:
        a=args[0]

def funcWithKwargs(**kwargs):
    if 'size' in kwargs:
        a=kwargs['size']

def funcWithPosArgs(size, compression):
    ...
#1, 2, 3 are packed into args
funcWithVarArgs(1,2,3)
# 'size':10 is stored into kwargs
funcWithKwargs(size=10)
# Arguments unpacking
argList=[20,10]
funcWithPosArgs(*argList)
```

Listing 3.7 Example of varargs kwargs and argument unpacking.

- *Positional vs. keyword-based:* In the first case, the association between arguments and parameters is based on their order. In the second case, arguments are passed as name-value pairs, which makes it possible to pass them in an arbitrary order. In Listing 3.6, the two invocations of the function `exampleFun` in lines 6 and 8 are done by passing positional arguments and keyword-based arguments, respectively.

- *Required vs. optional:* In the function definition, a parameter can have a default value

(*e.g.,* in Listing 3.6 the `numLines` parameter has a default value of 1) that makes its passing optional (omitted at line 10).

- *Fixed vs. variable:* To allow for passing a variable number of arguments, the function definition can contain a "varargs" parameter, which name is preceded by a "*". When the function is invoked, the arguments exceeding the fixed parameters will be mapped to items of a list. For example, in Listing 3.7 the passed arguments 1, 2, and 3 will be mapped to items of the list `args`. Similarly, if the parameter name is preceded by "**", then parameters passed by keywords will become a dictionary.

- *Argument unpacking:* Passing a list to a function and preceding its name with a "*" causes the unpacking of the list items onto positional parameters. For example, in line 17 of Listing 3.7 the list `argList` is passed to `funcWithPosArgs`, and its items are mapped onto `size` and `compression`. Similarly, if an argument is preceded by "**", it is assumed to be a dictionary and its items are mapped onto parameters whose names correspond to the keys.

```
1  def posOnlyArgs(size=1,compression=10,/,fileName="bar"):
2      print(size,compression,fileName)
3
4  def kwOnlyArgs(size=1,*,compression=10):
5      print(size,compression)
6
7  #These are correct
8  posOnlyArgs(10,20)
9  posOnlyArgs(10,fileName="foo")
10 #This raises an error
11 posOnlyArgs(compression=2)
12
13 #These are correct
14 kwOnlyArgs(10)
15 kwOnlyArgs(compression=5)
16 #This raises an error
17 args=[10,20]
18 kwOnlyArgs(*args)
```

Listing 3.8 Positional-only and kewword-only arguments

- *Positional-only parameters:* It is possible to enforce that some parameters can only be passed by position. This is achieved by following a list of parameters with a "/". Then,

all parameters preceding the "/" must be passed by position, whereas the remaining ones can be either passed by position or by keyword. In Listing 3.8, the invocations in lines 8 and 9 are correct, whereas the one in line 11 raises an error: `compression` cannot be passed by keyword.

- *Keyword-only parameters:* Since Python 3.8 [69], and similarly to the above case, all parameters following a "*" can only be passed by keyword. In Listing 3.8, the invocations in lines 14 and 15 are correct, whereas the one in line 18 raises an error. Keyword-only arguments may avoid the unintentional unpacking of a longer-than-expected list onto wrong parameters.

3.3 Python Abstract Syntax Tree

Python provides native support to navigate its Abstract Syntax Tree (AST). Specifically, the *ast* module, features a parsing method that transforms source code into a typed tree representation. The obtained Python AST nodes are decorated with information including source code line details (start and end lines).

The abstract grammar, defined within the AST tree module, outlines the structure of Python's syntax. It encompasses various constructs, including modules, statements, expressions, and more. The grammar provides a hierarchical representation of Python code, detailing the syntax and relationships between different elements such as function definitions, assignments, conditionals, and loops. In the context of this thesis, particular emphasis is placed on the following aspects[2]:

```
1  stmt = | For(expr target, expr iter, stmt* body, stmt* orelse,
     string? type_comment)
2         | While(expr test, stmt* body, stmt* orelse)
3         | If(expr test, stmt* body, stmt* orelse)
4
5  expr =  | Lambda(arguments args, expr body)
6          | IfExp(expr test, expr body, expr orelse)
7          | ListComp(expr elt, comprehension* generators)
8          | SetComp(expr elt, comprehension* generators)
9          | DictComp(expr key, expr value, comprehension* generators)
10         | GeneratorExp(expr elt, comprehension* generators)
11         | Call(expr func, expr* args, keyword* keywords)
```

Listing 3.9 AST tree.

[2]https://docs.python.org/3/library/ast.html

In the AST representation Listing 3.9, here are the elements of interest for this thesis. The "For" statement involves a target variable, iteration, and optional clauses, directing the flow of repeated execution. The "While" statement controls code execution based on a specified condition, and the "If" statement provides conditional branching. On the expression side, the "Lambda" expression facilitates the creation of anonymous functions. The "IfExp" expression handles ternary conditional operations, while various comprehensions like "ListComp," "SetComp," "DictComp," and "GeneratorExp" succinctly represent complex data structures and iterations. Furthermore, the "Call" expression encapsulates function calls, including operations like "map," "filter," and "reduce."

The AST module provides a **walk** tree method, which, when given an AST node as a parameter, traverses each node within the subtree of the provided parameter. In essence, it recursively visits all nodes.

The combination of the **walk** method and the typed nature of AST nodes simplifies the traversal and retrieval of various information. For instance, it enables tasks such as locating all instances of extracting software metrics, such as the count of list comprehensions.

3.3.1 The `for` and `ListComp` Nodes

The following two code fragments and their AST representations illustrate Python AST structure:

- The 'for' loop code listing 3.10, is represented by the following Abstract Syntax Tree (AST) listing 3.11:

The AST structure consists of nodes corresponding to different syntactic elements in the code. The top-level node is a `For` node, representing the for loop. Within this node, there is a `Name` node for the loop variable (`n`), another `Name` node for the iterable (`numbers`), and a nested `If` node representing the conditional statement. Inside the `If` node, there is an `Expr` node, representing an expression, and a `Call` node, signifying a method call (`append` on `new_list`). The `Call` node contains an `Attribute` node, indicating the attribute `append` of the object `new_list`, and a `BinOp` node for the multiplication operation (`n * 2`).

```
1 for n in numbers:
2     if condition:
3         new_list.append(n * 2)
```

Listing 3.10 A simple 'for' example.

```
For(
  target=Name(id='n', ctx=Store()),
  iter=Name(id='numbers', ctx=Load()),
  body=[
    If(
      test=Name(id='condition', ctx=Load()),
      body=[
        Expr(
          value=Call(
            func=Attribute(
              value=Name(id='new_list', ctx=Load()),
              attr='append',
              ctx=Load()),
            args=[
              BinOp(
                left=Name(id='n', ctx=Load()),
                op=Mult(),
                right=Constant(value=2))],
            keywords=[]))],
      orelse=[])],
  orelse=[])
```

Listing 3.11 AST simple for example.

```
new_list = [n * 2 for n in numbers if condition]
```

Listing 3.12 A simple list comprehension example.

```
ListComp(
    elt=BinOp(
        left=Name(id='n', ctx=Load()),
        op=Mult(),
        right=Num(n=2)
    ),
    generators=[
        comprehension(
            target=Name(id='n', ctx=Store()),
            iter=Name(id='numbers', ctx=Load()),
            ifs=[Name(id='condition', ctx=Load())])])
```

Listing 3.13 AST simple list comprehension example.

- The list comprehension code example 3.12, which is represented by the following Abstract Syntax Tree (AST) example 3.13:

`ListComp` node, indicating a list comprehension. Within this node, there is an `elt`node representing the list element (`n * 2`), and a `generators` list containing a `comprehension` node representing the comprehension itself (`for n in numbers if condition`). Inside the `comprehension` node, there is a `target` node for the loop variable (`n`), an `iter` node for the iterable (`numbers`), and an `ifs` list containing a `Name` node representing the condition (`condition`). This hierarchical structure captures the syntactic relationships and logic within the original Python code, illustrating how the list is generated by iterating over `numbers` with the condition `if condition`.

3.4 Statistic test

In this thesis, we have utilized various statistical tests. To provide a clearer understanding, consider the following fictitious example.Consider two variables: "bug" (presence or absence of bugs) and "functionality" (whether an element is functional or not). Assume that a sample of 50 elements has been drawn from a larger population for a study investigating the relationship between bugs and functionality. A contingency table has been constructed to present the distribution of elements categorized by their functionality status and bug presence. The table is presented Table 3.1:

Table 3.1 contingency table example

	Bug	**Not Bug**	**Total**
Functional	16	4	20
Not Functional	10	20	30
Total	26	24	50

The counts representing functional, non-functional, bug-afflicted, and bug-free elements are termed marginal totals. The aggregate total (the sum of all elements represented in the contingency table) is indicated in the bottom right corner.

This table allows for a concise overview of the distribution: 16 functional elements exhibit bugs, 10 non-functional elements are affected by bugs, 4 functional elements are bug-free, and 20 non-functional elements are free of bugs. The strength of the association between bugs and functionality can be quantified through measures such as the odds ratio. The odds ratio, in general terms, is a statistical measure used to assess the association between two variables. In this context, it compares the odds of bugs occurring in functional elements to the odds of

bugs occurring in non-functional elements, providing insights into the relationship between bugs and functionality.

To do this, we will use the following formulas:

$$P(\text{Bug} \mid \text{Functional}) = \frac{\text{Number of bugs among functional elements}}{\text{Total functional elements}}$$

$$P(\text{Bug} \mid \text{Not Functional}) = \frac{\text{Number of bugs among non-functional elements}}{\text{Total non-functional elements}}$$

$$P(\text{Bug} \mid \text{Functional}) = \frac{16}{20} = 0.8$$

$$P(\text{Bug} \mid \text{Not Functional}) = \frac{10}{30} = 0.333$$

Now, we can calculate the odds ratio:

$$\text{Odds Ratio} = \frac{0.8/(1-0.8)}{0.333/(1-0.333)}$$

$$\text{Odds Ratio} = \frac{0.8/0.2}{0.333/0.667} = \frac{4}{0.5} = 8$$

Statistical significance regarding this relationship can be assessed using various tests, including Pearson's chi-squared test and Fisher's exact test, provided that the sample is representative of the population. If significant disparities exist in the proportions of elements across different categories, it suggests a contingency between the variables. In such cases, the variables are considered dependent; otherwise, they are deemed independent.

3.4.1 Non-parametric tests

The Chi-square test, the data should be frequencies or counts of cases, the levels of variables should be mutually exclusive, each subject should contribute to only one cell of the table,

and study groups should be independent. Additionally, variables should be categorical and have expected values of at least 5 in at least 80% of the cells, with no expected value less than one. This requirement is best met when the sample size is at least equal to the number of cells multiplied by 5. Adhering to these assumptions is essential for proper application and adequate interpretation of the χ^2 test.

Fisher's Exact Test relies on several assumptions, including random sampling, independence of observations, and mutual exclusivity of groups, to ensure the accuracy of its results. When applying Fisher's Exact Test, it's essential to assess whether these assumptions are met in the dataset under analysis. In cases where cell counts are low, such as when there are fewer than 10 observations per cell, Fisher's Exact Test is often preferred due to its robustness in such scenarios where assumptions necessary for other tests may be violated. In the contingency table provided above, with a total of 50 observations example table 3.1, the use of Fisher's Exact Test was motivated by the presence of cells containing a low number of observations, such as the cell with 4 cases of non-functional features, where the sample size is less than 10. This choice ensures the validity and reliability of the statistical analysis when investigating differences between categorical variables.

3.4.2 Generalized Linear Mixed Models

In assessing the fit of Generalized Linear Mixed Models (GLMMs) [70], several key assumptions play a pivotal role. One fundamental assumption pertains to the distribution of random effects, particularly the expectation that both intercepts and slopes follow a normal distribution. Although assessing this assumption is challenging, visual methods such as histograms or Q-Q plots offer valuable insights. A symmetric bell-shaped histogram or Q-Q plot aligning closely with a diagonal line suggests conformity with the normal distribution assumption. However, addressing deviations from normality in random effects poses complexities, as transformations typically applied in linear models are not feasible in GLMMs.

Another critical assumption concerns the appropriateness of the chosen link function, which dictates the relationship between predictors and outcomes. Notably, common link functions like the logit for binomial data or the log for Poisson data may not always accurately capture this relationship. To evaluate this assumption, comparing predicted values to actual outcomes provides a practical approach. By comparing the averages of outcomes to predicted values, deviations indicative of misfit can be identified, akin to the Hosmer-Lemeshow test for logistic regression models. Remedies for potential mis-specifications include altering the link function, transforming predictors, or categorizing continuous predictors.

Lastly, ensuring appropriate estimation of variance constitutes another critical aspect. Unlike

linear models, where variance is constant, variance in GLMMs is a function of the estimated parameter, making its assessment more intricate. Nonetheless, fit statistics like the generalized chi-square, comparing model residuals to theoretical variance, offer a means to evaluate variance estimation. A chi-square statistic close to 1 signifies adequate variance estimation, while deviations indicate potential overdispersion or underdispersion. Adjusting models for overdispersion can mitigate this issue, leading to more conservative p-values.

In this thesis, we opted to use GLMR, implemented through the 'lme4' [71] package in R, to address these complexities and ensure the robustness of our analysis, aiming to obtain more accurate and reliable estimates while enhancing the validity and interpretability of our findings.

CHAPTER 4 A STUDY ON THE PYTHONIC FUNCTIONAL CONSTRUCTS' UNDERSTANDABILITY

This chapter presents the outcomes of a controlled experiment involving 209 developers, aimed at evaluating the understandability of key Pythonic functional constructs—specifically lambdas, comprehensions, and map/reduce/filter functions—in comparison to their procedural counterparts.

The study sought to address the overarching goal of assessing the understandability of Pythonic functional constructs through a threefold approach. Developers were tasked with modifying code, utilizing both functional constructs and procedural alternatives, to compare the understandability of different implementations. Additionally, participants were asked to provide insights into the situations where the use of functional constructs is preferable.

The findings of the study revealed nuanced insights into the impact of functional constructs on code understandability. Notably, code snippets employing lambdas were found to be more amenable to modification compared to their procedural counterparts. However, this was not uniformly observed in the case of comprehensions. Perceived understandability favored procedural implementations, with code snippets relying on procedural constructs generally considered more readable than their functional counterparts. Importantly, while functional constructs contribute to code conciseness and potential improvements in maintainability and performance, they were identified as posing challenges in terms of debugging.

The implications of these results extend to the realm of developer education, emphasizing the need for a nuanced understanding of when and where to employ functional constructs. Furthermore, the findings highlight the importance of prioritizing quality assurance activities in code sections involving functional constructs and suggest avenues for enhancing tool support to alleviate debugging challenges faced by developers.

4.1 Study Definition and Planning

The *object* of this study are Pythonic functional constructs, and, specifically, lambdas, comprehensions, and MRF. The *quality focus* refers to their understandability, especially during a change task. The *perspective* is of researchers interested in distilling guidelines for developers, educators, and researchers/tool builders to facilitate the comprehension and maintenance of Python code. The *context* consists of 209 paid developers recruited through the Prolific platform, and code snippets written using functional constructs, as well as their procedural

alternatives. The study aims to address the following three research questions:

1. **RQa$_1$:** *How difficult is it to modify code snippets relying on functional constructs compared to their procedural alternatives?* This research question aims to assess, through change tasks, the understandability of code written using functional constructs versus their procedural alternatives.

2. **RQa$_2$:** *To what extent do functional constructs influence the perceived source code understandability?* Here we assess the perceived understandability of a code snippet using a functional construct versus its procedural alternative. This is achieved by showing two functionally equivalent code snippets written with and without functional constructs.

3. **RQa$_3$:** *Why do developers use functional constructs, and in which scenarios?* The last research question qualitatively investigates the reasons why and when developers use functional constructs, but also the perceived disadvantages.

4.1.1 Context Selection - Study Participants

The study was conducted by leveraging 209 paid developers recruited using the Prolific [72] platform. As highlighted by Tahaei and Vaniea [73], this is the only platform that could involve an adequate number of participants who pass the programming questions. Note that Prolific has been previously used in software engineering research on defect prediction [74], or tool evaluation [75]. Existing studies [76–78] highlighted the "vague meaning of programming experience". Therefore, we have complemented a set of constraints on the Prolific platform, *e.g.,* in terms of education, skills, and working experience to recruit developers, with further filters, *e.g.,* filtering out all respondents not able to complete any of the six change tasks. Specifically, we have selected the participants on several requirements: (i) the subject of study was computer science, (ii) they declared to know computer programming, (iii) Python was listed as one of the known programming languages, and (iv) for the education level, we limited our choices to technical/community college, undergraduate (*e.g.,* bachelor), graduate (*e.g.,* master), or doctorate degree.

Given that we estimated a task duration between 40 minutes and 1 hour, and considering our available budget, we offered a payment of 12 UK Pounds-hour (the minimum as per Prolific's policy is 8 UK Pounds-hour).

4.1.2 Context Selection - Study Objects

Since our experiment shows code snippets to developers and asks them to perform certain tasks, we need to properly select representative ones. To avoid artificial code snippets, we started identifying random instances of the investigated functional constructs from an existing dataset [79]. The selection was made to select, for each type of functional construct, a set of code elements with varying complexity. The complexity is computed as McCabe's cyclomatic complexity [80], by adding one to the number of conditionals, iterations, and generators.

In particular, we picked: 10 lambda usages with a complexity between 1 and 4; 10 comprehension usages, with a complexity between 2 and 6; and 3 map, 3 reduce, and 3 filter usages.

After the selection, we isolated each code snippet from its surrounding context, by creating constant initialization for the used variables. Then, for each code example, one author produced the procedural alternative, that has been double-checked by a different author, both having at least three years of Python programming experience. Specifically, we replaced lambda with an explicit function definition, while comprehensions have been replaced by nested loops and conditionals. For MRF, we replaced their invocation with the explicit implementation of the required operation, *e.g.,* iterating through all the elements of a list and applying to all of them the desired operation. As a last step, we made sure that each code snippet produced an output so that it was possible to objectively assess its outcome, as well as we verified that, given the same input, the output of the functional code snippet was perfectly similar to the one obtained with the "hand-crafted" procedural alternative.

4.1.3 Dependent and independent variables

The study *dependent variable* is the code understandability, measured as change task correctness for RQa_1, and perceived understandability for RQa_2. The *independent variable* is the use of a functional construct, as opposed to its procedural alternative. While the analysis has been done on multiple constructs (lambdas, comprehensions, and MRF), this is not a variable of the study, as we do not directly compare constructs with each other. In addition, we account for *co-factors* related to constructs' complexity and developers' characteristics. In the following section, we further detail how variables are measured, and what co-factors are.

4.1.4 Experiment Design and Planning

The experiment consists of a series of questions, asked to each participant through a Web-based questionnaire (a Google form). The questionnaire was organized into 4 sections.

```
lst=[[[i + j + k for k in range(4, 7)] for j in range(1, 4)] for i
   in range(2, 5)]
print(lst)
```

Code snippet with functional constructs: Group 6

Question: Please modify the code so that it will generate a matrix where the elements in the rows are summed if the indexes are all even, otherwise are multiplied.

Figure 4.1 Example of change task for list comprehensions

```
lst=[]

for i in range(2, 5):
  lt=[]
  for j in range(1, 4):
    l=[]
    for k in range(4, 7):
      l.append(i + j + k )
    lt.append(l)
  lst.append(lt)

print(lst)
```

Code snippet with procedural constructs: Group 5

Question: Please modify the code so that it will generate a matrix where the elements in the rows are summed if the indexes are all even, otherwise are multiplied.

Figure 4.2 Example of change task for the procedural counterpart of the list comprehension reported in Fig. 4.1

In the **first section**, we assess the participants' understanding of code snippets by letting them perform six change tasks. This has been previously done in other studies on program comprehension (*e.g.,* [23, 81, 82]), allowing for an objective evaluation, and better reflecting a scenario in which a developer needs to understand a code snippet during evolution tasks. Specifically, we show a code snippet and ask participants to modify it. Three tasks include code snippets using functional constructs: one with lambda, one with comprehension, and one with map/reduce/filter. The other three tasks (interleaved with the former) feature procedural alternatives of functional code that different participants are changing. Fig. 4.1 illustrates an example of a change task for list comprehension (assigned to group 6) while its

Table 4.1 Assignment of change and comparison tasks across the ten experimental groups. MRF means map/reduce/filter. Note that for each task, there is the identifier of the task, followed (for the change tasks) by a letter indicating whether the snippet used the (F)unctional or (P)rocedural paradigm, and the complexity in parenthesis (for lambda and comprehension only).

Group	Change Tasks						Comparison Tasks		
	Lambda$_1$	**Lambda$_2$**	**Comp$_1$**	**Comp$_2$**	**MRF$_1$**	**MRF$_2$**	**Lambda**	**Comp.**	**MRF**
1	lambda-a-F (1)	lambda-g-P (4)	comp-a-F (2)	comp-f-P (3)	filter-a-F	map-c-P	lambda-h (2)	comp-b-(4)	reduce-a
2	lambda-b-P (2)	lambda-g-F (4)	comp-a-P (2)	comp-f-F (3)	filter-a-P	map-c-F	lambda-c (3)	comp-d (5)	reduce-c
3	lambda-a-P (1)	lambda-h-F (2)	comp-b-P (4)	comp-g-F (2)	map-a-F	reduce-b-P	lambda-g (4)	comp-f (3)	filter-a
4	lambda-b-F (2)	lambda-i-P (3)	comp-b-F (4)	comp-g-P (2)	map-a-P	reduce-b-F	lambda-d (2)	comp-a (2)	filter-c
5	lambda-c-F (3)	lambda-h-P (2)	comp-c-F (2)	comp-h-P (4)	reduce-a-P	filter-b-F	lambda-b (2)	comp-j (6)	map-c
6	lambda-c-P (3)	lambda-d-F (2)	comp-c-P (2)	comp-h-F (4)	reduce-a-F	filter-b-P	lambda-i (3)	comp-g (2)	map-a
7	lambda-d-P (2)	lambda-i-F (3)	comp-d-P (5)	comp-i-F (3)	map-b-F	filter-c-P	lambda-a (1)	comp-h (4)	reduce-b
8	lambda-e-F (3)	lambda-j-P (2)	comp-d-F (5)	comp-i-P (3)	map-b-P	filter-c-F	lambda-f (1)	comp-e (3)	reduce-a
9	lambda-e-P (3)	lambda-f-F (1)	comp-e-P (3)	comp-j-F (6)	filter-b-F	reduce-c-P	lambda-e (3)	comp-c (2)	map-b
10	lambda-f-P (1)	lambda-j-F (2)	comp-e-F (3)	comp-j-P (6)	filter-b-P	reduce-c-F	lambda-e (3)	comp-i (3)	map-c

procedural alternative, assigned to a different experimental group (group 5), is depicted in Fig. 4.2.

The **second section** consists of three questions asking the perceived ease of understandability of a functional construct and its (functionally equivalent) procedural alternative. Each participant receives three pairs of code snippets, one related to lambda, one to comprehension, and one to MRF. We ask participants to perform the assessment using a 5-level Likert scale [83], as illustrated in Fig. 4.3. To mitigate possible ordering bias, we change the order in which we show the code snippets: some participants receive the functional before the procedural, and vice versa. Furthermore, to mitigate risks due to "random" answers, we also ask the participants to describe the behavior of the code snippet, so we were sure they understood its meaning.

The **third section** consists of four questions asking the reasons for using functional constructs, as well as the procedural alternatives, *e.g., "If you use lambda functions regularly, can you please explain the reasons why you are using them compared to their procedural alternatives? Please use None as an answer in case you do not use them regularly."*

The **fourth section** (optional), collects demographics. We ask about (i) the highest degree earned, (ii) the current working position, (iii) the years of development experience, (iv) and, the frequency of usage of each of the considered functional constructs by providing as options: Never; Less than 25% of the development tasks; Between 25% and 75% of the development tasks; and More than 75% of the development tasks. The declared usage frequency will be used as one of the criteria to discard responses.

Table 4.1 summarizes the assignment of change (RQa_1) and comparison (RQa_2) tasks to the study participants. For the change tasks, we indicate the type of construct being studied,

