Comprehensive Analysis of Programming Language Idioms and Their Application to Software Quality

Zejun Zhang

A thesis submitted for the degree of Doctor of Philosophy

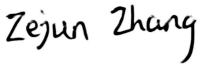
School of Computing
ANU College of Engineering & Computer Science

May, 2024

Declaration

I, Zejun Zhang, declare that this thesis titled, 'Comprehensive Analysis of Programming Language Idioms and Their Application to Software Quality' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.



22 November 2024

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Zhenchang Xing. While he often appears to take a relaxed approach with us, his meticulous approach to research and forward-thinking mindset have been relentlessly inspiring. Every discussion with him leaves me simultaneously feeling both challenged and motivated. His guidance has not only refined my research perspective but also instilled in me a rigorous attitude to scientific research. I am truly grateful for his mentorship, which has played a pivotal role in shaping my academic journey.

I am also tremendously grateful to Professor Xin Xia, who introduced me to Prof. Xing. Professor Xia's quick thinking and efficient work style have been incredibly inspirational, encouraging me to adopt a more agile and decisive approach in my own work. Special thanks go to Professor John Grundy. His approachable nature and infectious enthusiasm for research have left a lasting impression on me. I fondly remember our extended lunchtime conversations at Monash University, where he would eagerly discuss various research directions for over an hour, sharing his expansive knowledge and unbridled passion for the field. I would also like to extend my heartfelt thanks to Professor David Lo. Despite his vast academic influence and numerous students, he remains remarkably approachable and polite. He has always been willing to help, responding to my queries with enthusiasm and providing valuable assistance.

I also wish to acknowledge the numerous researchers who have provided assistance and guidance throughout my journey. Your contributions have been indispensable to my progress.

I am immensely grateful for the companionship and unwavering support of researchers and my fellow students. At Zhejiang University, I owe a debt of gratitude to Lingfeng Bao, Chao Ni, XingHu, Zhiyuan Wan, Xiaoxue Ren, Zhongxin Liu, Junwei Zhang, Yanming Yang, Kaiwen Yang, Yuanrui Fan, Shengyi Pan, Zhipeng Gao, Jiakun Liu, Neng Zhang, Chao Liu, Haoye Wang, Bo Yang, Fangcheng Qiu, Zhan Qi, Yi Gao, Weixin Lin, Zhuang Liu, and Kexing Chen. Similarly, at ANU, I am thankful for the camaraderie and support of Dehai Zhao, Jiamou Sun, Mulong Xie, Jieshan Chen, Mengyu Chen, Yanqi Su, Xinyuan Ye, Shidong Pan, and Zhen Tao. Their companionship in both life and research has been invaluable to me, and I extend my deepest gratitude to each and every one of them.

Lastly, I owe my profound thanks to my parents. Whenever I faced challenges or frustrations in my research and life, they were my first point of call. My mother, in particular, has been a source of endless support, offering her insightful perspectives and listening patiently to my problems. Her encouragement has been crucial to my progress, and I sincerely believe she shares half of my academic achievements. I love you both dearly, and your support means the world to me.

Abstract

Achieving high-quality code is fundamental to the success and sustainability of software development projects. Software quality encompasses multiple facets, including maintainability, readability and performance, each posing its unique challenges. Achieving these goals requires a deep understanding of best practices and effective coding conventions. Programming idioms, known for their their conciseness and improved performance, offer substantial benefits but also introduce complexity that can hinder their adoption. Compared to idioms in other programming languages, Pythonic idioms are particularly well-regarded and receive significant attention from the Python community and developers. This thesis explores the enhancement of software quality through the strategic use of Pythonic idioms and systematic bi-directional transformation between idiomatic and non-idiomatic Python code, focusing on three critical aspects: code maintainability, readability, and performance optimization.

Firstly, we introduce an automated refactoring tool, RIdiom that transforms non-idiomatic Python code into idiomatic forms, significantly enhancing maintainability and reducing manual effort. By leveraging Large Language Models (LLMs) in a hybrid knowledge-driven approach, we synergize rule-based methods with the adaptability of LLMs to improve code detection and refactoring accuracy. This integration addresses common challenges in code transformation, boosting both precision and consistency. Additionally, our approach promotes the consistent use of best practices across codebases, ultimately contributing to more maintainable and robust software. To aid developers' understanding and proper utilization of Pythonic idioms, we explain these idioms into equivalent non-idiomatic code, bridging comprehension gaps and ensuring semantic accuracy. This intuition is particularly beneficial for developers less familiar with Pythonic constructs, helping them to avoid common pitfalls and write more idiomatic Python code. Furthermore, we conduct an empirical study to evaluate the performance impacts of Pythonic idioms, providing developers with clear, evidence-based guidelines for optimal code performance. Our study reveals that while Pythonic idioms generally enhance performance, their impact can vary based on specific usage contexts and data characteristics. Our findings reveal nuanced insights into the benefits and limitations of Pythonic idioms, supporting developers in writing more maintainable, readable and efficient code.

Keywords: Programming Idioms, Software Quality, Code Maintainability, Code Readability, Performance Optimization

Contents

Figures

Tables

Abbreviations

ANU	**A**ustralian **N**ational **U**niversity
AST	**A**bstract **S**yntax **T**ree
ARI	**A**nalytic **R**ule **I**nterface
DE	**D**eviance **E**xplained
GAM	**G**eneralized **A**dditive **M**odel
LLM	**L**arge **L**anguage **M**odel
LOC	**L**ine **O**f **C**ode
VM	**V**irtual **M**achine

Chapter 1

Introduction

In the realm of software engineering, achieving high-quality software is paramount for ensuring the success and sustainability of software development projects. This pursuit necessitates expertise in various dimensions, including code maintainability, readability and performance optimization. In this thesis, we focus on advancing the quality of software development through the lens of Pythonic idioms. Figure 1.1 shows the overview of our research works.

Our research addresses significant gaps in understanding, utilizing, and integrating Pythonic idioms to improve code quality. We first developed the first automatic refactoring tool to transform non-idiomatic code into idiomatic code, thus enhancing the maintainability of the codebase. The second work capitalizes on the capabilities of Large Language Models (LLMs) to create a hybrid knowledge-driven approach that combines rule-based methods with LLM adaptability for more accurate and scalable code idiomatization. In our third effort, we examine the challenges developers face in comprehending Pythonic idioms and provide equivalent non-idiomatic code to facilitate better understanding. Lastly, we investigate the performance impacts of Pythonic idioms using a comprehensive empirical study, revealing nuanced insights into their effects on both synthetic and real-world code.

These contributions collectively aim to bridge the gaps in knowledge and tooling for Pythonic idioms, providing developers with the resources and automation needed to write more maintainable, readable and performant code.

1.1 Motivations and Goals

1.1.1 Improving Code Maintainability through Automated Refactoring

Code maintainability is a crucial aspect of software quality. Maintainable code is easier to update, extend, and debug, reducing the long-term cost and effort associated with the software lifecycle. Code refactoring, which involves restructuring existing code without altering its external behavior, is a key technique for enhancing code maintainability. By improving the internal organization of the code, refactoring can make the codebase more understandable and adaptable to changes.

Pythonic idioms, known for their brevity and performance advantages, are often considered best practices that promote readability and maintainability. However, the diverse and intricate nature of these idioms presents a significant challenge for automated refactoring. Despite the clear benefits of using Pythonic idioms, developers often struggle to refactor non-idiomatic code into its idiomatic counterparts due to the lack of automated support tools.

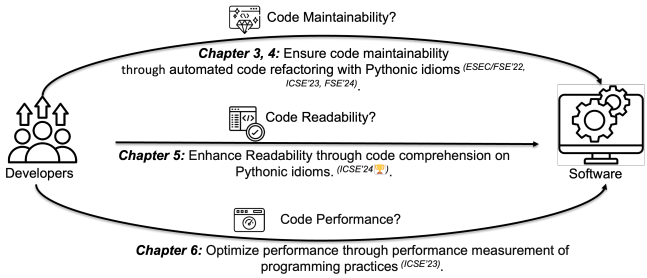


Figure 1.1: The overview of our research work.

Unfortunately, there are currently no tools available that can automatically refactor non-idiomatic Python code with Pythonic idioms. To our knowledge, only Pylint and Teddy provide limited support for Pythonic idioms. Pylint can detect chain-comparison and truth-value-test patterns, but its suggestions are often unclear, as evidenced by developers' confusion over simplifying chained comparisons. Teddy collects non-idiomatic code fragments and their idiomatic counterparts, suggesting replacements for three Pythonic idioms. However, both tools still require developers to manually perform the refactorings.

Therefore, my goal is to develop an automated refactoring tool that can automatically refactor non-idiomatic code with Pythonic idioms. This would significantly enhance code maintainability by reducing the manual effort required and promoting the consistent use of Pythonic idioms.

1.1.2 Enhancing Readability through Code Comprehension

Code readability is a cornerstone of software quality, fundamentally influencing system reliability, maintainability, and security. High readability ensures that code is understandable and accessible to different developers, which in turn facilitates easier debugging, updating, and extending of the software. This is crucial for guaranteeing that a program performs as expected across various conditions, thereby minimizing the risk of errors that could lead to major system failures or security vulnerabilities. Consequently, the methodologies and tools designed to enhance code readability and correctness play a vital role in both academic research and industrial practice.

One of the most effective ways to enhance code readability is by improving code comprehensibility. When developers can easily understand the code they are working on, they are better equipped to identify and rectify potential issues, implement new features correctly, and maintain the system over time. Clarity in code facilitates better peer reviews, more effective debugging processes, and more accurate modifications, all contributing to the overall correctness of the software.

Pythonic idioms represent concise and efficient programming practices that align with coventions of Python programming language. These idioms embody best practices and standard conventions, often leading to more readable and maintainable code. For instance, appending elements to a list with list comprehension idioms not only makes the code more succinct but also leverages optimizations built into the language.

However, despite their advantages, Pythonic idioms pose a significant challenge due to their inherent complexity and diversity. Many idioms abstract intricate operations into seemingly simple constructs, which can be perplexing for developers unfamiliar with them. This lack of understanding can lead to misinterpretation and misuse, ultimately compromising code correctness.

Despite the critical role of Pythonic idioms in writing correct and efficient code, there is a notable gap in research focusing on improving developers' comprehension of these constructs. Most existing studies emphasize the formal aspects of programming languages or tools for error detection, without addressing the cognitive processes by which developers learn and understand idiomatic patterns.

To bridge the gap in understanding and utilizing Pythonic idioms effectively, we aim to deeply analyze the challenges developers face in understanding Pythonic idioms. Based on this analysis, we will provide equivalent non-idiomatic code composed of common syntax from various programming languages. This will help developers, even those with limited experience in Python, to correctly understand and use Pythonic idioms.

1.1.3 Analyzing Code Performance Through Measurement

Code performance is a critical aspect of software quality, directly affecting user experience, resource utilization, and overall system efficiency. However, achieving optimal performance is often challenging due to the multitude of ways in which code functionality can be implemented, each with varying runtime efficiencies that may not be readily apparent to developers.

One significant hurdle is the confusion developers face regarding the performance implications of idiomatic versus non-idiomatic code. While Pythonic idioms are celebrated for their readability and conciseness, their impact on performance is not always intuitive. For example, the performance differences between list comprehensions (an idiomatic construct) and traditional for-loops (a more

straightforward but less idiomatic approach) can lead to heated debates among developers. Such discussions often hinge on personal programming experience and anecdotal evidence rather than empirical data, further muddling the understanding of idiomatic efficiency.

This confusion extends beyond idioms to encompass a broader range of performance-related issues. Developers frequently lack the tools and knowledge necessary to make informed decisions about which programming constructs and patterns will yield the best performance in a given context. This knowledge gap can result in suboptimal code that, while functionally correct, does not perform as efficiently as possible.

To address these challenges, our goal is to conduct systematic research into the performance impacts of Pythonic idioms compared to their non-idiomatic counterparts. By rigorously benchmarking and analyzing these constructs, we aim to provide clear, evidence-based guidelines that can help developers make better-informed decisions about code performance.

1.2 Main Works and Contributions

This thesis consists of four primary works: automated refactoring with Pythonic idioms, hybrid knowledge-driven refactoring to Pythonic idioms leveraging Large Language Models (LLMs), enhancing the understanding of Pythonic idioms with equivalent non-idiomatic code, and uncovering the performance mysteries of Pythonic idioms through empirical evidence. All works have been published in top-tier conferences, and all source code, models, and datasets are available in GitHub repositories.

In the first work, to help developers use Pythonic idioms, we developed the first automatic refactoring tool that detects nine types of non-idiomatic code (referred to as anti-idiom code smells) and refactors them into idiomatic code implementing the same functionalities. Unlike existing work that searches popular Python books and online sources to create catalogs of idioms, we identify unique Pythonic idioms by contrasting the language syntax of Python with Java, resulting in the identification of nine pythonic idioms, four of which were previously unrecognized. We confirmed the validity of these idioms through the Python language specification and online materials. Our tool formulates atomic AST rewriting operations for each idiom to refactor anti-idiom code. Applied to 7,638 Python projects, our tool detected and refactored over 2,252,022 anti-idiom code smells with high accuracy, achieving 100% detection accuracy for six idioms and 100% refactoring accuracy for eight idioms, with minimal errors. We also submitted 90 pull requests to 84 projects for real-world validation, receiving positive feedback on 62% of these requests, indicating the practical value of our refactoring tool. The feedback from developers has revealed concerns about the readability and performance of some Pythonic idioms, providing a rich dataset for further investigation into these areas. We published this work at ESEC/FSE 2022 (Zhang, Xing, Xia, Xu and Zhu, 2022), and later developed the RIdiom tool, which was published at ICSE 2023 (Zhang, Xing, Xu et al., 2023a).

In the second work, leveraging the capabilities of Large Language Models (LLMs), we developed a hybrid knowledge-driven approach to help developers use Pythonic idioms. This approach combines the deterministic nature of rule-based methods with the adaptability of LLMs, addressing the challenges of code miss, wrong detection, and wrong refactoring. Our method involves three core modules—knowledge, extraction, and idiomatization—which use Analytic Rule Interfaces (ARIs) and LLM prompts to transform non-idiomatic code into idiomatic code effectively. We evaluated our approach on nine Pythonic idioms identified by existing studies and extended it to four new idioms, achieving superior performance in accuracy, F1-score, precision, and recall compared to existing methods. Our results demonstrate high effectiveness and scalability, paving the way for new opportunities in code idiomatization. We published the work at FSE 2024 (Zhang, Xing, Xiao et al., 2024).

In the third work, to help developers understand Pythonic idioms, we focused on explaining nine unique Pythonic idioms that pose comprehension challenges due to their exclusive syntax. By analyzing questions on Stack Overflow and usage in GitHub repositories, we identified common misunderstandings and potential negative effects of these idioms, such as performance issues and bugs. To address this, we developed a novel approach that detects idiomatic code using AST nodes and rewrites it into non-idiomatic code, ensuring clarity and semantic accuracy. Applied to 7,577 GitHub repositories, our tool achieved 100% detection accuracy and 99-100% rewriting accuracy.

A user study with 20 students showed that providing explanatory non-idiomatic code improved comprehension by 63.5% and reduced completion time by 22.6%. These results demonstrate our tool's effectiveness in enhancing developers' understanding and confidence in using Pythonic idioms. This work was published at ICSE 2024 (Zhang, Xing, Zhao et al., 2024) and were honored with the ACM Distinguished Paper Award.