```
i = [1, 2, 3, 4, 5, 6]

def function(i):
    return tuple((-1 if j is None else j for j in i[1:4]))

print(function(i))
```

```
i = [1, 2, 3, 4, 5, 6]
print((lambda i: tuple((-1 if j is None else j for j in i[1:4])))(i))
```

Please select a choice:

1. The first code is definitely easier to understand than the second code
2. The first code is slightly easier to understand than the second code
3. There are no differences in terms of understandability
4. The second code is slightly easier to understand than the first code
5. The second code is definitely easier to understand than the first code

Figure 4.3 Example of understandability task for lambda and its procedural alternative

whether we show the (F)unctional construct or its (P)rocedural alternative, and, for lambda and comprehension, the complexity of the construct within the code snippet (in parenthesis). As the table shows, we have 10 experimental groups, designed with the following constraints:

- For lambda and comprehension, participants in the same group receive code snippets with constructs of different complexity. As MRF mainly applies a function (which from the client side may even be a black box) to each data structure entry, complexity was not considered a relevant factor, so we have not considered their complexity in both the experimental design and the analysis methodology.
- The comparison tasks are designed by showing the functional before the procedural or vice versa to different experimental groups.
- For MRF, the experimental groups are designed so that each participant receives a map example, a reduce example, and a filter example, *i.e.,* one for the first change task, one for the second change task, and one for the comparison.

4.1.5 Experiment Execution

Before experimenting, its design was submitted to a University Ethical Board for their approval, as the study involved humans. After receiving the approval, we proceeded with the operation, administering the tasks to the study participants. To avoid having the same person in different experimental groups to earn money (this was explicitly disallowed by the protocol

we published), we created 10 different Web forms, and with them, 10 sequential "Studies" on Prolific. After each participant completed the questionnaire, one author (i) checked whether the participant had not already participated in a different previous experimental group, and (ii) performed a sanity check to guarantee questions were answered. If such checks passed, the author approved the payment, otherwise, the participant was discarded. Where necessary (*i.e.,* having fewer participants), we extended the invitation so that we ended up with at least 20 answers for each experimental group. Out of 248 returned questionnaires, we discarded 39, of which 18 did not pass the sanity check, 14 have never used any of the studied functional constructs, and 7 did not provide any correct answer for all six change tasks.

4.1.6 Results Collection and Processing

After the collection of the results from the 10 experimental groups, we analyzed the answers. First, we excluded participants who declared having never used any of the considered functional constructs. However, we kept those that only used some constructs, *e.g.,* lambda and comprehension, but they are only considered for the questions related to that construct.

For the **change tasks**, each code snippet produced by the participants was scrutinized to check whether the change was implemented and the output was not faked (*e.g.,* the code snippet just produced some code generating the required output). After that, the code snippets were executed through a test script, comparing the scripts' output with the expected one. The outcome of each change task was considered *correct* if the test passed, *wrong* in case of run-time errors or failing test. Using the results of this analysis, we also excluded (from all the analyses of our study) 7 participants who performed all tasks wrongly.

For the **comparison tasks**, there are two aspects to account for, namely the description of the functionality implemented by the showed code snippet and the perceived understandability. While the latter was trivial to collect and analyze, for the former, the two authors cooperatively scrutinized the responses describing the code snippet's behavior. Each response was classified into four categories: (i) correct, (ii) somewhat correct (*e.g.,* some details about the behavior were missing, but, overall, the respondent understood the general behavior), (iii) wrong, and (iv) automatically generated. For the latter, the authors used GPTZero [84] to check whether the produced text was likely to be AI-generated. We have considered this to be the case if the outcome of GPTZero reads: *"Your text is likely to be written entirely by AI"*.

For the **reasons for using functional constructs, as well as their procedural alternatives**, we applied open card-sorting [85] on the provided answers. Also in this case, we excluded all answers likely to be AI-generated based on the outcome of GPTZero [84]. On

the remaining answers, two authors (annotators in the following) performed a joint categorization of 20 responses to agree on the labeling criteria. After that, they used an online spreadsheet to independently categorize the remaining responses. Specifically, the annotators could select a category from a list of possible ones. When a new category was required, the annotator could add it to the list, so it became available to both of them. Note the sheet was designed so that each annotator could assign multiple categories to the same answer when a respondent provided multiple reasons for using a construct. To assess the quality of the manual labeling, we used Krippendorff's α [86] reliability coefficient. This procedure can handle missing codes, as well as the presence of multiple categories assigned to the same answer. The two annotators achieved an $\alpha = 0.87$ (very strong) concerning the reasons for using functional constructs, and $\alpha = 0.70$ (strong) for reasons behind preferring the procedural alternatives.

4.1.7 Analysis Methodology

In the following, we describe the methodology used to address the three research questions.

To address $\mathbf{RQa}_1$, for each functional construct, we exclude data points for respondents who never used the construct. This left 160 valid responses for lambdas, 192 for comprehensions, and 159 for MRF. After that, we statistically analyze the remaining data points.

Specifically, to perform the statistical analysis of the results, we need to use a procedure that tests the relationship between the correctness of the change task and various factors besides the main experimental factor. Also, the procedure needs to check the longitudinal, within-subjects effect. To this aim, we use a mixed-effect logistic model, leveraging the *glmer* function of the *R lme4* package [71]. We use such a model because it is suitable for an experiment with dependent samples. More in detail, the participants (through their identifiers) are the model's random effect, the dependent variable is the change task correctness, and the independent variables are:

- **Main Factor:** Whether the code snippet was shown with the *functional* construct or with its *procedural* alternative;
- **Complexity:** The effect of the constructs' complexity (for lambdas and comprehensions only) and its interaction with the main factor;
- **Approvals:** The participants' number of previous task approvals on Prolific;
- **Usage Frequency:** The participant self-declared frequency usage for that type of construct;
- **Student:** Whether the participant is currently a student, as declared on Prolific.

Since the *glmer* procedure involves multiple factors for which p-values are computed, we adjust them using the Benjamini-Hochberg procedure [87].

As the experiment design foresees, for MRF, the longitudinal analysis with the *glmer* model will only be possible by ignoring the construct type. The latter is mainly due to having only two change tasks, one with one of the three functions and a different one with a different function. For this reason, for MRF, we perform an additional (unpaired, between subjects) analysis for each specific function through a logistic generalized linear model (*glm*). We complement the test with the Odds Ratio (OR) effect size measure, where the OR reads as the increasing odds to flip the dependent variable to *true* (*i.e.,* correct answer) for a unitary increase of the independent variable, or, for categorical dependent variables, the increasing odds when the variable assumes that value.

To address $\mathbf{RQa}_2$, for each functional construct, we report the comparative, perceived level of understandability using diverging stacked bar charts. Furthermore, we show the relationship between the results and (i) the declared frequency of usage of the different functional constructs, as well as (ii) the construct complexity (for lambda and comprehension only). This is done with an ordinal logistic regression model, implemented through the *polr* function of the *R MASS* package [88]. In this case, the OR indicates how the unitary increment of an independent variable increases the odds of a unitary increase of the dependent variable. Also in this case, p-values are adjusted with the Benjamini-Hochberg procedure [87].

Finally, to address $\mathbf{RQa}_3$, we report and discuss the elicited reasons for using functional constructs, as well as their procedural alternatives.

4.2 Study Results

In the following, after reporting the demographics of the study participants, we report and discuss the study results.

4.2.1 Participants' Demographics

Although demographic questions were optional, all participants answered them. In terms of highest education qualification, 122 out of 209 participants hold a bachelor's degree, 41 have a master's degree, 4 have received a Ph.D., and the remaining 42 have a high school diploma while being undergraduates. For what concerns development experience, 26 out of 209 participants declare to have between 5 and 10 years of development experience, while 183 have less than 5 years of experience. Hence, our sample is, in its majority, representative of junior/average-seniority developers.

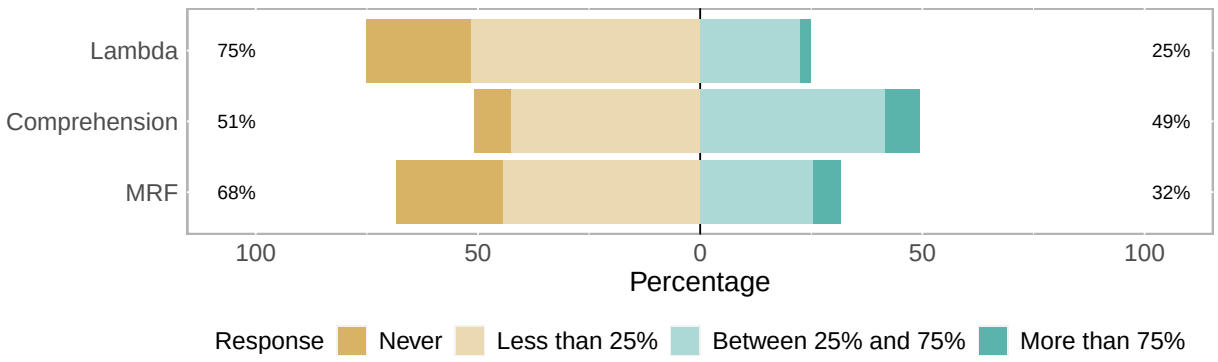

Figure 4.4 Declared construct usage.

Fig. 4.4 summarizes the declared usage of the functional constructs object of the study. As the figure shows, 49% of the participants use comprehensions for over 25% of their tasks, while for MRF this happens only for 32% of the participants, and for lambda only for 25% of them.

Table 4.2 Number and percentage of correct/wrong tasks for functional constructs and their procedural alternatives

Construct	**Functional**		**% Corr.**	**Procedural**		**% Corr**
	Corr.	**Wrong**		**Corr.**	**Wrong**	
Lambda	112	98	53.33	115	95	54.76
Compr.	99	111	47.14	114	96	54.29
MRF	80	130	38.10	85	125	40.48

4.2.2 RQa$_1$: How difficult is it to modify code snippets relying on functional constructs compared to their procedural alternatives?

Table 4.2 reports the number of correct and wrong tasks for functional constructs and their procedural alternatives. Looking at it, we can only state that for comprehension, the percentage of correct answers is lower (47.14%) for the functional than for the procedural alternative (54.29%). For lambda and MRF, the difference is again in favor of the procedural, yet the difference is 1-2% only.

Table 4.3 shows the results of the mixed-effect model for the lambda construct. As the table shows:

1. There is a statistically significant effect of the main factor. Specifically, giving procedural code, in general, reduces the odds of $OR = 0.11$ to implement a correct change;

Table 4.3 RQa_1: Mixed-effect logistic regression relating the use of lambdas with the correctness of the change task

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**	
438.3	468.5	-211.2	422.3	314	
SCALED RESIDUALS:					
Min	**1Q**	**Median**	**3Q**	**Max**	
-1.98	-0.95	0.49	0.85	1.44	
RANDOM EFFECTS:					
Groups				**Variance**	**Std.Dev.**
User (Intercept)				0.2299	0.4795
Number of obs: 322, groups: User, 160					
FIXED EFFECTS:					
	OR	**Estimate**	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	3.73	1.32	1.12	1.17	0.24
MainFactorProc	0.11	-2.19	0.67	-3.29	**<0.01**
Compl.	0.75	-0.29	0.18	-1.55	0.24
Usage Freq.	1.32	0.27	0.23	1.20	0.24
Approvals	1.00	-0.00	0.00	-1.41	0.14
StudentTrue	0.35	-1.04	0.88	-1.18	0.24
MainFactorProc:Compl.	2.67	0.98	0.28	3.56	**<0.01**

2. The complexity alone does not play a statistically significant effect;
3. There is a positive statistically significant interaction between the main factor and the complexity of the construct. Specifically, if the participant receives procedural code, increasing its complexity increases 2.67 times the odds of correctly implementing the change, compared to those who receive its functional alternative;
4. Participant-related metrics (Usage Frequency, Approvals, and Student status) do not play a statistically significant role.

Overall, we can conclude that when the complexity of the lambda increases, it is easier to correctly modify the code snippet by operating on its procedural alternative, *i.e.,* modify a separate function.

Table 4.4 shows the results of the mixed-effect model for the comprehension construct. In this case:

1. The main factor has a statistically significant effect. However, in this case, giving procedural code increases the odds of correctly implementing the change ($OR = 6.11$);
2. The complexity alone does not have a statistically significant effect;
3. The main factor has a statistically significant interaction with the complexity, yet the effect is the opposite of what was reported for lambdas. When procedural code is used,

Table 4.4 RQa_1: Mixed-effect logistic regression relating the use of comprehensions with the correctness of the change task

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**	
534.7	566.3	-259.3	518.7	378	
SCALED RESIDUALS:					
Min	**1Q**	**Median**	**3Q**	**Max**	
-1.64	-0.96	0.62	0.99	1.59	
RANDOM EFFECTS:					
Groups				**Variance**	**Std.Dev.**
User (Intercept)				0	0
Number of obs: 386, groups: User, 192					
FIXED EFFECTS:					
	OR	**Estimate**	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	1.15	0.14	0.89	0.16	0.88
MainFactorProc	6.11	1.81	0.60	3.00	**0.02**
Compl.	1.02	0.02	0.11	0.15	0.88
Usage Freq.	0.86	-0.15	0.16	-0.94	0.61
Approvals	1.00	-0.00	0.00	-1.12	0.61
StudentTrue	1.19	0.18	0.66	0.27	0.88
MainFactorProc:Compl.	0.65	-0.43	0.17	-2.58	**0.03**

an increase of complexity reduces the odds of $OR = 0.65$ for correctly implementing the change. Possibly, developers better inspect loops and conditionals in a comprehension expression than in a (longer) procedural code.

4. Participant-related metrics do not play a statistically significant role.

For MRF (see Table 4.5) we were unable to observe any statistically significant difference. However, as mentioned before, the paired analysis is not sufficient for MRF, because participants received, over the two change tasks, code snippets dealing with different functions.

Therefore, we performed an unpaired analysis for each function whose results (see Tables 4.6, 4.7, 4.8) did not report any statistically significant difference. A possible reason could be the relative simplicity of the functions called by the MRF (needed by the controlled experiment setting), and, consequently, of their procedural alternative.

RQa_1 Summary: Procedural code is easier to modify than complex lambdas. In contrast, complex comprehensions are easier to modify than procedural code for list construction. For MRF, no statistically significant differences were found.

Table 4.5 RQa_1: Mixed-effect logistic regression relating the use of MRF with the correctness of the change task

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**	
443.0	465.6	-215.5	431.0	314	
SCALED RESIDUALS:					
Min	**1Q**	**Median**	**3Q**	**Max**	
-0.92	-0.89	-0.76	1.12	1.52	
RANDOM EFFECTS:					
Groups				**Variance**	**Std.Dev.**
User (Intercept)				0	0
Number of obs: 320, groups: User, 159					
FIXED EFFECTS:					
	OR	**Estimate**	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	1.64	0.50	0.96	0.52	0.93
MainFactorProc	0.95	-0.05	0.23	-0.23	0.93
Usage Freq.	0.74	-0.30	0.18	-1.60	0.55
Approvals	1.00	-0.00	0.00	-0.29	0.93
StudentTrue	0.94	-0.07	0.76	-0.09	0.93

Table 4.6 RQa_1: Logistic regression relating the use of map with the correctness of the change task (AIC=121)

	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	16.48	1370.51	0.01	0.99
MainFactorProc	0.67	0.48	1.39	0.82
Usage Freq.	-0.03	0.38	-0.09	0.99
Approvals	0.00	0.00	-0.07	0.99
StudentTrue	-15.75	1370.51	-0.01	0.99

4.2.3 RQa_2: To what extent do functional constructs influence the perceived source code understandability?

Fig. 4.5 shows the perceived understandability of different functional constructs compared to their procedural alternatives. Values on the 1-2 side indicate a strong preference towards the procedural alternative, while 4-5 indicate a better-perceived level of understanding for the functional constructs. The study participants almost always prefer the procedural code, and this is particularly true for lambdas, where only 6% of the participants have a better-perceived level of understanding for the functional alternative. As shown in the RQa_1 results, this is the construct for which an increasing level of complexity makes the changes more error-prone. Comprehension is, instead, the construct for which there is a relatively larger percentage of positive answers (26%). Indeed, as found in the RQa_1 results, when the complexity

Table 4.7 RQa_1: Logistic regression relating the use of reduce with the correctness of the change task (AIC=118)

	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-13.20	1696.36	-0.01	0.99
MainFactorProc	-0.26	0.47	-0.56	0.99
Usage Freq.	-0.83	0.44	-1.87	0.30
Approvals	0.00	0.00	-0.36	0.99
StudentTrue	14.38	1696.36	0.01	0.99

Table 4.8 RQa_1: Logistic regression relating the use of filter with the correctness of the change task (AIC=157)

	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-15.80	1383.44	0.01	0.99
MainFactorProc	-0.06	0.40	-0.16	0.99
Usage Freq.	-0.18	0.33	-0.54	0.99
Approvals	0.00	0.00	-0.76	0.99
StudentTrue	15.56	1383.44	0.01	0.99

increases, the study participants made fewer errors when modifying comprehensions than when modifying their procedural alternatives. The percentage of positives for MRF (18%) is in between lambda and comprehension, with small differences between the three functions. In this case, while the perceived level of understanding appears to be in favor of the procedural alternative, this was not reflected by better performance in the change tasks.

Table 4.9 reports the results of the ordinal logistic regression investigating the relationship between the perceived ease of understanding of the constructs and the construct usage frequency, as well as the constructs' complexity (except for MRF). In all cases, the ease of understanding positively correlates with usage frequency, with statistically significant p-values for lambda and MRF, and marginally significant p-value for comprehension. The complexity is statistically significant, with a negative effect ($OR = 0.77$) for comprehension, while it is not statistically significant for lambda.

RQa_2 Summary: Functional constructs are, in general, perceived as more challenging to understand than their procedural alternatives. Such developers' perception depends on the constructs' usage frequency, as well as the construct's complexity.

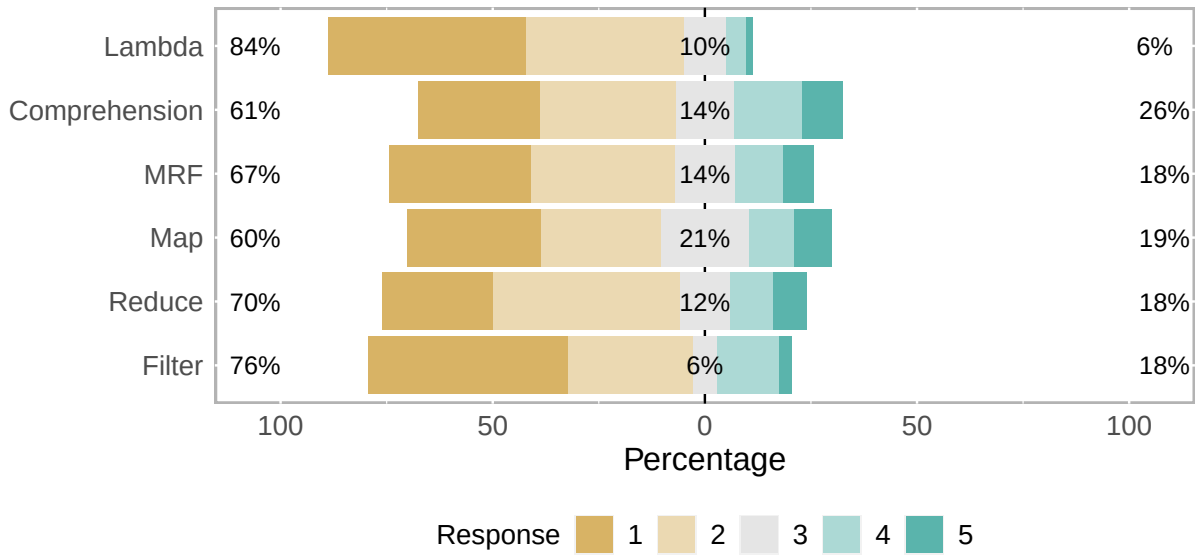

Figure 4.5 Perceived understandability of functional constructs, compared to their procedural alternative (low values indicate a preference towards the procedural)

4.2.4 RQa$_3$: Why do developers use functional constructs, and in which scenarios?

Table 4.10 reports the results of the open coding for what concerns the reasons for using lambdas, comprehensions, and MRF, as well as their procedural alternatives. As it can be seen, the first five reasons are shared across both functional constructs and their procedural alternatives. *Size* is specific for using functional constructs, while *Lack of knowledge*, *Project constraints*, and *Simplify debugging* are specific for preferring their procedural alternatives.

In the following, we provide a discussion of the reasons, together with examples of what is being highlighted by our participants.

Coding time. When implementing or enhancing features, developers tend to opt for the paradigm that ensures better productivity. Specifically, for lambdas, respondents mention the saved time instead of defining a function. Of course, the latter is true in all cases where the function does not require to be used more than once in the code, *i.e.,* P$_{65}$ stated *"it's a waste of time to define a function if all I'm ever going to do with it is pass it to another function once"*. As regards the usage of comprehensions, the save in coding time seems, as reported by P$_{184}$, directly related to the total number of lines to be written *"to gather together a sub-list"*. For MRF, developers motivate their use because as stated by P$_{161}$, using MRF, it is possible to *"effortlessly reuse a function throughout different sections of your code"*. However, deciding whether to use the functional/procedural alternative is a trade-off

Table 4.9 RQa_2: Ordinal logistic regression results of the relationship between perceived understandability of the functional constructs and (i) participants' usage frequency, and (ii) constructs' complexity (except for MRF)

Lambda (90 data points)					
	OR	**Estimate**	**StdError**	**t-value**	**p-value**
Usage Freq.	2.29	0.83	0.31	2.65	**0.02**
Compl.	0.88	-0.13	0.18	-0.73	0.47
Comprehension (120 data points)					
	OR	**Estimate**	**StdError**	**t-value**	**p-value**
Usage Freq.	1.49	0.40	0.22	1.84	0.07
Compl.	0.77	-0.26	0.11	-2.27	**0.04**
MRF (103 data points)					
	OR	**Estimate**	**StdError**	**t-value**	**p-value**
Usage Freq.	1.79	0.58	0.25	2.31	**0.02**

Table 4.10 Reasons for using functional and procedural code

Reason	Lambdas	Comp.	MRF	Proc.
Coding time	11	12	10	2
Ease of use	4	10	8	7
Maintainability	14	9	14	6
Performance	7	28	23	6
Readability/Understandability	19	89	33	27
Size	41	76	39	–
Lack of knowledge	–	–	–	16
Project constraints	–	–	–	5
Simplify debugging	–	–	–	9

between saved coding time and code readability, *i.e., "They save time and don't compromise the readability as much as lambda"*(P_{59}).

As the last column of Table 4.10 shows, coding time may also be a reason for opting towards the procedural alternative. However, by preferring the procedural alternative, reduced coding time may lead to less readable code, hence they may be preferred when nobody else has to look at the same piece of code, *i.e.,* P_1 stated that *" if I am going to be the only looking at my code I care less about how clean it looks. It takes less time to just manipulate variables directly rather than sending them to a function and getting the output."*

Ease of use. Coding time is not the only way to measure developers' productivity. A different proxy can be related to how simple it would be to deal with the construct, taking into account the task at hand. Specifically, for lambdas, P_{194} stated that *"they are much*

easier to write and can usually fit on one line". About comprehensions, participants highlight the simplicity in translating the logic of the feature to be implemented, *i.e.,* P_{133} stated *"the way I think can be easily translated into the code".* For MRF, developers recognize the simplicity in properly applying the same operation *"to all members of a list.'* (P_{120}). Ease of use, is also a reason for opting towards the procedural alternative, *e.g.,* P_{44} reported:*"it's sometimes easier to write code idea straightforward than making it short and clean".*

Maintainability. For what concerns lambdas, developers mention the simplicity in changing them when needed, as well as the reduction of redundant code due to creating functions, *e.g., "they reduce boilerplate"* (P_{31}). The latter also applies to comprehensions, for which P_{195} highlights that *"they can help to reduce code duplication and make the code more maintainable".* Furthermore, P_{180} mentions the reduced defect-proneness *"mak[e] the code more predictable, easier to test, and less prone to bugs."* As for MRF, maintenance becomes easier as one avoids writing the same logic over and over again, and, more importantly, *"already implemented constructs behave better than ad-hoc written constructs"* (P_{75}). However, seven respondents point out maintainability reasons in favor of the procedural paradigm. This is the case in scenarios where it is important to have better control of the code under development, as well as flexibility or customization options, *e.g.,* P_{114} highlighted: *"such as when the operation to be performed on each element of a list depends on complex conditions or involves side effects".*

Performance. Functional programming is mainly intended to avoid side effects, as well as to improve performance [10, 16]. This has been confirmed by our respondents. Specifically, for lambdas, P_{164}mentions that *"They are more efficient, as they can take advantage of the computational resources available on the server and can be executed in parallel".* For comprehensions, P_{108} reports that they *"can be faster than for loops because they are optimized for this use case in the Python interpreter".* However, P_{55} highlights that speed matters only when processing large datasets. This is also true for MRF, for which P_{93} states that *"they're useful when working with large iterable since they perform lazy evaluation, which prevents the program from using more memory than needed."* Despite such developers' perceptions, recent research showed contradictory results in terms of the performance benefits introduced by Pythonic idioms [89]. Finally, five respondents highlight scenarios where procedural may be better for performance. As an example, *"procedural alternatives may be more performant especially for certain types of operations or data structures. This can be important in high-performance or large-scale applications"* (P_{186}).

Readability/Understandability. Readability/understandability may be better for functional constructs, or for their procedural alternatives, based on the task at hand. As an ex-

ample, for comprehensions, P_{96} states that *"they're more elegant and aren't any less readable in simple cases, for instance when the condition is simple"*. For MRF, there are contradictory opinions. On the one hand, they *"express intent better"* (P_{16}). On the other hand, *"they can quickly do the job they are meant to do, however, they are also possibly confusing"* (P_{122}). Furthermore, developers who prefer procedural code mention that functional constructs *"can be more difficult to read and understand than traditional for loops, especially for developers who are less familiar with functional programming concepts"* (P_{55}).

Size. This category relates to choice due to writing shorter code, which could impact productivity, understandability, and maintainability. As expected, this reason arises only when looking at why developers prefer functional constructs. 71 respondents state this for comprehensions, *e.g.,* P_{126} reports: *"List comprehensions are typically a single line of code which can make them easier to read and understand compared to longer multi-line for loops or map/-filter functions."*, yet, also, in this case, there may be trade-offs, *e.g., "it's important for me to use them appropriately and not sacrifice readability for conciseness"* (P_{139}). 35 and 33 respondents mention size as a reason for using lambda, *e.g., "they allow for concise function definitions in a single line of code"* (P_{71}), and MRF, *e.g., "to write more compact code as long as readability doesn't suffer"* (P_{40}).

Lack of knowledge. Sometimes, developers make their decisions based on their level of experience with programming language features. Indeed, 16 respondents admit relying on the procedural paradigm due to *"a lack of familiarity or preference for imperative programming styles"* (P_{143}).

Project constraints. Four respondents state that the decision behind adopting the procedural paradigm is simply dictated by the requirements or constraints of the projects they are contributing. For instance, P_{202} reports that *"non-functional code may be required or preferred in certain environments or domains that do not support functional programming"*, while P_{34} states that *"some projects may require the use of non-functional programming constructs due to compatibility requirements"*. One aspect being highlighted by P_{139} deals with the presence of legacy code, where *"If a project has a lot of legacy code that uses non-functional constructs[,] it may be more practical to continue using those constructs, rather than rewriting the code using functional constructs"*.

Simplify debugging. Seven respondents highlight preferring the procedural paradigm due to the challenges encountered when debugging "functional" code, also because of the lack of suitable tool support. Indeed, by adopting functional constructs, it is hard to trace the flow of data through multiple operations. For instance, P_{135} reports that *"traditional loops or conditional statements may be easier to debug than functional constructs, as they provide*

more visibility into the program flow and state." Furthermore, the difficulty in debugging may become more problematic when sharing code, *e.g., "comprehensions are difficult to debug, so I tend to avoid them when I work in groups"* (P_{172}).

RQa_3 Summary: The main reasons for using functional constructs are writing less code, (perceived) improved performance, and maintainability. However, debugging functional constructs is seen as challenging.

4.3 Implications

This section discusses the study's implications for researchers/tool builders, practitioners, and educators.

Researchers and Tool builders should improve the support systems for writing, testing, and debugging functional constructs. By enhancing the tools available, developers can be empowered to create more robust and efficient code. To this extent, AI-empowered techniques may support the automated completion, error detection, and even repair of such constructs. For what concerns testing, a big gap is represented by the lack of coverage analysis and debugging tools specific for functional constructs, *e.g.,* loops and conditionals in comprehensions, which makes the quality assurance activity more challenging. Another useful recommendation would be to spot possible performance anti-patterns or, in general, bad smells in functional constructs, hence suggesting appropriate refactoring operations [90]. Furthermore, researchers could delve deeper into the realm of study, employing sophisticated techniques such as eye-tracking technology and other monitoring tools. By conducting in-depth analyses, they can gain profound insights into how developers comprehend and use these constructs. This research might not only enrich our understanding of the cognitive processes involved but also pave the way for the design of even more intuitive and user-friendly programming tools.

Practitioners should ponder the adoption of functional constructs, factoring in the unique project requirements, coding styles employed, and the complexity of the task to implement. Developers have a trade-off: on the one hand, as we found in RQa_3, developers may choose the constructs that better reflect their "thinking" about the piece of functionality to be implemented. On the other hand, if a project follows a coding style with fairly limited use of functional constructs, introducing them through refactoring or when adding new code may not necessarily be beneficial, but it may rather harm software understandability. From a different perspective, practitioners should be mindful of the impact on future maintenance activities, especially for newcomers joining the project. Last, but not least, practitioners should per-

form specific testing of functional constructs, *e.g.,* by achieving constructs' (branch) coverage. Also, whenever available, developers should use static analysis tools capable of suggesting potential bugs or refactoring related to such constructs. For example, *Pylint* [91] features some checks specific to lambdas and comprehension (*e.g., unnecessary lambda*, *unnecessary comprehension*, or *consider using comprehension*).

Finally, *Educators* should not only teach the syntax of functional constructs, but also train students about their quality assurance, *e.g.,* through dedicated code inspection and test cases, and understanding. Moreover, it may be desirable to outline and discuss different development scenarios in which such constructs could be useful or, instead, counterproductive, negatively affecting program comprehension.

4.4 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. The main threat is related to how we measure the understandability of the functional constructs against their procedural alternatives. We have chosen to assess the understandability during a change task (RQa_1), as was done in previous research [23, 81, 82], as well as to ask about the *perceived understandability* in a comparative assessment (RQa_2). It is possible that what we measure might not reflect the achieved level of the construct's understanding. To mitigate this threat, we correlated, for each construct, the *perceived* understandability in terms of ability to perform a change task (RQa_1) with the perceived understandability as declared for the comparison task (RQa_2) using the Kendall's Tau correlation [92]. For all functional constructs, results are never statistically significant (p-value=0.4 or greater), with a negligible correlation coefficient (< 0.05). Based on this analysis, we can state that the perceived understandability does not reflect the correctness of the change task.

Threats to *internal validity* concern factors that could have influenced our results. We controlled for factors such as constructs' complexity, as well as participants' demographics, *i.e.,* the declared level of usage of the construct, the number of approvals on Prolific, and whether the participant is a student. We are aware that there could be other factors, beyond our control, that could influence our results. However, we did our best in keeping the functional code as close as possible to the original code, and translating each code snippet into the equivalent procedural alternative with the same length, complexity, and same identifiers (but loop variables). Finally, for RQa_3, we used GPTZero [84] to check for AI-generated answers, yet, given the study settings, we could not exclude that respondents leveraged the Web to provide us insights.

Threats to *conclusion validity* concern the relationship between the experimentation and outcome. To answer our research questions, wherever appropriate, we used tests suitable to the nature of the observed data (*i.e.,* logistic models, as well as ordinal logistic models). Since multivariate models involve multiple factors, p-values are adjusted with the Benjamini-Hochberg correction [87]. Finally, we checked the reliability of our qualitative analysis by employing a suitable inter-rater agreement measure, *i.e.,* Krippendorff's α [86].

Threats to *external validity* concern the generalizability of our findings. The main threat is due to the representativeness of Prolific workers as real software developers. We have tried to mitigate this threat as much as possible with (i) participants' pre-screening, and (ii) several filters on the collected results. Nevertheless, replications in other contexts, *e.g.,* classrooms or industrial settings, are desirable. In particular, experienced Python developers, well used to such constructs, may exhibit different performance. Truly, it might be difficult to recruit a large number of such participants, and this was our primary rationale for choosing Prolific [72].

4.5 Conclusion

This chapter reported the results of a controlled experiment in which we studied the effect of some functional Pythonic constructs—and specifically lambdas, comprehensions, and map/reduce/filter (MRF)—on program comprehension. The experiment was conducted with 209 paid developers recruited from *Prolific* and consisted of change tasks, comparative assessment of the perceived understandability, and questions about the developers' reasons for (not) using such constructs. Results of the study show that:

1. Developers using lambda better performed their change tasks than when using the procedural alternatives, although this is no longer true when complexity increases. The opposite happens for comprehensions, which are generally more error-prone than their procedural alternatives. However, when complexity increases, comprehension becomes advantageous, likely because they do not scatter complexity. No statistically significant differences were found for MRF.

2. Developers generally perceive functional constructs as more challenging to understand than their procedural alternatives. However, this naturally depends on the usage frequency of such constructs and, for comprehension, on the constructs' complexity.

3. While developers acknowledge code size reduction as a pro for using functional constructs (without affecting maintainability), they still face challenges when debugging code following the functional paradigm.

In summary, given our participants (with mainly junior or medium-level seniority), results tell that the difficulty of understanding and changing the studied constructs may vary from one construct to the other, and also depend on the constructs' complexity.

4.6 Data Availability

The replication package [93] contains (i) the questions and code examples used in the experiments, (ii) the questionnaire forms, (iii) detailed results of the qualitative analysis, and (iv) the study results and the R script to analyze them.

CHAPTER 5 AN EMPIRICAL STUDY ON THE FAULT-INDUCING EFFECT OF FUNCTIONAL CONSTRUCTS IN PYTHON

This chapter delves into the results of a comprehensive investigation conducted on 200 open-source Python projects, comprising approximately 630,000 commits, to understand the relationship between changes involving functional constructs—specifically lambdas, comprehensions, and map/reduce/filter functions—and the likelihood of inducing fixes. Additionally, the study explores how the complexity of these constructs and the survival time of resulting bugs contribute to the overall fault-proneness.

Functional programming, primarily designed to prevent functions from causing side effects on data structures, has gained prominence for its potential benefits in performance-critical scenarios and bug reduction [10,16]. While some languages are purely functional, many allow a mix of functional and non-functional constructs, introducing a choice for developers that may impact both performance and code understandability [13, 14, 94–96].

This investigation focuses on functional constructs in Python, a language renowned for its popularity, particularly in data science and machine learning applications. Python's prevalence on GitHub further underscores its significance in contemporary software development [97]. The study concentrates on three key functional constructs: lambda functions, list/dictionary/set comprehensions, and map/reduce/filter functions. These constructs were chosen due to their frequent occurrence in Python programs [12] and their refactoring potential [89, 90, 98].

While functional constructs offer anticipated benefits, the effectiveness of their utilization varies among developers, especially in languages traditionally dominated by imperative constructs [15, 18]. This chapter investigates whether changes involving functional constructs have higher chances of inducing fixes, the nature of such fixes (addition or modification), the varying likelihood among different functional constructs, the influence of McCabe cyclomatic complexity on fault-proneness, and the survival time of bugs associated with different constructs. The study combines a quantitative analysis of 633,803 commits with a qualitative examination to gain a holistic understanding of scenarios where functional constructs lead to bug fixes.

Results demonstrate that changes to functional constructs are 1.15 times more likely to directly induce fixes than other changes, with some constructs—particularly lambdas and comprehensions—exhibiting higher odds of inducing fixes than others. Surprisingly, the complexity of functional constructs does not significantly impact fault-proneness. The survival

time of bugs introduced by different functional constructs varies, providing insights into their respective debugging challenges. Qualitative analysis aligns with quantitative findings and further highlights instances where functional constructs were refactored into non-functional ones.

The outcomes of this study contribute valuable insights for the development community, guiding the improvement of Integrated Development Environments (IDEs) and program analysis tools to better support developers employing functional constructs. Furthermore, it underscores the importance of prioritizing code review and testing activities for sections of code involving functional constructs, acknowledging their potential intricacies.

Structure of the Chapter: Section 5.1 details the study's design and planning, while Section 5.2 presents and discusses the results. The qualitative analysis and implications are discussed in Section 5.3, and potential threats to the study's validity are outlined in Section 5.4, and finally, Section 5.5 concludes the chapter and suggests possible avenues for future research.

5.1 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 higher chances to induce fixes than other changes, whether their complexity has an impact on such chances, and how long bugs introduced in different construct types survive over time.

The *quality focus* is the software fault-proneness, which may be influenced by the use of such programming constructs and their complexities. 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 aims to address the following five research questions:

- **RQb$_1$:** *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.
- **RQb$_2$:** *How do the odds to induce fixes vary between additions of functional constructs and their modifications?* We investigate whether changes adding functional constructs are more likely to induce fixes than changes simply modifying them. The conjecture we would like to verify is whether functional constructs tend to be "born fault-prone" and whether changes made to them still induce further bugs.

- **RQb$_3$:** *How does the fault-proneness vary among different types of functional constructs?* This research question aims at verifying whether developers may experience different levels of difficulty when using different types of functional constructs, namely lambdas, comprehensions, and map/reduce/filter functions.
- **RQb$_4$:** *Is there a relation between the cyclomatic complexity of functional constructs and their fault-proneness?* We investigate whether the fault-proneness of different functional constructs varies when considering their complexity. The conjecture we would like to verify is whether the fault-proneness of the functional constructs increases, as soon as their complexity increases.
- **RQb$_5$:** *What is the survival time of bugs involving different functional constructs?* Finally, we investigate how long bugs involving functional constructs remain in the code by studying their survivability. Specifically, we verify whether different functional constructs impact such survival time.

5.1.1 Context Selection

We answer our research questions by mining the change history of 200 open-source Python projects hosted on GitHub, selected as follows. We relied on an existing dataset of engineered software projects, made available by [3], by selecting only the ones mainly written in Python. For each project, we used the GitHub API to extract metadata 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.

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: the project (i) is not a tutorial, book code, or code examples, (ii) has its commit history and issues descriptions mostly written in English, and (iii) has at least one commit in the last year (from January to December 2021). The selected 200 projects have a size (in KLOC) between 0.09 and 1,208 (median 24). The complete list of projects is available in our online appendix [99].

5.1.2 Data Extraction

Once repositories of the projects have been cloned, we analyzed their evolution history to identify (i) changes to functional constructs, and (ii) bug fixes, together with their fix-inducing changes. Furthermore, to address RQb$_4$, for each fix-inducing change dealing with a functional construct, we computed the cyclomatic complexity of the construct before the fix. Note that, the analysis has been performed by considering all commits of all branches, excluding merge commits.

Analysis of Changes to Functional Constructs

One option to analyze Python changes would have been to leverage *Gumtree* [100]. Although this is the state-of-the-art for AST-based, fine-grained differencing analysis, for Python it leverages the *pythonparser* module [101], 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 [5], which was used in our work. We parsed each Python file and then traversed its tree extracting relevant nodes. Each AST node is decorated with information including the beginning and ending line/column. Furthermore, each call node, *i.e.,*`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 comprehensions and lambdas are explicitly represented as AST nodes, *i.e.,* the AST features the `ast.DictComp, ast.ListComp, ast.SetComp` and `ast.Lambda` nodes. `Map, reduce, and filter` functions, instead, are call nodes (`ast.Call`) whose `id` is equal to `map`, `reduce`, or `filter`.

The extraction process involved the entire set of Python files modified in each commit. Specifically, for each file, we recorded the file name, the commit hash along with its date, and the identified functional constructs, including their locations (*i.e.,* starting and ending lines). For each commit, we parsed the modified Python files in their version before and after the commit. If a file was added, all identified functional constructs within it are treated as added. Similarly, if a file was removed, its functional constructs were considered 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. For each functional construct within a modified code churn, considering the initial and final locations of both the functional construct and the code churn in the two file versions, we examine the following three scenarios:

- If the construct is present in both versions and its lines have been changed, we conclude the functional construct is ***modified***;
- If the construct is present solely in the "before" version, we consider it as ***deleted***; and
- If the construct is present solely in the "after" version, we consider it as ***added***.

Note that the approach is not able to distinguish semantic-altering changes from semantic-preserving ones, *e.g.,* refactoring. We did not want to exclude refactoring actions from the considered changes, as previous research has found that also semantically irrelevant

changes [102] might induce fixes [49, 103]. In our qualitative analysis, presented in Section 5.3.3, we discuss a sample of such cases.

Table 5.1 Descriptive statistics of the type of changes involving the functional constructs object of the study

Construct	Type of change	Q1	Median	Q3
Lambdas	Addition	18	63	171
	Modification	8	27	135
	Removal	5	18	52
	All	37	168	468
List comp.	Addition	34	99	281
	Modification	18	66	241
	Removal	12	32.5	90
	All	74	237	745
Dictionary comp.	Addition	0	2	10
	Modification	0	0	3
	Removal	0	0	2
	All	0	6	28
Set comp.	Addition	0	0	0
	Modification	0	0	0
	Removal	0	0	0
	All	0	0	2
Map	Addition	2	9	31
	Modification	0	4.5	21
	Removal	0	2.5	9
	All	3	20	78
Reduce	Addition	0	0	0
	Modification	0	0	0
	Removal	0	0	0
	All	0	0	2
Filter	Addition	0	2	5
	Modification	0	0	3
	Removal	0	0	1
	All	0	4	13

Table 5.1 reports descriptive statistics (first, second, and third quartile) of changes involving functional constructs in the studied projects. Note that the "All" line does not discriminate between the type of change, *i.e.,* addition, modification, and removal. The table shows that changes related to some constructs, such as `reduce`, `filter`, set, and dictionary comprehensions, are rare.

Analysis of Fix-Inducing Changes

The fix-inducing analysis has been performed by leveraging a lightweight version of the SZZ algorithm [6], similar to the approach taken in previous studies that explored the connection between code naturalness and defects [47]. This is because, differently from Java—where other authors have identified curated projects properly handling defects through an issue tracker [49]—we noticed that Python projects do not consistently use the issue trackers, and often contain fixes not even linked to the issue tracker. We could have limited our attention to the subset of fixes explicitly linked, but this could have biased the dataset [104]. For this reason, we decided to rely on a simpler, lightweight identification of fix commits, *i.e.,* commits whose message matches the following regular expression:

$$\texttt{"}\backslash b(fix|fixed|fixing|fixes|bugs?)\backslash b\texttt{"}\ ^{1}.$$

Once fix commits have been identified, our analyzer determines their fix-inducing commits by leveraging *PyDriller* [105] features and the `git blame` (applied to the lines affected by the fix). In doing so, we discard "cosmetic" changes, *i.e.,* changes involving white spaces and/or comments only. Through the `git blame -p` (porcelain option), we were also able to handle file renaming. To sum up, for each changed line belonging to fixed Python files, we obtain a candidate introduction location, *i.e.,* commit, file name, and line number.

Combining Functional Changes and Fix-Inducing Changes

To address the five research questions, we combine the two aforementioned analyses, *i.e.,* changes to functional constructs and fix-inducing changes. Specifically, for each changed code churn in a commit, we identify whether (i) it induces a fix, and if yes, which lines of the churn induce the fix, (ii) the churn contains a change to functional constructs, and if yes, which ones, and whether the construct was added or modified, and (iii) the functional construct change and the fix-inducing change occur on the same line.

5.1.3 Analysis Methodology

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

To address RQb_1, we compare the proportion of fix-inducing changes involving functional constructs and those not involving functional constructs. The latter has been done following

[1] $\backslash b$ stands for matching word boundaries.

Table 5.2 Descriptive statistics of cyclomatic complexity for the studied functional constructs

Construct	Min	Q_1	Median	Q_3	Max
Lambdas	1	1	1	1	7
List comp.	2	2	2	3	11
Dictionary comp.	2	2	2	3	9
Set comp.	2	2	2	3	6
Map	1	1	1	1	4
Reduce	1	1	1	2	3
Filter	1	1	1	1	6

a methodology similar to a previous work analyzing the extent to which refactoring induces a fix [49], except that we operate at the granularity of changed churns, *i.e.,* consecutive lines changed in a source code file. Specifically, we divide the changed churns into:

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;
4. The churn does not affect a functional construct and does not induce a fix.

Then, we compare the proportions of (1) over (2) and (3) over (4) using Fisher's exact test [106], 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 changed churn, *i.e.,* 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 code churn (*i.e.,* while one changed a functional construct, the fix has been induced). After that, 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 a functional construct and induced a fix on the same line where the functional construct was 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 to address RQb_1, there could be confounding factors influencing the odds a change has to induce a fix. Among others, we consider the changed churn size, since [41] pointed out that a large change has higher odds to induce a fix. Of course, there could be further factors that we do not consider as part of this study, as discussed in Section 5.4. To consider this effect, we build a binominal, mixed-effect generalized linear model. This model considers the relationship between the dependent variable (whether a change induces a fix), and the independent variables, *i.e.,* the churn size, whether the change affected a functional construct, along with the interaction between the two independent variables. The model considers the project 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 lme4* package [71]. Since the mixed model involves multiple comparisons, its p-values have been adjusted using the Benjamini-Hochberg correction [87].

To address RQb_2, we look at whether a changed churn has involved (i) the addition of new functional constructs, and/or (ii) the modification of existing functional constructs. Similarly to RQb_1, we leverage a mixed-effect model to determine whether a fix-inducing change is correlated with additions, modifications, and their interaction with the changed churn size. In this context, controlling the effect of size is particularly important, as additions may tend to be larger than simply modifications.

For RQb_3, we analyze how the fault-proneness varies among different types of functional constructs. To this aim, we categorize functional changes into three types: (i) affecting lambdas, (ii) affecting comprehensions, and (iii) affecting map/reduce/filter functions. 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 changed churn involved any of the three types of functional constructs.

To address RQb_4, we analyze the variability in fault-proneness of functional constructs across various complexity levels by creating distinct mixed-effect models, one for each functional construct. As this research question explores the variation in fault-proneness odds with increasing complexity levels, we only consider the changes involving the investigated functional constructs. The dependent variable in the models represents whether a change to a functional construct, *i.e.,* lambda, comprehension, or map/reduce/filter functions, is fix-inducing, while the independent variables are:

- The McCabe's cyclomatic complexity of the functional construct before the fix.
- The size of the changed churn.

- For map/reduce/filter functions, we also consider the involvement of lambdas as the invoked function, aiming to explore potential interactions between lambdas and the invocations of the map/reduce/filter functions.

We conjecture that the previous analysis is unlikely to reveal a significant effect size, as the complexity of the identified constructs is expected to be low, and in any case, bound to relatively small values, as shown in Table 5.2. In other words, as the estimated odds would refer to a unitary increase of complexity, that would be very low because of the limited bounds of such complexity. For this reason, we divide the constructs into two groups, those having a *high complexity*, when the complexity is greater than the third quartile of the distribution, and the remaining ones, having a *low complexity*. Then, we perform a Fisher's exact test comparing the fault-proneness of constructs having low and high complexity. More specifically, the test is conducted considering a confusion matrix with two dimensions, *i.e.,* on the one side, changes to a given type of functional construct inducing a fix or not, and, on the other side, changes to functional constructs with low and high complexity.

Finally, as it was done in studies analyzing how long bugs [107] and code smells [108] tend to survive in a software project, to address RQb_5, we rely on a statistical method called survival analysis [109], *i.e.,* a method aimed at analyzing the time duration until one or more events happen, on fixes involving functional constructs, as well as on fixes not containing any functional construct. A survival model consists of:

1. An *hazard function*, $h(t)$, representing the risk that an event occurs for a time delta, as compared to the hazard of a baseline of interest $h_0(t)$;

2. The *effect parameters*, representing the hazard for specific covariates (*i.e.,* independent variables).

In our context, we use as a proxy for the longevity of a bug the number of days (*i.e.,* survival time) between the fix-inducing change and the corresponding bug fix commit. We perform survival analysis using the Cox proportional hazard model [109]. Specifically, given the values of the N covariates $X_i = (X_{i1}, \ldots, X_{iN})$ for a subject i, the model estimates the coefficients $\beta_1, \ldots \beta_N$ for the function:

$$h(t) = h_0(t) \cdot e^{\beta X}$$

The term $h_0(t)$ corresponds to the risk when covariates are equal to zero. Note that the exponential of a parameter β_j is the hazard rate, the change in hazard when the covariate $X_{,j}$ increases by one. Specifically, considering two individuals, *i.e.,* X and Y, their hazard

ratio is independent of time t, and just depends on the covariates levels:

$$\frac{h_{Y(t)}}{h_{X(t)}} = \frac{e^{\beta_1 \cdot Y_1 + \ldots \beta_N \cdot Y_N}}{e^{\beta_1 \cdot X_1 + \ldots \beta_N \cdot X_N}}$$

In other words, if we set the levels of all covariates but the $j-th$ explanatory variables to zero, we observe that the hazard rate between Y and X is equal to $e^{\beta_j \cdot (Y_j - X_J)}$, and thus for a unit of change of the risk factor it corresponds to e^{β_j}.

$$\frac{h_{X_{ij}}(t)}{h_{X_0}(t)} = \frac{h_0(t) \cdot e^{\beta_j}}{h_0(t) \cdot e^{\beta_1 \cdot 0 + \ldots \beta_N \cdot 0}} = e^{\beta_j}$$

Other than building Cox proportional regression model using the type of construct as covariate, we use Kaplan-Meier curves [109] to visualize the display hazard ratios—computed as the exponential of the covariate coefficients β_j—over time, as well as forest plots [109] showing the confidence intervals of covariate hazard ratios. Covariates are retained if p-value < 0.05. To create the Cox regression models and to plot the Kaplan-Meier curves and forest plots, we use the *survival* [110] *R* package, while we rely(p on the *prodlim* [111] *R* package to extract the median survival times, representing the point when the population is reduced by 50%.

5.2 Study Results

This section discusses the results for the five research questions defined in Section 5.1.

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

To address RQb_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 5.3 reports the results of Fisher's exact test together with the ORs and their confidence interval. An OR greater than one indicates that a commit where a functional construct is added or modified has higher odds than other commits of inducing a bug. As shown in the first line of Table 5.3, the commits adding or modifying a functional construct have significantly higher odds ($OR = 2.23$) of inducing a bug than other changes. If we consider that a functional construct induces a bug only if a line where it has been added/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 of being directly responsible for fixes than other changes. The higher OR for the churn level may indicate a high proneness of the code surrounding functional constructs (regardless of the specific line) to induce fixes.

Table 5.3 RQb_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 5.4 RQb_1: Mixed-effect logistic regression relating changes to functional constructs, size of the changed churn, 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**

Despite that, it is still possible that what we observed 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 correlate with fix-inducing changes. Table 5.4 reports the results of the binomial mixed-effect logistic regression, considering functional changes, changed churn sizes, and their interaction as independent variables while the project is the random effect. Results point out that the change to functional construct, the changed churn 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 changed churn size

increases the odds by $e^{0.12} = 1.13$. In other words, a change involving $\simeq 6$ lines would have the same odds of inducing a fix as a single change to a functional construct.

RQb_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 fixes.

5.2.2 *RQb_2:* How do the odds to induce fixes vary between additions of functional constructs and their modifications?

Table 5.5 RQb_2: Mixed-effect logistic regression relating the type of change to functional constructs (addition/modification) 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**
Modified Func.	0.24	0.02	13.73	**<0.001**

Table 5.5 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 fault-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.

Table 5.6 RQb_2: Mixed-effect logistic regression relating the type of change to functional constructs (addition/modification), 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**
Modified Func.	0.70	0.03	22.73	**<0.001**
Added Func:Churn Size	0.02	0.01	1.54	0.1510
Churn Size:Modified Func.	-0.11	0.01	-9.79	**<0.001**

Similarly to what has been done for answering RQb_1, we have evaluated the effect of the changed churn size. As shown in Table 5.6, only the change to existing functional constructs, the changed churn size, and their interaction have a statistically significant effect on the likelihood that the change is inducing a fix. In particular, the change 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 $e^{0.12} = 1.13$. Furthermore, 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.

RQb_2 Summary: Introducing new functional constructs is more likely to result in fixes compared to modifications to existing constructs. However, such a higher effect is captured by the changed churn size for changes adding new functional constructs.

Table 5.7 RQb_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**
Lambdas	0.98	0.02	58.40	**<0.001**
Comprehensions	0.66	0.02	43.75	**<0.001**
Map/Reduce/Filter	0.28	0.04	7.19	**<0.001**

5.2.3 *RQb_3:* How does the fault-proneness vary among different types of functional constructs?

Table 5.7 shows the results of the mixed-effect logistic regression model performed to check the extent to which the fault-proneness of a change varies among different types of functional constructs. 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 constructs, regardless of their interaction with other factors. As the table shows, all functional constructs have a statistically significant correlation with fix-inducing changes. Changes dealing with lambdas have the highest odds of inducing a fix (*i.e.,* $OR = e^{0.98} = 2.66$), followed by changes handling 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 (see Table 5.1). Although the latter needs to be further investigated, it may be that lambdas and comprehensions have a more complex syntax that makes bug introduction easier.

RQb_3 Summary: The fault-proneness of changes involving functional constructs varies based on the type of functional construct, with changes handling lambdas having the highest odds to induce a fix, followed by comprehensions.

Table 5.8 RQb_4: Mixed-effect logistic regression relating the complexity and size of lambdas with fix-inducing changes

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**
39477.61	39517.25	-19734.81	39469.61	148546.00
SCALED RESIDUALS:				
Min	**1Q**	**Median**	**3Q**	**Max**
-0.94	-0.16	-0.15	-0.15	22.98
RANDOM EFFECTS:				
Groups			**Variance**	**Std.Dev.**
Project (Intercept)			3.72	1.93
Number of obs: 148,550, Groups: Project: 191				
FIXED EFFECTS:				
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-4.10	0.19	-22.16	$<$ **0.001**
Churn Size	-0.00	0.00	-1.48	0.14
Complexity	-0.15	0.08	-1.89	0.059

5.2.4 *RQb_4:* Is there a relation between the cyclomatic complexity of functional constructs and their fault-proneness?

The results obtained from RQb_1 to RQb_3 suggest how functional constructs are more prone to induce fixes. In this research question, we investigate the impact of their complexity on fault-proneness. As explained in Section 5.1, we only consider changes involving a given functional construct, and therefore, odds to induce fixes correlated with an increase of complexity. Tables 5.8, 5.9 and 5.10 show the results of the mixed-effect logistic regression models performed to check the extent to which the fault-proneness of lambdas, comprehensions, and map/reduce/filter functions are explained by constructs' complexity and churn size.

For lambdas (see Table 5.8), the effect size for churn size is negligible and not statistically significant, and, surprisingly, the complexity coefficient is negative, meaning that a unit increase of complexity reduces the odds of the construct to induce fixes (OR $= e^{-0.15} = 0.86$). However, it is only marginally significant (p-value=0.059). Given that 75% of lambdas have a complexity of one, we conjecture that developers may focus on introducing or modifying lambdas with higher complexity.

Table 5.9 reports the mixed-effect logistic regression model for comprehensions. As for lambdas, the churn size plays a negligible role, with a negative coefficient for complexity (OR $= e^{-0.17} = 0.84$). While it is unlikely that complexity has a "protective" role, we conjecture that, since low complexity constructs are the vast majority (75% of comprehensions have a complexity of three or less), developers may pay less attention to them, hence they induce

Table 5.9 RQb_4: Mixed-effect logistic regression relating the complexity and size of comprehensions with fix-inducing changes

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**
36620.08	36659.30	-18306.04	36612.08	133885.00
SCALED RESIDUALS:				
Min	**1Q**	**Median**	**3Q**	**Max**
-1.04	-0.21	-0.12	-0.08	22.38
RANDOM EFFECTS:				
Groups			**Variance**	**Std.Dev.**
Project (Intercept)			2.54	1.59
Number of obs: 133,889, Groups: Project: 198				
FIXED EFFECTS:				
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-3.39	0.15	-23.30	$<$ **0.001**
Churn Size	-0.00	0.00	-21.46	$<$ **0.001**
Complexity	-0.17	0.03	-5.55	$<$ **0.001**

more fixes than those with high complexity.

Finally, Table 5.10 shows the model for the map, reduce, and filter functions. These constructs are analyzed together due to their commonality—they all iteratively apply a function over a collection—and the relatively small number of instances. For these constructs, we consider an extra covariate, namely the use of a lambda function instead of a function call, *i.e., lambdaFalse* indicates the covariate level where a function call is used instead of a lambda. Similar to previous cases, the effect size for churn size is negligible, and complexity's significance is marginal (p-value=0.085) with an $OR = e^{-0.24} = 0.79$. This is unsurprising since the complexity is computed considering the lambda function being used.However, if a lambda function is not involved, the odds of inducing fixes decrease. Indeed, as the table shows, the absence of lambdas in the invocation reduces the odds of inducing fixes, *i.e.,* the OR is $e^{-0.84} = 0.43$.

Given the complexity distributions shown in Table 5.2, we can notice how the largest majority of the studied constructs has a cyclomatic complexity of one or two. While mixed-effect logistic regression models show a non-obvious correlation of fix-inducing changes with cyclomatic complexity, the modeling of complexity as an integer variable prevents us from discerning the distinct effects between high and low complexity. Therefore, as explained in Section 5.1, we decided to divide constructs into those with a low complexity (less than, or equal to the third quartile), and those with a high complexity (greater than the third quartile). As shown in Table 5.11, for lambdas, the odds for inducing fixes in case of high complexity are

Table 5.10 RQb_4: Mixed-effect logistic regression relating the complexity and size of map/reduce/filter functions with fix-inducing changes

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**
4543.78	4583.07	-2266.89	4533.78	19107.00
Scaled residuals:				
Min	**1Q**	**Median**	**3Q**	**Max**
-1.27	-0.18	-0.10	-0.05	15.17
Random effects:				
Groups			**Variance**	**Std.Dev.**
Project (Intercept)			2.27	1.51
Number of obs: 19,112, Groups: Project: 169				
Fixed effects:				
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-3.03	0.23	-13.28	$<$ **0.001**
Churn Size	-0.00	0.00	-7.87	$<$ **0.001**
Complexity	-0.24	0.14	-1.72	0.085
lambdaFalse	-0.84	0.10	-8.64	$<$ **0.001**

0.63 of those for low complexity. Nevertheless, even in this case, the p-value is only slightly significant. Similar results (OR=0.79) are obtained for list comprehensions (see Table 5.12), this time with a statistically significant p-value. The results are similar for dictionary comprehensions (OR=0.83), yet the p-value is not significant. Instead, as Table 5.13 shows, for set comprehensions, we have an odd ratio greater than one ($OR = 2.38$). Similar results are obtained for the `filter` function (see Table 5.13, OR=3.22), while for the `map` function, although the OR is 4.39, the results are not statistically significant (p-value=0.17). Finally, for the `reduce` function (Table 5.15), the OR cannot be computed, as fix-inducing changes only occur for low-complexity cases.

Table 5.11 Contingency table and Fisher's exact test results when discretizing complexity of lambdas

	Low Complexity	**High Complexity**
Not fix-inducing	142,526	721
Fix-inducing	5,286	17
Fisher's exact test	p-value: 0.08	OR=0.63 [0.37 - 1.03]

Table 5.12 Contingency table and Fisher's exact test results when discretizing complexity of list comprehensions

	Low Complexity	**High Complexity**
Not fix-inducing	126,263	2,556
Fix-inducing	4,991	79
Fisher's exact test	***p*-value: 0.03**	OR=0.79 [0.62 - 0.98]

Table 5.13 Contingency table and Fisher's exact test results when discretizing complexity of set comprehensions

	Low Complexity	**High Complexity**
Not fix-inducing	2,790	91
Fix-inducing	91	7
Fisher's exact test	***p*-value: 0.04**	OR=2.38 [0.89 - 5.26]

RQb_4 Summary: The effect of cyclomatic complexity on fault-proneness for changes involving functional constructs varies based on construct type. When discretizing complexity into two levels (high and low), we found that, except for set comprehensions and filter functions, complexity does not have an adverse role. Finally, changes involving map, reduce, and filter functions also containing lambdas have the highest odds of inducing a fix.

5.2.5 *RQb_5:* What is the survival time of bugs involving different functional constructs?

RQb_5 explores the time detected bugs remain in the system through survival models, namely the Cox proportional hazard regression. The covariates are the different types of functional constructs. Given the results of RQb_4, we do not include complexity and churn size among covariates. Time is measured in days, and, as explained in Section 5.1, only constructs whose churns underwent a fix are part of the analysis, because we cannot know that there is a bug if we do not observe a fix.

Table 5.14 Contingency table and Fisher's exact test results when discretizing complexity of filter functions

	Low Complexity	**High Complexity**
Not fix-inducing	8,013	164
Fix-inducing	187	12
Fisher's exact test	***p*-value: <0.001**	OR=3.12 [1.56 - 5.88]

Table 5.15 Contingency table and Fisher's exact test results when discretizing complexity of reduce functions

	Low Complexity	**High Complexity**
Not fix-inducing	974	113
Fix-inducing	41	0
Fisher's exact test	***p*-value: <0.001**	

Table 5.16 reports the estimates of the Cox model for different construct types. as well as for fix-inducing changes on churns not including the studied functional constructs (labeled as *other changes*). Note that, as done in the second part of RQb_4, to better analyze the contribution of different constructs, we consider different types of comprehensions (list, dictionary, and set comprehension) as well as map, reduce and filter separately. The constructs with statistically significant p-values are lambdas, list and set comprehensions, and (marginally significant, with a p-value of 0.08) reduce. Of these constructs:

1. Set comprehensions have a hazard rate greater than one (1.642), *i.e.,* bugs involving churns with this construct get fixed quicker than other bugs;
2. Lambdas also have a hazard rate greater than one, yet the effect is relatively smaller (1.167);
3. List comprehensions have a hazard rate slightly smaller than one (0.930), *i.e.,* bugs involving churns with this construct get fixed slightly slower than other bugs;
4. Reduce has a hazard rate smaller than one (0.761), indicating a slower fixing time, even though, as said above, the p-value is only marginally significant.

Fig. 5.1 displays forest plots showing the confidence intervals for hazard rates obtained from Table 5.16. The corresponding Kaplan-Meier plots are shown in Fig. 5.2. The figure also shows, as reference (hazard rate=1) the *other changes* not involving these constructs. From the plot, it can be noticed that reduces and set comprehensions play opposite roles, as discussed above, *i.e.,* they are placed on the opposite side of the reference value of one. This is also intuitively confirmed by the Kaplan-Meier plots of Fig. 5.2 which shows how the curve for reduce slowly decays, whereas a quick drop occurs for set comprehensions.

The median survival times are reported in Table 5.17 together with their interquartile ranges. We observe median survival times are spread evenly, with a minimum of (about) one or two months (filter and set comprehensions respectively) and a maximum of 18 months. Note that the population size for certain constructs is small, and, strictly speaking, hazards are

Table 5.16 RQb_5: Cox detailed hazard model for the type of functional construct, including other changes

Variable	**Exp(Coef)**	**Confidence Intervals**	***p*-value**
Lambdas	1.167	[1.14 - 1.20]	< **0.001**
List comp.	0.930	[0.90 - 0.96]	< **0.001**
Dictionary comp.	1.078	[0.97 - 1.20]	0.156
Set comp.	1.642	[1.35 - 2.00]	< **0.001**
Map	1.026	[0.93 - 1.13]	0.597
Reduce	0.761	[0.56 - 1.03]	0.08
Filter	1.184	[1.03 - 1.36]	**0.017**
Concordance = 0.504 (se = 0)			
Likelihood ratio test = 177.693 on 7 df			< 0.001
Wald test = 186.3 on 7 df			< 0.001
Score (logrank) test = 187.113 on 7 df			< 0.001

Table 5.17 Median time and inter quartile range (IQR) for different functional constructs

Construct	**Median surv. time**	**IQR**(25^{th} , 75^{th})
Lambdas	94.17	[15.96, 566.16]
List comp.	239.64	[12.89, 739.05]
Dictionary comp.	240.25	[9.72, 692.87]
Set comp.	67.36	[8.69, 225.02]
Map	183.94	[31.32, 588.67]
Reduce	547.78	[18.10, 1268.88]
Filter	31.32	[3.16, 443.82]
Other changes	192.14	[19.85, 665.73]

not constant over time. This latter can be noticed because the Kaplan-Meier curves for different constructs (*e.g.,* filter and dictionary comprehensions) intersect. However, there may be other factors beyond those we considered (type of construct, cyclomatic complexity, and churn size) that play a role in influencing the survivability of a bug over time.

RQb_5 Summary: The hazard rate for the fault-proneness of changes involving functional constructs varies based on the type of construct, with a negligible effect of complexity, and no effect on the churn size. Filter and set comprehensions exhibit the highest hazard rates, whereas reduce the lowest ones. The median survival time of reduce is eight and 16 times greater than for set comprehensions and filter respectively.

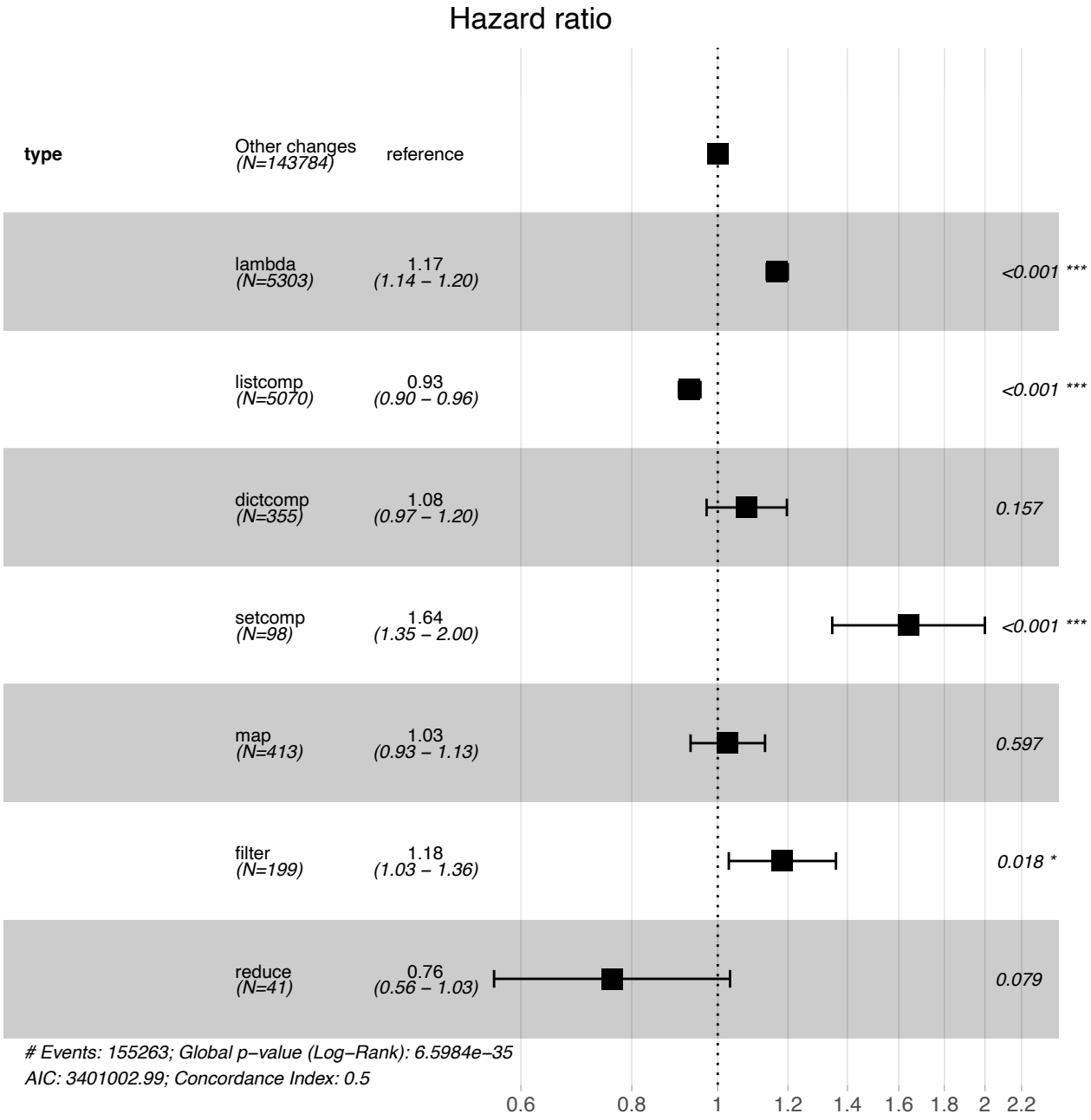

Figure 5.1 RQb_5: Forest plots for the model of Table 5.16. The area on the right (left) of the dashed vertical line defines a decrease (increase) of the risk.

5.3 Qualitative Analysis and Implications

In the following, we report the results of a qualitative analysis of a sample of the studied fixes, a discussion of some examples concerning the survival time, as well as the study's

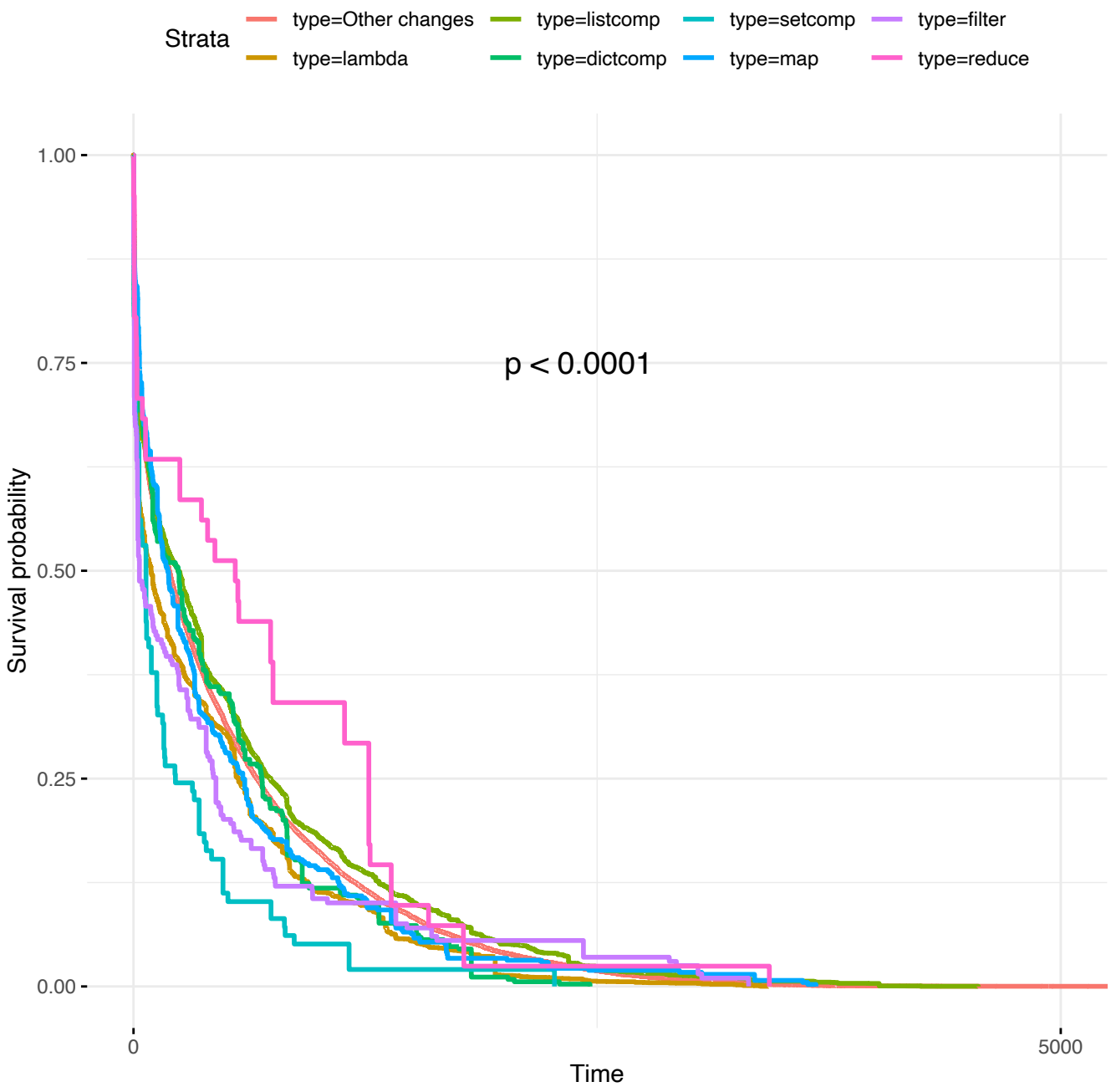

Figure 5.2 RQb$_5$: Kaplan-Meier plot - the *additional* strata represent the baseline, *i.e.,* fixes to churns not involving functional constructs.

implications for developers, educators, and researchers.

Table 5.18 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/d4ee5495	Issue #51: chart displaying dates into the future ...This is also handled with ...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.

5.3.1 Qualitative Analysis

This qualitative analysis complements the quantitative results reported in Section 5.2 and has a three-fold goal: (i) validate our methodology, (ii) identify and discuss different scenarios in which functional constructs are changed or removed and (iii) gain insights on the actual functional constructs usage.

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 ±5% for a confidence level of 95%.

Two independent annotators (two authors) have manually analyzed each sample to determine whether the fix is a semantic-altering change involving a functional construct, adding some notes about the change rationale. As regards the latter, we did not want to perform a categorization, indeed we aimed 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 [112] inter-rater agreement of 0.63, which can be considered a strong agreement. A third annotator has finally addressed and resolved all validation discrepancies.

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 [108].

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 3.1.5. In this context, we found cases where developers admit their fix to functional construct in the commit message.

Table 7.11 reports some examples of documented changes to functional constructs, highlighting several reasons behind such changes. Specifically, several cases involve fixes within the construct, often related to a misinterpretation or misuse of the construct itself. For instance, Listing 5.1 shows a fix due to a list comprehension misuse, explicitly admitted within the commit message (#6) stating 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 what concerns the compatibility issues, we found a change (#4) related to Python 2 compatibility (*i.e.,* dictionary comprehension unsupported) which diff is shown in Listing 5.2; a case (#5) in which the fix concerned the replacements of several `filter` functions with comprehensions (as reported in Listing 5.3); a similar scenario, shown in Listing 5.4, has been found when considering the `map` function (#9), and, last but not least, we