In the fourth work, we addressed the challenge of lacking datasets with pairs of idiomatic and functionally-equivalent non-idiomatic Python code to study performance impacts. Leveraging a previously developed RIdiom refactoring tool, we created a large synthetic dataset and reused a real-project dataset to investigate the performance impact of nine Python idioms. Our synthetic dataset includes 24,126 code pairs, manually validated for correctness, while the real-project dataset comprises 54,879 code pairs with accompanying test cases. By executing the code in multiple virtual machine (VM) invocations and using statistical bootstrapping, we ensured reliable performance measurements. Our findings indicate that Pythonic idioms tend to speed up synthetic code moderately but often result in a slowdown in real-project code due to factors like library objects and complex computations. We also discovered that code complexity, variable scope, data properties, and execution path explain performance differences in synthetic code but offer limited insights for real-project code. This work helps developers understand the nuanced performance implications of Pythonic idioms in different contexts. This work was published at ICSE 2023 (Zhang, Xing, Xia, Xu, Zhu and Lu, 2023).

1.3 Thesis Outline

In this thesis, the works are organized as seven chapters.

Chapter 1 introduces the motivations and contributions of this research.

Chapter 2 investigates related works and compares the difference between our work with them.

In Chapter 3, we introduce RIdiom, an automated tool for refactoring non-idiomatic Python code into idiomatic forms, along with its design, implementation, and evaluation.

In Chapter 4, we present a hybrid knowledge-driven approach leveraging Large Language Models (LLMs) and Analytic Rule Interfaces (ARIs) to refactor non-idiomatic Python code into idiomatic forms, demonstrating its effectiveness and scalability across both established and new Pythonic idioms.

In Chapter 5, we introduce DeIdiom, a tool that transforms Pythonic idioms into non-idiomatic code to enhance readability and comprehension, and evaluate its accuracy and usefulness in helping developers understand idiomatic code correctly and efficiently.

In Chapter 6, we unveil the performance impact of Pythonic idioms through a comprehensive empirical study, analyzing both synthetic and real-project code to identify the conditions under which these idioms improve or hinder performance.

Chapter 7 summarize the works completed in this thesis and presents the future works that can be extended from this research.

Chapter 2

Literature Review

2.1 Programming Idioms

Evolution and Establishment of Programming Idioms. The concept of programming idioms, defined as commonly used patterns or conventions in programming, has been explored and established through a variety of studies and research efforts across different domains. Perlis and Rugaber (1979) are early proponents, proposing programming language constructs characterized by frquent occurrence, cohesive purpose, ease of identification, and versatility in application in 1979. They observed that APL contains a large vocabulary of idioms owing to its conciseness and functional orientation. Building on this foundation, subsequent research by Bosch (1996) focused on distinguishing between design patterns and language design. They asserted that design patterns constitute a fundamental aspect of a software engineer's paradigm, and it falls upon programming languages to accurately represent these concepts. While no language can encompass all concepts, they argue that proper language design can facilitate the representation of numerous concepts, including various design patterns. Langer (2001) delved into the importance of idiomatic programming in Java. Idioms, in this context, refer to the typical or characteristic ways of coding in Java to achieve certain tasks efficiently and effectively. Further studies by Bloch (2008) discussed common conventions and best practices that are widely accepted within the Java community. The author defined idioms as recurring patterns or practices that are considered effective and efficient solutions to common programming problems. He encouraged developers to embrace these idioms as they embody the collective wisdom of experienced Java developers and help streamline the development process. Cwalina and Abrams (2008) emphasized the significance of idiomatic code in the context of .NET development. Idioms are viewed as common patterns or practices that are widely accepted within the .NET community and contribute to the overall quality and consistency of code. Knupp (2013) discussed the importance of writing idiomatic Python code, which refers to code that follows the conventions, idioms, and patterns established by the Python community. It emphasizes the significance of idiomatic Python in terms of readability, maintainability, and overall code quality. In conclusion, idioms can be described as standardized practices or patterns that are widely accepted within a programming community. They play a crucial role across different programming languages, with many books and developers actively promoting their adoption. Particularly in the Python language, the Python community has been continuously developing and refining Pythonic idioms since 2000, aiming to enhance developers' programming efficiency and code quality. In conclusion, idioms can refer to programming practices and coding conventions that align with the core philosophy and style of coding within a specific programming context.

Mining of Programming Idioms. To promote the understanding and usage of programming idioms, many researches employ various approaches to mine programming idioms. One approach is based on the literature review. Alexandru et al. (2018), identified 19 Pythonic idioms (list-comprehension and dict-comprehension overlap with our work) from several books. They manually classified the idioms based on performance and readability. Phan-udom et al. (2020) collect 10 Pythonic idioms from presentations given by renowned Python developers (Knupp, 2013). Farooq and Zaytsev (2021) employed a grounded theory in a bottom-up approach to mine programming idioms. They first searched the world wide web for the most popular Python books and then used a set of keywords to scann literature and cross-referencing the results across books. They identified a total of 27 detectable idioms, of which five (list/set/dict-comprehension, chain-comparison, truth-test) overlap with those defined by RIdiom (Zhang, Xing, Xia, Xu and Zhu, 2022; Zhang, Xing, Xu et al., 2023b). Another

common approach is to mine programming idioms from source code. Allamanis and Sutton (2014) define code idioms as recurring syntactic fragments across projects with a consistent semantic role. They employed automatic mining techniques based on nonparametric Bayesian tree substitution grammar from a corpus of previously written, idiomatic software projects. Their findings reveal that code idioms manifest in various contexts: some are unique to specific projects, APIs, or programming languages. After that, Allamanis, Barr et al. (2016) proposed a technique for mining loop idioms, which are frequently occurring semantic patterns within loops, from extensive codebases to uncover meaningful patterns. They found that the loops within this dataset are straightforward and predictable: 90% of them consist of fewer than 15 lines of code (LOC), and 90% exhibit no nesting and possess very simple control structures. Following that, Sivaraman et al. (2022) introduced semantic enrichment to Abstract Syntax Trees (ASTs) using approximate dataflow information, extending the AST structure. They developed Jezero, a tool capable of learning semantic patterns from large codebases using tree structures augmented with dataflow information derived from a cost-effective, syntactic dataflow analysis. Their experiments, conducted on four APIs from Facebook's Hack codebase, demonstrated that Jezero outperformed baseline methods lacking dataflow augmentation in identifying refactoring opportunities within unannotated legacy code. Different from these works, we propose mining programming idioms by comparing the syntax of different programming languages (Zhang, Xing, Xia, Xu and Zhu, 2022; Zhang, Xing, Xu et al., 2023b). This method not only allows us to identify commonalities and differences in programming idioms across languages but also reveals unique idioms specific to each language. We implemented this idea by contrasting the syntax of Python and Java, which led to the identification of nine unique Pythonic idioms.

Programming Idioms Assistance. Numerous researchers have devised approaches and tools aimed at assisting developers in utilizing programming idioms effectively. Franklin et al. (2013) introduced the LAMBDAFICATOR tool, which automates two Java code transformations involving lambda expressions. Firstly, it transforms AIC (Anonymous Inner Classes) to lambda expressions. Secondly, it converts for loops over Collections to functional operators, utilizing lambda expressions. Radoi et al. (2014) proposed an automated approach to translate sequential, imperative code into a parallel MapReduce framework. By employing novel techniques such as group-by operations, their system, called MOLD, can effectively generate MapReduce implementations from sequential Java code targeting the Apache Spark runtime. David et al. (2017) introduced a methodology for conducting semantic reasoning and exploring potential refactoring options. Their approach enables the transformation of Java code that employs external iteration over collections into code utilizing Streams, a new abstraction introduced in Java 8. Reactive programming languages and libraries, such as ReactiveX, have gained significant traction, offering improved software design and witnessing widespread adoption in industry. Köhler and Salvaneschi (2019) proposed 2RX, an automated refactoring approach for converting asynchronous code to reactive programming paradigms. Midolo and Tramontana (2021) introduced a methodology for conducting semantic reasoning and exploring potential refactoring options. Additionally, *Pylint* (2022) can detect two types of non-idiomatic code that can be refactored into chain-comparison and truth-value-test, but it provides only simple refactoring suggestions that may not be intuitive to developers. Furthermore, Phan-udom et al. (2020) gathered 58 non-idiomatic Python code fragments and corresponding 55 idiomatic Python code examples, including three Pythonic idioms: list-comprehension, set-comprehension, and truth-value-test, which overlap with our work. However, due to the diversity and complexity of non-idiomatic Python code, there is currently no tool available, aside from our own, that can automatically refactor non-idiomatic Python code with Pythonic idioms. In our work, we offer the capability to automatically identify and refactor non-idiomatic Python code with Pythonic idioms, rather than merely providing simple suggestions or code examples.

2.2 Code Refactoring

Empirical Studies on the Impact of Refactoring on Code Quality. Fowler (2018) provided comprehensive guidance on improving the design and maintainability of existing software systems through refactoring techniques. It outlines various refactorings, which are disciplined techniques for restructuring code without changing its external behavior. Through practical examples and insights, Fowler demonstrates how refactoring can enhance code readability, extensibility, and overall quality. Kim et al. (2012) conducted a groundbreaking study assessing the impact of extensive, multi-year refactoring initiatives on inter-module dependencies and post-release defects in a significant organization. Their

findings highlight the potential for refactoring-driven changes to improve system safety and reliability relative to routine modifications. Furthermore, they emphasized the importance of augmenting tool support with features like refactoring-aware code reviews and benefit estimation efforts beyond automated refactorings in integrated development environments (IDEs). Al Dallal and Abdin (2017) conducted a systematic literature review, analyzing 76 primary studies to evaluate the impact of object-oriented code refactoring on various quality attributes. Their findings revealed that different refactoring scenarios may have diverse and sometimes conflicting effects on quality metrics. While certain refactorings like Extract Class and Move Method demonstrated improvements in cohesion, coupling, and complexity, others like Extract Method were observed to weaken cohesion and coupling attributes in most cases. Kaur and Kaur (2016) focused on a Java-based library software and utilized tools like JDeodorant to identify and Metrics plugin to assess complexity. Following refactoring, they recalculated the project's complexity and observed a notable reduction, indicating enhanced quality. This reduction in complexity not only improves code understandability but also lowers maintenance costs, making the software more sustainable in the long run. Lin et al. (2019) conducted an extensive analysis of refactoring operations across various open-source systems, focusing on their impact on code naturalness. Contrary to expectations, their findings suggest that refactoring does not always enhance the naturalness of the modified code. Moreover, they observed that the influence on code naturalness varies significantly based on the type of refactoring operation employed. Paixão et al. (2020) conducted an empirical study on software refactoring in modern code review, analyzing 1,780 reviewed code changes from 6 systems across two large open-source communities. They identified and classified developers' intents behind each change into 7 distinct categories. The study highlighted the prevalence of complex refactoring operations, such as moving or extracting classes, within explicit refactoring changes, and emphasized the importance of refining existing techniques for refactoring-aware code reviews to guide developers effectively. Based on these studies, we can conclude that while code refactoring can enhance code quality, it remains challenging to apply refactoring in the appropriate code locations and refactor code correctly.

Detection of Code Refactoring. Detection of code refactoring is important to prevents code misunderstanding and mitigates potential maintenance difficulties. Their approaches can be dividied into symptom-based, metric-based, visualization-based, search-based, probabilistic cooperative-based, and manual approaches. Romano and Scanniello (2018) developed an approach that utilizes RTA to detect redundant methods, resulting in the creation of a supporting tool called Dead Code Finder (DCF). Through empirical evaluation on four separate open-source Java desktop applications, they observed that DCF exhibited superior accuracy in identifying dead methods compared to the baseline tools. Chen, Liu et al. (2018) introduced FEED (Feature Envy Detector), a tool based on dataflow analysis, aimed at identifying instances of feature envy in code. Unlike previous approaches that treat entire methods as units for detection, FEED offers finer granularity in detection, thereby improving accuracy. While many existing approaches identify design flaws ad-hoc, focusing on software metrics, local code smells, or coarse-grained architectural anti-patterns, Peldszus et al. (2016) proposed a methodology for specifying object-oriented design flaws using compound rules that integrate code metrics, code smells, and anti-patterns in a modular fashion. Mumtaz et al. (2018) explored the complementary nature of Parallel Coordinates Plots and RadViz in analyzing multivariate software metrics to detect bad smells in software systems. The study introduces an interactive visual analytics system for automatic detection of bad smell patterns and investigates distinctive properties of outliers that may not be harmful but are noteworthy for other reasons. Gupta et al. (2018) introduced a mathematical model leveraging entropy from information theory to predict bad smells in open-source software, focusing on Apache Abdera. They collect bad smells using a detection tool from various components of the Apache Abdera project and compute different entropy measures. Through non-linear regression techniques, they forecasted potential bad smells in future software versions based on observed bad smells and entropy measures. Tsantalis, Mansouri et al. (2018) identified 15 Java refactoring types and proposed the RMiner tool. They achieved this by utilizing an AST-based statement matching algorithm, which identifies refactoring candidates without requiring user-defined thresholds. Dilhara, Ketkar et al. (2022) identified 18 refactoring types that involve fine-grained changes, such as alterations to method signatures or shifts in method bodies. They employed JavaFyPy to adapt CPatMiner, a leading tool for mining fine-grained change patterns, to Python. CPatMiner compares modified methods and their body statements across commits to detect and categorize fine-grained change patterns. In contrast to these works, we have designed detection rules and utilized four atomic operations to compose AST rewriting rules. This enables us to not only detect non-idiomatic code but also automatically refactor

it with Pythonic idioms. Our approach was implemented on Python code, and the high precision and recall validate its effectiveness.

Automated Refactoring. Automated refactoring provides an important avenue for developers to enhance code quality and streamline maintenance processes efficiently and swiftly. Wang, Yu et al. (2018) introduced refactoring algorithm rooted in complex network theory, aiming to optimize functionality distribution from a system perspective. By integrating move method, move field, and extract class refactorings, the algorithm targets "bad smells" stemming from cohesion and coupling issues across both inheritance and non-inheritance hierarchies. Through class-level multi-relation directed networks and method-level weighted undirected networks, the algorithm conducts refactoring preprocessing and operations, prioritizing "high cohesion and low coupling" principles based on weighted clustering. To address the dilemma of whether to apply a specific design pattern, Cinnéide (2000) devised automated support for introducing design patterns into existing object-oriented programs. This approach allows software engineers to select program entities, such as classes, objects, and methods, for applying the desired design pattern. Arcelli et al. (2015) introduced a methodology that combines the analysis of antipattern probability and refactoring benefits to identify effective solutions for enhancing software performance. Operating within a fuzzy context where precise threshold values are not defined, the approach generates a list of performance antipatterns along with their probabilities of occurrence in the model. Additionally, it links each antipattern with multiple refactoring alternatives, estimating their efficacy in improving software performance. Franklin et al. (2013) present a LAMBDAFICATOR tool to automatically perform two refactorings with lambda expressions. Radoi et al. (2014) propose an approach to automatically translate sequential, imperative code into a parallel MapReduce framework. Ouni et al. (2016) propose a multi-objective search-based approach for finding the optimal sequence of refactorings. Tsantalis, Mazinanian et al. (2017) propose a tool to refactor Type-2 and Type-3 clones by utilizing Lambda expressions. David et al. (2017) develop a tool to transform Java with external iteration over collections into code that uses Streams. Köhler and Salvaneschi (2019) develop a tool to automatically convert asynchronous code to reactive programming. Dilhara, Dig et al. (2023) introduced a novel automated workflow, PYEVOLVE, for mining frequently repeated code change patterns (CPATs), inferring transformation rules, and automatically applying them to new target sites. Furthermore, Dilhara, Bellur et al. (2024) introduced PyCraft, a tool designed to advance Transformation by Example (TBE) systems by automating the generation of unseen code variants leveraging Large Language Models (LLMs). PyCraft ensures correctness and usefulness of generated variants through fine-tuning of LLM hyper-parameters and comprehensive automatic checks, significantly enhancing TBE techniques and automation capabilities. Different from previous works, our focus lies in automatically and precisely applying AST rewriting rules based on semantic equivalence using four atomic operations to refactor code, achieving high precision and recall rates in implementing code refactoring of Pythonic idioms.