```
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 5.1 List comprehension misuse example

```
_inv_typemap = { v: k for k,v in _typemap.items() }
```

```
_inv_typemap = dict((v,k) for (k,v) in _typemap.items());
```

Listing 5.2 Unsupported dictionary comprehension example

found an instance (#8), which diff is shown in Listing 5.5, where the bug is due to a side-effect of Python moving the `reduce` function into a dedicated library, *i.e.,* in Python 3.x, `reduce()` was moved from the *builtins* module to the *functools*, requiring specific import for its use. Some developers decided to remove the `reduce()` function calls due to its unavailability as a built-in function. This is a clear example where a function construct is replaced with a non-functional alternative without altering the semantics of the implemented code. Last but not least, in another case (#1 in Table 7.11), the commit message clarifies the resolution of a crash that occurred during the retrieval of keys for an invalid user. The fix involved a change to a lambda function to consider two forgotten parameters (see Listing 5.6).

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. An interesting case is shown in Listing 5.7 [2], involving the extraction of a function invocation from a list comprehension, since such a construct is not optimized for function calls, leading to possible performance degradation [95, 96].

[2] `https://github.com/saltstack/salt/commit/1a9f03a`

```
files = filter(lambda x: x.endswith(".mo"), files)
files = [os.path.join(basedir, x) for x in files]
```

```
files = (f for f in files if f.endswith(".mo"))
files = [os.path.join(basedir, f) for f in files]
```

Listing 5.3 Filter functions replaced with list comprehensions example

```
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 5.4 Map function replaced with a list comprehension example

- 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), and 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, as well as cases where comprehension were translated in other functional constructs, *e.g.,* `map()`, `reduce()` in list comprehensions with aggregation functions, or list comprehensions into loops.

5.3.2 Survival Time

The time-fix-inducing changes that remain in the system may be impacted by several factors. One expects blocking and severe bugs' survival time to be very short. Even if we did not find a reference to blocking bugs, we found some bugs living in the systems for less than one day, such as the one in Listing 5.8 [3] where to fix an issue, *i.e.,"errorMessages":["Either the 'username' or the 'key' query parameters need to be provided"],"errors":*, there was the need to revert a previous change adding a `filter()` function.

[3] `http://tinyurl.com/yp4w7f7k`

```
numRows = reduce(max, [len(d[0]) for d in data])
```

```
numRows = max([len(d[0]) for d in data])
```

Listing 5.5 Replacing a functional construct with its non-functional alternative example

```
self.getKeysFor([node["to"]], lambda successJids, b: self.
    sendToContact(node) if len(successJids) == 1 else self.
    toLower(node), lambda: self.toLower(node))
```

```
self.getKeysFor([node["to"]], lambda successJids, b: self.
    sendToContact(node) if len(successJids) == 1 else self.
    toLower(node), lambda error_node, entity:
    on_get_keys_error(error_node, entity, node))
```

Listing 5.6 Lambda function not dealing with all possible corner cases

We also found several extreme cases where the bugs involving functional constructs remain in the system for a long time. For instance, the fix-inducing change in Listing 5.9 [4] removes both the `filter()` and the `lambda` functions for compatibility reasons after 763 days they have been introduced into the system. The fix-inducing changes aimed at ensuring compatibility across different Python versions have a survival time highly dependent on when the core team members decide to support the new Python version being released. Indeed, the example shown in Listing 5.10[5], replacing the `map()` function with a list comprehension, only remained in the system 11 days. While on the former, the survivability is high mainly because when the fix-inducing change occurred, Python v.3 was not in use within the system, for the latter, the survivability is lower because the project still used Python 3. Other than fixing compatibility issues, there are also cases where the fix involving a functional construct happening far from its introduction, aims at addressing styling issues or is part of a large cleanup operation. For instance, the change shown in Listing 5.11[6] applies a kind of extract method refactoring to have the same `reduce()` function being used more than once in the code. In this case, there might be a discussion on whether these changes should be seen as a purposeful bug fix stemming from a change made to address a specific issue, or if they are more of a side effect of restructuring and improvement, similar to cleaning up code smells [108].

[4] https://github.com/saltstack/salt/commit/8c418e0
[5] https://github.com/HIPS/autograd/commit/9224718
[6] https://github.com/pyload/pyload/commit/0b210a1

```
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 5.7 Functional construct definition misuse example

```
user_tuples = filter(None, jira_client.format_user(user) for
    user in response])
```

```
user_id_field = jira_client.user_id_field()
```

Listing 5.8 Delete `filter()` function to fix a bug - survival time 1 days

Last but not least, it is also important to highlight instances where the survival of the functional construct causing a fix does not align with an extreme case.Listing 5.12[7],[8] reports a fix involving a set comprehension. Specifically, there is the need to add a previously missed condition in Slack notifications rules to send notifications only to a subset of users (see the *if instanceof(participant, User)* clause). An interesting case to be discussed deals with a low-priority bug[9] that should be fixed by changing the logic implemented through a `lambda` function (see Listing 5.13[10]). By looking at the commit message, *i.e., ... remove the dirs_only parameter ...t doesn't work in some cases and isn't used anyway*, it is evident the bug is not critical, blocking the overall functionality of the system as a whole, so it is kind of expected that the bug survives in the system for more than five months (*i.e.,* 167 days).

To sum up, while quantitative findings (RQb_5) show that different functional constructs have different median survival times, our manual inspection was not able to identify any specific reason why dictionary, set comprehensions, and filter functions are removed faster than reduce functions. It is important to note that given the population size (see Fig. 5.1), we have inspected all fixes involving dictionary and set comprehensions, and reduce and filter functions. The only finding we have is that, very often, the functional construct is simply removed as part of other major changes. This is analogous to what the previous work found for code bad smell removals [108].

[7] `http://tinyurl.com/yc2vzd2c`
[8] `https://github.com/getsentry/sentry/pull/29495`
[9] `https://github.com/saltstack/salt/issues/16638`
[10] `https://github.com/etuttle/salt/commit/bbae532`

```
fields = dict( filter( lambda x: x[0] in glanceclient.v1.
    images.CREATE_PARAMS, kwargs.items()))
```

```
 fields = [(k, v) for (k, v) in six.iteritems(kwargs) if k in
     glanceclient.v1.images.CREATE_PARAMS]
```

Listing 5.9 textttfilter() and `lambda` functions removed to fix compatibility issues with Py3 - survival time 763 days

```
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 5.10 *‘map(...).index()‘ doesn’t work on Py3, just use a list comprehension - survival time 11 days*

5.3.3 Implications

The results of our study trigger implications for developers, educators, and researchers.

Developers should carefully ponder the use of functional constructs, balancing syntactic elegance (and shorter code) with understandability [18], also knowing that performance may or may not improve [95, 96]. Also, developers should not underestimate the risk of inducing fixes when adding or changing simple list comprehensions, lambdas, or map functions.Last, but not least, developers should give high priority to code review (*e.g.,* by using specific checklists) and testing of functional constructs, considering the complexity and the type of functional construct among the criteria to prioritize it. To complement code review, developers could also leverage static analysis tools, such as *pylint* [91], that point out potential problems to functional constructs, and suggest refactoring when appropriate.

Educators should, on the one hand, give proper emphasis to both the syntax and usage examples of functional constructs, showing typical misuse cases. On the other hand, they should also explain the pros and cons of using functional constructs during development, as to the need for properly reviewing/testing such code. Special attention should also be paid to constructs such as set and dictionary comprehensions. Given the findings of our study, we conjecture that these constructs are not sufficiently understood by developers.

Researchers could develop, similar to what was done for lambdas in Java [17,98], refactoring

```
salt = reduce(lambda x,y: x + y,[str(random.randint(0, 9))
   for i in range(5)])
salt_pw = salt + _salted_password(password, salt)
```

```
def _gensalt():
    return reduce(lambda x, y: x + y, str(random.randint(0,
   9)) for i in range(5)])
salt_pw = _salted_password(password, _gensalt())
```

Listing 5.11 Extract "function" to avoid retyping the same `reduce()` function more than once - survival time 739 days

```
user_ids = {user.id for users in participants_by_provider.
   values() for user in users}
```

```
user_ids = {
    participant.id
    for participants in participants_by_provider.values()
    for participant in participants
    if isinstance(participant, User)
    }
```

Listing 5.12 Adding missing condition in a previously defined set comprehension to retrieve only users who are instances of the User class - survival time 81 days

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, research should be targeted to better support debugging and fault localization on functional constructs. Existing IDEs do not support proper debugging of such constructs (*e.g.,* step-wise execution of comprehensions) nor branch or condition coverage during testing.

5.4 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 5.1.2, 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

```
# filter out the files or the dirs depending on flag
ret[bucket] += filter(lambda k: k.endswith('/') == dirs_only,
    filePaths)
```

```
# filter out the dirs
ret[bucket] += filter(lambda k: not k.endswith('/'),
   filePaths)
```

Listing 5.13 `filter()`a change in lambda example - survival time 167 days

there. 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 5.3, 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 5.3. However, through the achieved results, we cannot claim causation. Moreover, we have used appropriate models to evaluate the effect of the changed churn size together with the cyclomatic complexity of the functional constructs being added/modified. 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. Finally, while we performed an analysis of the survivability of fix-inducing changes for churns involving different types of functional constructs, and while we control for churn size and constructs' cyclomatic complexity, we are aware that there may be several other (uncontrolled) factors that may influence such survivability. Indeed, as observed in our qualitative analysis, there were instances where the constructs were removed as part of broader changes.

Threats to *conclusion validity* concern the relationship between experimentation and outcome. We have used appropriate statistical tests (Fisher's exact test, mixed-effect models, and Cox proportional hazard 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 [87] 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 functions). We do not know whether the obtained results would generalize to other programming languages, and above all to other functional 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 [99].

5.5 Conclusion

This chapter discusses the results of an empirical study aimed at analyzing the extent to which changes to three Python functional constructs—*i.e.,* lambdas, dictionary/list/set comprehensions, and map/reduce/filter functions—have higher odds of inducing fixes than other changes, how the code cyclomatic complexity influences this phenomenon, and how long bugs in churns containing functional constructs survive in the software system.

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

1. Changes to churns that affect functional constructs have more than twice higher odds to induce fixes than other changes, and 15% higher odds if the fix impacts the same line(s) of the functional construct. Also, a changed churn involving functional constructs has the same odds of inducing fixes as the addition/change of $\simeq 6$ lines of code.

2. Unsurprisingly, when a new functional construct is added, this induces fixes with higher odds than when it is modified. However, this highly depends on the changed churn size.

3. Lambdas have higher odds of inducing fixes, followed by comprehensions, whereas the effect of map/reduce/filter functions is more limited.

4. The constructs' cyclomatic complexity does not play a major role. On the contrary, we found that, for some constructs such as lambdas and list comprehensions, odds to induce fixes are higher for constructs with a low complexity than for constructs with a high complexity.

5. Bugs introduced in different constructs have different survival times. Filter functions and comprehensions exhibit the highest hazard rates, meaning that fixes to churns containing such constructs tend to be removed significantly quicker than other fixes.

Reduce functions, instead, exhibit the lowest hazard rate, indicating a longer survival for related bugs. That being said, with our qualitative analysis, we did not find any explicit cause-effect relationship between the construct type and the bug survival time. Instead, we found that constructs are changed (or removed) as a consequence of other, major changes.

6. 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 to an alternative one or to get rid of it, writing a non-functional alternative solution.

5.6 Data Availability

The replication package [99] comprises datasets extracted from a GitHub repository, which served as the basis for conducting diverse statistical analyses and constructing survival models in our study.

CHAPTER 6 LIST COMPREHENSION VERSUS FOR LOOPS PERFORMANCE IN REAL PYTHON PROJECTS

This chapter presents the outcomes of an empirical investigation designed to unravel the nuances surrounding the impact of list comprehensions on program execution time, particularly when juxtaposed with semantically equivalent procedural code.

Building upon the groundwork laid by previous studies, such as Zhang et al. [2], this chapter approaches the problem in two phases: a first experiment is meant to compare 1) synthetic but realistic code in-vitro 2) real projects performance based on test input data. experiments. The first set of experiments involves the substitution of small procedural code snippets, often encapsulated within a single loop, with list comprehensions. As anticipated, our results align with the findings of Zhang et al. [2]: when subjected to an iterative amplification—executing the transformed code thousands of times—list comprehensions exhibit a notable advantage in terms of execution speed.

However, as we transition from controlled experiments to the intricate landscape of real-world projects (a second set of experiments), the narrative takes an intriguing turn. In direct contrast to the consistent advantages observed in isolated experiments, our investigation into transforming for loops into list comprehensions reveals a spectrum of outcomes that defy a uniform trend. The magnitude of observed differences varies not only from program to program but also performance gain lacks practical significance in many instances. Even when improvements are discernible, their impact on specific program use cases is often negligible. This observation, coupled with previous research indicating a potential increase in fault-proneness with the use of functional constructs [1], prompts a cautious reflection on the default adoption of functional programming practices.

The structure of this chapter unfolds as follows: Section 6.1 elucidates the methodology employed for transforming `for` loops into list comprehensions. Section 6.2 outlines the study design and planning, providing a roadmap for the ensuing analysis. Results and their implications are meticulously presented and discussed in Section 6.3, while Section 6.4 addresses potential threats to the validity of our findings. Finally, Section 6.5 concludes this chapter, offering insights into the broader implications of our study and paving the way for future research directions.

6.1 Converting Loops into List Comprehensions

We have developed, conceived, and implemented an approach to automatically convert `for` loops into comprehension. The approach we have followed has commonalities with the one proposed by Zhang *et al.* [113]. More specifically, Zhang *et al.* [113] proposed a set of transformation rules for a variety of Python non-idiomatic code. Among these rules, they also considered a rule to map for loops into list comprehensions. Unfortunately, the rules they propose, at least for list comprehensions, are not able to capture the `for` loops variability *e.g.,* two nested loops where the appended value depends on both loops. Therefore, we decided to develop our set of transformation rules, rules handling the most common cases but also more complex cases (see rules three, four, and five).

To devise conversion heuristics, we have, first of all, analyzed the set of 200 projects from the Zampetti *et al.* dataset [1], mining list comprehension constructs. We found that 1) List comprehensions rarely have a high McCabe complexity (usually it is lower than three or four); 2) Similarly, when `for` constructs are used to build a list, the code snippet does not have a high McCabe complexity (usually below three); and 3) in the end, the `for` loops being good candidates for refactoring follow a limited set of (five) structure patterns.

Each candidate `for` is transformed piece-wise. However, since sometimes the transformation rules fail, each transformed `for` is tested before moving on to the next one. In this way at the end of the process, we are sure that the new code behaves like to old one over available test cases. We are aware that this is a weaker condition than a fully-fledged semantic equivalence check. Based on that, we defined our conversion approach, consisting of the following steps:

1. We parse each Python file and, using the set of for loop patterns, we identify candidate code fragments (AST nodes) for transformation.

2. As we plan to use the project's available test cases to execute the code and measure performance, we check whether the identified transformation candidates are traversed by test case execution using *pycoverage*;

3. For the identified and traversed (by at least one test case)transformation candidate, by applying transformation rules, restructure the AST. This results in removing a single `for` and synthesizing the equivalent list comprehension;

4. By executing the test cases, we check the effect of the transformation. If the test cases pass, we accept the transformation, otherwise, we reject it and keep the code unchanged.

In the following, we detail the different transformation rules.

6.1.1 F2L.1 : A simple for loop

```
for _ in range(1000):
    number_lst.append(random())
```

Listing 6.1 First case: A simple for loop example.

```
number_lst = [random() for _ in range(1000)]
```

Listing 6.2 First case: Equivalent in list comprehension simple example.

Listing 6.1, shows the simplest case of a transformation. In this case, the `for` loop does not have nested control-flow instructions, and the body contains only a single statement. To convert the loop, two phases are needed. First, we need to retrieve relevant `for` elements. Then, we synthesize a list comprehension statement. When traversing the `for` sub-tree we need to:

- Locate and extract the append `target` *i.e.,* number_lst.
- Identify and extract the iteration variable, `iter` iteration variable *i.e.,* the list generated by the range() function.
- Identify and extract the loop's `body`.

In this case, the loop body contains one statement and one node. A body traversal allows us to identify the AST call (ast.Call) node where the invoked function name is `append`, its parameter (*i.e.,*`random()` is the expression generating the value appended to the list. With the identified and extracted `for` elements, and their sub-trees a new list comprehension can be synthesized. To build a comprehension node we need:

- A `target`. In this case, it will be the target previously extracted number_lst;
- `iter`, the iterator. In this example, it is `the generator created by the range() function.`

Once the list comprehension is created, we create an `a ssign` node assigning its value to the list comprehension created (assignment right-end-side) and setting its target to the variable extracted from the append statement (assignment left-end-side) *i.e.,* number_lst.

6.1.2 F2L.2 : A for loop with an if block inside

```
for i in range(1000):
    if a[i] % 2 == 0:
        number_lst.append(i)
```

Listing 6.3 Second case: a for loop with an if block inside example.

```
number_lst = [i for i in range(1000) if a[i] % 2 == 0]
```

Listing 6.4 Second case: Equivalent in list comprehension example.

The second transformation rule deals with loops containing a single `if`. The conversion process, as shown in Listing 6.3, is slightly more complicated as the body is an `if` statement that must be transformed to synthesize the proper list comprehension elements.

Similarly to the previous case, the transformation requires the append parameter, the target, and the iteration variable. Also, we need to extract the if-statement test expression *i.e.,*`a[i] % 2 == 0`. The process is the same as the previous case, except that the list comprehension statement now contains an if-expression where the test expression is `a[i] % 2 == 0`.

6.1.3 F2L.3 A for loop with an if-else block inside:

In the case of Listing 6.5, the transformation is complicated by the presence of both `if` branches the `then` and the `else` branch. Each branch has its append call, and thus parameter (*i.e.,*`i` and `i*2` respectively). This forces the use of an `if` expression with two branches accounting for the different values.

```
for i in range(1000):
    if a[i] % 2 == 0:
        number_lst.append(i)
    else:
        number_lst.append(i * 2)
```

Listing 6.5 Third case: A for loop with an if-else block inside example.

```
number_lst = [i if a[i] % 2 == 0 else i * 2 for i in range(1000)]
```

Listing 6.6 Third case: Equivalent in list comprehension example.

6.1.4 F2L.4 : For loop with intermediate variable

As shown in Listing 6.7 the value added to the list depends on intermediate variables that can be defined inside or outside or both the loop. Since we are converting non-functional code

to functional code, the loop disappears. Therefore, whatever is inside its body disappears too.

In a nutshell, we need to trace appended variables to their definitions, *i.e.,* where values come from. To do this, we implemented a simplified symbol table tracing 1) scopes, 2) definitions, and 3) uses of variables. Our symbol table is limited to the scope defined inside a single file. It handles simple cases and thus arrays, references, and virtual calls are not handled.

Our simplified symbol table allows tracking the line where a variable was defined, its assigned value (more precisely right-end-side), and the scope it belongs to. Using this information, we can retrace the last definition's right-end-side in the correct list comprehension place. For example, the more complex example of Listing 6.9 is properly handled and transformed into the code of Listing 6.10.

```
for i in lst_a:
    c = i ** 2
    lst_b.append(c)
```

Listing 6.7 Fourth case : with an intermediate variable inside the loop example.

```
lst_b = [i ** 2 for i in lst_a]
```

Listing 6.8 Fourth case: Equivalent in list comprehension example.

```
a = 5
for i in lst_a:
    c = i ** 2
    lst_b.append(c * a)
```

Listing 6.9 Fourth case : with an intermediate variable inside and outside of the loop example.

```
lst_b = [i ** 2 * 5 for i in lst_a]
```

Listing 6.10 Equivalent in list comprehension of for loop with an intermediate variable inside and outside of the loop example.

6.1.5 F2L.5 : Two nested nested for loops

```
for a in lst_a:
    for b in lst_b:
        mylist.append(a*b)
```

Listing 6.11 Fifth case: simple nested for loop example.