2.3 Code Comprehension

Studies on Program Comprehension. Research on program comprehension started more than 30 years ago (Sackman et al., 1968; Siegmund, 2016). Shneiderman (1976) investigated programmer behavior through experimental exploration. The paper delves into how programmers interact with code and explores various aspects of their behavior during software development tasks, shedding light on important insights into developer practices. Shaft and Vessey (1995) conducted a study where programmers were tasked with understanding a program while vocalizing their thoughts, offering insights into developers' approaches when working with code in both familiar and unfamiliar domains. They observed that developers formulate hypotheses in familiar domains and draw inferences when dealing with unfamiliar ones. Similarly, Mayrhauser and Vans (1993) conducted a study where developers completed maintenance tasks while thinking aloud, revealing that developers create various mental models of the source code and transition between them. However, the use of think-aloud protocols is often avoided due to the significant effort involved in recording, transcribing, and analyzing the data. Xia et al. (2018) extended the investigation of program comprehension beyond IDE interactions by leveraging the ActivitySpace framework to analyze Human-Computer Interaction (HCI) data across various applications. Their findings underscore the need for research efforts aimed at reducing program comprehension time through improved code quality, documentation, search engines, and IDE design.

Supporting Program Comprehension. Program comprehension is essential for ensuring code correctness, as developers need to understand the codebase thoroughly to identify and fix bugs, implement new features, and maintain the software effectively. Program comprehension accounts for over 50% of the time allocated to software maintenance (Corbi, 1989; Xia et al., 2018; Zelkowitz et al., 1979). Many researches used different approaches (e.g., think-aloud protocols, memorization and comprehension tasks) to measure program comprehension (Shao and Wang, 2003; Siegmund, 2016; Soloway and Ehrlich, 1984; Xia et al., 2018; Yu and Zhou, 2010). Gopstein et al. (2017) summarized code patterns that can lead to a significantly increased rate of misunderstanding versus equivalent code without the patterns. Brun et al. (2023) found blindspots in Python and Java APIs result in vulnerable code and suggested to develop tools to recognize blindspots in APIs. Bhattacharjee et al. (2022) proposed approach transforms the Agglomerative Hierarchical Clustering (AHC) tree into a more manageable structure through cluster flattening, while also providing natural text summaries for abstract nodes derived from method comments. Mészáros et al. (2019) utilized LSP to improve the code comprehension experience inside code editors. Lanza et al. (2005) developed CodeCrawler to visualize object-oriented software for program comprehension. In contrast to prior research, we novelly propose to explain code using commonly shared syntax found in multiple programming languages, ensuring accurate comprehension even for developers with limited experience. Specifically, we apply this method to elucidate Pythonic idioms and automate the conversion of idiomatic code into equivalent non-idiomatic code, facilitating program comprehension across nine Pythonic idioms.

2.4 Code Performance

Performance Measurement. Performance measurement serves as a foundational tool to assess, benchmark, and improve software performance systematically and strategically. Georges et al. (2007a) highlighted the challenges in benchmarking Java performance due to various influencing factors and non-determinism during runtime. The paper emphasizes the need for statistically rigorous data analysis to avoid misleading conclusions by discussing the diversity of methodologies used for Java performance evaluation. The paper proposes to quantify both startup and steady-state performance and introduces JavaStats software for obtaining performance metrics rigorously. He, Manns et al. (2019) introduced PT4Cloud, a novel cloud performance testing methodology that addresses the challenge of obtaining accurate performance results in cloud environments. By leveraging non-parametric statistical approaches, PT4Cloud offers reliable stop conditions to achieve highly accurate performance distributions with confidence bands. He, Liu et al. (2021) proposed to leverage advanced statistical tools such as block bootstrapping, the law of large numbers, and autocorrelation to properly account for internal dependencies within cloud performance test data. Laaber, Würsten et al. (2020) introduced a novel technique for dynamically stopping software microbenchmark executions to achieve significant reductions in execution time while maintaining result quality. By implementing three statistical stoppage criteria, the proposed approach can reduce Java Microbenchmark Harness (JMH) suite execution times by up to 86.0% while retaining result quality for the majority of benchmarks. Kalibera and Jones (2020) proposed a statistical model inspired by methods used in other scientific fields, enabling the quantification of uncertainty in performance measurements and the design of experiments that account for various sources of non-determinism. This framework offers more accurate and interpretable summaries of experimental results, enhancing the repeatability, reproducibility, and validity of quantitative findings. To address non-determinism in code execution time, we executed each code 35 times across 50 VM invocations, following previous studies. We also utilize the approach of quantifying performance changes with effect size, as proposed by Kalibera and Jones (2013, 2020), to validate the reliability of our performance measurements.

Performance Issues. Many studies Chen, Yu et al. (2019); Chen, Shang et al. (2020); Demeyer (2005); Huang, Ma et al. (2014); Luo et al. (2016); Nguyen, Adams et al. (2012); Nguyen, Nagappan et al. (2014); Song and Lu (2017); Trubiani et al. (2018) analyze source code to identify performance regression or improvement. Alcocer et al. (2019) help developers understand performance regressions and improvements by contrasting performance variations and source code changes at package, class, and method level on three Pharo applications with 15 versions. Jin et al. (2012) learn guidance for software practitioners by studying 109 real-world performance bugs from five software systems. Daly et al. (2020) presents mechanisms to identify the commit causing a specific performance degradation event by running a collection of tests periodically. Laaber et al. and He et al. (He, Liu et al., 2021; He, Manns et al., 2019; Laaber, Würsten et al., 2020) focus on reducing the execution time of microbenchmarking

tests and ensuring the accurate performance results through a early stop based on a threshold on Java Microbenchmark Harness (JMH) suite. Chen, Shang et al. (2020) extract metrics from code commits to predict whether commits cause performance regression. They find changes to loops are the only important factor causing performance regression. Traini et al. (2021) extract 55 refactoring operations from the commits of 20 Java systems and execute the benchmarks before and after a commit. They find Extract Class/Interface and Extract Method have higher chances of triggering performance regression. Leelaprute et al. (2022) analyzes the performance of some Python features (e.g., collections.defaultdict, lambda, generator expression) and two idioms (list/dict comprehension) with different input sizes. For list/dict comprehension, they find that as the number of added elements increases, the time difference between idiomatic code and non-idiomatic code becomes larger, which is consistent with our result. However, since they only compare the time difference on one toy code example, they do not observe the wide distribution of performance differences of list/dict comprehension as our results show, neither the discrepancies between synthetic and real-project code. Furthermore, our study involves 7 more idioms and is done on two large scale datasets.

2.5 Large Language Models

Emperical Studies on Large Language Models. The advancements in Large Language Models (LLMs) have opened up new opportunities for Automated Software Engineering (ASE). Many researches conducted comprehensive and in-depth analysis for the capability of LLMs on ASE tasks. Shin et al. (2023) explored the effectiveness of GPT-4 with three prompting engineering techniques against 18 fine-tuned LLMs across three (Automated Software Engineering) ASE tasks: code generation, code summarization, and code translation. Quantitative analysis indicates that GPT-4's performance varies across tasks and prompting strategies, with conversational prompts showing promise. User study results suggest conversational prompting yields significant improvement, highlighting its potential for ASE tasks, while fully automated prompt engineering requires further study and enhancement. Chang et al. (2023) provided a comprehensive review of evaluation methods for Large Language Models (LLMs), focusing on what to evaluate, where to evaluate, and how to evaluate. It covers evaluation tasks ranging from natural language processing to ethics and education, and discusses evaluation methods and benchmarks crucial for assessing LLM performance. The paper also highlights success and failure cases of LLMs in various tasks and identifies future challenges in LLM evaluation, emphasizing the importance of evaluation in advancing LLM development. Zheng et al. (2023) presented a comprehensive analysis of the intersection between software engineering and large-scale language models (LLMs), covering 123 relevant studies across different software engineering tasks. By categorizing these tasks and providing application examples, the review aims to assist researchers in identifying and addressing challenges in applying LLMs to software engineering. Additionally, the review offers insights into the performance of LLMs in various software engineering tasks and explores reasons for performance variations, providing valuable guidance for developers in optimizing LLMs effectively. Wang, Huang et al. (2024) offered a comprehensive review of the utilization of large language models (LLMs) in software testing, analyzing 52 relevant studies from both the software testing and LLM perspectives. The work identifies common tasks in software testing where LLMs are applied, such as test case preparation and program repair, and examines the types of prompt engineering and techniques used with these models. Besides, the work highlights key challenges and potential opportunities in understanding the use of LLMs in software testing. Differentiating from prior research, our focus lies in harnessing the potential of Large Language Models (LLMs) in software engineering tasks. We propose an approach that integrates LLMs with a knowledge base to automate and enhance the efficiency of code refactoring, specifically targeting the adoption of Pythonic idioms.

Software Engineering Tasks on Large Language Models. Building on the achievements of Language Models (LLMs) like GPT-3 (Brown et al., 2020) and GPT-4 (OpenAI, 2023) in the field of Natural Language Processing (Chen, Zhang et al., 2022; Creswell et al., 2022; Lemieux et al., 2023; Ren, Ye et al., 2023; Wei et al., 2023; Wu et al., 2022; Yang et al., 2022), researchers are now delving into their potential applications in software engineering (Dilhara, Bellur et al., 2024; Feng and Chen, 2023; Fu et al., 2022; Huang, Yuan et al., 2023; Huang, Zou et al., 2023; Jain et al., 2022; Li, Choi et al., 2022; Liu et al., 2023; *OpenAI Codex* 2023; Peng et al., 2023). These applications span a variety of tasks, including program analysis, code debugging, and code generation. For program analysis task, Huang, Zou et al. (2023) introduced a chain-of-thought approach based on LLMs, comprising four steps: extracting structure hierarchy, isolating nested code blocks, generating control flow graphs (CFGs) for these nested blocks,

and amalgamating all CFGs. This approach surpasses existing CFG tools in terms of both node and edge coverage, particularly for incomplete or erroneous code. Peng et al. (2023) presented TYPEGEN, which generates prompts incorporating domain knowledge, then feeds them into LLMs for type prediction. This approach uses few annotated examples to achieve superior performance compared to rule-based type inference approaches. For program debugging task, Feng and Chen (2023) proposed a two-phase approach utilizing a chain-of-thought prompt to guide LLMs in extracting S2R entities, followed by matching these entities with GUI states to replicate the bug reproduction steps. This demonstrates that instructing LLMs through prompts can effectively achieve bug replay. For code generation task, Ren, Ye et al. (2023) introduced KPC, a knowledge-driven prompt chaining-based code generation approach that leverages fine-grained exception-handling knowledge extracted from API documentation to enhance LLM-based code generation. By employing a divide-and-conquer strategy and iterative coding practice, KPC refines generated code from misuse to mastery through a chain of modular prompts for LLMs. This approach involves constructing an API knowledge base from official documentation and chaining knowledge-driven prompts to verify and handle exceptions in the generated code. In this work, we focus on making Python code idiomatic with Pythonic idioms. Unlike previous approaches that exclusively design prompts for instructing LLMs to generate code or complete tasks, we propose a hybrid approach using APIs generated by LLMs to extract three elements, along with prompts to guide LLMs in performing the idiomatization task. It demonstrates that LLMs and rule-based approaches complement each other which can assist researchers to solve software engineering tasks better.

Chapter 3

Automated Refactoring of Non-Idiomatic Python Code with Pythonic Idioms

This chapter was published as

Z. Zhang, Z. Xing, X. Xia, X. Xu and L. Zhu (2022). '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. New York, NY, USA: Association for Computing Machinery, pp. 696–708. ISBN: 9781450394130. DOI: 10.1145/3540250.3549143

Compared to other programming languages (e.g., Java), Python has more idioms to make Python code concise and efficient. Although Pythonic idioms are well accepted in the Python community, Python programmers are often faced with many challenges in using them, for example, being unaware of certain Pythonic idioms or do not know how to use them properly. Based on an analysis of 7,638 Python repositories on GitHub, we find that non-idiomatic Python code that can be implemented with Pythonic idioms occurs frequently and widely. Unfortunately, there is no tool for automatically refactoring such non-idiomatic code into idiomatic code. In this paper, we design and implement an automatic refactoring tool to make Python code idiomatic. We identify nine Pythonic idioms by systematically contrasting the abstract syntax grammar of Python and Java. Then we define the syntactic patterns for detecting non-idiomatic code for each pythonic idiom. Finally, we devise atomic AST-rewriting operations and refactoring steps to refactor non-idiomatic code into idiomatic code. We test and review over 4,115 refactorings applied to 1,065 Python projects from GitHub, and submit 90 pull requests for the 90 randomly sampled refactorings to 84 projects.

3.1 Introduction

Programming (or code) idioms are widely present in programming languages (*Programming Idioms* 2022). They represent notable programming styles and features of a programming language. Python is well known for its Pythonic idioms (Merchante and Robles, 2017). Many books and online materials (Alexandru et al., 2018; Hettinger, 2013; Knupp, 2013; Merchante and Robles, 2017; Reitz and Schlusser, 2016; Slatkin, 2020) promote the use of Pythonic idioms for not only concise coding styles but also improved performance. For instance, consider the chain-comparison idiom. In Figure 3.1, In the non-idiomatic code (or anti-idiom code) "-size <= x.indices(size)[0] and x.indices(size)[0] <= size", the corresponding idiomatic code is "-size <= x.indices(size)[0] <= size". The idiomatic code, compared to its non-idiomatic counterpart, not only enhances code conciseness but also improves runtime efficiency. In spite of the benefits of Pythonic idioms and the availability of many online materials, our investigation of some highly viewed Python questions on Stack Overflow suggests that developers are often unaware of Pythonic idioms or do not know when and how to use Pythonic idioms properly (see the examples in Table 6.1 and the analysis in Section 3.2.1).

Due to these challenges in using Pythonic idioms, developers may implement a functionality in a non-idiomatic way without using Pythonic idioms. We study 7,638 Python projects on GitHub (see

Non-Idiomatic Code:

```
-size <= x.indices(size)[0] and x.indices(size)[0] <= size
```

Idiomatic Code:

```
-size <= x.indices(size)[0] <= size
```

Figure 3.1: The non-idiomatic code and the corresponding idiomatic code for the chain comparison idiom.

Section 3.2.1) and find that non-idiomatic code that can be implemented with Pythonic idioms is widely present at the repository, file, method and statement level (see Table 3.1). Non-idiomatic code and idiomatic code co-exist in many repositories, files or even methods. Non-idiomatic code syntax exists in other mainstream programming languages (e.g., Java). In contrast, Pythonic idioms have "uncommon" syntax. Developers, even those with little Python programming experience, can still write non-idiomatic code. However, to use Pythonic idioms, they would need to learn new syntax or need some tool supports.

Although online documentation of Pythonic idioms provide rich learning materials, they cannot directly support programming with Pythonic idioms. To the best of our knowledge, only two tools provide limited support for using Pythonic idioms. Among the nine Pythonic idioms , *Pylint* (2022) can detect two types of non-idiomatic code which can be refactored into chain-comparison and truth-value-test respectively. However, it offers only a simple refactoring suggestion which may not be intuitive to developers. For example, for the code "`a < 0 and b > 1 and c < 1 and d > 2`", Pylint suggests "Simplify chained comparison between the operands"[1]. Unfortunately, the developer did not understand this suggestion initially. When Pylint developer further explained that the code can be refactored into "`a < 0 and b > 1 > c and d > 2`", the developer understood what to do and left a comment "Would it be an idea for Pylint to output the suggested refactor to the user?" which received thumbs up by other developers. Phan-udom et al. (2020) collects 58 non-idiomatic code fragments and corresponding 55 idiomatic code (three Pythonic idioms list-comprehension, set-comprehension and truth-value-test overlap with 9 idioms in Figure 3.4. It detects non-idiomatic code similar to the collected 58 examples and recommends corresponding idiomatic code examples. Developers still have to manually refactor the non-idiomatic code.