```
mylist = [a * b for a in lst_a for b in lst_b]
```

Listing 6.12 Fifth case : Equivalent in list comprehension example.

The last rule addresses the conversion of a simple nested `for` loop as shown in Listing 6.11, where the appended value depends on both loops. In this case, the loops are flattened, and the loop variables populate a list comprehension containing two "nested" `for` expressions.

Table 6.1 Number of instances of list comprehensions with their complexity

Complexity	Number of instances
2	180,787
3	57,773
4	3,788
5	1,045
6	246
7	96
8	14
9	21
10	2
11	2

6.1.6 List comprehension complexity

As explained above, our aim is to convert into list comprehensions loops that have a fairly low cyclomatic complexity. To understand what is the distribution of cyclomatic complexity for list comprehensions, we analyzed those of the dataset from Zampetti *et al.* [1].

Such data set is composed of 200 Python of nontrivial *industrial-like* characteristics. As shown in Table 6.1, complexity two (and three) have the overwhelming majority. This means that, by simply applying the first two rules, it would be already sufficient to translate the vast majority of candidate `for` loops into the corresponding list comprehensions.

6.2 Study Design

The *goal* of this study is to investigate if there is a performance gain (execution speed improvement) when for are replaced by list comprehensions. In other words, this study focuses on list comprehension and for loop and aims to compare the execution speed of the loop (`for`), and the (semantically-equivalent) list comprehension to analyze the differences. The *perspective* is of both practitioners and researchers. Practitioners aim to develop faster code,

while researchers may be interested in understanding under what conditions list comprehensions will lead to faster code. The *context* accounts for both synthetic code fragments, as well as eleven open-source Python projects hosted on GitHub.

The study aims to address the following two research questions:

- **RQc$_1$:** *Is list comprehension code running faster than its equivalent for code?* Here we would like to establish a theoretical baseline by analyzing the amplified execution of synthetic, yet realistic code, and comparing the performance of `for` loop code against semantically equivalent list comprehensions.
- **RQc$_2$:** *Does transforming for loops into list comprehensions result in faster real-world projects?* Any real-world project is composed of a mix of various constructs. Here we investigate what is the effect of transforming for-loops into their equivalent list comprehension. We would like to study how many `for` loops are worth being transformed, how many are successfully transformed, and whether the transformation has any significant performance effect.

6.2.1 Context Selection

Our study has been conducted on two datasets, one of real-world code and one of synthetic code snippets. The real-world projects dataset is a subset of the 200 projects in the dataset of Zampetti *et al.* [1]. The basis of these projects was the existing list of engineered software projects made available by Munaiah *et al.* [3]. We selected from the 200 projects a subset of (eleven) projects that 1) have at least one `for` loop candidate for transformation into list comprehensions; 2) have *pytest* test cases; and 3) successfully install and run (successfully) at least one test case. At the same time, from the entire dataset of 200 projects, we studied the frequency of list comprehension and for loops to identify the most frequent cases. Simple list comprehension, having a McCabe cyclomatic complexity of two or three, are the vast majority. Then, based on the inspection of several code snippets, we defined a set of five exemplary `for` loops with the equivalent list comprehension code. This second small sample has the purpose of comparing the two paradigms on synthetic yet realistic code.

Table 6.7 reports summary information about the chosen projects. We can observe that each of these projects already uses list comprehensions *i.e.,* developers are aware of Python idioms. Their size ranges between a few dozen files to 260. Projects cover different domains from networking (dnspython.git), to database (sqlparse.git), to Elasticsearch query formulation support (elasticsearch-dsl-py.git). A common feature of these projects is the availability of `pytest` test cases allowing us to quantify the speed gain when introducing list comprehensions.

6.2.2 Data Extraction

Data collection was performed on an AMD Ryzen 7 3800X, with 48 GBytes RAM. Input and output data were stored on a fast NVME SSD. The running environment was AlmaLinux 9 and Python 3.10.2.

Data extraction for the five code snippets was straightforward. To better observe the differences for synthetic code snippets, we amplified their execution by iterating it 100,000 times.

For the eleven real projects, data extraction was a bit more complicated. We first transformed each project and incrementally identified successfully transformed for. As explained in Section 6.1 we parsed each Python file through the Python AST module[1]. Then, we traversed its parse tree by extracting relevant nodes (`for` loops). We leveraged the node names to identify the different elements of `for` loops (and, in particular, their iterators, bodies, and invoked functions). Whenever a for loop sub-tree matched a transformation rule, we then generated its equivalent list comprehension and replaced with it the `for` loop. The output of this incremental approach (see Section 6.1), is an equivalent code where some of for loops have been replaced. Finally, we executed (in isolation) each project, and collected its execution time. Each experiment for each code snippet(project) was repeated 32 times [114] to limit the effect of the operating system and execution environment.

Execution times were measured considering them from an user's perspective, *i.e.,* the whole time a user has to wait when executing the program. This includes both the user and the system time, and not just the user time as it is typically done in performance analysis.

6.2.3 Data Analysis Methodology

As a metric to assess the difference of performances between `for` and comprehensions, and similarly to Zhang *et al.* [2], we computed the performance index ρ. The ρ is defined as the ratio between the execution time before the transformation and the execution time after the transformation. We considered all the possible couples (time before, time after) and graphically represented the distributions of ρ for the different projects as box plots.

For each project's differences, we report the Tukey five-number summary (minimum, first quartile, median, third quartile, and maximum). Then, we applied the Wilcoxon [7] signed rank test (which is a non-parametric, paired test, recommended to use for randomized experiments like this one [114]) to check for the presence of statistically significant differences. Also, we complemented the statistical test by the Cliff's delta [8] effect size measure.

[1] `https://docs.python.org/3/library/ast.html`

6.3 Study Results

This section reports the results of our study, to address the research questions formulated in Section 6.2.

Table 6.2 Time performance of code using F2L.1 in seconds for 100,000 iterations

F2L.1	**Listing: 6.1**	**Listing: 6.2**
Minimum	0.015	0.015
Q1	0.016	0.016
Median	0.016	0.016
Q3	0.018	0.018
Maximum	0.032	0.032
Effect size	+0.15 (S)	

Table 6.3 Time performance of code using F2L.2 in seconds for 100,000 iterations

F2L.2	**Listing: 6.3**	**Listing: 6.4**
Minimum	0.076	0.074
Q1	0.077	0.074
Median	0.079	0.074
Q3	0.081	0.075
Maximum	0.110	0.077
Effect size	+0.95 (L)	

Table 6.4 Time performance of code using F2L.3 in seconds for 100,000 iterations

F2L.3	**Listing: 6.5**	**Listing: 6.6**
Minimum	0.083	0.078
Q1	0.085	0.079
Median	0.087	0.080
Q3	0.101	0.090
Maximum	0.271	0.273
Effect size	+0.56 (L)	

6.3.1 RQc_1: Is list comprehension code running faster than its equivalent for code?

Tables 6.2-6.6 report execution time summary statistics and effect size for the synthetic code snippets. For example, Table 6.2 shows the summary statistics obtained for Listing 6.1 and 6.2, *i.e.,* the `for` code snippet version and its list comprehension counterpart.

Table 6.5 Time performance of code using F2L.4 in seconds for 100,000 iterations

F2L.4	**Listing: 6.7**	**Listing: 6.8**
Minimum	0.044	0.038
Q1	0.045	0.038
Median	0.045	0.038
Q3	0.046	0.039
Maximum	0.055	0.045
Effect size	+0.98 (L)	

Table 6.6 Time performance of code using F2L.5 in seconds for 100,000 iterations

F2L.5	**Listing: 6.11**	**Listing: 6.12**
Minimum	23.70	11.20
Q1	23.80	11.27
Median	23.87	11.29
Q3	24.04	11.36
Maximum	26.10	11.67
Effect size	+1.0 (L)	

As explained in Section 6.2.2, differences have been amplified by executing 100,000 times each snippet. As one can notice, the effect size sign is always positive. A positive sign means that the transformed code is faster. However, for snippets in Listing 6.1 (6.2) to Listing 6.7 (6.8), the time differences are generally below one second. The exception is the more complex example in Listing 6.11 (6.12) where the nested `for` makes a significant difference: list comprehension is twice as fast as its `for` counterpart.

Overall, from the study with synthetic snippets—as results reported in Table 6.2 to Table 6.6 highlight—it clearly emerges that list comprehensions are faster than `for` loops. However, as one can expect, the magnitude of such a difference depends on the content of the `for` body. Moreover, without amplification (and this would be what happens in real projects), the observed magnitudes likely vanish. Since, as shown in Table 6.1, very simple list comprehensions are the majority of the cases, the gain in real life may be marginal.

6.3.2 *RQc*$_2$: Does transforming for loops into list comprehensions result in faster real-world projects?

Tables 6.7-6.9 report results about the 11 studied real projects. From Table 6.7, we can observe that transformable `for` loops are a fraction of the total `for` loops. The percentage of the transformed loop varies between 15% (*e.g.,* pre-commit-hooks) and 55% in `pycparser`. In other words, we expect to observe a measurable performance change. The `sqlparse` project

Table 6.7 Results of transformation rules on real projects

Project	Files		for loops			Num Listcomp	Total passed
	total	with for	total	candidate	transformed	before	tests
dnspython	201	73	307	48	29	41	1216
mesa	122	46	184	29	8	56	118
jasmin	188	72	238	162	64	5	227
spotipy	66	28	44	5	2	26	36
sqlparse	35	16	49	4	0	15	416
pycoin	263	96	364	44	19	125	1809
pycparser	45	22	241	134	67	36	129
pexpect	92	28	68	12	2	25	234
sslyze	121	45	123	24	6	34	152
pre-commit-hooks	75	32	66	10	0	10	404
elasticsearch-dsl-py	70	27	101	4	1	18	330

contains four `for` candidates, however, no one, once transformed, passed the test cases. We kept the project as a reference point because any observed difference, in this case, can only be

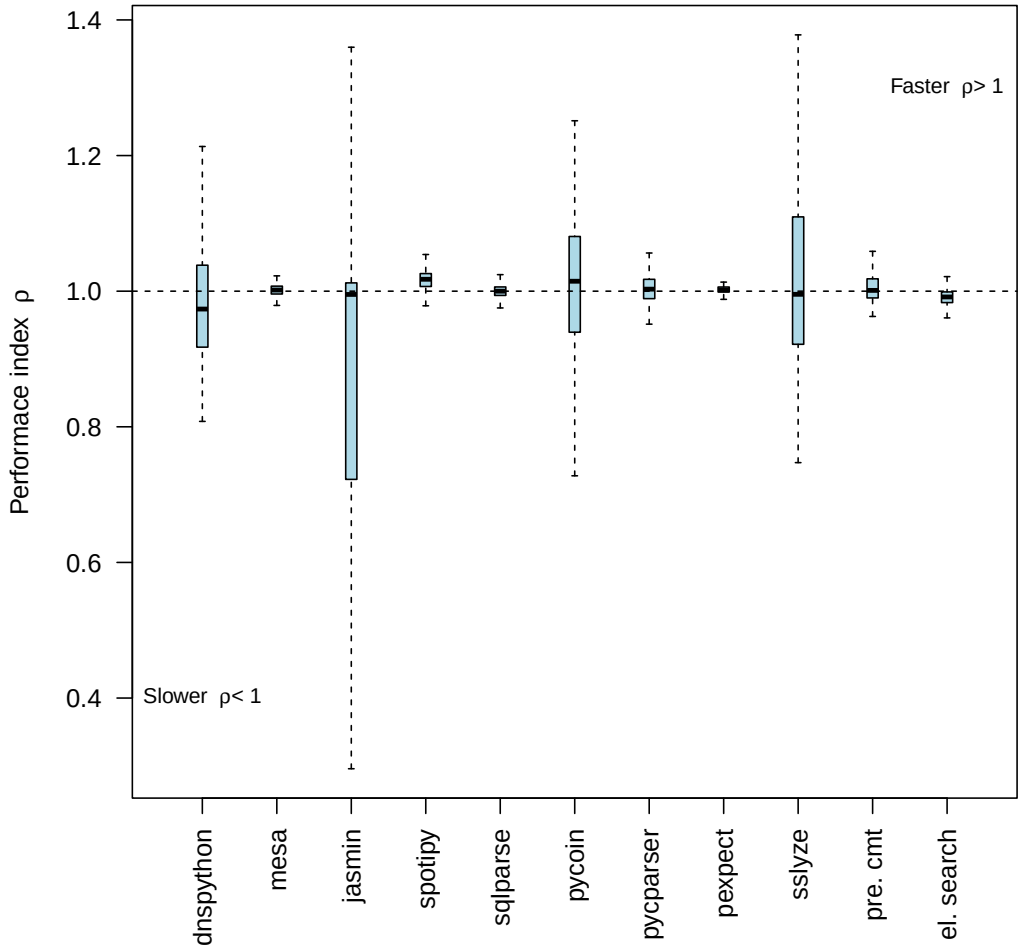

Figure 6.1 Performance index for the studied systems.

due to execution environment variability. Table 6.8 reports the number of applied rules per type and project. In agreement with Table 6.1, simple `for` loops take the lion's share. Also, it is worth noting that in out of the nine cases in which rule F2L.5 was applied, `for` loops would not have been transformed by the approach proposed by Zhang *et al.* [113]. Also, in the studied 11 projects, we did not find `for` loops matching the fourth rule *i.e.,* loop with intermediate variables.

Table 6.9 reports descriptive statistics, p-values from the Wilcoxon test, and Cliff's delta computed out of the 32 projects' executions. Compared to synthetic code, the use of list comprehensions does not necessarily result in a gain. This is in line with the findings of Zhang *et al.* [2]. However, results heavily vary from one system to the other. In

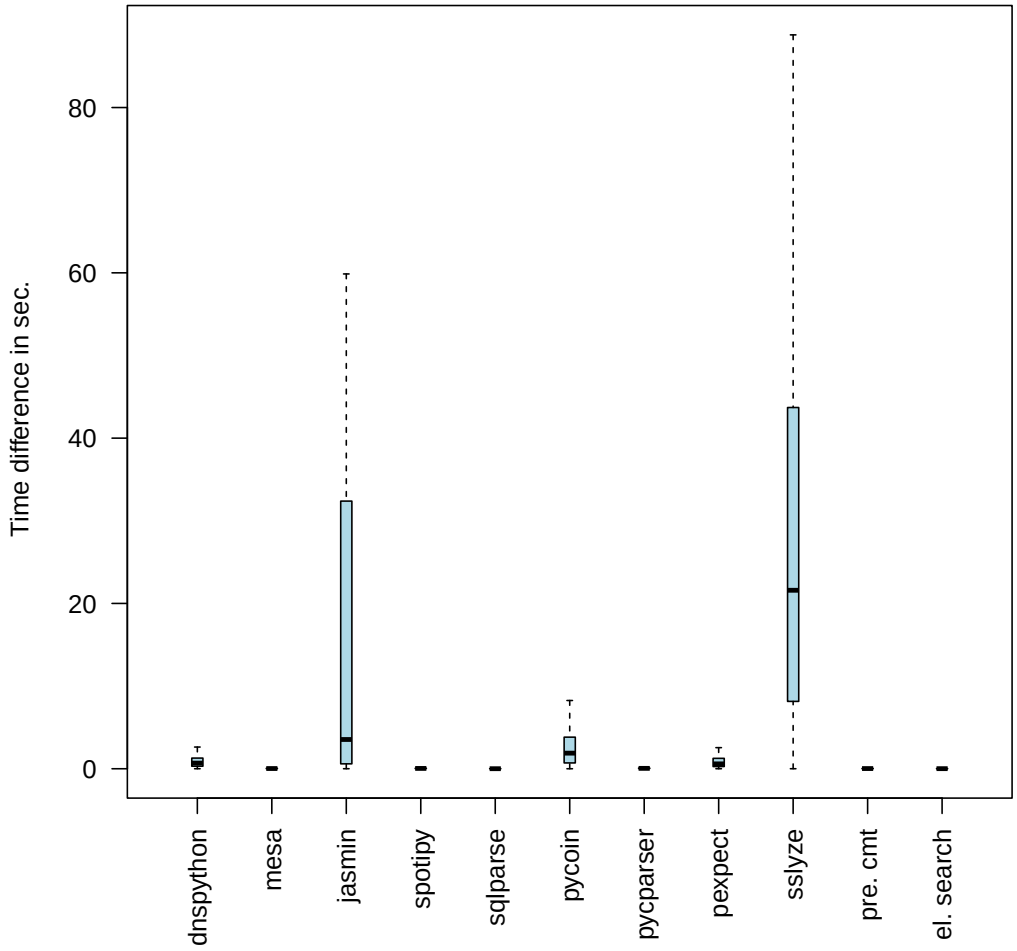

Figure 6.2 Performance time difference in second for the studied systems.

nine cases (out of eleven), despite the list comprehensions being faster, the observed differences are not statistically significant with an effect size that only in two cases, `spotipy` and `elasticsearch-dsl-py`, is large.

Understanding the effects of our transformations for the `dnspython`, `jasmin`, `pycoin`, `spotipy`, and `sslyze` projects is more complicated. This is because these projects heavily leverage network services. For example, `dnspython`[2] is a toolbox supporting domain name resolution. Similarly, `spotipy` [3], a lightweight Python Spotify library (*i.e.,* supporting music streaming). For the other three projects, the dependency from the network is even more

[2]`https://github.com/rthalley/dnspython`
[3]`https://github.com/spotipy-dev/spotipy`

Table 6.8 Number of list comprehensions from real projects that were successfully transformed and passed the tests (for each rule)

Project	F2L.1	F2L.2	F2L.3	F2L.4	F2L.5
dnspython	14	11	0	0	4
mesa	4	2	0	0	2
jasmin	43	16	0	0	5
spotipy	2	0	0	0	0
sqlparse	0	0	0	0	0
pycoin	12	6	1	0	0
pycparser	47	12	8	0	0
pexpect	2	0	0	0	0
sslyze	6	0	0	0	0
pre-commit-hooks	0	0	0	0	0
elasticsearch-dsl-py	1	0	0	0	0

complex. `jasmin` [4] is an open-source text message gateway; `sslyze`[5] is an SSL/TLS scanning tool. `pycoin` [6] is a collection of utilities to deal with bitcoin and bitcoin-like systems. While `dnspython` and `spotipy` could somewhat be tested locally, minimizing the dependency from the network speed, for the last three systems, there is no easy way to mitigate the randomness due to the Internet access. Despite we executed each system multiple times and discarded extreme outliers, those systems still exhibit trends likely due to network bottlenecks.

Similarly to what was done by Zhang *et al.* [2], Figure 6.1, reports the *rho* performance index. Differently from what Zhang *et al.* [2] show in Figure 1 of their paper, in our study the ρ index is almost always close to one, and never surpassing 1.5. In practice, we never observed a speed doubling.

Last, but not least, it is important to discuss the magnitude of differences. Glancing over Table 6.9 it is clear that, from the user's point of view differences are not really relevant. Indeed, in most cases, differences are in the range of a few seconds. This fact is also underlined by the boxplots of Figure 6.2. Following the same approach used for 6.1, we computed the absolute execution time difference for all pairs of before and after transformation executions, such time difference is always below ten seconds, except for `jasmin` and `sslyze`. As explained above, `jasmin` and `sslyze` heavily depend on the network, and therefore the measured performance is affected by that.

In conclusion, we can state that, for real projects executed with their test cases, there is no

[4] `https://github.com/jookies/jasmin`
[5] `https://github.com/nabla-c0d3/sslyze`
[6] `https://github.com/richardkiss/pycoin`

appreciable gain in nine cases out of eleven. Even considering `jasmin` and `sslyze`, where the performance variation is seriously affected by the network, as explained above, the median of response time differences is always below 20 seconds.

6.4 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. First of all, this type of threat may concern the extent to which we are conducting a performance measurement that reflects what happens in practice. For this reason, on the one hand we have created a baseline with synthetic code snippets. On the other hand, we have measured the execution time of real projects (by their test cases). Also, measurements have been performed only when the test cases cover the transformed constructs.

As explained in Section 6.2.2, the performance measurements have been performed by considering the whole response time and not the user's time, to put ourselves in the perspective of a user experiencing the program's response time.

Last, but not least, similar to Zhang *et al.* [2], we performed the performance analysis by relying on the ρ performance index.

Threats to *internal validity* concern factors, internal to our study, that could influence our results. One factor is surely due to the noisiness of performance analysis. To mitigate this threat, we repeated the executions 32 times. Nevertheless, as we have discussed in our results, the execution time of some programs that heavily depend on network services is still likely to be affected by the response time of these services and by the network's speed.

A further threat could be related to mistakes committed in the transformation. We mitigated this threat by skipping cases for which the test outcome was different between the transformed and the original code. However, we cannot exclude that this could have biased our analysis toward certain cases for which the transformation was successful.

Threats to *conclusion validity* concern the relationship between experimentation and outcome. To ensure reliable conclusions, we complemented descriptive statistics with suitable statistical tests (Wilcoxon signed rank test) and effect size measures (Cliff's delta).

Threats to *external validity* concern the generalizability of our findings. While the study has been performed by considering both synthetic snippets and real projects, whose number is relatively small (11), and, therefore, further studies are certainly required. Furthermore, our study limits to list comprehensions vs. `for` loops. However, the same analysis should be repeated on other Pythonic functional constructs, including further types of comprehensions. Last, but not least, our study has been performed by transforming `for` loops into list

Table 6.9 Descriptive statistics of execution times before (Bfr) and after (Afr) transformation for each project (tests were executed 32 times). The effect size is indicated as N: Negligible, S: Small, M: Medium, L: Large

Dataset	Min		Q1		Median		Q3		Max		p-value	Effect size
	Bfr	Afr	Bfr	Afr	Bfr	Afr	Bfr	Afr	Bfr	Afr		
dnspython	8.99	9.19	9.21	9.58	9.50	9.83	10.71	10.58	12.83	11.14	0.11	-0.24 (S)
mesa	4.69	4.70	4.74	4.73	4.76	4.75	4.78	4.77	5.20	4.85	0.24	+0.17 (S)
jasmin	20.06	21.38	48.61	76.93	77.39	77.66	77.95	78.20	81.25	78.89	0.21	-0.18 (S)
spotipy	2.14	2.11	2.17	2.12	2.18	2.14	2.18	2.15	2.29	2.23	<0.001	+0.64 (L)
sqlparse	0.94	0.94	0.95	0.95	0.96	0.95	0.96	0.96	1.25	1.26	1	+0.01 (N)
pycoin	25.30	24.61	25.87	25.30	26.48	26.17	28.49	28.61	40.50	35.28	0.23	+0.17 (S)
pycparser	3.27	3.25	3.32	3.31	3.35	3.34	3.40	3.38	3.85	3.43	0.44	+0.11 (N)
pexpect	168.76	168.86	169.65	169.48	169.87	169.56	170.75	169.99	171.12	171.31	0.02	+0.35 (M)
sslyze	211.63	194.78	218.58	218.98	225.06	231.03	261.93	255.04	283.57	283.25	0.77	-0.04 (N)
pre-commit-hooks	1.68	1.68	1.69	1.70	1.71	1.70	1.74	1.722	1.84	1.87	0.80	+0.04 (N)
elasticsearch-dsl-py	0.92	0.93	0.93	0.94	0.94	0.94	0.94	0.95	1.29	1.31	<0.001	-0.57 (L)

comprehensions, and not vice versa. It would be therefore interesting to also replicate the experiment in the opposite direction.

6.5 Conclusion

We conducted a study on the effect—in terms of runtime performance–of transforming Python `for` loops (procedural code) into their list comprehensions (functional code) equivalent structures. Specifically, we wanted to investigate whether to what extent the use of list comprehensions in place of `for` loop leads to a faster code.

Our results show that, when it comes to synthetic code, the use of functional constructs is more effective compared to the use of procedural programming constructs. However, when it comes to real-life projects, the conclusions are not the same. By running projects' test cases, we measured the user-perceived execution times (*i.e.,* user time plus system times). For the eleven analyzed Python projects, we found the difference is almost always small or negligible. Differently from previous work, we did not find a 100% (or greater) performance increase when using functional constructs, but, rather, way smaller percentages. In terms of absolute figures, the median difference in execution times is always below 20 seconds. The largest observed difference was of about 90 seconds in an SSL/TLS network scanner. However, this difference is likely due mostly to network access and congestion. In most cases, and for nine projects out of eleven, systems difference never exceeds ten seconds. Given our findings, developers should carefully consider whether or not to transform `for` loops into list comprehensions as previous studies provided evidence of list comprehensions defect-proneness [1].

6.6 Data Availability

The replication package [115] contains (i) the clones of the considered projects git repositories, (ii) the scripts used for experiments, and (iii) the obtained results.

CHAPTER 7 THE RELATIONSHIP BETWEEN DIFFERENT PYTHON ARGUMENT-PASSING MECHANISMS AND FIXES

The exchange of information between functions or methods through argument passing is one of the elements that developers consider with more care during (integration) testing and code review. Indeed, specific integration testing coverage criteria have been defined for that [64]. Programming languages provide different ways of passing arguments, allowing, for example, the use of a positional or keyword-based notation, to define default values, and to pass a variable number of arguments.

This chapter examines argument-passing mechanisms in Python, which as already explained in previous chapters is the second-most popular programming language nowadays on GitHub [116], and widely used for a large set of applications, including those related to data science and machine learning. Like most of the other programming languages, Python supports a positional argument-passing mechanism, as well as default values, but it also allows the passing of arguments in arbitrary order through keyword-value pairs and has several variants of those. Furthermore, Python allows various combinations in the same function definition; some parameters may be positional and others keyword-based. However, not all combinations are legal.

In summary, Python has been designed to provide developers with convenient syntactical elements to make their work more efficient, and to make the source code more concise. As it always happens, while programming language constructs are introduced with good intentions, sometimes they can make the lives of developers more difficult. Indeed, previous research has coined the term "atoms of confusion" to define code elements that confuse developers, making program comprehension more difficult and, incidentally, causing the introduction of defects. Moreover, by leveraging mining software repository approaches such as the SZZ algorithm [6], previous research has investigated the extent to which certain program elements, *e.g.,* pointers [46], functional constructs [18, 94, 117], or even the source code naturalness [47] induce more fixes than others.

So far, little is known about whether different Python parameter-passing mechanisms impact defect-proneness. This may be relevant to investigate because some mechanisms may induce less confusion, *e.g.,* a keyword-based mechanism may avoid associating an actual parameter with the wrong formal one. Also, while certain parameter passing styles such as ordered (positional arguments) or variable length are common to many programming languages, keyword or keyword-only are less common, and thus developers may be less familiar, with

and misuse them.

This chapter reports the results of an empirical study that investigates the extent to which different parameter-passing mechanisms available in Python induce fixes. More specifically, we study this construct in two different places, *i.e.,* the function definition level and the call sites.

The study has been conducted on a set of 200 Python projects, obtained starting from a list of "engineered" projects [3] as described in previous chapters. This study examines:

1. Whether different signature parameter passing mechanisms have higher odds of inducing fixes;
2. What are the specific odd increases related to specific features in function signature definition, while controlling for confounding factors;
3. Whether different argument passing mechanisms induce fixes in the function call sites; and
4. What are the specific odd increases related to specific features in argument passing.

The study shows how some features such as positional parameters or variable parameters induce more fixes to function signatures, whereas a keyword-based passing mechanism induces significantly fewer fixes than others. The results not only pave the way to better code completion tools aimed at mitigating mistakes in argument passing but also, at prioritizing code review and testing towards certain code elements.

The structure of this chapter is organized as follows. Section 7.1 details the study design and planning. Results are presented and discussed in Section 7.2. The qualitative analysis and the implications of the study are discussed in Section 7.3, while Section 7.4 reports the study's threats to validity. Section 7.5 concludes the chapter.

7.1 Study Design

The *goal* of this study is to investigate whether changes affecting Python function argument passing—at signature/body and call site level—induce more fixes than other changes, and what parameter-passing mechanisms are responsible for inducing more fixes than others. The *perspective* is that of researchers interested in understanding the downside of different parameter passing and argument construction mechanisms. The *context* consists of the evolutionary history of 200 open-source Python projects hosted on GitHub.

The study working dataset is available in our replication package [118].

The study addresses the following four research questions:

- **RQd$_1$:** *To what extent do changes to parameter-passing in function definitions induce more fixes than others?* This research question investigates whether changes involving function signatures with different parameter-passing mechanisms have higher odds of inducing fixes than other changes.
- **RQd$_2$:** *How does the fault-inducing proneness in function definitions vary among different types of parameter passing mechanisms?* This research question follows up the previous one, investigating whether changes involving certain parameter passing mechanisms are more likely to induce fixes.
- **RQd$_3$:** *To what extent do changes to function argument-passing in call sites induce more fixes than others?* While the first two research questions look at the function definition, here we determine whether changes to arguments in call sites induce more fixes than other changes.
- **RQd$_4$:** *How does the fault-inducing proneness in call sites vary among different argument passing mechanisms?* This research question deepens the analysis of RQd$_3$, investigating what the specific argument passing mechanisms that induce more fixes than others.

7.1.1 Context Selection

We answer the research questions by mining the change history of 200 open-source Python projects hosted on GitHub and selected through the following procedure. We rely on an existing dataset of engineered software projects made available by Munaiah *et al.* [3] as explained in the chapter 5.1.1

Table 7.1 reports descriptive statistics about the studied projects, as well as the occurrences of functions with different parameter passing options, *i.e.,* positional (*args*), *defaults*, *varargs*, *kwargs*, positional-only (*posonlyargs*), and keyword-only (*kwonlyargs*).

The study concerns more than 580,439 functions contained in 67,742 Python files, 2,804,230 function definition changes, and 12,579,843 call site changes. We decided to exclude from the analysis changes related to *posonlyargs* as they are only 12, likely because this feature was introduced with Python 3.8 only. We also excluded function definition (call sites) outliers. In both cases, we discarded the extreme outliers: function definition (call sites) with more than

20 parameters and cyclomatic complexity greater than 20 (more than 100 expressions and more than five nested calls). These cases are likely to be code bad smells, and fault-prone themselves [119].

Table 7.1 Five number statistics of the studied Python repositories: Size (KLOC), number of files, and studied constructs.

Metric	Min	1Q	Median	3Q	Max
Size (KLOC)	0.92	9.28	23.60	64.56	1208.31
Files	4.00	54.00	133.00	351.50	9406.00
Functions	40.00	444.5	1156.0	3012.0	39718.0
args	48.21	91.09	95.63	98.04	100
defaults	1.87	9.21	13.91	19.22	64.05
varargs	0	0.86	1.87	3.14	32.18
kwargs	0	1.47	3.13	5.72	51.44
kwonlyargs	0	0	0	0.04	6.38
posonlyargs	0	0	0	0	0.06

7.1.2 Data Extraction

We analyzed the evolution history of the repositories described in Section 7.1.1 to identify, (i) changes to function definitions (both signatures and body), changes to call sites, and, (ii) fixes and fix-inducing changes. Then, we combined the different pieces of information to perform the analysis of the results described in Section 7.1.3. The analysis has been performed by considering all commits of all branches, excluding merge commits.

Analysis of Changes to Function Definitions

We parsed each Python file modified in each commit using the Python 3.10 AST module [5], and then traversed its parse tree extracting relevant nodes. We leveraged the node names to identify the different elements of function definitions (in particular of their signatures, their formal parameters, and associated AST types) and of function invocations (actual parameters and associated AST types).

Each AST node is decorated with information including the beginning and ending line/-column. For each function definition node, `ast.FunctionDef`, we extract its `ast.Name`, the function identifier, and, if present, the tuple of formal arguments. Notice that, while `posonlyargs`, `args`, `kwonlyarg`, `kw_defaults`, and `defaults` are always present, `vararg` and `kwarg` are optional. Call site nodes, `ast.Call`, are expressions `ast.expr`. A call site is modeled through three lists. The first list models the call sequence (*e.g.,*`ast.parse(string)`

– access path calls). The second and the third list store the sub-trees for the positional and keyword-passed parameters respectively. Each list item is an `ast.expr` traversing it we collect the different kinds of expressions (*e.g.,* list, constant, if-expression, or call) composing the passed argument.

For each commit, Python files have been parsed in their version before and after the commit. If a file was added, all functions (and methods) identified are considered as added. Similarly, if a file is removed, its function (and methods) are considered 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. Overall, we record the starting and ending location for each function (and signature) and the starting and the ending location of each code churn being modified in a commit, plus all information extracted from git commits and ASTs. Whenever a function (or method) is involved in a change, given the two file versions, we consider the following three cases:

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

Note that when a function is added (deleted), all its call sites are considered added (deleted), however, when changed a call site is considered changed if and only if it is contained in a churn. Due to context diff limitations and location function movements, to uniquely identify each function in the code, we represent it through a 256-bit fingerprint, computed after removing comments and white spaces. This fingerprint allows for matching the function along the software change history.However, the approach does not distinguish semantic-altering changes from semantic-preserving changes (*e.g.,* refactoring). Therefore, we may detect irrelevant changes [102]. In our qualitative analysis (Section 7.3), we discuss a sample of such cases.

Analysis of Fix-Inducing Changes

The 5.1.2 section delineates the methodology utilized for analyzing fixes in Python projects, highlighting the use of a lightweight version of the SZZ algorithm due to the inconsistent use of issue trackers in these projects. While some studies on SZZ suggest not considering the first commit of a file [120], in our study, we intentionally decided to not do so, as we are also interested in parameter-passing definitions or call sites "born as defect-prone", *i.e.,* defects are introduced when creating the code. This conjecture was, for example, found to be true

for code bad smells [108] as well as for functional constructs [117].

Combining Function Changes and Fix-Inducing Changes

As the last step of our analysis, we combine the two aforementioned analyses, *i.e.,* changes to function definitions (signatures and call sites) and fix-inducing changes. Given a change in a commit, we identify (i) whether it induces a fix, and if yes, what lines of the churn induce the fix, (ii) whether the churn contains a change to function its signature/call site and if yes which ones, and whether the function (signature or call site) was added or modified, and (iii) whether the function change and the fix-inducing change occur on the same line.

7.1.3 Analysis Methodology

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

To address RQd_1, we compare the proportion of fix-inducing changes involving function signature per type of passing mechanism. We consider different types of function parameter passing types shown in the bottom part of Table 7.1 (types).

Given each type T of parameter passing mechanism:

1. The changed churn affects a function signature featuring a given type of parameter-passing mechanism T, and it induces a fix;
2. The changed churn affects a function signature featuring a given type of parameter-passing mechanism T, but it does not induce any fix;
3. The changed churn affects a function signature, not featuring the specific type of parameter-passing mechanism T, yet it induces a fix;
4. The changed churn does not affect a function signature featuring a given type of parameter-passing mechanism T, and it does not induce a fix.

To determine whether the passed parameter mechanism induces a fix, similar to what previous work has done on functional constructs [117], we can use Fisher's exact test [106] to compare the proportions of (1) over (2) and (3) over (4), and also compute 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 is the ratio between the odd of the experimental group (the changes with argument type T in our case) while the control group concerns all other types. Due to multiple comparisons, we adjust p-values using the Benjamini-Hochberg correction [87]. Unlike previous work [117], we do not compare these changes with all other changes, because we are interested in performing a

relative comparison among functions featuring different types of arguments.

Also, when it comes to the fix, an issue affecting the function parameters may be fixed in different ways, *i.e.,* (i) the fix impacts the Signature Only (SO), or the fix impacts both the signature and the body (B2). For this reason, we divide our analysis considering the two different types of fixes. Truly, a fix related to a bug occurring in the signature may be performed in the body only. However, analyzing whether the fix was related to a signature-related problem would, at the minimum, require identifying a dependency (*e.g.,* through data and control flow analysis) between the two. For this reason, we limited our attention to the cases where the fix involved the signature.

While the aforementioned analysis could be enough for addressing RQd_1, there could be confounding factors influencing the odds a change has to induce a fix. Among others, we consider the changed churn size (referred as *churn_size* in the reported models) since previous literature [41] points out that a large change has higher odds of inducing a fix, the file size (*file_size*), the function size (*func_size*) and McCabe cyclomatic complexity (*cmplx*) [121]. We mitigate this threat in RQd_2, where 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.,* (i) the presence of certain features in the changed signature, and (ii) confounding factors listed above, and whenever pertinent the number of parameters (*i.e., num_parms*). 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* [122] *lme4* package [71]. 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. Given this is a logistic model, the OR is the exponential of the estimate.

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 7.4.

To address RQd_3 and RQd_4, we applied the same methodology followed for RQd_1 and RQd_2, while considering function call sites instead of function definitions. However, it is important to note that size- and complexity-related co-factors are computed on the caller source code and not on the callee, *i.e.,* the *cmplx* co-factor refers to the cyclomatic complexity of the callee. The intuition is that a function call within a more complex or longer code may be more likely to induce a fix.

As for RQd_3, we consider as a feature the presence or not of arguments, the presence of

positional arguments (*args*), and keyword-based arguments (*kwargs*).

For RQd_4 we consider as (nominal) features (i) the presence of positional-based and keyword-based arguments *args* and *kwds*, respectively, and (ii) factors controlling the changed churn size, caller function size, and file size. Namely, *args* equals to *true* if the call site contains at least a positional argument, *false* otherwise. The same happens for *kwds*. Also, we repeat the analysis considering, instead of nominal factors, two numerical factors, counting the number of positional and keyword-based arguments.

Finally, we conjecture that for what concerns call sites there could be further factors, related to the call site, that may influence the fix-proneness, *e.g.,* the presence of expressions or lambda.

Besides the quantitative analysis addressing the four research questions, we perform a qualitative analysis on a sample of changes to function signatures that induced a fix and call sites that induced a fix. This qualitative analysis is described in Section 7.3.

7.2 Study Results

This section discusses the results addressing the research questions defined in Section 7.1.

Table 7.2 RQd_1: Fisher's exact test for the B2 fix inducing changes.

Metric	OR	95% CI		p-value
args	1.004	0.91	1.11	1
defaults	1.094	1.017	1.176	**0.026**
varargs	1.383	1.196	1.59	$<$ **0.001**
kwargs	1.275	1.132	1.431	$<$ **0.001**
kwonlyargs	0.885	0.441	1.587	1
noargs	0.923	0.828	1.026	0.195

7.2.1 *RQd_1:* To what extent do changes to parameter-passing in function definitions induce more fixes than others?

To address RQd_1, we compare the proportion of fix-inducing changes occurring in commits modifying function definition signature (SO), signature, and body (B2) versus other changes. As explained, we set the diff context to zero and thus the overlap occurs at the exact line(s) level.

Table 7.2 and Table 7.3 report the results of Fisher's exact test, the ORs, and their confidence interval, considering data from all the studied Python projects for the B2 and SO cases

Table 7.3 RQd_1: Fisher's exact test for the SO fix inducing changes.

Metric	OR	95% CI		P-value
args	2.220	2.018	2.458	**<0.001**
defaults	0.630	0.591	0.671	**<0.001**
varargs	0.444	0.373	0.525	**<0.001**
kwargs	0.493	0.433	0.559	**<0.001**
kwonlyargs	0.835	0.516	1.279	0.338
noargs	0.413	0.372	0.457	**< 0.001**

respectively. An OR greater than one indicates that commits in which a function signature is added or modified—and the signature contains the elements indicated in the table row—have higher odds of inducing a fix than other commits where such a feature is not in the changed signature.

The presence of positional arguments (*args*) has no significant effect on increasing the odds of inducing B2 fixes (Table 7.2, yet, as one may expect, they have a significant effect on signature fixes (Table 7.3 – p-value < 0.001) and OR is 2.22. *i.e.,* the presence of positional arguments means twice the chances to induce a fix compared to cases when positional arguments are not changed.

Similar observations can be made for *varargs* for what concerns the B2 changes (OR = 1.38). Instead, when changes concern the signature only, the OR is only 0.47. One possible explanation is that a *varargs* parameter is then unpacked in the function body across individual parameters, and mistakes also impact the function body. Also, passing parameters by keywords (*kwargs*) correlates with inducing more fixes jointly on the body and signature with an OR of 1.27, while if the change only affects the signature the OR is 0.46. A further analysis considering confounding factors (RQd_2) is required to better understand such observations.

Unsurprisingly, the absence of arguments means that changes are less likely to induce fixes on the signature, yet these are still possible, *e.g.,* through adding new parameters. Finally, all other features have a milder or insignificant effect.

RQd_1 Summary: Changes to signatures in the presence of positional arguments induce more fixes than other changes. This also happens for *varargs* and *kwargs*, this time with fixes affecting both signatures and body. Other changes have milder or insignificant effects.

Table 7.4 RQd_2: Mixed-effect logistic regression relating the mechanism of passing parameters with fix-inducing changes involving both signature and body (B2).

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**
65798.92	65945.47	-32887.46	65774.92	1487662.00
SCALED RESIDUALS:				
Min	**1Q**	**Median**	**3Q**	**Max**
-0.42	-0.07	-0.04	-0.02	252.74
RANDOM EFFECTS:				
Groups			**Variance**	**Std.Dev.**
Id (Intercept)			2.10	1.45
Number of obs: 1487674, Groups: Project: 200				
FIXED EFFECTS:				
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-5.94	0.12	-48.65	$<$ **0.001**
file_size	-0.00	0.00	-20.02	$<$ **0.001**
churn_size	-0.00	0.00	-1.92	0.055
func_size	0.00	0.00	3.61	$<$ **0.001**
func_cplx	-0.03	0.01	-3.29	**0.001**
num params	0.03	0.01	3.31	**0.001**
args	-0.09	0.06	-1.64	0.101
defaults	0.11	0.04	2.36	**0.018**
varargs	0.29	0.09	3.30	**0.001**
kwargs	0.02	0.07	0.26	0.793
kwonlyargs	-0.56	0.31	-1.82	0.069

7.2.2 *RQd_2:* How does the fault-inducing proneness in function definitions vary among different types of parameter passing mechanisms?

In RQ_2 we investigate the joint effect of the different parameter passing mechanisms using mixed-effect logistic mode. Also, the model controls for size and complexity effects, as well as for the random effect, *i.e.,* the project.

Table 7.4 and Table 7.5 reports the results of the mixed-effect logistic model for the B2 and SO respectively. We recall that the OR is equal to $e^{Estimate}$, and therefore values close to zero indicate a negligible effect, negative values indicate a mitigating factor, and positive ones an increase of the fix-proneness in the presence of the given factor.In other words,$args(kwarg)$ models the presence or absence of positional arguments (keywords arguments) in the signature *i.e.,* we are not concerned with the number of positional (or keyword) arguments just with the presence of such parameter-passing mechanisms. We observe that in both cases size does not play a role (*i.e.,* churn, function, and file size).

Table 7.5 RQd_2: Mixed-effect logistic regression relating the mechanism of passing parameters with fix-inducing changes involving the signature (SO).

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**
95033.72	95180.27	-47504.86	95009.72	1487662.00
Scaled residuals:				
Min	**1Q**	**Median**	**3Q**	**Max**
-1.47	-0.07	-0.04	-0.02	1342.23
Random effects:				
Groups			**Variance**	**Std.Dev.**
Id (Intercept)			2.02	1.45
Number of obs: 1487674, Groups: Project: 200				
Fixed effects:				
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-7.36	0.12	-59.63	< **0.001**
file_size	0.00	0.00	4.29	< **0.001**
churn_size	-0.00	0.00	-41.30	< **0.001**
func_size	0.00	0.00	11.36	< **0.001**
func_cplx	-0.13	0.01	-18.47	< **0.001**
num params	0.24	0.00	70.20	< **0.001**
args	0.58	0.06	10.42	< **.001**
defaults	-0.10	0.04	-2.64	**0.008**
varargs	-0.26	0.09	-2.80	**0.005**
kwargs	-0.48	0.07	-6.79	< **0.001**
kwonlyargs	0.09	0.23	0.40	0.69

The cyclomatic complexity of the function/method (*cmplx*) has a negative effect for both B2 and SO. This seems counterintuitive as high complexity should be likely to correlate with more defects. We conjecture that, when a complex function or method is changed, developers tend to pay more attention. Recent empirical results in a study on the understandability of Pythonic constructs corroborate such a conjecture [18].

When fixes involve both the signature and body (B2, Table 7.4), we can notice an increase of fix-proneness due to the presence of *varargs* (the OR is $e^{0.29} = 1.34$, which means a 34% increase. In this case, it is quite expected that fixes involve the function body too because such fixes likely impact the way the list is unpacked onto multiple parameters. Interestingly, the effect is in the opposite direction when fixes concern the signature only (SO, Table 7.5), yet this can be explained by the fact that such fixes almost always involve the body too and that if only the signature was changed *varargs* is added or incorporate and replace some of previous parameters.

The effect related to positional parameters (*args*) is statistically significant when the fixes

involve the signature only (SO, Table 7.5), and in this case the fix-proneness increase is of $e^{0.58} = 1.79$, *i.e.,* 79%. Differently from what was observed when analyzing individual factors (RQd_1), the presence of default parameters has the opposite effect, *i.e.,* correlating with a reduction of fix-proneness for signatures (OR=$e^{-0.10} = 0.9$) and increasing (slightly) the risk for body and signature (*i.e.,* 11 %). A similar effect can be observed for the presence of *kwargs*. In this case, the OR is $e^{-0.48} = 0.61$. This has a plausible explanation, that, however, requires confirmation from the investigations of RQd_3 and RQd_4. In essence, passing parameters by keywords avoids mistakes related to the order exchange of positional parameters, however, this is not statistically significant when the change involves both the signature and the body.

kwonlyargs has been conceived as a way to reduce the risks when passing varargs preceding other parameters. Indeed, in their presence the odd to induce fixes involving both signature and body (Table 7.4) are reduced, as the OR is $e^{-0.56} = 0.57$. The effect on fixes involving signatures only is not statistically significant.

All other observed effects, including those of control factors, either have a very small effect size (*i.e.,* less than 5% increase or decrease in terms of fix-proneness) or are not statistically significant (*i.e.,* p-value$>$ 0.05). The only exceptions, for fixes involving signature only (Table 7.5), are represented by (i) the number of parameters which, as expected, increases the odd (OR=$e^{0.24} = 1.27$), and (ii) the complexity, that has a negative effect (OR=$e^{-0.13} = 0.87$).

RQd_2 Summary: While *varargs* may be useful to developers, their presence tends to induce more fixes of $\simeq$ 34% than cases where this is not present. On the opposite, *defaults* and *kwargs* reduce the odds to induce fixes of about 10% and 39%, as they may help to avoid mistakes due to positional parameters.

Table 7.6 RQd_3: Fisher's exact test for the call sites fix inducing changes.

Arguments	OR	95% CI		p-value
args	1.08	1.03	1.14	**<0.001**
kwargs	0.90	0.87	0.94	**<0.001**
has arguments	1.42	1.23	1.64	**<0.001**

7.2.3 RQd_3 : To what extent do changes to function argument-passing in call sites induce more fixes than others?

As Table 7.6 shows, and as expected, the presence of arguments (*has arguments*) increases the fixes to call sites by 40%. Also, if arguments are positional, there is a small odd increase

(8%), whereas passing by keywords, makes a difference on the opposite side, decreasing the odds by 10%.

In essence, as expected, and confirming the results found in the analysis of function definitions, positional(keyword) arguments play statistically significant a role in (reducing) fix-inducing changes. As for RQd_1, also in this case the results need to be further analyzed by accounting, as we will do in RQd_4, for project-related and confounding factors.

RQd_3 Summary: The presence of parameters increases the odds to induce fixes in a function invocation of 40%. Moreover, there is an increase of 8% in the presence of positional arguments, while keyword-based parameters decrease by 10% the odds of inducing fixes.

Table 7.7 RQd_4: Mixed-effect logistic regression relating the call site keyword and argument passing mechanisms with fix-inducing changes.

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**
163230.05	163324.84	-81607.02	163214.05	1034499.00
Scaled residuals:				
Min	**1Q**	**Median**	**3Q**	**Max**
-0.64	-0.16	-0.08	-0.04	77.56
Random effects:				
Groups			**Variance**	**Std.Dev.**
Id(Intercept)			2.79	1.67
Number of obs: 1034507, Groups: Project: 200				
Fixed effects:				
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-4.59	0.13	-36.12	< **0.001**
file_size	0.00	0.00	-29.76	< **0.001**
churn_size	0.00	0.00	-24.05	< **0.001**
func_cplx	-0.01	0.00	-6.88	< **0.001**
func_size	0.00	0.00	1.04	0.297
args	0.08	0.03	2.86	**0.004**
kwargs	0.07	0.02	3.10	**0.002**

7.2.4 *RQd_4* : How does the fault-inducing proneness in call sites vary among different argument passing mechanisms?

As a last analysis for RQd_4, we study the effect of different kinds of features present in positional and keyword-based arguments. The results of the mixed-effect logistic regression model shown in Table 7.7 are in substantial agreement with RQd_3 findings, controlling, however,

for confounding factors (size and complexity-related factors) and the projects' random effect. More specifically, Table 7.7 indicate a small effect of both positional and keyword-based parameters (OR=$e^{0.08} = 1.08$ and $e^{0.07} = 1.07$ respectively, *i.e.,* a substantial equivalence in terms of proneness to induce fixes. The effect of confounding factors is negligible ($e^0 = 1$). Note that we also repeated the analysis by considering, instead of nominal factors for *args* and *kwd*, numerical ones considering the number of arguments passed with each of the two mechanisms. However, we obtained very similar results, and therefore, we do not report them in the chapter.

Results of this mixed-effect logistic regression model are reported in Table 7.8 is based on factors, not the numbers. The table is divided into three regions, with the top one reporting confounding factors, the second the effect of positional arguments, and the third of keyword-based arguments.

The top part of Table 7.8 indicates that as in the previous analysis reported in Table 7.7, confounding factors have no role. By looking at the second and third regions of the table, we can see that, in general, certain language features make the positional argument passing more problematic than the keyword-based one, yet common issues exist.

Passing constant arguments has a positive correlation with fix-inducing changes for both positional and keyword-based arguments, with the same OR of $e^{0.15} = 1.16$. The same happens for dictionaries, yet in this case, we have an OR of $e^{0.20} = 1.22$ for positional arguments and $e^{0.10} = 1.11$ for keyword-based arguments.

From previous studies, it has emerged that lambda functions are problematic in terms of fault-proneness and understandability. [18,117]. This finding is confirmed in Table 7.8: passing a lambda through positional arguments increases the fix-inducing odds of $e^{0.26} = 1.30$, whereas there is no statistically significant effect for keyword-based arguments. Similarly, the use of comprehensions increases the OR to $e^{0.22} = 1.25$, and the same happens for `if` expressions. Finally, passing a list via a positional argument slightly increases the odds ($e^{0.10} = 1.11$) of fix-inducing changes. In all these cases, no significant effect is found for keyword-based arguments.

The only case for which keyword-based arguments have a greater fix-proneness than positional arguments is the use of function calls, for which the OR is of $e^{0.40} = 1.49$. While we do not have an explanation of why such a difference exists, we conjecture that function calls in arguments, because of the complexity, stability, or, simply, API evolution of the invoked function, may induce fixes regardless of the passing mechanism.

Table 7.8 RQd_4: call site fix inducing affecting factors.

AIC	**BIC**	**logLik**	**deviance**	**df.residuals**
162946.50	163254.59	-81447.25	162894.50	1034481.00
SCALED RESIDUALS:				
Min	**1Q**	**Median**	**3Q**	**Max**
-0.74	-0.16	-0.08	-0.04	67.44
RANDOM EFFECTS:				
Groups			**Variance**	**Std.Dev.**
Id (Intercept)			2.76	1.66
Number of obs: 1034507, Groups: Project: 200				
FIXED EFFECTS:				
	Estimate	**Std.Error**	**z value**	**Pr(>|z|)**
(Intercept)	-4.64	0.13	-36.67	< **0.001**
file_size	0.00	0.00	-29.84	< **0.001**
churn_size	0.00	0.00	-23.87	< **0.001**
func_size	0.00	0.00	-0.89	0.372
func_cplx	-0.01	0.00	-4.71	< **0.001**
args_call	0.03	0.02	1.41	0.158
args_comp	0.22	0.08	2.79	**0.005**
args_constant	0.15	0.02	8.42	< **0.001**
args_dict	0.20	0.04	5.47	< **0.001**
args_expr	0.02	0.03	0.73	0.463
args_if	0.22	0.10	2.16	**0.031**
args_lambda	0.26	0.08	3.16	**0.002**
args_list	0.10	0.03	2.91	**0.004**
args_slice	-0.04	0.09	-0.40	0.69
args_star	-0.07	0.10	-0.72	0.471
kwd_call	0.40	0.04	9.78	< **0.001**
kwd_comp	-0.03	0.23	-0.14	0.888
kwd_constant	0.15	0.04	4.27	< **0.001**
kwd_dict	0.10	0.05	2.01	**0.044**
kwd_expr	-0.07	0.03	-2.29	**0.022**
kwd_if	-0.08	0.19	-0.43	0.666
kwd_lambda	0.15	0.12	1.19	0.233
kwd_list	0.06	0.06	0.98	0.328
kwd_slice	0.04	0.30	0.14	0.887
kwd_star	0.22	0.73	0.30	0.768

RQd_4 Summary: When controlling for project- and change-related factors, the effect of the type of passing arguments is statistically significant, yet very similar for both positional and keyword-based arguments. When looking closer at the programming language features involved in the arguments, we found more significant cases of fix-inducing odd increases for positional than for keyword-based arguments. Language features, such as dictionaries, lambda functions, comprehensions, if conditionals, and lists, exhibit higher odds of fix-inducing changes when used with positional arguments than with keyword-based arguments. The only case for which keyword-based arguments exhibit higher fix-inducing odds is represented by function calls.

7.3 Qualitative Analysis

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

Table 7.9 Qualitative analysis: manual classification of function definition fixes.

Category	**SO**	**B2**	Total
BUG	88 (48.9%)	55 (52.4%)	143 (50.2%)
FALSE POSITIVE	18 (10.0%)	5 (4.8%)	23 (8.1%)
MAJOR RESTRUCTUCTURING	27 (15.0%)	31 (29.5%)	58 (20.3%)
READABILITY	47 (26.1%)	14 (13.3%)	61 (21.4%)
Total	180 (62.2%)	105 (36.8%)	285

Table 7.10 Qualitative analysis: manual classification of calls sites fixes.

Category	**Positionals**	**Keywords**	**Mixed**	Total
ADDED	46 (15.4%)	2 (4.3%)	3 (4.8%)	51 (12.5%)
BUG	120 (40.3%)	29 (61.7%)	40 (64.5%)	189 (46.4%)
FALSE POSITIVE	50 (16.8%)	4 (8.5%)	4 (6.5%)	58 (14.3%)
MAJOR RESTRUCTURING	63 (21.1%)	9 (19.1%)	14 (22.6%)	86 (21.1%)
READABILITY	19(6.4)	3 (6.4%)	1 (1.6%)	23 (5.7%)
Total	298 (73.2%)	47 (11.6%)	62 (15.2%)	407

This qualitative analysis complements the quantitative results reported in Section 7.2 and has a two-fold goal: (i) validate our methodology, and (ii) identify and discuss different scenarios in which parameters (arguments) are changed, removed or added. We extracted two statistically significant, randomly stratified samples one for function definitions and one for call sites (*i.e.,* call sites inside modified function definitions). Each sample, was extracted from the subset of changed functions that were 1) tagged as bug fix; and 2) for which one or more keywords (*e.g.,* argument , passed, posonlyargs, vararg, etc.) in the commit message pointed to a parameter passing issue. The full list of keywords is available in the replication package [118]. In other words, for function definitions, out of the overall 5,659(11,624) B2 (SO) bugs we sampled data from the remaning 766(836) functions. At the end our first sample was made up by 285 modified functions, 180 B2 and 105 SO ensuring a ±5% for a confidence level of 95% each. The call sites dataset contains 19,032 entries tagged as buggy out of which 1982 contain an API related keyword in the commit log. To ensure a ±5% for a confidence level of 95% we selected 407 calls out of these 1982. Extracted samples are stratified based on 1) the type of parameter passing, *e.g.,* positional versus keywords; 2) the number of parameters/arguments; and 3) the projects. To increase variability, only one sample per file and commit was taken.

Table 7.11 Examples of documented fixes to functional constructs.

E #	Commit URL	Commit Message
E 1	https://github.com/appium/python-client/commit/9c597a3	...doc: Add script to generate sphinx doc ...
E 2	https://github.com/getsentry/sentry/commit/e59b74c248b99981be1d7dd289ca065d72ffaa02	...serialize() had '*args' following keyword arguments ..
E 3	https://github.com/HIPS/autograd/commit/f6394db	Fixed Python 2.7 syntax error - keyword arg following *args
E 4	https://github.com/saltstack/salt/commit/04671d0	fix overly loose signature ...
E 5	https://github.com/saltstack/salt/commit/d34a8fe	... We need to use **kwargs after the *names or *packages references...
E 6	https://github.com/saltstack/salt/commit/203f7ab	..Unused argument 'pillar' in ext_pillar() ...
E 7	https://github.com/saltstack/salt/commit/f25164e	... fixing wrong params in _post_message ...
E 8	https://github.com/mitmproxy/mitmproxy/commit/36c04f1	fix flowfilter.match args
E 9	https://github.com/tyiannak/pyAudioAnalysis/commit/ce65d5d	fix wrong function argument
E 10	https://github.com/Tribler/tribler/commit/9a6cb35	Fixes taking into account the new dispersy parameter.
E 11	https://github.com/saltstack/salt/commit/5624865	redis.StrictRedis.hmset expects a dict as its second arg.
E 12	https://github.com/matrix-org/synapse/commit/63ef607	Fix the test to pass the right number of args to the Store constructors.

In a nutshell, we extracted two samples 1) a sample of 285 fixes, in which there is a line-level overlap between the change to the parameter (for the function definition) ; and 2) a sample of 407 fixes, in which there is a line-level overlap between the change and a function call or one of its arguments.

Each fix in the sample was manually analyzed by two independent annotators (two authors) in two distinct validation steps, to determine whether the fix is a semantic-altering change involving a parameter (argument), 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 or major restructuring (not filtered out by the diff), (ii) readability changes *e.g.,* renaming a parameter; (iv) changes altering the semantics (likely bug fixes); and (v) false positives. During this validation, the annotators achieved a Cohen's kappa [112] inter-rater agreement of 0.7, which can be considered a strong agreement. All the discrepancies have been solved during a discussion meeting.

Table 7.9 and Table 7.10 report the distribution of fixes belonging to the different categories for the manually validated samples.

Overall, in the validated samples, we found about 50% of cases in which the fixes changed or removed a parameter (argument) and altered the function signature (or call site). However, some differences exists between definitions and call sites. Notice that for function definitions, Table 7.9 , (*i.e.,* B2 and SO) we are interested in verifying if the fix is effectively a semantic fix impacting 1) both the signature and the body (B2); or 2) the signature only, SO. On the other hand, inspecting call sites we also tagged data at a finer granularity level, to verify if different passing mechanisms differs from the fix standpoint. The two pieces of information complement each other, bottom line each call site tells the story of how parameter are passes to the function definition. Differently for function definition, call sites can be added along the function life, in Table 7.10 the label `ADDED` accounts form call sites added in the fix.

For function definitions we observe that Table 7.9 differences between SO and B2 can be explained by the B2 SO nature. Indeed, Unsurprisingly `READABILITY` changes are higher for SO, changes to parameter names, added comments or annotation most of the times to no require changing the body. Function and parameter annotation have been introduced in Python 3.5. It allows for adding type information to parameters and function definitions *e.g.,* to specify the returned type or the parameter expected type [123] [124].

On the other hand more likely that the body and signature are restructured together, this explain why B2 has almost the double `MAJOR RESTRUCTURIG`. On the opposite side ifis more likely a falso positive when only the signature is modified and indeed `FALSE POSITIVE` is 10% for SO while for B2 is only 4.8%. more common when only the signature is altered, while false positives are lower when body and signature are impacted. Not surprisingly major restructuring mostly impacts both body and signature. Overall, even if we aggregate false positives and readability, we conclude semantic fixes are effectively the largest fraction close to 70% of fixes.

Call site results, Table 7.10, as expected show that `READABILITY` only pertains to fraction of changes with an average of 5.7% of cases. Bugs still have the lion share with major `MAJOR RESTRUCTURIG` in he second place, this mens there was a major code block removal and replacement, *i.e.,* the function (or call site) was incidentally removed along with other changes. These include cases in which the function definition (possibly including one or more call sites) was part of a larger change where the fix was not intended to change parameters or arguments. This is unsurprising and consistent with previous studies on software quality, *e.g.,* showing that code smells are mainly removed because the affected code is removed or replaced [108]. `FALSE POSITIVE` tops at 16.8% for positional parameter passing.

More interestingly Table 7.10 shows the evidence that not all parameter passing mechanisms are easily fix-inducing changes or equally adopted. Perhaps due to a higher familiarity with more traditional programming languages, passing by keywords is only one-sixth of positional parameter passing, and, oftentimes the two mechanisms are used together. One premise of passing by keywords was to have a more robust parameter passing mechanism less prone to bugs, however, Table 7.10 tells another story positional parameter passing is less fix-inducing change prone. Indeed, mixing the two mechanisms is even slightly risker, this is not surprising for example, positional-only arguments are specific to Python. We surmise this points to the need a specific training and tools tuned to support passing by keywords and mixed cases.

In the following, we discuss some examples extracted while classifying the two stratified samples and deemed interesting, examples listed in Table 7.11