In this chapter, we develop the first automatic refactoring tool that detects 9 types of non-idiomatic code (referred to as anti-idiom code smells) and refactors these anti-idiom code smells into idiomatic code implementing the same functionalities. Existing works (Farooq and Zaytsev, 2021; Merchante and Robles, 2017; Sivaraman et al., 2022) search popular Python books and online sources to create catalogues of python idioms that are not unique to Python. Different from them, to identify unique Pythonic idioms, we contrast the language syntax of Python and the other mainstream programming language (Java in this work) because non-idiomatic code syntax is similar to those of other languages but idiomatic code has unique syntax. As a result, our analysis identifies 9 Pythonic idioms 4 of which were not identified by previous studies. We confirm the validity of these Pythonic idioms through the Python language specification and online materials (Hettinger, 2013; Knupp, 2013; Slatkin, 2020). For each pythonic idiom, we define syntactic patterns for detecting non-idiomatic code fragments that implement the same functionality as the pythonic idiom. Following the refactoring principle (one small step at a time) (Fowler, 2018), we formulate four atomic AST rewriting operations and compose these atomic operations for each pythonic idiom for refactoring anti-idiom code with the corresponding pythonic idiom.

To evaluate the code smell detection and refactoring accuracy of our approach, we apply our tool to 7,638 Python projects which detects and refactors over 2,252,022 anti-idiom code smells. We verify the refactoring results by both testing and code review. Our approach achieves 100% smell detection accuracy for six idioms and 100% refactoring accuracy for eight idioms. It makes only a few rare detection errors and only one refactoring error due to the limitation of Python static analysis and the complex program logic. To explore the usefulness of our code refactoring tool in practice, we randomly sample 10 refactorings respectively for each pythonic idiom, and submit in total 90 pull requests to 84 projects to make the project members review our refactorings. As a result, we receive 57 replies from 54 projects, of which 34 accept our pull request with praise of our refactorings and 28 replies merge the pull requests into their repositories. Our results show developers care about pythonic idiom refactorings, and our refactorings have been well received in

[1] https://github.com/PyCQA/pylint/issues/5800

practice. The developers' feedback on the rejected pull requests reveal some interesting concerns about the readability and performance of Pythonic idioms which deserve further study. The dataset of anti-idiom code and corresponding idiomatic code produced in this work provides the first large-scale test bed to systematically investigate such concerns.

In summary, we make the following contribution in this chapter:

- To the best of our knowledge, we are the first to automatically detect non-idiomatic code and refactor it into idiomatic code for 9 widely used Pythonic idioms.
- Through the evaluation on a large number of real-world Python projects, we confirm the high accuracy, practicality and usefulness of our refactoring tool.
- Our work creates the first large-scale dataset of anti-idiom code smells and corresponding idiomatic code for studying and validating the claims and concerns about Pythonic idioms.

3.2 Formative Study

We conduct an empirical study of Pythonic coding practices to answer the following three research questions:

RQ1: What are the coding practices concerning Pythonic idioms and anti-idiom code smells?

RQ2: What are the challenges for writing idiomatic code?

3.2.1 RQ1: Python Coding Practices

Data Preparation

To understand the coding practices with respect to Pythonic idioms and anti-idiom code smells, we crawl the top 10,000 repositories using Python by the number of stars from GitHub. 7,638 repositories can be successfully parsed using Python 3. We collect 506,765 Python source files from these repositories. We then detect the occurrence of idiomatic code and the anti-idiom code that can be refactored with Pythonic idioms in these Python source files. All nine Pythonic idioms can be detected by analyzing abstract syntax trees (ASTs). List/set/dict-comprehension and loop-else idioms directly correspond to AST nodes, we can directly detect such idiomatic code instances. For star-in-function-call, we extract starred node in function call node. For truth-value-test, we extract the test node corresponding to an object. For chain-comparison, assign-multiple-targets and for-multiple-targets, we extract operators of the compare node, the value and targets of the assign node, and the target of for node with elements greater than 1, respectively. We then collected idiomatic code instances using the method. To validate the accuracy of our approach to identify idiomatic code, we randomly selected 100 instances of idiomatic code for each Pythonic idiom and then two authors independently verified their correctness. Both authors confirmed that all sampled idiomatic code was correct, with no instances of false positives. This result underscores the reliability of our approach in identifying idiomatic code of Pythonic idioms. For non-idiomatic code, we use our detection rules (see Section 3.3.2) to detect non-idiomatic code instances. We count the number of repositories, files, methods and statements that contain the instances of idiomatic code and non-idiomatic code. Note that a repository, file or method may contain both Pythonic idioms and refactorable non-idiomatic code.

Result

Figure 3.2 shows the numbers of repositories, files and methods using different types of Pythonic idioms and anti-idiom code smells. We see that although Pythonic idioms are well adopted in the Python projects, there are still non-trivial usage of non-idiomatic code that can be refactored with Pythonic idioms. Furthermore, a repository, file or method generally uses multiple types of idiomatic and non-idiomatic code. There are non-trivial numbers of repositories and files that use 5 or more types of Pythonic idioms or contain 5 or more types of non-idiomatic code that can be refactored with Pythonic idioms.

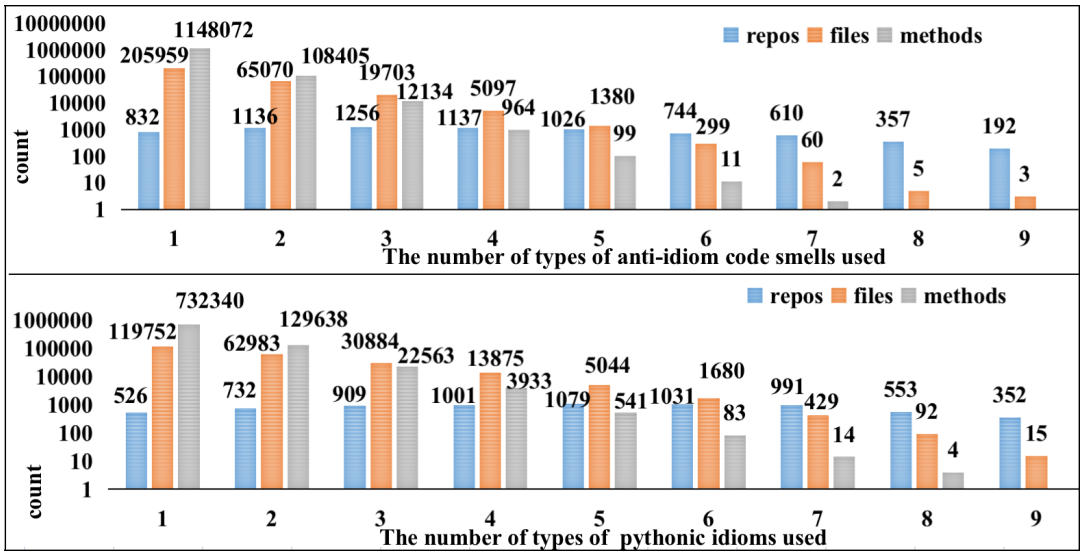


Figure 3.2: The usage of Pythonic idioms and anti-idiom code smells in repositories, files and methods

Table 3.1 summarizes the number of repositories, files and methods containing non-idiomatic code smells (the NPy column), Pythonic idiom (PyId) and both non-idiomatic and idiomatic code (⋂) for each Pythonic idiom. We see that many repositories, files or even methods often have a mix of idiomatic and non-idiomatic code to achieve the same functionality. For example, a large number of repositories and files mix the use of the list-comprehension idiom and non-idiomatic list operation, and 8,258 methods contain both idiomatic list comprehension and non-idiomatic list operation that can be refactored into idiomatic list comprehension. Among the nine Pythonic idioms, set-comprehension and loop-else have relatively low mix usage, while truth-value-test and assign-multiple-targets have high mix usage in methods.

Table 3.1: Python coding practices with respect to Pythonic idioms and anti-idiom code smells

Idiom	Repository			File			Method		
	NPy	PyId	⋂	NPy	PyId	⋂	NPy	PyId	⋂
List Comprehension	3,814	6,161	3,619	17,732	97,775	10,040	24505	219414	8258
Set Comprehension	700	1,048	319	1,279	4,724	151	1,512	7,151	115
Dict Comprehension	2,348	3,167	1,600	7,837	20,871	1,636	10,001	32,941	1,113
Chain Comparison	4,334	2,690	2,252	26,017	11,066	3,095	39,045	17,241	2,467
Truth Val Test	5,885	6,930	5,728	67,991	179,184	46,303	135,861	578,141	48,550
Loop Else	806	1,204	65	1,399	3,602	133	1,644	4,537	127
Assign Multi	7,271	4,074	4,068	288,578	28,193	26,452	1,173,508	49,560	46,075
Star in Func Call	2,336	4,185	1,840	7,671	30,566	1,977	11,325	64,492	2,303
For Multi	2,018	5,372	1,812	5,314	57,063	1,953	7,245	104,893	1,923

Table 3.2 summarizes the occurrence of non-idiomatic code that can be refactored with an idiom (the Non-idiomatic column) and the occurrence of a type of Pythonic idiom (the Idiomatic column) at the statement level. Sum is the sum of the two occurrences and % is the percentage of the non-idiomatic code out of the Sum. For the five Pythonic idioms (set-comprehension, dict-comprehension, truth-value-test, loop-else and star-in-func-call), the percentages of non-idiomatic code are about 14.6%-22.9%. List-comprehension and for-multi-targets have low non-idiomatic percentages (7.0% and 6.3% respectively). This may be due to the popularity of these two idioms in the Python community. Most of online resources we read mention these two idioms. These two idioms have been used more than 349K and 127K times. Although the percentage of non-idiomatic code is low, there are still large numbers of non-idiomatic code fragments (26,510 and 8,490) that can be refactored with the

Table 3.2: Statistics of Pythonic idioms and anti-idiom code smells at the statement level

Idiom	Non-Idiomatic Code	Idiomatic Code	Sum	Percentage
List Comprehension	26,510	349,912	376,422	0.070
Set Comprehension	1,596	9,304	10,900	0.146
Dict Comprehension	10,695	42,149	52,844	0.202
Chain Comparison	53,811	26,913	80,724	0.667
Truth Value Test	197,667	1,023,539	1,221,206	0.162
Loop Else	1,644	5,552	7,196	0.228
Assign Multi Targets	1,934,165	70,700	2,004,865	0.965
Star in Func Call	17,444	90,292	107,736	0.162
For Multi Targets	8,490	127,305	135,795	0.063

list-comprehension idiom and the for-multi-targets idiom respectively. In contrast, chain-comparison and assign-multi-targets have high non-pythonic percentages (66.7% and 96.5% respectively). Expression comparison and assignment are very basic programming constructs no matter in Python or other programming languages. However, developers may not realize unique Pythonic idioms for comparison and assignment. For example, developers are surprised when they see Python can make comparison for more than two operands[2].

3.2.2 RQ2: Challenges in Writing Idiomatic Code

We examine Stack Overflow questions to understand the challenges in writing idiomatic code. We search the questions for each Pythonic idiom using the "python" tag and the Pythonic idiom name. We examine the returned top 30 questions and summarize the challenges in using Pythonic idioms in the discussions. We summarize three key challenges. Table 3.3 shows some representative examples. Many questions have very high view counts which indicate common information needs. The #C column lists the challenge index as discussed below. One question may involve several challenges. However, we observe that the three challenges have a progressive relationship. For example, when developers do not know the meaning of an idiom, they would also further ask how to use the idiom correctly. Therefore, we list only the most fundamental challenge.

(1) Developers do not know certain Pythonic idioms. For example, for the dict-comprehension idiom (the 4th row in Table 3.3), the developer knows list comprehension but he/she does not know whether he/she can initialize dictionary in a similar way. The question has been viewed about 1,000,000 times. Although it was asked 12 years ago, it was still actively discussed about 1 month ago (as of this paper writing). Idioms in Python are more than those in other mainstream languages (Alexandru et al., 2018), which brings challenges for developers to learn and write idiomatic Python code. To give another example, consider the chain-comparison idiom (the 1st row in Table 3.3). In this case, a developer is unaware of the existence of the chain-comparison idiom and expresses confusion, leaving a comment like, "I didn't know you could do that in Python. Was really scratching my head on this one." This question has been viewed about 116,000 times.

(2) Developers know certain Pythonic idioms but they do not understand what the idioms can do. Take the loop-else idiom (the 7th row in Table 3.3) as an example. While the developer knows the else clause is legal in Python, they struggle to understand its purpose in loop statements. This question has received approximately 245,000 views and remained a topic of active discussion over the past 2 months. Similary, consider the question for using asterisk operator "*" in the function call (the second last row in Table 3.3) which has been viewed about 234,000 times. The developer notices a single asterisk (`zip(*x)`) can be used before a parameter in function calls, but he/she does not know what this means and what it can be used for. Actually, the `*x` is to unpack `x` into multiple arguments. Knowing what an idiom can be used for is the pre-requisite for using it in practice.

(3) Developers know what a Pythonic idiom can do but they do not know how to use them properly. The 2nd row of Table 3.3 shows such an example. The developer wants to refactor a list initialization

[2] https://stackoverflow.com/questions/26502775/simplify-chained-comparison

using list comprehension. Unfortunately, he/she does not know whether and how if-else statement can be used in the list-comprehension idiom. The question has been viewed about 165,000 times. In fact, list-comprehension has complex syntax and it may nest multiple loops and multiple if statements. Developers have to read and understand this complex syntax in order to use the list-comprehension idiom properly.

Table 3.3: Challenges in writing idiomatic Python code

Idiom	#C	Question
List Comprehension	(3)	**Question:** Here is the code I was trying to turn into a list comprehension:... Is there a way to add the else statement to this comprehension? Asked 11 years; Active 2 years; Viewed 165k times
Set Comprehension	(1)	**Question:** Fastest way to generate a random-like unique string with random length in Python 3 **Answer:** ...Use a set comprehension to produce a series of keys at a time to avoid having to look up and call the set.add() method in a loop... Asked 4 years; Active 6 months ago; Viewed 16k times
Dict Comprehension	(1)	**Question:** I like the Python list comprehension syntax. Can it be used to create dictionaries too? For example, by iterating over pairs of keys and values: Asked 12 years; Active 28 days ago; Viewed 1.0m times
Chain Comparison	(1)	**Question:** I write the following statement: if x >= start and x <= end: ... the tooltip tells me that I must simplify chained comparison What have I missed here? **Comment:** Thanks, I didn't know you could do that in Python. Was really scratching my head on this one. Asked 7 years; Active 2 years ago; Viewed 100k times
Truth Value Test	(1)	**Question:** Does Python have something like an empty string variable where you can do: if myString == string.empty: Asked 10 years; Active 19 days ago; Viewed 2.5m times
Loop Else	(2)	**Question:** Why does python use 'else' after for and while loops? Asked 9 years; Active 2 months ago;Viewed 245k times
Assign Multiple Targets	(1)	**Question:** Python assigning two variables on one line **Answer:** ...use sequence unpacking: self.a, self.b = a, b Asked 8 years; Active 8 years; Viewed 19k times
Star in Func Call	(2)	**Question:** What does the * operator mean in Python, such as in code like zip(*x) or f(**k)? Asked 11 years; Active 12 months ago; Viewed 234k times
For Multiple Targets	(2)	**Question:** Tuple unpacking in for loops Asked 9 years; Active 1 month ago; Viewed 222k times

3.3 Approach

We now present our refactoring tool for improving idiomatic coding practices. Figure 3.3 shows the three steps for designing and implementing our refactoring tool. These three steps answer three technical questions respectively: 1) how to identify programming idioms unique to Python; 2) how to detect anti-idiom code that can be implemented in Pythonic idioms; 3) how to refactor non-idiomatic code with Pythonic idioms in a systematic and extensible way. Rather than relying on mining code

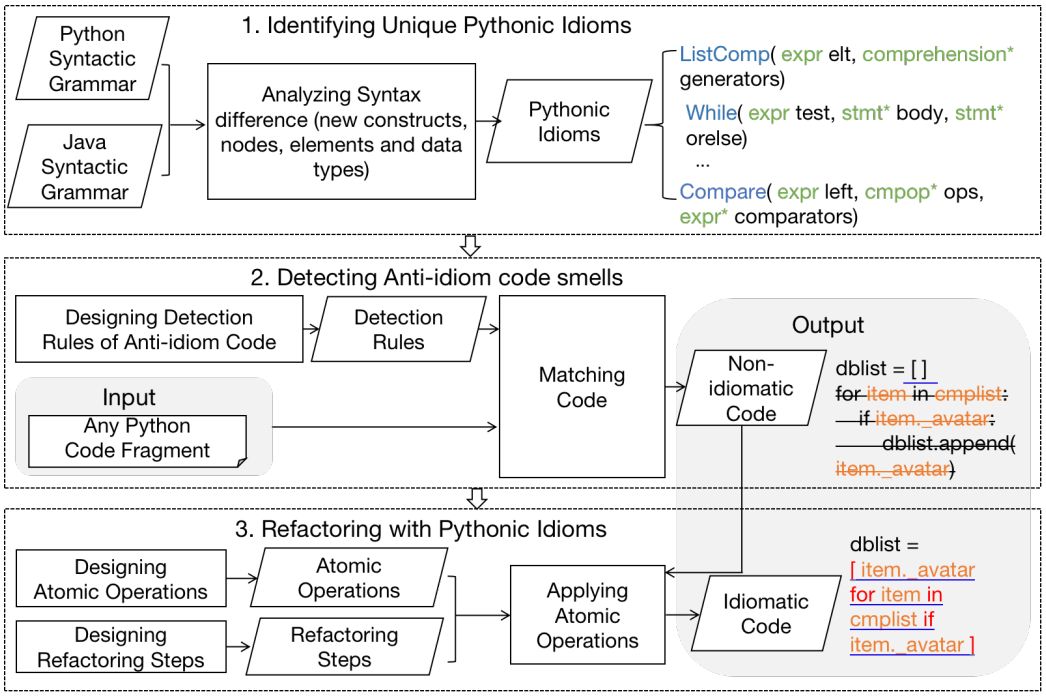


Figure 3.3: Approach overview

patterns or personal programming experience, our solution is built on the effective analysis of Python language syntax and specification.

3.3.1 Identifying Unique Pythonic idioms

Pythonic idioms are dispersed across diverse resources (Bader, 2017; Code., 2014; Hettinger, 2013; Knupp, 2013). Although prior work (Alexandru et al., 2018; Farooq and Zaytsev, 2021) conducted a literature review from online materials to summarize twenty-seven Pythonic idioms, the approach has three disadvantages. The one is that there are many Python syntaxes that also exist in many programming languages, e.g., assert and @staticmethod. Few developers write their own non-idiomatic code to implement the same functionality without these idioms. The second is that there are many Python APIs of Python modules to help Python users directly use these APIs to reduce the coding workload. For example, the collection module and the itertools module contain several APIs (e.g., collections.defaultdict, zip_longest and groupby). However, these APIs not only generally exist in other programming languages, and if we consider APIs as Pythonic idioms, it can lead to endless Pythonic idioms. The last is they usually mention only some popular Pythonic idioms (e.g., list-comprehension and truth-value-test) repeatedly based on personal programming experience, collecting pythonic idioms from online materials is ad-hoc and may miss important pythonic idioms. We find they lost four idioms including loop-else, assign-multi-targets, for-multi-targets and star-in-func-call.

In contrast to the previous works, our focus is on Python syntax that is unique to the Python programming language. Python grammar syntax plays a crucial role in Python software development because it defines the set of specifications that govern how Python code should be written. Regrettably, there is still a lack of a systematic way to explore Pythonic idioms within Python grammar syntax that are unique to Python programming language. Particularly, not all Python syntaxes are Pythonic. For example, For and If statements are basically all common programming grammar syntaxes, they are not Pythonic (unique) to Python programming language.

To identify unique Pythonic idioms that is unique to the Python programming language, we compare Python abstract syntax trees (AST) and the abstract syntax trees (AST) of the other programming language. We chose Java as the comparison object for the following reasons:

(1) Python and Java are currently among the most popular programming languages. According to the latest ranking report from IEEE Spectrum (*Top Programming Languages* 2023), Python and Java rank first

Pythonic Idioms	AST Representation	Code Examples	
(1) Python supports new AST nodes which do not exist in Java.			
list/set/dict-comprehension: Allow conditional construction of an iterable using for and if clauses with one code line	**ListComp** (expr elt, comprehension* generators) **SetComp** (expr elt, comprehension* generators) **DictComp** (expr key, expr value, comprehension* generators)	**new_cols = []** **for col in old cols:** **new_cols.append(** **col + postfix)**	new_cols = **[col + postfix** **for col in old cols]**
star-in-func-call: Unpack an iterable to the positional arguments in a function call	**Starred** (expr value, expr_context ctx):	nn.Linear(**gate channels[0],** **gate channels[2])**	nn.Linear(***gate channels[:4:2])**
(2) Python AST nodes contain new child nodes that are not present in Java.			
loop-else: A loop statement has an else clause	While(expr test, stmt* body, **stmt* orelse**) For(expr target, expr iter, stmt* body, **stmt* orelse**, string? type_comment)	is_root = True for children in resource: ... is_root=False break **if is_root:** ...	for children in resource: ... break **else:** ...
(3) A Child node of a Python AST node can be repeated multiple times that Java cannot.			
chain-comparison: Chain multiple comparison operations into one comparison operation	Compare(expr left, **cmpop* ops, expr* comparators**)	**a > b and a < 1**	**b < a < 1**
assign-multi-tar: Assign multiple values to multiple variables in an assign statement	Assign(**expr* targets**, expr **value**, string? type_comment)	**self. ad = device** **self. al4a_client = None**	**self. ad, self. al4a_client = device, None**

for-multi-tar Unpack the iterated target of a for statement	For(**expr target**, expr iter, stmt* body, **stmt* orelse**, string? type_comment)	for **sample** in family.samples:
		if **sample[0]** > 2:
		...
			for **e0, *e** in family.samples:
			if **e0** > 2:
			...

(4) A Child node of a Python AST node support more comprehensive data type			

truth-value-test: Directly check the "truthiness" of an object	While(**expr test**, stmt* body, stmt* orelse)
	If(**expr test**, stmt* body, stmt* orelse)
	Assert(**expr test**, expr? msg)
	BoolOp(**boolop** op, **expr* values**)
		embedding dim % 2 == 0	**not embedding dim**

Figure 3.4: Nine Pythonic idioms.

and second, respectively, indicating their widespread influence and application in the programming community.

(2) Python and Java are both general-purpose programming languages widely used in various applications (Gosling et al., 2005; Kuhlman, 2009). Compared to other purpose-specific programming languages (e.g., Javascript and SQL), Python and Java have more universal syntax and can cover a wider range of programming needs.

(3) Java's syntax is similar to Python's syntax (Dilhara, Ketkar et al., 2022), e.g., both have assignment and control flow statements. This similarity provides a solid basis for comparison. When choosing Java, we also considered other popular programming languages with generic purpose such as C#, C++ and C. Given that both Java and C# exhibit syntax similarities to C++, which, in turn, derives from C (*C Sharp and Java: Comparing Programming Languages* 2022; *Comparison of C Sharp and Java* 2022), either of them could serve as suitable comparison candidates. Ultimately, our decision to proceed with Java was influenced by its current status as the most popular programming language, second only to Python.

After that, we propose a hierarchical comparative approach across four levels of analytical granularity to analyze syntax differences between Python and Java. We begin by examining each Abstract Syntax Tree (AST) node defined in the Python Language Specification (*Python Abstract Grammar* 2022). If a corresponding counterpart does not exist in Java, we categorize it as a Pythonic idiom. Otherwise, we further check its children nodes. If one child node does not exist in Java, we categorize it as a Pythonic idiom. Otherwise, we further check its children nodes. If one child node does not exist in Java, we categorize it as a Pythonic idiom. Otherwise, we further check if the children node can be repeated any number of times or support more comprehensive data type but Java does not support. If this condition is met, we categorize it as a Pythonic idiom. In total, we collect 12 Pythonic idioms, with four of them (loop-else, for-multi-targets, ass-multi-targets, and star-in-func-call) not previously identified by previous works. Figure 3.4 shows the collected Pythonic idioms with code examples. The details of Pythonic idioms as follows:

(1) **Python supports new AST nodes which do not exist in Java.** This includes four syntactic constructs: ListComp, SetComp, DictComp and Starred which correspond to four idioms, list comprehension, set comprehension, dictionary comprehension and single asterisk operator. The list/set/dict-comprehension create iterable object with one line of code (*PEP 202-List comprehension* 2023; *PEP 274-Dict comprehension* 2023). The single asterisk operator is usually used to unpack the an iterable

into positional arguments inside a function call (star-in-func-call) (Developers, 2013). We also identify several other new constructs, such as Yield, With and GeneratorExp. However, it is very inconvenient to implement the same functionality as these new constructs in a non-idiomatic way. As such, the Yield-, With- or GeneratorExp-equivalent non-idiomatic code is too complex to safely refactor. Therefore, we do not consider Yield, With and GeneratorExp in this work.

(2) **Python AST nodes contain new child nodes that are not present in Java.** Although Python and Java share same AST nodes, the Python AST nodes incorporate child nodes that are absent in Java's AST nodes. This includes the Loop construct which consists of the For and While statement. For example, the For statement of Python adds orelse node (i.e., the loop-else idiom). The orelse node is executed after the loop iterator is exhausted, unless the loop ends prematurely due to a break statement (Knupp, 2013).

(3) **A Child node of a Python AST node can be repeated multiple times that Java cannot.** This includes the Assign statement with multiple targets (the assign-multi-targets idiom), the ChainComp with multiple operators (the chain-comparison idiom), and the For statement with multiple targets (the for-multi-targets idiom). For example, the assignment statement of Python allows multiple variables to be assigned simultaneously. A useful scenario for assign-multi-targets is to swap variables without creating temporary variables.

(4) **A Child node of a Python AST node support more comprehensive data type.** This includes the truth-value-testing idiom (Developers, 2023; Knupp, 2013). In Python, any object (e.g., string, numeric type and sequences) can be directly tested for truth value. For example, we can directly check if a variable "a" of list data type is empty with "`if not a`" instead of "`if a == []`".

3.3.2 Non-Idiomatic Code Detection

A Pythonic idiom possesses its own semantics and specific usage scenarios, aiding Python developers in expressing their intentions in a concise and efficient way. We observe that non-idiomatic code corresponding to a Pythonic idiom exhibits a deterministic AST node composition which satisfies some conditions (aka syntactic pattern). Therefore, we begin by identifying this composition and subsequently formulate the conditions that must be met by the AST node composition. Table 3.4 presents our detection rules and refactoring steps along with illustrative examples.

List/Set/Dict Comprehension

The list/set/dict-comprehension idiom is used for the object initialization (1st row in Table 3.4). It can continuously append elements to an iterable with just a single line of code. The corresponding non-idiomatic code should include an assign statement (*ass*) and a For statement (*for*). Given that Comprehension supports specified keywords including for, if, and if-else, the non-idiomatic code must meet the `MatchCompre` condition, meaning that the for statement corresponds to the syntax grammar of Comprehension.

The `MatchCompre` condition is summarized in Algorithm 1. To illustrate, in Figure 3.5, the Assign statements in the 2nd and 3rd lines are removed by replacing "`class_name`" and "`attribute`" in the 4th and 5th lines with the values assigned in the 2nd and 3rd lines. Furthermore, if the code block comprises only one statement, the algorithm proceeds to analyze the type of statement (whether it is a For node, an If node with or without an else clause, or involves appending elements to a list/set/dictionary). The primary objective is to ascertain whether the block intends to append elements to a data structure such as a list, set, or dictionary. For instance, in Figure 3.5, the 2nd line is identified as an If node (line 4), and its body involves appending elements to a list (line 8). Consequently, the code in Figure 3.5 aligns with the `MatchCompre` condition.

Table 3.4: Examples of detection and refactoring of anti-idiom code smells

Idiom	Detection Rules and Examples of Code Pairs	Refactoring Steps
list/set/dict comprehension	**Composition:** $P = [ass, for]$ **Conditions:** 1. ass.value ∈ {[], set(), dict(), {}} 2. `MatchCompre`(for.body) **AST:** | - Assign (*ass*) | | - targets | | - value | - ... | - For (*for*) | - body | - ... **Non-Idiomatic Code:** dblist = [] for item in cmplist: if item._avatar dblist.append(item._avatar) **Idiomatic Code:** dblist = [item._avatar for item in cmplist if item._avatar]	1: $comp$=Create("ListComp"/"SetComp"/"DictComp") 2: Traversing for, keep copying its children to $comp$ 3: **If** ass and for have the different parents **then** 4: Replace(for, ass) 5: Replace(ass.value, $comp$) 6: **Else** 7: Replace(ass.value, $comp$) 8: Remove(for)
chain comparison	**Composition:** $P = [blop]$ **Conditions:** 1. $blop$.op = And 2. $cmp_1 \in blop$.value ∧ $cmp_2 \in blop$.value 3. cmp_1 and cmp_2 have a common operand **AST:** |- BoolOP (*blop*) |- op |- values |- ... |- Compare (*cmp1*) |- ... |- Compare (*cmp2*) **Non-Idiomatic Code:** cp >= 178208 and cp <= 183983 **Idiomatic Code:** 183983 >= cp >= 178208	1: com=Create("Compare") 2: Traversing cmp_1 and cmp_2 to copy their children to com 3: **If** Num($blop.values$)> 2 **then** 4: Replace(cmp_1, com) 5: Remove(cmp_2) 6: **Else** 7: Replace($blop$, com)
truth test	**Composition:** $P = [cmp]$ **Conditions:** 1. cmp.parent ∈ {test, BoolOP} 2. cmp.ops ∈ {Eq, NotEq} 3. $\{cmp.left,\ cmp.comparators\} \cap EmptySet \neq \varnothing$ **AST:** |- test / BoolOP |- ... |- Compare (*cmp*) |- left |- comparators |- ops **Non-Idiomatic Code:** docs == [] and size > 3 **Idiomatic Code:** not docs and size > 3	1: com=Create("Compare") 2: Traversing cmp_1 and cmp_2 to copy their children to com 3: **If** Num($blop.values$)> 2 **then** 4: Replace(cmp_1, com) 5: Remove(cmp_2) 6: **Else** 7: Replace($blop$, com)

loop else	**Composition:** $P = [ass, for, if]$
	Conditions:
	1. for.orelse = ∅ and Break ∈ for.body
	2. $\forall stmt_j (stmt_j = Break \wedge stmt_j \in for.body \rightarrow \exists s(s \in \{Assign, If\} \wedge stmt_j.parent = s.parent \wedge (c_1 : \texttt{DiffSem}(s, if.test) \wedge \texttt{SameSem}(ass, if.test)) \vee c_2 : (\texttt{SameSem}(s, if.test) \wedge \texttt{OppositeSem}(ass, if.test) \wedge if.orelse \neq \varnothing))))$
	AST: |- Assign (*ass*) |- ... |- For / While (*loop*) | | - body | | - ... |- If (*if*)
	Non-Idiomatic Code:
	always_true = False
	for term in clause:
	if ...:
	last_clause= clause
	always_true = True
	break
	if not always_true:
	filtered_res.append(...)
	Idiomatic Code:
	for term in clause:
	if ...:
	last_clause= clause
	break
	else:
	filtered_res.append(...)
		1: **If** c_1 **then**
		2: Replace($for.orelse$, $if.body$)
		3: **If** $if.orelse$ is not $None$ **then**
		4: Copy($if.else$, Index($stmt_j$))
		5: **If** $ass.targets$ does not occur in other statements in detection rules **then**
		6: Remove(ass)
		7: **Else**
		8: Replace($for.orelse$, $if.orelse$)
		9: Copy($if.body$, Index($stmt_j$))
		10: **If** $ass.targets$ does not occur in other statements in detection rules **then**
		11: Remove(ass)
		12: **If** ass = Assign **then**
		13: Remove(s)
		14: Remove(if)