```
def get_desired_capabilities(app
    =None):
```

```
def get_desired_capabilities(app
    : Optional[str] = None) ->
    Dict[str, Any]:
```

Listing 7.1 Signature and parameter readability improvement

Readability changes mostly impact function definitions. This is, for example, what happened in Example 1 of Table 7.11, shown in Listing 7.1. In this case, the developers added type information and a script to automatically generate documentation using SPHINX [125].

```
def serialize(objects, user=
    None, serializer=None, *args,
     **kwargs):
```

```
def serialize(objects, user=
    None, serializer=None, **
    kwargs):
```

Listing 7.2 Multiple misuses: *args before **kwargs

Mixing packing/unpacking operators, posonlyargs, kwargs, and using dictionaries allows plenty of flexibility, yet it may lead to mistakes. Developers point exactly to this, see Example 2 (code in Listing 7.2) the commit message states: *serialize() had ‘*args‘ following*

```
return serialize([objects], user
   =user, serializer=serializer,
    *args, **kwargs)[0]
```

```
return serialize([objects], user
   =user, serializer=serializer,
    **kwargs)[0]
```

Listing 7.3 Wrong order of parameter types

*keyword arguments. This results in '*args' creating an error, as Python globs too many arguments into keyword args. After some discussion, the decision was to remove the problematic and unused '*args'.*, in file `src/sentry/api/serializers/base.py` line8. Such a change has ripple effects on call sites, developers also changed the function return (line 17 - code in Listing 7.3) where the "*args" argument is removed due to signature change propagating the change to all call sites..

```
def check_grads(f, *args, fwd=
  True):
```

```
def check_grads(f, *args, **
  kwargs):
   fwd = kwargs.pop('fwd', True
  )
```

Listing 7.4 Keyword misuse following *args example

Example 3 (code in Listing 7.4) is another example in which some confusion can occur. While the function definition does not contain a defect itself, Python does not allow keyword arguments before a positional argument, and thus a call such as `check_grads(f, mat, fwd=False, 'hello')` raises an error since there is a positional argument (*'hello'*) follows keyword argument (*fwd*). Note that the commit message is misleading, in fact, the issue is still there, because positional arguments cannot follow keywords arguments. The correct call should be: `check_grads(f, mat, 'hello', fwd=False)` or something alternative, provided that keyword-passing only occurs at the end of the call.

In Example 5 (code in Listing 7.5), the commit message states: *The change ... added a keyword argument before the *args definition, which doesn't work. We need to use **kwargs after the *names or *packages references.* In other words, the unpacking was performed using the wrong mechanism.

```
def _get_pkg_info(failhard=True
  , *packages):
```

```
def _get_pkg_info(*packages, **
  kwargs):
...
failhard = kwargs.pop('failhard'
  , True)
...
```

Listing 7.5 Wrong type order: keyword before *args

```
def status(name, *args, **
  kwargs):
```

```
def status(name, sig=None):
```

Listing 7.6 Fix overly loose signature example

```
def ext_pillar(minion_id,
  pillar, command):
```

```
def ext_pillar(minion_id,#
  pylint:disable=W0613
               pillar,# pylint:
  disable=W0613
               command):
```

Listing 7.7 Unused parameter workaround example

Example 4 (code in Listing 7.6) and Example 6 (code in Listing 7.7) are of less complicated changes. In the first case, the goal is to *fix overly loose signature*, *i.e.,* a signature accepting too much (in terms of passing arguments), while in the second case, an unused parameter was silenced to avoid warnings.

Example 10, code in Listing 7.13 is another example where a class design change, adding a new `dispersy` attribute has ripple affect on several signatures, forcing call sites fix. Call sites fix are among other the following:

- Rename and add a new argument, Example 7 (code in Listing 7.8);

```
slack = _post_message(channel,

   message,

   channel,

   api_key)
```

```
 slack = _post_message(channel,

   message,

   username,

   as_user,

   api_key)
```

Listing 7.8 Adding new arguments

```
elif flowfilter.match(flow, self
   .flt):
```

```
elif flowfilter.match(self.flt,
   flow):
```

Listing 7.9 Swapping arguments

```
aS.hmm_segmentation(wavFile,
   hmmModelName, plot_res=True,
   gt_file=gtFile)
```

```
aS.hmm_segmentation(wavFile,
   hmmModelName, plot_results=
   True, gt_file=gtFile)
```

Listing 7.10 Wrong argument passing

- Arguments swap, Example 8 (code in Listing 7.9);
- Fixing the expected argument Example 9 (code in Listing 7.10)
- Passing a non-unpacked dictionary (instead of a ***kwargs*), Example 11 (code in Listing 7.11); and

```
return server.hmset(key, **
    fieldsvals)
```

```
return server.hmset(key,
    fieldsvals)
```

Listing 7.11 Wrong usage of dictionary unpacking

```
self.store = PresenceStore(hs)
```

```
self.store = PresenceStore(None,
    hs)
```

Listing 7.12 Forgotten passed argument

```
  def join_community(self,
   my_member):
  ...
  self.my_channel =
   ChannelCommunity.
   create_community(self.
   my_member,

   integrate_with_tribler =
   False)
```

```
  def join_community(self,
   dispersy, my_member):
  ...
  self.my_channel =
   ChannelCommunity.
   create_community(self.
   _dispersy,