assign multi targets	**Composition:** $P = [ass_1, ..., ass_n]$ **Conditions:** 1. $\forall (ass_i, ass_k)(n \geq k > i > 0 \wedge (\sim \texttt{isDepend}(ass_k, ass_i) \vee \texttt{isDepend}(ass_k, ass_i) \rightarrow \exists ass_j (k > j > i \wedge ass_j.targets = ass_i.value)))$ **AST:** | - Assign (ass_i) | | - targets | | - value | - ... | - Assign (ass_n) **Non-Idiomatic Code:** (ass_1/ass_i): f = d[0] (ass_2/ass_j): d[0] = d[e] (ass_n/ass_k): d[e] = f **Idiomatic Code:** (ass_1): d[0], d[e] = d[e], d[0]	1: $value$=Create("Tuple") 2: Traversing P to copy its children to $value$ 3: Replace($ass_1.value$, $value$) 4: $targets$=Create("Tuple") 5: Traversing P to copy its children to $targets$ 6: Replace($ass_1.targets$, $targets$) 7: **For** i from 2 to n **do** 8: Remove(ass_i)
star-in func calls	**Composition:** $P = [arg_1, ..., arg_n]$ **Conditions:** 1. $\forall arg_i (n+1 > i > 0 \wedge arg_i \in Call.args \wedge arg_i = Subscript)$ 2. $\forall (arg_i, arg_j)(n+1 > i,\ j > 0 \wedge arg_i.value = arg_j.value)$ 3. the slice sequence of $arg_1, ..., arg_n$ is an arithmetic sequence **AST:** | - Call | - args | - ... | - Subscript (arg_1) | - value | - slice | - ... |- Subscript (arg_n) **Non-Idiomatic Code:** load_crowdhuman_json(sys.argv[1] , sys.argv[2], sys.argv[3]) **Idiomatic Code:** load_crowdhuman_json(*sys.argv[1:4])	1: $subs$=Create("Subscript", P) 2: Traversing P to compute and create the slice node of $subs$, and copy the value of Subscript node value to the value of $subs$ 3: $star$=Create("Starred", $subs$) 4: Replace(arg_1, $star$) 5: **For** i from 2 to n **do** 6: Remove(arg_i)

for multi targets	**Composition:** $P = [for]$
	Conditions:
	1. `Num`(for.targets)=1
	2. `isUseSubscript`(for.targets, for.body)
	AST: | - For (*for*) | - body | - targets | - iter
	Non-Idiomatic Code:
	for interval in intervals:
	if interval[1] - interval[0] > min_len:
	part = utter[interval[0] : interval[1]]
	...
	Idiomatic Code:
	for e0, e1, *e in intervals:
	if e0 - e1 > min_len:
	part = utter[e0 : e1]
	...
		1: Getting a variable mapping pair Map, where each element consists of a original variable and a new variable
		2: $target$=Create("Tuple", Map)
		3: Replace($for.targets$, $target$)
		4: **For** e in traverse($P.body$) **do**
		5: **If** e in Map **then**
		6: Replace(e, $Map[e]$)

Algorithm 1 MatchCompre (*block*)

```
1: simplify block by eliminating Assignment statements
2: if the number of statements of block is equal to 1 then
3:    stmt=block[0]
4:    if stmt is a For node or stmt is an If node without else clause then
5:       return MatchCompre(stmt.body)
6:    else if stmt is an If node with else clause then
7:       return if both stmt.body and stmt.else are appending elements to a list/set/dictionary.
8:    else if stmt is appending elements to list/set/dictionary then
9:       return True
10:   else
11:      return False
12:   end if
13: end if
```

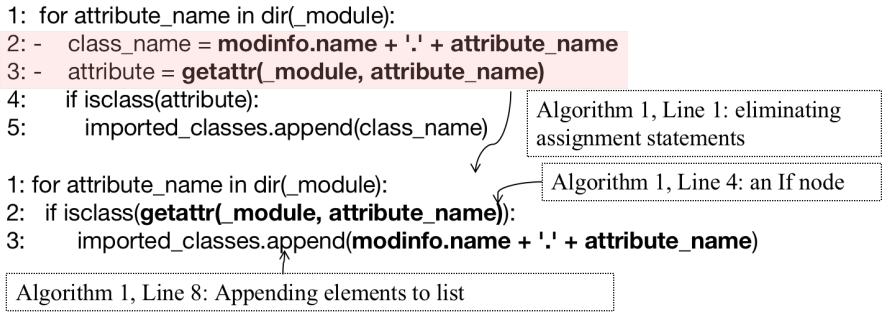


Figure 3.5: Example: MatchCompre algorithm of list/set/dict-comprehension.

The list-comprehension idiom is used for the list initialization (2nd row in Table 3.4). The rule first finds an empty assignment statement $stmt_1$ (e.g., "dblist = []"). Then, it finds a for statement $stmt_n$ which iteratively adds elements to the target ("dblist") of $stmt_1$. There cannot be other statements using the target "dblist" of $stmt_1$ between $stmt_1$ and $stmt_n$ to lest the "dblist" is modified (i.e., $isNotUse(stmt_1.target, stmt_1, stmt_n)$). Since the $stmt_n$ corresponds to the comp node of the ListComp construct which only supports for clause and if clause, the rule checks whether $stmt_n$ satisfies the *MatchCompre* condition, i.e., if the $stmt_n$ corresponds to the syntax grammar of Comprehension. The detection rule for the non-idiomatic code of the set-comprehension and the dict-comprehension idiom are the same.

Chain Comparison

The chain-comparison "a op_1 b op_2 c ... y op_n z" is equivalent to "a op_1 b and b op_2 c and ... y op_n z" **chainCompare**. The non-idiomatic code of the chain comparison must be a *BoolOp*-and expression which contains at least two compare nodes. Moreover, the two compare nodes have the same operands. For example, for the expression "`cp >= 178208 and cp <= 183983`" (3rd row in Table 3.4), the `cp` is the common operand of the two compare nodes, and the expression can be refactored as "`183983 >= cp >= 178208`".

Truth Value Test

The truth-value-test idiom is used for checking the "truthiness" of an object. Generally, when developers want to compare whether an object is equal or is not equal to a value, many programming languages use "==" or "!=" operator to achieve the functionality. In Python, any object can be directly tested for truth value, so developers do not need to use "==" or "!=" operator to test truth value. Python documentation specify the built-in objects in *EmptySet* (e.g., [] and set()) are considered as false value. Therefore, if a statement directly compares an object to the element of *EmptySet*, it will be regarded as a non-idiomatic code of the truth value test. However, not all compare nodes are refactorable with truth value test. For example, "`a!=[]`" in "`return a!=[]`" cannot be refactored because "`return a`" changes the code semantic. According to Python syntax, the non-idiomatic code of truth

value test corresponds to a test-type node. Therefore, our rule checks whether a compare node is the child of a test-type node, for example, the "runs([]) == []" is the child of an if-node "if runs([]) == []" (4th row in Table 3.4). Since the if-node is a test-type node, the compare node "runs([]) == []" is refactorable to a truth-value-test.

Loop Else

The else clause of the loop statement is executed after the iterator is exhausted, unless the loop was ended prematurely due to a break statement. The non-idiomatic way of implementing a loop-else generally has an assignment statement $stmt_1$ to flag current state, a for statement $stmt_n$ which contains a statement s to change the current state and a break statement $stmt_j$ to end the loop, and an if statement $stmt_{n+1}$ after the for statement $stmt_n$ to check the current state to execute different operations. There are four circumstances: c_1 and c_2 complement each other, and c_3 and c_4 complement each other.

The c_1 satisfies the following semantic conditions: the semantic of the assignment statement $stmt_1$ is the same as the semantic of the test node of if statement $stmt_{n+1}.test$, and the semantic of assignment statement s is different from the semantic of $stmt_1$ where s and the break statement $stmt_j$ are at the same scope. These semantic conditions are designed because the non-idiomatic code of loop-else implies two execution paths (5th row in Table 3.4): $stmt_1 \nrightarrow s$ and $stmt_1 \rightarrow stmt_{n+1}$ or $stmt_1 \rightarrow s$ and $stmt_j \nrightarrow stmt_{n+1}$.

The c_2 satisfies the following semantic conditions: the semantic of the assignment statement $stmt_1$ is the opposite of the semantic of the test node of the if-statement $stmt_{n+1}.test$, the if-statement $stmt_{n+1}$ has an else clause, and the semantic of the assignment statement s is the opposite of the semantic of $stmt_1$ where s and the break statement $stmt_j$ are at the same scope. The c_2 condition is a complement to the c_1 condition. If $stmt_{n+1}$ has an else clause and $stmt_{n+1}.test$ has the opposite semantic with $stmt_1$, it indicates that the else clause has the same semantic as $stmt_1$. Therefore, the code satisfying the c_2 condition is also refactorable to a loop-else. For example (5th row in Table 5), if we change $stmt_{n+1}.test$ "good_partition" into "not good_partition" and add an else clause to the if statement, the code satisfies the c_2 condition.

The c_3 satisfies the following semantic conditions: the semantic of the assignment statement $stmt_1$ is the same as the semantic of test node of the if-statement $stmt_{n+1}.test$, and the semantic of the if-statement s in the body of the loop statement $stmt_n$ is different from the semantic of $stmt_1$, and the body of the if-statement s contains the break statement $stmt_j$. The c_3 is a variant of c_1 and c_2. The c_1 and c_2 requires an assignment s to change the current state, but c_3 uses an if statement s to detect the change of the current state and break the loop, such as "if not good_partition: break".

The c_4 satisfies the following semantic conditions: the semantic of the the assignment $stmt_1$ is the opposite of the semantic of test node of the if-statement $stmt_{n+1}.test$, the if-statement $stmt_{n+1}$ has an else clause, and the semantic of the test node of if statement $s.test$ in the body of the loop-statement $stmt_n$ is the opposite of the semantic of $stmt_1$ and the body of the if-statement s contains the break statement $stmt_j$. The c_4 complements c_3, in the same vein as c_2 complements c_1.

Assign Multiple Targets

The assign-multiple-targets idiom is to assign multiple values at the same time in one assignment statement. For several consecutive assignment statements, if an assignment statement $stmt_k$ does not use the result of an assignment statement $stmt_i$ before it, these assignment statements are refactorable to assign-multi-targets. When an assignment statement $stmt_k$ uses the result of the an assignment statement $stmt_i$ before $stmt_k$, the code usually is to swap variables by creating temporary variables. For such non-idiomatic code, it requires that the target of a statement $stmt_j$ between the $stmt_i$ and the $stmt_k$ is the same as the value of $stmt_i$. For example (2nd row in Table **??**), $stmt_k$ "d[e] = f" uses the target "f" of $stmt_i$ "f = d[0]", and the target "d[0]" of the $stmt_j$ "d[0] = d[e]" is the same as the value "d[0]" of the $stmt_i$ "f = d[0]". This sequence of assignments via a temporary variable can also be refactored with the assign-multiple-targets idiom.

Star in Function Calls

The star-in-function-call idiom is usually used to unpack an iterable to the positional arguments in a function call (*PEP 448-Additional Unpacking Generalizations* 2023). The non-idiomatic way of passing a

sequence of arguments is that the subscript sequence of multiple consecutive parameters of a function call is an arithmetic sequence of the same variable. For example, "1, 2, 3" is an arithmetic sequence where the common difference is 1 for accessing the first, second and third element of "sys.argv" (3rd row in Table 3.4). It can be refactored into "*sys.argv[1:4:1]".

For Multiple Targets

The non-idiomatic code of the for-multiple-targets idiom only contains one variable as the target of for statement p. The body of p uses the subscript expression to get elements of the variable. For example (the last row of Table 3.4), the code uses "interval[0]" and "interval[1]" to get the elements of the variable "interval" inside the body of for loop. Instead, the elements of "interval" can be accessed using a for-multiple-targets idiom.

3.3.3 Refactoring with Pythonic idioms

According to Fowler (2018), a refactoring constitutes a series of small, behavior-preserving transformations. In adherence to this principle, we scrutinize the AST transformations necessary for converting a piece of anti-idiom code into idiomatic code. We identify four atomic AST-rewriting operations applicable across all idioms and subsequently compose these atomic operations into the refactoring steps for each Pythonic idiom. The four atomic operations are outlined below:

(1) **Copy(s, i):** This operation copies the node s from non-idiomatic code to the position i of a node of idiomatic code. If the node at the position i is empty, we copy s into the position i. Otherwise, we insert s into the position i. Since a refactoring does not change the code semantics, many parts of non-idiomatic code can be copied to the resulting idiomatic code. For example, in the list-comprehension idiom (1st row in Table 3.4), both the target node item and the iter node cmplist of non-idiomatic code are copied to the corresponding target and iter position of the comprehension node, respectively. Another instance is in the chain-comparison idiom (2nd row in Table 3.4), where we copy operands of the Compare node from non-idiomatic code to the position of operands in a new Compare node.

(2) **Create(s, *info):** This operation constructs a node of type s with information $*info$, where * represents any amount of information. To refactor non-idiomatic code with Pythonic idioms, it is sometimes necessary to create new AST nodes or elements that don't have corresponding parts in the non-idiomatic code. For instance, in the truth-value-test idiom (3rd row in Table 3.4), we need to create a Not node. In another example, for the star-in-function-call idiom (second-to-last row in Table 3.4), we need to create a Starred node with subscript information from the non-idiomatic code.

(3) **Remove(s):** This operation removes the node s from the AST of non-idiomatic code, which is no longer needed in idiomatic code. Generally, refactoring non-idiomatic code into idiomatic code will reduce the lines or tokens of code. Therefore, it is natural to remove nodes that are no longer used. For example, in the loop-else idiom (4th row in Table 3.4), we need to remove the initial flag assignment "good_partition = True" and the flag-update statement "good_partition = False" that are no longer needed when the loop-else idiom is used. In another example, for the assign-multi-targets idiom (5th row in Table 3.4), we remove assign statements from $stmt_2$ to $stmt_n$.

(4) **Replace(s, t):** This operation replaces the node s of non-idiomatic with the node t obtained through code transformation. For example, in the chain-comparison idiom (2nd row in Table 3.4), we replace the original expression "cp >= 178208 and cp <= 183983" with the resulting chain-comparison "183983 >= cp >= 178208". In another example, for the for-multi-targets idiom (the last row in Table 3.4), we replace "interval[0], interval[1]" with "interval_0, interval_1" respectively.

The refactoring steps for each pythonic idiom are presented in the 3rd column of Table 3.4. The green line numbers show the steps performed to transform the examples of non-idiomatic code on the left into the idiomatic code on the right in the 2nd column of Table 3.4.

For instance, to refactor the non-idiomatic code example into a list comprehension code (1st row in Table 3.4), we first create a ListComp node *comp* and then traverse the for statement $stmt_n$ to copy its children to the *comp* node (lines 1-2). This involves copying "item._avatar" to the position of $stmt_n.elt$ (i.e., elements to add to the list). Since $stmt_n$ and $stmt_1$ are at the same scope (line 6), we directly replace $stmt_1.value$ with *comp*, and then remove $stmt_n$ (lines 7-8).