   self.my_member,

   integrate_with_tribler =
   False)
```

Listing 7.13 Added new passed argument

- Passing the correct number of arguments, Example 12 (code in Listing 7.12).

7.4 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. The first threat is related to how we define “fixes" in our study. We are fully aware that not all fixes declared as such in the commit messages are related to defects [126, 127]. However, as explained in Section 7.1, in our study, we are interested in fixes in general, also those related to the improvement of code readability and addressing potential problems by removing static analysis tool warnings. This is consistent with what was done in previous work [117]. Furthermore, we perform a qualitative analysis on a statistically significant sample of fix-inducing changes to mitigate such a threat. Also, we are aware that the considered regular expression to match fixes may miss some possible cases.

Threats to *internal validity* concern factors internal to our study that could influence our findings. While the study we have conducted shows a correlation between certain features in parameter definition and argument passing, this does not show causation. Also in this case, the threat is mitigated through a qualitative analysis.

We control for the effect of confounding factors such as the function size and complexity on the presence of fix-inducing changes. Certainly, it is possible that further metrics could better control such types of confounding factors.

Our study suffers from all the intrinsic limitations of the SZZ algorithm [128]. In particular, when a problem is induced in a signature, the corresponding fixing action can occur in the signature itself (only), or in both the signature and the function body. However, in principle, there may be cases in which the fix only occurs in the body, and these are not captured by the SZZ algorithm. The latter would require an SZZ implementation comprising a control and data-flow analysis between the signatures and the changed code. An example of such a case is a fix occurring in the *saltstack/salt* repository[1]

Threats to *conclusion validity* concern the relationship between observation and outcome. The findings of all four research questions are supported by suitable statistical procedures, *i.e.,* Fisher’s exact test and mixed-effect logistic regression models. In the presence of multiple tests, p-values are adjusted using suitable adjustment procedures (Benjamini-Hochberg [87]). Last, but not least, the magnitude of the observed differences is quantified using effect-size measures (Odds Ratio).

Threats to *external validity* concern the generalizability of our findings. First, our results are

[1] `https://github.com/saltstack/salt/commit/5de0b93/salt/modules/saltutil.py`

limited to the analysis of 200 Python open-source projects hosted on GitHub. It is possible that extending the study to further projects would lead to different results. Moreover, other programming languages provide similar features for argument passing, and it would be useful to check whether the obtained findings would be different in these languages.

7.5 Conclusion

This paper empirically analyzed the extent to which different features related to parameter passing, and different Python argument-passing (*i.e.,* positional or keyword-based) relate to fix-inducing changes. The analysis has been performed on 200 engineered, highly-forked open-source projects hosted on GitHub. The results of the study highlight that:

1. The presence, in function definitions, of positional parameters increases the odd to induce fixes.
2. In the same way, the definition of a variable argument (vararg) parameter also increases the odd to induce fixes.
3. Instead, default values and keyword-based arguments have an opposite effect, *i.e.,* they reduce the odd to induce fixes.
4. When looking at call sites, while additional arguments increase the odds to induce fixes when such arguments are positional, the increase is about 80% greater than when they are keyword-based.

In essence, and also based on the results of our qualitative analysis, mechanisms such as the definition of default values and the use of keyword-based argument passing tend to aid developers in avoiding making mistakes that often occur with a positional argument-passing. The qualitative analysis also shows how a proportion of fixes (30% for function definitions and 5% for call sites) is mainly related to readability enhancement.

Our study paves the way to integrate mechanisms in IDE to support developers when designing and defining the function signature and then using the functions at call sites. For example, special care should be used if default values are defined in keyword-based mechanisms. Last, but not least, our findings may also serve as a guide to prioritize code review or focus testing activities to certain call sites.

CHAPTER 8 CONCLUSION

This thesis embarks on an exploration of the Python functional programming ecosystem, utilizing empirical analyses to dissect various facets of this programming paradigm. The focal points include Python constructs such as list comprehensions, dictionary comprehensions, set comprehensions, lambdas, and higher-order functions.

Within this exploration, we introduce a research hypothesis:

> To what extent do Pythonic functional constructs impact code understandability, fault proneness, and performance, and how do these aspects vary across the different constructs under study?

This hypothesis drives our investigation, leading to the formulation of four specific research sub-questions listed in the following:

- **RQa:** *How does the presence of functional constructs impact the difficulty of code modification and source code understandability?*

Prior to addressing this question, we conducted a controlled experiment, detailed in Chapter 4. This experiment delved into the impact of functional Pythonic constructs, specifically lambdas, comprehensions, and map/reduce/filter (MRF), on program comprehension. Focusing on three research questions, the study explored the difficulty of modifying code snippets, the influence of functional constructs on perceived source code understandability, and the reasons and scenarios for developers using these constructs.

The results revealed nuanced insights:

- Developers using lambdas demonstrated better performance in change tasks compared to procedural alternatives, with this advantage diminishing as complexity increased.
- Comprehensions were generally more error-prone than procedural alternatives, with advantages emerging in higher complexity scenarios. No significant differences were found for MRF.

Participants perceived functional constructs as more challenging to understand than procedural alternatives, with variations based on usage frequency and, for comprehension, on

construct complexity. Developers acknowledged code size reduction as a benefit of using functional constructs, but challenges emerged during debugging in the functional paradigm.

The study highlighted that the difficulty of understanding and changing functional constructs varies, influenced by the specific construct and its complexity. Notably, the findings are based on participants with predominantly junior to medium-level seniority.

To enhance the generalizability of the results, future work aims to replicate the study in diverse contexts, including field studies with experienced Python professionals and experiments in classroom settings. Additionally, there is a need for tools to support developers in writing, testing, and debugging functional constructs, addressing the challenges identified in this study. The insights gained contribute to a deeper understanding of the practical implications of using functional constructs in Python programming.

- **RQb:** *To what extent do modifications involving functional constructs influence the frequency of fixes in the source code?*

In this empirical study, we thoroughly investigated the impact of changes to functional constructs in Python on the likelihood of inducing fixes, addressing various research questions. The findings reveal significant insights into the fault-proneness of functional constructs, encompassing types, cyclomatic complexity, and bug survivability introduced during changes involving these constructs.

In Chapter 5, we provided a detailed response to RQb_1, unveiling that changes affecting functional constructs have more than twice the odds of inducing fixes compared to other changes. Moreover, distinctions between additions and modifications of functional constructs are highlighted, with additions showing higher odds of inducing fixes, dependent on the size of the changed churn.

Analyzing different types of functional constructs, including lambdas, comprehensions, and map/reduce/filter functions, the research identifies variations in fault-proneness. Lambdas exhibit higher odds of inducing fixes, followed by comprehensions, while map/reduce/filter functions show a more limited effect.

Contrary to expectations, the study found that the cyclomatic complexity of functional constructs does not significantly impact their fault-proneness. In some cases, constructs with low complexity displayed higher odds of inducing fixes than those with high complexity.

Examining the survival time of bugs related to different functional constructs, the study identifies varying hazard rates. Filter functions and comprehensions exhibit higher hazard rates, indicating quicker removal of related bugs, while reduced functions show the lowest

hazard rate, suggesting longer bug survival. However, the qualitative analysis suggests that construct type may not be the sole determinant of bug survival, as constructs are often changed or removed due to broader system modifications.

Despite these valuable insights, it's essential to consider the study's limitations, including its focus on Python and specific functional constructs. The external validity of the findings might not extend to other programming languages or additional functional constructs. Replication studies and further investigations into constructs like immutability are recommended for a more comprehensive understanding.

- **RQc:** *Does the transformation of "for" "loops" into list comprehensions significantly contribute to performance improvements in real-world projects?*

Detailed in Chapter 6, delved into the runtime performance impact of transforming Python "for" loops into their equivalent list comprehensions, aiming to address two key research questions: (1) whether list comprehension code runs faster than its equivalent "for" loop code, and (2) if transforming "for" loops into list comprehensions results in faster real-world projects.

For (1), our analysis of synthetic, yet realistic code snippets revealed that list comprehensions tend to perform better than semantically equivalent "for" loop code. However, the results were nuanced when applied to real-world projects (2). The study, conducted on eleven Python projects, measured user-perceived execution times through test cases. The findings indicated that while there were instances of performance improvement, the differences were generally small or negligible. The median difference in execution times across projects remained below 20 seconds, challenging the notion of a substantial performance gain.

The study identified potential threats to construct validity, internal validity, conclusion validity, and external validity. To mitigate these threats, we utilized synthetic code snippets, performed thorough performance measurements, and applied statistical tests and effect size measures to draw reliable conclusions.

The results suggest that developers should carefully consider whether transforming "for" loops into list comprehensions is worthwhile, as previous studies have also indicated potential defect-proneness associated with list comprehensions. Future work aims to extend the dataset, both quantitatively and qualitatively, to gain a deeper understanding of the conditions under which the adoption of list comprehensions brings substantial benefits. Additionally, exploring the performance impact in the opposite direction—transforming list comprehensions into "for" loops—would provide a more comprehensive understanding of the trade-offs involved in using these constructs.

- **RQd:** *In what ways do changes related to parameter-passing in function definitions and call sites affect fault proneness?*

In Chapter 7, we conducted an empirical study that systematically investigated the relationship between different features related to parameter passing in Python function definitions and call sites, and the induction of fixes in open-source projects. The study addressed four research questions, focusing on the extent of fixes induced by changes to parameter-passing in function definitions and argument-passing in call sites.

The key findings of the study are as follows:

- The presence of positional parameters and variable argument (vararg) parameters in function definitions increases the odds of inducing fixes.
- Conversely, the definition of default values and the use of keyword-based arguments in function definitions reduce the odds of inducing fixes.
- In call sites, the addition of positional arguments increases the odds of inducing fixes, with a substantially greater impact compared to the addition of keyword-based arguments.
- The qualitative analysis complemented the quantitative results, revealing that a significant proportion of fixes, particularly in function definitions (30%) and to a lesser extent in call sites (5%), is related to readability enhancements.

The study's insights suggest opportunities for integrating mechanisms into Integrated Development Environments (IDEs) to support developers in designing and defining function signatures and using functions at call sites. Developers should exercise caution when using default values in keyword-based mechanisms. The findings also provide guidance for prioritizing code review or focusing testing activities on specific call sites.

While the study offers valuable insights into the relationship between parameter-passing mechanisms and fixes, its external validity is limited to the analysis of 200 Python open-source projects on GitHub. Future work could extend the study to include a broader set of projects and explore similar features in other programming languages. Overall, this research contributes to enhancing our understanding of the factors influencing fix-inducing changes related to parameter and argument passing in Python code.

8.1 Limitations

While the research endeavors to provide a comprehensive view, it's essential to acknowledge certain limitations. The scope of the empirical analyses may not cover all potential scenarios, and the generalizability of the findings could be influenced by the specific set of projects and contexts considered. Additionally, the dynamic nature of the software development landscape may introduce variability not entirely captured in this study. Here are a few points that may affect the results :

1. Project Selection:
   The study examined a set of 200 projects, however, we do not have information on the specific types and domains of these projects. Different types of software projects can have distinct characteristics that may impact development practices and outcomes.

2. Developer Details:
   Understanding the background and experience of developers is essential. Factors such as the number of developers per project, their experience level, coding backgrounds, and any specialized skills they bring to the table could significantly influence project outcomes.

3. Company Culture:
   The owning company of a project plays a pivotal role. Corporate culture, development methodologies, and budget considerations can vary widely among companies, potentially impacting the software development process and outcomes.

4. Budgetary Considerations:
   The study briefly mentions the potential impact of budget, but a more in-depth exploration of how varying budgetary constraints may affect development practices, timelines, and ultimately, the quality of the software would provide valuable insights.

5. Validation Process:
   The study does not touch upon the specifics of the verification and validation process. Understanding whether there is a formal validation process in place, the criteria for validation, and how it aligns with industry standards could shed light on the reliability of the study's findings.

6. Coding Style Charter:
   The existence of a coding style charter and its adherence by developers is a crucial factor influencing the consistency and maintainability of the codebase. This could impact the overall quality of the software.

7. Testing Quality and Coverage:
   Information on the quality and types of tests conducted is vital. Additionally, how the detection and resolution of errors or bugs influence the overall coverage of tests could provide insights into the robustness of the software.

In summary, a more detailed exploration of these aspects as well as the tool chains adopted in the development and deployment process would enhance the study's comprehensiveness and provide a richer understanding of the factors influencing software development outcomes.

8.2 Future Research

For future research, we can delve into three aspects related to Python:

- Python for Developer: how developers interact with Python, focusing on development tools, user experience in programming, and ease of learning for newcomers.
- Python Performance: Investigate aspects related to the performance of Python by analyzing possible optimizations, coding best practices to enhance efficiency, and the impact of different approaches on application performance.
- Python Code Evolution: Study how Python code evolves, concentrating on best practices for managing legacy code, strategies for migrating to new Python versions, and the impact of language evolution on the maintenance of existing code.

By addressing these three aspects, we can gain a better understanding of how Python aligns with developers, how it can be optimized for maximum performance, and how it can be effectively managed for language evolution. This would not only enhance the productivity of developers working with Python but also improve the quality and sustainability of applications developed in this language.

8.2.1 Python Developer

In Chapter 4, various aspects of Python were thoroughly examined, encompassing code length, algorithmic complexity, and diverse functionalities. It has been noted that there isn't always a clear or universally applicable guideline regarding these aspects. Consequently, code comprehension remains a highly personal experience, varying from individual to individual, and is not easily generalizable.

Furthermore, it is important to highlight that the study did not comprehensively cover some fundamental and Python-specific programming constructs. Among these, the use of type

hints with the typing module and the exploration of closures deserve in-depth analysis. Type hints, for instance, provide the opportunity to add type annotations to functions, potentially enhancing code readability and maintainability. Closures, on the other hand, enable advanced functionality by capturing variables from the outer environment, significantly influencing code design and modularity.

Understanding these additional elements of Python programming can provide a more holistic perspective on how developers interact with the Python programming language.

The central question that arises is: how can this be improved? Establishing a comprehensive code charter and defining best practices tailored to the type of project or application domain could be a beneficial approach. Such an initiative would standardize coding aspects, including code length, complexity, and the usage of different Python constructs, ultimately enhancing team collaboration. Conducting experiments or project analyses, with a specific focus on the developer experience, could provide valuable insights into whether technical evolution is primarily influenced by individual preferences or shaped by the specific requirements of each domain. This is especially crucial given Python's widespread prevalence and its ongoing evolution.

In parallel, the adoption of Large Language Models (LLM) and coding support tools has introduced a paradigm shift in the coding landscape. Developers now have access to advanced tools that offer code suggestions, corrections, and even code generation. This technological advancement poses both opportunities and challenges for developers, necessitating a thoughtful integration of these tools into the development workflow. The consequences of this adoption are not only technical but also impact the way developers approach problem-solving and code creation.

Regarding debugging in a more sophisticated IDE, such as PyCharm or Visual Studio Code with Python extension, handling complex constructs like list comprehensions becomes more accessible. Developers can set breakpoints, inspect variables, and step through the code execution, facilitating a granular understanding of the list comprehension's behavior during runtime. This underscores the importance of leveraging powerful integrated development environments to streamline the debugging process and enhance overall code quality. Future works should consider exploring the impact of these coding support tools and advanced integrated development environments on developers' productivity and software project quality.

8.2.2 Python Performance

The study delves into the impact of transforming Python `for` loops into list comprehensions in terms of execution performance. While functional constructs prove effective for synthetic code, the results vary for real-world projects. However, it is crucial to note that this study exclusively focused on comparing `for` loops and list comprehensions. Other Python structures, such as user-defined functions (`def`) and lambda functions, could also undergo similar comparisons in a comparable context.

For instance, exploring which approach (user-defined function or lambda) performs better in specific situations would be interesting. Additionally, other aspects of the Python language, like data manipulation, could be included in similar analyses.

This study was conducted on a limited sample of projects, mainly due to the complexity associated with executing multiple projects and the variability of results. Future work could consider expanding the scope of comparisons, accounting for different types of transformations, application domains, data types, and data volumes. This would provide a more in-depth understanding of the relative performances of various structures and approaches in diverse scenarios.

8.2.3 Python Code Evolution

The evolution of Python, celebrated for its simplicity and user-friendliness, nonetheless presents significant challenges. The transition from Python 2 to Python 3 marked a major milestone, introducing substantial language syntax and functionality changes. This thesis exclusively focused on Python 3, a decision that unraveled its complexity.

A concrete example of this complexity lies in the analysis of the reduce function. Initially, I was misled, thinking it had been removed, only to later realize it had been relocated. This observation underscores the challenges that changes in the location of functionalities can introduce during Python's evolution. Another notable example is the evolution of lambda expressions in Python. They started as neat, anonymous functions, but over time, they changed the way parameters are handled. Early versions had a simple list of parameters without type annotations. However, with Python 3.0, things got fancier, allowing developers to use type annotations in lambdas for clearer code. Python 3.8 even brought in the Walrus Operator (:=), shaking things up with more concise code.

The execution of projects added a layer of challenges. Ensuring the use of appropriate versions is not always explicitly mentioned in project configurations or requirements. Developers often encounter compatibility issues with dependencies, and each library may not work seamlessly

with the latest versions of Python, demanding extra effort from developers.

Different ways to install libraries contribute to this complexity. Python supports multiple package management tools, such as pip, conda, and others, each with its own set of quirks and conventions. The choice of package manager can impact the compatibility and installation process. Additionally, libraries themselves may have version-specific requirements or behaviors. Some libraries are optimized for certain Python versions, and developers need to be aware of these nuances to ensure smooth integration into their projects.

An intriguing aspect of this evolution pertains to the modification of the AST (Abstract Syntax Tree) structure from one Python 3 version to another. This internal evolution adds further complexity, necessitating ongoing adaptation of static analysis tools and development practices.

While Python is celebrated for its user-friendliness, its evolution is not without difficulties. Transitions between major versions and continual adjustments in dependencies and internal structures require proactive management and constant adaptation of development practices. These challenges, though sometimes perceived as obstacles, underscore the importance of understanding and mastering the language's evolution.

In addressing these issues, the crucial question is how to approach this evolution consciously and adaptively, especially considering different application domains. Future work could involve developing strategies for managing Python's evolution, emphasizing awareness of this evolution, and implementing tailored practices. This might include creating best practice guidelines, domain-specific recommendations, and tools facilitating smooth transitions between versions. In essence, while acknowledging the benefits of evolution, it is essential to explore ways to control it more effectively and seamlessly integrate it into the development process.

REFERENCES

[1] F. Zampetti, F. Belias, C. Zid, G. Antoniol, and M. Di Penta, "An empirical study on the fault-inducing effect of functional constructs in python," in *IEEE International Conference on Software Maintenance and Evolution*, 2022.

[2] Z. Zhang, Z. Xing, X. Xia, X. Xu, L. Zhu, and Q. Lu, "Faster or slower? performance mystery of python idioms unveiled with empirical evidence," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1495–1507.

[3] 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.

[4] D. Spadini, F. Palomba, T. Baum, S. Hanenberg, M. Bruntink, and A. Bacchelli, "Test-driven code review: an empirical study," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019, pp. 1061–1072.

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

[6] 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.

[7] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[8] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[9] R. Dyer and J. Chauhan, "An exploratory study on the predominant programming paradigms in python code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 2022, pp. 684–695.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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, 2013, pp. 543–553.

[17] 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.

[18] C. Zid, F. Zampetti, G. Antoniol, and M. Di Penta, "A study on the pythonic functional constructs' understandability," in *Proceedings of the 45th ACM/IEEE International Conference on Software Engineering (ICSE 2024), May 14-20 2024, Lisbon, Portugal*. ACM/IEEE, 2024.

[19] I. Schröter, J. Krüger, J. Siegmund, and T. Leich, "Comprehending studies on program comprehension," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 308–311.

[20] J. C. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," *Empir. Softw. Eng.*, vol. 24, no. 1, pp. 417–443, 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9621-x

[21] J. Johnson, S. Lubo, N. Yedla, J. Aponte, and B. Sharif, "An empirical study assessing source code readability in comprehension," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 513–523. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00085

[22] R. A. McCauley, B. Hanks, S. Fitzgerald, and L. Murphy, "Recursion vs. iteration: An empirical study of comprehension revisited," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE 2015, Kansas City, MO, USA, March 4-7, 2015*. ACM, 2015, pp. 350–355.

[23] C. S. Yu, C. Treude, and M. F. Aniche, "Comprehending test code: An empirical study," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 501–512.

[24] A. Jbara and D. G. Feitelson, "How programmers read regular code: a controlled experiment using eye tracking," *Empirical software engineering*, vol. 22, pp. 1440–1477, 2017.

[25] N. Peitek, J. Siegmund, and S. Apel, "What drives the reading order of programmers?: An eye tracking study," in *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 342–353.

[26] J. Bauer, J. Siegmund, N. Peitek, J. C. Hofmeister, and S. Apel, "Indentation: simply a matter of style or support for program comprehension?" in *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 154–164.

[27] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund, "Program comprehension and code complexity metrics: A replication package of an fmri study," in *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 168–169.

[28] S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, and O. Adesope, "Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization," *Empirical Software Engineering*, vol. 25, pp. 2140–2178, 2020.

[29] S. Hanenberg, "An experiment about static and dynamic type systems doubts about the positive impact of static type systems on development time," vol. 45, 10 2010, pp. 22–35.

[30] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "Program indentation and comprehensibility," *Commun. ACM*, vol. 26, no. 11, p. 861–867, nov 1983. [Online]. Available: https://doi.org/10.1145/182.358437

[31] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *The Psychological Review*, vol. 63, no. 2, pp. 81–97, 1956.

[32] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K. Yeh, and J. Cappos, "Understanding misunderstandings in source code," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, Sept. 4-8, 2017*. ACM, 2017, pp. 129–139.

[33] D. Gopstein, H. H. Zhou, P. G. Frankl, and J. Cappos, "Prevalence of confusing code in software projects: atoms of confusion in the wild," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 2018, pp. 281–291.

[34] C. Langhout and M. Aniche, "Atoms of confusion in java," in *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 2021, pp. 25–35.

[35] D. Gopstein, A. Fayard, S. Apel, and J. Cappos, "Thinking aloud about confusing code: a qualitative investigation of program comprehension and atoms of confusion," 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*. ACM, 2020, pp. 605–616.

[36] B. de Oliveira, M. Ribeiro, J. A. S. da Costa, R. Gheyi, G. Amaral, R. M. de Mello, A. Oliveira, A. F. Garcia, R. Bonifácio, and B. Fonseca, "Atoms of confusion: The eyes do not lie," in *34th Brazilian Symposium on Software Engineering, SBES 2020, Natal, Brazil, October 19-23, 2020*. ACM, 2020, pp. 243–252.

[37] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, "Antipatterns: Refactoring software, architectures, and projects in crisis," 1998.

[38] F. Palomba, A. D. Lucia, G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues," *Advances in Computers*, vol. 95, pp. 201–238, 2015.

[39] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013*, 2013, pp. 187–196.

[40] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany.* IEEE Computer Society, 2011, pp. 181–190.

[41] 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.

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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.

[49] 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

[50] 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

[51] 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

[52] 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.

[53] 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

[54] 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

[55] 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.

[56] S. Ajami, Y. Woodbridge, and D. G. Feitelson, "Syntax, predicates, idioms - what really affects code complexity?" in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 66–76.

[57] D. Gopstein, H. H. Zhou, P. Frankl, and J. Cappos, "Prevalence of confusing code in software projects: Atoms of confusion in the wild," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 281–291. [Online]. Available: https://doi.org/10.1145/3196398.3196432

[58] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, and R. Gheyi, "An investigation of misunderstanding code patterns in c open-source software projects," *Empirical Software Engineering*, vol. 24, pp. 1693–1726, 2019.

[59] J. Dolado, M. Harman, M. Otero, and L. Hu, "An empirical investigation of the influence of a type of side effects on program comprehension," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 665–670, 2003.

[60] J. A. S. da Costa, R. Gheyi, F. Castor, P. R. F. de Oliveira, M. Ribeiro, and B. Fonseca, "Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension," *Empirical Software Engineering*, vol. 28, no. 4, p. 81, 2023.

[61] V. Bogachenkova, L. Nguyen, F. Ebert, A. Serebrenik, and F. Castor, "Evaluating atoms of confusion in the context of code reviews," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 404–408.

[62] A. J. Offutt, "Unit testing versus integration testing," in *Proceedings IEEE International Test Conference 1991, Test: Faster, Better, Sooner, Nashville, TN, USA, October 26-30, 1991*. IEEE Computer Society, 1991, pp. 1108–1109.

[63] S. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," *Softw. Test. Verification Reliab.*, vol. 11, no. 3, pp. 207–225, 2001.

[64] Z. Jin and A. J. Offutt, "Coupling-based criteria for integration testing," *Softw. Test. Verification Reliab.*, vol. 8, no. 3, pp. 133–154, 1998.

[65] A. Hamidi, G. Antoniol, F. Khomh, M. D. Penta, and M. Hamidi, "Towards understanding developers' machine-learning challenges: A multi-language study on stack overflow," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021.

[66] A. Nagpal and G. Gabrani, "Python for data analytics, scientific and technical applications," in *2019 Amity International Conference on Artificial Intelligence (AICAI)*, 2019, pp. 140–145.

[67] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976. [Online]. Available: https://doi.org/10.1109/TSE.1976.233837

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

[69] "Python.org", ""pep 570 – python positional-only parameters https://peps.python.org/pep-0570/"."

[70] *Explanatory Item Response Models : A Generalized Linear and Nonlinear Approach*. Springer New York, March 2013.

[71] 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.

[72] P. Inc., "Prolific https://www.prolific.co," 2023, (Last access: 08/11/2023).

[73] M. Tahaei and K. Vaniea, "Lessons learned from recruiting participants with programming skills for empirical privacy and security studies," in *1st International Workshop on Recruiting Participants for Empirical Software Engineering*, 2022.

[74] J. Jiarpakdee, C. Tantithamthavorn, and J. C. Grundy, "Practitioners' perceptions of the goals and visual explanations of defect prediction models," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 2021, pp. 432–443. [Online]. Available: https://doi.org/10.1109/MSR52588.2021.00055

[75] B. Reid, M. d'Amorim, M. Wagner, and C. Treude, "NCQ: code reuse support for node.js developers," *IEEE Trans. Software Eng.*, vol. 49, no. 5, pp. 3205–3225, 2023.

[76] F. Ebert, A. Serebrenik, C. Treude, N. Novielli, and F. Castor, "On recruiting experienced github contributors for interviews and surveys on prolific," in *International Workshop on Recruiting Participants for Empirical Software Engineering*, 2022.

[77] B. Reid, M. Wagner, M. d'Amorim, and C. Treude, "Software engineering user study recruitment on prolific: An experience report," *CoRR*, vol. abs/2201.05348, 2022.

[78] D. Russo, "Recruiting software engineers on prolific," 2022.

[79] F. Zampetti, F. Belias, C. Zid, G. Antoniol, and M. D. Penta, "An empirical study on the fault-inducing effect of functional constructs in python," in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*. IEEE, 2022, pp. 47–58.

[80] T. Mccabe, "Cyclomatic complexity and the year 2000," *IEEE Software*, vol. 13, no. 3, pp. 115–117, 1996.

[81] G. Scanniello and M. Risi, "Dealing with faults in source code: Abbreviated vs. full-word identifier names," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 2013, pp. 190–199.

[82] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato, "How developers' experience and ability influence web application comprehension tasks supported by UML stereotypes: A series of four experiments," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 96–118, 2010.

[83] A. N. Oppenheim, *Questionnaire design, interviewing and attitude measurement*. Bloomsbury Publishing, 2000.

[84] Open AI, "Gptzero https://gptzero.me/," 2023, (Last access: 08/11/2023).

[85] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[86] K. Krippendorff, "Computing krippendorff's alpha-reliability," 2011.

[87] 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.

[88] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S*, 4th ed. New York: Springer, 2002, iSBN 0-387-95457-0. [Online]. Available: http://www.stats.ox.ac.uk/pub/MASS4

[89] Z. Zhang, Z. Xing, X. Xia, X. Xu, L. Zhu, and Q. Lu, "Faster or slower? performance mystery of python idioms unveiled with empirical evidence," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1495–1507.

[90] Z. Zhang, Z. Xing, X. Xia, X. Xu, and L. Zhu, "Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 2022, pp. 696–708.

[91] Logilab and Pylint contributors, "Pylint," 2023. [Online]. Available: https://pylint.pycqa.org/

[92] J. D. Long and N. Cliff, "Confidence intervals for kendall's tau," *British Journal of Mathematical and Statistical Psychology*, vol. 50, no. 1, pp. 31–41, 1997.

[93] C. Zid, F. Zampetti, G. Antoniol, and M. Di Penta, "Replication package for the paper: "A Study on the Pythonic Functional Constructs' Understandability"," December 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8191782

[94] C. Zid, F. Belias, F. Khomh, M. Di Penta, and G. Antoniol, "List comprehension versus for loops performance in real python projects: Should we care?" in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2024) , Tue 12 - Fri 15 March 2024 Rovaniemi , Finland*. ACM/IEEE, 2024.

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

[96] 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)," 2022.

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

[98] 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.

[99] F. Zampetti, C. Zid, G. Antoniol, and M. Di Penta, "An Empirical Study on the Fault-Inducing Effect of Functional Constructs in Python (extension)," Dec 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10462655

[100] 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

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

[102] 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

[103] 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 *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 104–113.

[104] 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.

[105] 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

[106] 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.

[107] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "How long does a bug survive? an empirical study," in *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, 2011, pp. 191–200. [Online]. Available: https://doi.org/10.1109/WCRE.2011.31

[108] 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

[109] J. Rupert G. Miller, *Survival Analysis, 2nd Edition*. John Wiley and Sons, 2011.

[110] T. M. Therneau, *A Package for Survival Analysis in R*, 2023, r package version 3.5-7. [Online]. Available: https://CRAN.R-project.org/package=survival

[111] T. A. Gerds, *prodlim: Product-Limit Estimation for Censored Event History Analysis*, 2023, r package version 2023.08.28. [Online]. Available: https://CRAN.R-project.org/package=prodlim

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

[113] Z. Zhang, Z. Xing, X. Xia, X. Xu, and L. Zhu, "Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 696–708. [Online]. Available: https://doi.org/10.1145/3540250.3549143

[114] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1–10. [Online]. Available: https://doi.org/10.1145/1985793.1985795

[115] Anonymous, "Replication package for the paper: "List Comprehension Versus for Loops Performance in Real Python Projects: Should we Care?"," november 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10149938

[116] GitHub, "Top programming languages in 2022, https://octoverse.github.com/2022/top-programming-languages."

[117] F. Zampetti, F. Belias, C. Zid, G. Antoniol, and M. Di Penta, “An empirical study on the fault-inducing effect of functional constructs in python,” in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*, 2022, pp. 47–58. [Online]. Available: https://doi.org/10.1109/ICSME55016.2022.00013

[118] Anonymous, “Dataset for the paper "The Relationship Between Different Python Argument-Passing Mechanisms and Fixes: An Empirical Study",” Apr. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7870870

[119] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[120] 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.

[121] T. McCabe, “A complexity measure,” *IEEE Trans. on Softw. Eng.*, no. 4, pp. 308–320, 1976.

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

[123] Python.org, “Python: support for type hints, https://docs.python.org/3/library/typing.html.”

[124] ——, “Pep 287 – restructuredtext docstring format, https://peps.python.org/pep-0287/.”

[125] Sphinx, “Sphinx - python documentation generator, https://www.sphinx-doc.org/.”

[126] 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.

[127] K. Herzig, S. Just, and A. Zeller, “It’s not a bug, it’s a feature: how misclassification impacts bug prediction,” in *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013*. IEEE / ACM, 2013, pp. 392–401.

[128] 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.