In cases where $stmt_1$ and $stmt_n$ are at different scopes (line 3), the Remove operation for $stmt_1$ is not executed. This is because $stmt_n$ may not be executed after executing $stmt_1$. Therefore, we only replace $stmt_n$ with $stmt_1$ and then update the value of $stmt_n$ (lines 4-5).

3.4 Evaluation

This section reports the evaluation of our approach. We focus on two aspects: the correctness and usefulness of refactoring anti-idiom code smells with pythonic idioms:

RQ1: How accurate is our approach when refactoring real-world anti-idiom Python code with pythonic idioms?

RQ2: Do code refactorings performed by our approach have practical value for real-world projects?

3.4.1 RQ1: Correctness of Refactorings

Motivation

Refactorings involve complex logic for detecting anti-idiom code smells and applying code transformation. We would like to confirm the design and implementation of our approach is robust and correct on real-world Python code.

Method

As described in Section 3.2.1, we collect 7,638 repositories from GitHub which can be successfully parsed using Python 3, and collect 506,765 Python source files from these repositories. We apply our refactoring tool to these Python source files to detect nine types of anti-idiom code statements and refactor these statements with pythonic idioms. We use both testing and code review to verify the correctness of refactorings. Particularlly, since no benchmark dataset was available prior to this work, and constructing a comprehensive dataset is both labor-intensive and potentially unrealistic, conducting large-scale recall evaluations is infeasible at this stage. Additionally, given that code refactoring tasks prioritize precision and correctness over recall, we focus on evaluating the correctness of the tool in this work, without incorporating recall at scale. In the future, based on the tool of this work, we plan to develop a more comprehensive benchmark to appropriately assess the tool's recall performance and provide a more holistic evaluation of its effectiveness.

Testing based verification. To determine the test cases that cover the detected non-idiomatic code fragments, we first collect the fully qualified names of all methods called by a test method using the DLocator tool (Wang, Li et al., 2020). If the method that contains a non-idiomatic code fragment belongs to the list of the methods called by the test method, we consider this test method as a test case for the non-idiomatic code fragment. Note that one test case may test one or more methods, and one method may undergo one or more different types of refactorings. Then, to execute the test cases successfully, we install the packages that the project depends on by reading its requirements.txt. Note that not all test cases can be executed successfully because of several problems, such as requiring other non-python packages or to manually configure some parameters. We filter out such fail-to-execute test cases.

In this work, we use Pytest (Hunt, 2019), a popular Python unit testing frame which also supports the Python's default unittest tool (**unittest**). We run the test cases on the original methods with non-idiomatic code fragments to ensure they pass successfully. Then we run the test cases again on the refactored methods. If the refactored methods pass the test cases, we consider the detection of anti-idiom, non-idiomatic code fragments and the corresponding code refactorings are correct. For the refactorings that fail the test cases, two authors independently analyze the failure causes. The two authors have more than three years of Python development experience. They examine the detected non-idiomatic code fragments and the idiomatic code obtained by the refactorings, and determine if the failures are caused by non-idiomatic code smell detection or pythonic idiom transformation. A detection failure means a detected non-idiomatic code fragment is not refactorable, e.g, it cannot be safely refactored into semantic-equivalent idiomatic code. If the failure is caused by non-idiomatic code detection, we do not double count it as the failure of pythonic idiom transformation. The two

Table 3.5: Accuracy of anti-idiom code smell detection (d-acc) and idiomatic code transformation (r-acc)

Idiom	Testing				Code Review		
	#Refs	#TCs	d-acc	r-acc	#Refs	d-acc	r-acc
List-Comprehension	132	391	1	1	100	1	1
Set-Comprehension	21	39	1	1	100	1	1
Dict-Comprehension	102	297	1	1	100	1	1
Chain-Compa	309	837	1	1	100	1	0.99
Truth-Test	641	1680	0.986	1	100	1	1
Loop-Else	37	98	1	1	100	1	1
Assign-Multi-Tar	1802	4565	0.999	1	100	1	1
Star-in-Func-Call	86	201	0.977	1	100	0.98	1
For-Multi-Tar	85	314	1	1	100	1	1
Total	3215	7216	0.995	1	900	0.998	0.998

authors discuss to resolve their disagreement and reach the consensus. Finally, we compute the accuracy of anti-idiom code smell detection and idiomatic code refactoring for each pythonic idiom.

Code review based verification. We randomly sample 100 pairs of non-idiomatic code fragments and the corresponding idiomatic code fragments for each pythonic idiom. Then the two authors independently review these code pairs, and determine if the non-idiomatic code fragments are detected correctly and if the idiomatic code fragments are refactored correctly. They discuss to resolve their disagreement and reach the consensus. Based on their code review results, we compute the accuracy of anti-idiom code smell detection and idiomatic code refactoring for each pythonic idiom.

Result

Table 3.5 presents the analysis results. #Ref and #TCs of the Testing column are the number of refactorings with successfully-executed test cases and the corresponding number of test cases. #Ref of the Code Review column is the number of refactorings we reviewed. d-acc and r-acc are the accuracy of non-idiomatic code smell detection and idiomatic code transformation respectively. In total, we successfully test 3,215 refactorings from 479 repositories and reviewed 900 refactorings from 672 repositories. Overall, our approach is very robust on real-world code. It achieves 100% accuracy of detection and refactoring for five pythonic idioms, i.e., list-comprehension, set-comprehension, dict-comprehension, loop-else and for-multi-targets. It achieves 100% detection accuracy for chain-comparison, and 100% refactoring accuracy for truth-value-test, assign-multi-targets and star-in-func-call.

Detection failure analysis. Our verification identifies 15 detected non-idiomatic code fragments which are not refactorable, including 9 for truth-value-test, 2 for assign-multi-targets and 4 for star-in-func-call. For example, for the truth-value-test "if `xpath_results == []`", if "`xpath_results`" is an empty string, the if-condition is false. However, the idiom "`if not xpath_results`" will be true if "`xpath_results`" is an empty string. Therefore, "`if xpath_results == []`" cannot be refactored into "`if not xpath_results`". Other non-refactorable truth-value-test cases suffer from the same problem.

The two non-refactorable assign-multi-targets failures are caused by the limitation of the Python static parsing. For example, for the two assignment statements, $stmt_1$: `lib=...` and $stmt_2$: `tmpl='{lib}'`. "`lib`" of $stmt_2$ is a variable because Python uses curly brackets to insert variables in string. $stmt_2$ uses the target "`lib`" of $stmt_1$, so the two statements cannot be refactored into "`lib,tmpl=...,'{ lib}'`" because it will make `tmpl` use the old value of `lib`. Since the parser parses '{lib}' into a string constant and does not parse the "`lib`" inside the brackets into a variable, we loss the information of data dependency and mistakenly identify the statements as refactorable.

Table 3.6: Results of our refactoring pull requests

Idiom Category	Accepted	Rejected	Merged	#Repo
List Comprehension	6	3	5	10
Set Comprehension	5	3	4	10
Dict Comprehension	5	0	5	10
Chain Comparison	4	4	3	10
Truth Value Test	3	3	3	10
Loop Else	5	1	4	10
Assign Multi Targets	3	2	1	10
Star in Func Call	1	6	1	10
For Multi Targets	2	1	2	10
Total	34	23	28	84

For the star-in-func-call idiom, both testing and code review find two non-refactorable non-idiomatic code fragments identified by our detection tool. The reason is that our tool does not consider the semantic of Python slice. For example, for the code "`self.add_circle_arc(p[-1], p[0], p[1])`", "`-1, 0, 1`" is an arithmetic sequence. However, Python list grows linearly and is not cyclic, as such slicing does not wrap (from end back to start going forward) as we expect.

Code transformation failure analysis. Our tool makes only 1 code transformation error for chain-comparison. For the code "`type is not None and self._meta_types and type not in self._meta_types`", it has three comparison operations and "`type`" is the common operand of the first and the third operation. Therefore, we refactor it into "`None is not type not in self._meta_types and self._ meta_typestype`". However, if "`self._meta_types`" is None, the refactoring will report a TypeError at runtime because None is not iterable. As our tool does not analyze the priority of comparison operations and adjust their order when refactoring the code, the resulting code encounters the runtime error.

3.4.2 RQ2: Usefulness of Refactorings

Motivation

Our tool is the first refactoring tool for pythonic idioms. We would like to know how well Python developers accept the refactorings our tool makes and what opinions they have towards pythonic idiom refactoring in practice.

Method

We randomly sample 10 refactorings (including the original non-idiomatic code fragments and the resulting idiomatic code after refactoring) for each type of pythonic idiom. The sampled refactorings come from 84 repositories. We fork the repository corresponding to the non-idiomatic code fragment and commit a pull request with the resulting idiomatic code. Readers can find the list of the 90 pull requests in our replication package. We collect the developers' responses to our pull requests, and count how many pull requests have been accepted or rejected by developers. Among the accepted pull requests, we further count how many pull requests have been merged into the repositories.

Result

Table 4.4 presents our experiment results. Among 90 pull requests, we received 57 responses, including 34 accepted and 23 rejected. 28 of the accepted pull requests have been merged into the repositories. The six pull requests that are not merged as they are not yet tested. The 63% (57/90) response rate indicates that Python developers pay attention to the pythonic idiom refactorings. The 60% acceptance

rate and the 50% merge rate among the responses provide the initial evidence of our refactoring tool's practicality and usefulness.

Among the accepted pull requests, many developers praise the refactoring pull requests we made to their repositories. For example, two developers praise the dict-comprehension refactoring (''{ name: mod for name, mod in module.named_modules() if isinstance(mod, _ConvNd)}'') "Thanks for the contribution, looks great!". Many developers confirm that the suggested refactorings are more pythonic, such as "Definitely more pythonic!" on a list-comprehension refactoring. Some developers express the interests in refactoring other places with the same pythonic idiom, such as "I will change the other place to a more pythonic style ..." inspired by our pull request for an assign-multiple-targets refactoring.

We analyze the rejected pull requests and summarize four main concerns developers have about pythonic idiom refactorings: *readability*, *performance*, *systematic refactoring*, and *inertia*.

Readability. 13 out of 23 reject responses are concerned that the pythonic idioms make the code less readable. For example, the developer comments on a suggested star-in-func-call refactoring (''*clip.size([2:4:1])'') : "While your change is indeed feasible, I believe the original style is more readable". Even with the readability concern, some developers express that they learn something from the suggested refactorings. For example, the developer comments on the chain-comparison refactoring (''sessions is None is metrics''): "... Interesting, ..., I learned something today though, thanks." As another example, the developer worries that refactoring may loss specific information for the truth-value-test. For example, a developer replied "I feel like asserting it to empty dict is more explicit and readable" if ''assert deepdiff.DeepDiff(...) == {}'' is refactored into ''assert not deepdiff.DeepDiff(...)''.

Performance. 3 reject responses are concerned about the performance or memory usage. For example, for the list-comprehension refactoring, the developers reject the pull request because they are not sure that the performance improvement would be significant in their project. For the star-in-func-call refactoring which refactors ''ss[0], ss[1], ss[2], ss[3], ss[4], ss[5], ss[6]'' into ''*ss[0:7:1]'', the developer believes the refactoring can cause memory fragmentation.

Systematic refactoring. 3 reject responses indicates that developers do not want to refactor the project in an ad-hoc way. Two responses are discouraged to refactor only one code fragment of the project. For example, although the developers reject our pull request for a set-comprehension refactoring, they propose that such refactorings should be applied to the whole project rather than by a single pull request to just one place. In another reject response, the developer replies that "Waf is just a tool for us. We don't need style patches for it." for a list-comprehension refactoring. In fact, we believe these responses confirm the need for systematic pythonic idiom refactoring tool like ours. Our tool can scan and refactor the whole project and dependent packages. It was just we submitted only some randomly sampled refactorings to the projects.

Inertia. 4 rejects are because the developers prefer the original code. For example, a developer replies "Thanks for the suggestion, I prefer the existing code" for a star-in-func-call refactoring. And some developers would like to accept pull requests to fix bugs instead of code refactoring, e.g., the developer replies to a for-multiple-targets refactoring: "I think it's better to leave the RUBI code alone for now unless there is work to fix it.".

3.5 Discussion

3.5.1 Pythonic Coding Practices

Refactoring is a widely adopted practice to improve code quality. A wide range of refactorings have been proposed to address code smells such as code clones, feature envy, shotgun surgery. Our work introduces a new type of code smell, i.e., non-idiomatic code that can be refactored with Pythonic idioms. Our empirical study on GitHub repositories and Stack Overflow questions calls for the tool support for assisting developers in using Pythonic idioms consistently. Our refactoring tool is the first tool of this kind. The evaluation on a large number of Python projects provides positive and encouraging feedback on the prototype. In the future, researchers could integrate this tool into GitHub Actions workflows to enable automated refactoring of idioms, ensuring consistent and concise coding style across projects. Meanwhile, some developer feedback raises concerns about the readability

and performance of Pythonic idioms. This calls for the careful validation of the conciseness and performance of Pythonic idioms. However, existing online materials are anecdotal and mostly based on personal programming experience. Our work produces a large dataset of non-idiomatic versus idiomatic code from real-world projects, which serves as an excellent testbed to empirically investigate the general claims and concerns about Pythonic idioms.

3.5.2 Threats to Validity

Threats to internal validity relate to two aspects in our work: (1) the errors in the implementation of code refactoring tool and (2) personal bias in evaluating accuracy of code refactoring. For the aspect (1), we have double-checked the code and verified the accuracy of our tool implementation by manually examining a large number of refactoring instances outputted by each step of our tool. As for the aspect (2), two authors with more than three years of Java and Python programming experience check the accuracy of refactoring instances independently. Furthermore, we collect a large number of real test cases to test the refactored code.

Threats to external validity relate to the generalizability of experiment results. To alleviate this threat, we built a large-scale dataset of 7,638 repositories and 506,765 Python files. To explore whether our code refactoring has practical value for developers, we submitted 90 pull requests to project members to review. The number of pull requests is larger than existing user studies in previous works Gao et al. (2021); Pan et al. (2020); Zhang, Huang et al. (2020). We release our tool and data in GitHub[3] for public evaluation.

3.6 Conclusion and Future Work

This chapter designs and implements the first automatic refactoring tool for nine types of Pythonic idioms. Our tool is motivated by the empirical observation of the challenges in writing pythonic code from the Stack Overflow discussions and of the wide presence of non-idiomatic code in thousands of real-world Python projects. Rather than relying on idiom mining, literature review or personal programming experience, our approach identifies Pythonic idioms and define non-idiomatic syntactic patterns and idiomatic code transformation steps through the systematic analysis of Python abstract syntax grammar. Our tool is robust and correct in detecting anti-idiom code smells and refactor these smells in real-world Python projects. The refactorings made by our tool have been well received and praised by the Python developers. In the future, we will integrate our refactoring tool into the open-source linting tool (e.g., Pylint). We will systematically investigate the readability and performance concerns about Pythonic idioms based on the large-scale refactoring dataset our tool produces.

[3]https://github.com/idiomaticrefactoring/CodeRefactoringPythonIdioms

Chapter 4

Hybrid Knowledge-Driven Refactoring to Pythonic Idioms Leveraging Large Language Models

Z. Zhang, Z. Xing, X. Xiao et al. (2024). 'Refactoring to Pythonic Idioms: A Hybrid Knowledge-Driven Approach Leveraging Large Language Models'. In: *2024 ACM International Conference on the Foundations of Software Engineering (FSE '24)*. FSE 2024. , Brazil, Brazil, , Association for Computing Machinery. DOI: 10.1145/3597503.3639101

Pythonic idioms are highly valued and widely used in the Python programming community. However, many Python users find it challenging to use Pythonic idioms. Adopting rule-based approach or LLM-only approach is not sufficient to overcome three persistent challenges of code idiomatization including code miss, wrong detection and wrong refactoring. Motivated by the determinism of rules and adaptability of LLMs, we propose a hybrid approach consisting of three modules. We not only write prompts to instruct LLMs to complete tasks, but we also invoke Analytic Rule Interfaces (ARIs) to accomplish tasks. The ARIs are Python code generated by prompting LLMs to generate code. We first construct a knowledge module with three elements including ASTscenario, ASTcomponent and Condition, and prompt LLMs to generate Python code for incorporation into an ARI library for subsequent use. After that, for any syntax-error-free Python code, we invoke ARIs from the ARI library to extract ASTcomponent from the ASTscenario, and then filter out ASTcomponent that does not meet the condition. Finally, we design prompts to instruct LLMs to abstract and idiomatize code, and then invoke ARIs from the ARI library to rewrite non-idiomatic code into the idiomatic code. Next, we conduct a comprehensive evaluation of our approach, RIdiom, and Prompt-LLM on nine established Pythonic idioms in RIdiom. Our approach exhibits superior accuracy, F1-score, and recall, while maintaining precision levels comparable to RIdiom, all of which consistently exceed or come close to 90% for each metric of each idiom. Lastly, we extend our evaluation to encompass four new Pythonic idioms. Our approach consistently outperforms Prompt-LLM, achieving metrics with values consistently exceeding 90% for accuracy, F1-score, precision, and recall.

4.1 Introduction

Pythonic idioms refer to programming practices and coding conventions that align with the core philosophy and style of the Python programming language (Alexandru et al., 2018; Farooq and Zaytsev, 2021; Slatkin, 2020; Zhang, Xing, Xia, Xu and Zhu, 2022). RIdiom in Chapter 3 identified nine Pythonic idioms by comparing syntax differences between Python and Java. Farooq and Zaytsev (2021) conducted a literature review to identify and explore the usage of twenty-seven Pythonic idioms. An example of a Pythonic idiom is the chain-comparison idiom, which allows comparing multiple variables in one comparison operation, such as "-size <= x.indices(size)[0] <= size". The Python community continually strives to design and improve them to achieve code conciseness and improved performance (Alexandru et al., 2018; developers, 2000). For the above example of chain-comparison, in contrast to the non-idiomatic equivalent, "-size <= x.indices(size)[0] and x.indices(size)[0] <= size", the chain-comparison simplifies code and improves performance. Given these benefits, the community and renowned Python developers actively promote the widespread adoption of Pythonic

idioms (Bader, 2017; Beazley and Jones, 2013; Code., 2014; Hettinger, 2013; Knupp, 2013; Slatkin, 2020).

However, previous studies (Alexandru et al., 2018; Zhang, Xing, Xia, Xu and Zhu, 2022) have indicated that Python users often be unaware of Pythonic idioms or unsure of how to correctly use Pythonic idioms, as Pythonic idioms are scattered across various materials, and are known for their versatile nature (Bader, 2017; Beazley and Jones, 2013; Code., 2014; Slatkin, 2020). For example, chain-comparison idiom also supports "in" operator (e.g., "`line <= r[1] in rlist`"), yet this usage often goes unnoticed by many Python users. To help Python users use Pythonic idioms, refactoring non-idiomatic code with Pythonic idioms emerges as a promising solution (Phan-udom et al., 2020; Zhang, Xing, Xia, Xu and Zhu, 2022). Through pilot studies, we find the task faces three challenges including code miss, wrong detection and wrong refactoring because of the versatile nature of Pythonic idioms. Code miss refers to **missing non-idiomatic code that can be refactored with Pythonic idioms**. Wrong detection refers to **misidentifying non-refactorable non-idiomatic code with Pythonic idioms as refactorable**. Wrong refactoring refers to **giving wrong idiomatic code for refactorable non-idiomatic code with Pythonic idioms**.

A most related work, RIdiom in Chapter 3, employs a rule-based approach to establish detection rules and refactoring procedures to refactor the non-idiomatic code into idiomatic code for nine Pythonic idioms. However, it is noteworthy that when confronted with intricate instances of non-idiomatic code, the reliance on pre-defined, inflexible rules cannot overcome the above three challenges (see RIdiom examples of Figure 4.1 in Section 4.2). Even when identified, formulating rules to refactor it into idiomatic code is challenging. On the other hand, in light of the success of Large Language Models (LLMs), users can simply describe natural language prompts to instruct LLMs to perform specific software engineering tasks, such as code generation (Dong et al., 2023; Fried et al., 2023; Nijkamp et al., 2022; *OpenAI Codex* 2023; Vaithilingam et al., 2022) and program synthesis (Huang, Yuan et al., 2023; Huang, Zhu et al., 2023; Huang, Zou et al., 2023; Peng et al., 2023). It inspires us to explore LLMs for code idiomatization, wherein we observe their powerful ability in certain scenarios, e.g., code ① of Figure 4.1 can be correctly refactored with set comprehension by LLMs. However, without knowledge guiding, LLMs may make obvious mistakes that can be easily avoided using rules because of the inherent randomness and black boxes of LLMs (see Prompt-LLM examples of Figure 4.1 in Section 4.2).

This observation underscores the insufficiency of relying solely on rule-based approach or LLMs. It motivates us to propose a hybrid approach that combines the determinism inherent in rule-based approach with the adaptability offered by LLMs. Specifically, our hybrid approach comprises three core modules: **a knowledge module**, **an extraction module**, and **an idiomatization module**. For each module, we write prompts to instruct LLMs to complete tasks or invokes Analytic Rule Interfaces (ARIs) to complete tasks. ARIs are Python code generated by prompting LLMs. The knowledge module is to construct a knowledge base consisting of three elements of non-idiomatic code of thirteen Pythonic idioms and an ARI library. The three elements are ASTscenario (the usage scenario of non-idiomatic code), ASTcomponent (the composition of non-idiomatic code) and Condition (the condition that refactorable non-idiomatic code must meet). The ARI library consists of ARIs to extract three elements and auxiliary ARIs to rewrite non-idiomatic code into idiomatic code. In the extraction module, for any syntax-error-free Python code, we invoke ARIs from the ARI library to extract ASTscenario and ASTcomponent that satisfies the condition, which will be input into the idiomatization module. The idiomatization module consists of three steps: abstracting code, idiomatizing code and rewriting code. We first abstractly represent the code of ASTcomponent by prompting LLMs. We then idiomatize the abstract code through LLM prompts, producing an abstract idiomatic code. Finally, we utilize ARIs to rewrite the non-idiomatic code into idiomatic code by using the abstract idiomatic code.

We conduct two experiments to evaluate the effectiveness and scalability of our approach. For effectiveness, we examine nine Pythonic idioms identified by RIdiom (Zhang, Xing, Xu et al., 2023b). To determine a complete, correct and unbias benchmark, we randomly sample methods from the methods of each Pythonic idiom collected by the RIdiom tool in Chapter 3. We independently run our approach, RIdiom and Prompt-LLM for the sampled methods to generate code pairs, and invite external workers to verify the correctness of code pairs by each approach manually. Then two authors and external workers discuss and resolve the inconsistencies. The metrics of accuracy, F1-score, precision, and recall were employed for evaluating results. The results demonstrate our approach

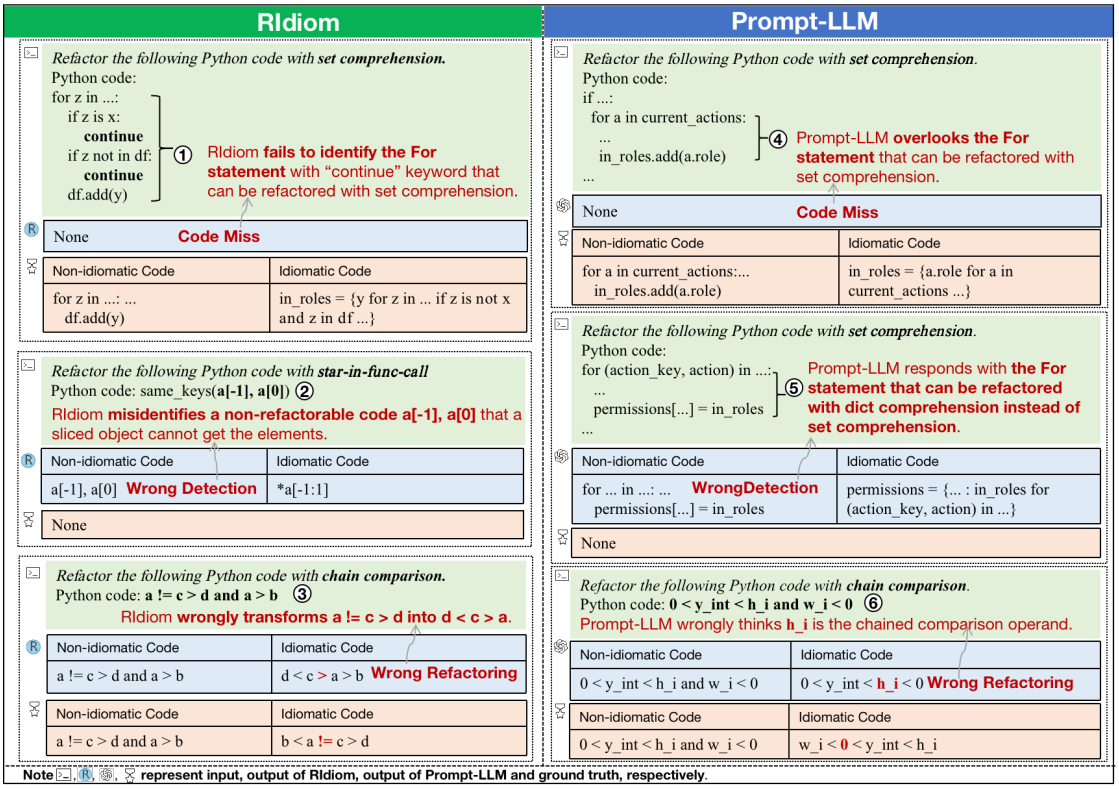


Figure 4.1: Motivating examples

achieves the best performance in accuracy, F1-score and recall compared to RIdiom and Prompt-LLM and achieves comparable precision with RIdiom. To evaluate the scalability of our approach, we choose four new Pythonic idioms not covered by RIdiom in Chapter 3. To avoid the bias of the benchmark, we randomly sample 600 methods from all methods in RIdiom. To ensure the correctness and completeness of our approach, we following the same process in Section 4.4.1. Since RIdiom does not support idiomatization for the four Pythonic idioms. We do not run RIdiom on the new four idioms. Our approach consistently outperformed in accuracy, F1-score, precision, and recall, all surpassing 90% for each Pythonic idiom, which shows that our approach can be effectively extend to new Pythonic idioms.

In summary, the contributions of this chapter are as follows:

- This is the first work to exploit LLMs into code idiomatization with Pythonic idioms, paving the way for new opportunities in code idiomatization.
- We propose a hybrid knowlege-driven approach with ARIs and prompts based on LLMs to refactor non-idiomatic code into idiomatic code with Pythonic idioms.
- We conduct experiments on both established and new Pythonic idioms. The high accuracy, F1-score, precision and recall verify the effectiveness and scalability of our approach. We provide a replication package (*Replication Package* 2023) for future studies.

4.2 Motivation

Although using Pythonic idioms can improve the conciseness and performance (Alexandru et al., 2018; Leelaprute et al., 2022; Zhang, Xing, Xia, Xu and Zhu, 2022; Zhang, Xing, Xia, Xu, Zhu and Lu, 2023), refactoring non-idiomatic Python code with Pythonic idioms for a given Python code is not easy. RIdiom in Chapter 3 is the state-of-the-art rule-based approach that formulates detection and refactoring rules to automatically refactor non-idiomatic code into idiomatic code for nine Pythonic idioms. Recently, large language models (LLMs) have achieved great success in various software engineering tasks (Brown et al., 2020; Feng and Chen, 2023; Huang, Zou et al., 2023; OpenAI, 2023; Peng et al., 2023; Ren, Ye et al., 2023). LLMs can directly complete various tasks by receiving natural

language prompts as input, a process we refer to as Prompt-LLM. To explore challenges encountered in this endeavor for the two approaches, we randomly collect ten Python methods crawled by RIdiom. The methods may contain several non-idiomatic code that can be refactored with a Pythonic idiom. Next, we apply RIdiom and Prompt-LLM to each Python method for nine Pythonic idioms from RIdiom. Two authors collaborate to check results of RIdiom and Prompt-LLM and then identify challenges encountered by the two approaches. We summarize three challenges that are described as follows.

(1) Code Miss: miss non-idiomatic code that can be refactored with Pythonic idioms. The code written by Python users comes in various styles, the code may contain several refactorable non-idiomatic code with a Pythonic idiom and the form of non-idiomatic code may be diverse. Missing the refactorable non-idiomatic code can lead to redundant code and performance degradation. Unfortunately, code missing is common in the two approaches. On the one hand, given the diverse and intricate nature of non-idiomatic code patterns, some instances may pose challenges that surpass the capabilities of straightforward rule-based identification. For example, code ① of the RIdiom column of Figure 4.1 is a "for" statement with two "continue" statements that can be refactored with set-comprehension. Since set-comprehension does not support "continue" keyword, RIdiom wrongly assumes the code cannot be refactored. Actually, we can change "z is x" and "z not in df" into "z is not x" and "z in df", and then we use the "and" to connect the two conditions to remove continue statements. On the other hand, in a codebase, instances of non-idiomatic code are distributed throughout, necessitating a comprehensive scan of the entire codebase to identify such occurrences. Unlike rule-based programs that deterministically scan Python code from start to end, LLMs operate as black boxes. This non-deterministic nature can inadvertently lead to the oversight of refactorable non-idiomatic code (Alexandru et al., 2018; Farooq and Zaytsev, 2021; Zhang, Xing, Xia, Xu, Zhu and Lu, 2023). For example, code ④ of the Prompt-LLM column in Figure 4.1 shows LLMs miss a "for" statement that can be refactored with set-comprehension.

(2) Wrong Detection: misidentify non-refactorable non-idiomatic code with Pythonic idioms as refactorable, which can lead to misunderstandings among Python users regarding Pythonic idioms and potentially introducing bugs into the codebase. Although not common in RIdiom, it should not be ignored. The detection rules of RIdiom are human-defined, and developers may overlook the nuances of code structures and Python syntax semantics, leading to false discoveries. For example, for ② of Figure 4.1, RIdiom determines "-1, 0" is an arithmetic sequence, so it wrongly thinks that "a[-1], a[0]" can be obtained by a sliced object "a[-1:1]". However, since Python list grows linearly and is not cyclic, as such slicing does not wrap (from end back to start going forward) as we expect, the "a[-1:1]" actually is empty. On the other hand, although LLM has powerful abilities, its flexibility and adaptability usually cause inappropriate or off-topic response. For example, code ⑤ of the Prompt-LLM column in Figure 4.1 shows that the Prompt-LLM wrongly classifies a "for" statement that can be refactored with dict-comprehension as refactorable non-idiomatic code with set-comprehension. Actually, we can add a condition to check whether the for statement has an "add" function call to filter out the wrong detection. For another example, non-idiomatic code of chain comparison should have two comparison operations. However, Prompt-LLM often mistakenly suggests refactoring a single comparison operation with chain comparison, even though it cannot be refactored in this way. For instance, when encountering a single comparison operation like "start is not None", Prompt-LLM wrongly assumes it can be refactored using chain comparison.

(3) Wrong Refactoring: give wrong idiomatic code for refactorable non-idiomatic code with Pythonic idioms. It occurs in identifying refactorable non-idiomatic code but wrongly refactoring it, resulting in inconsistency in code behavior before and after refactoring. The diversity and complexity of such non-idiomatic code make both the rule-based approach and the Prompt-LLM approach prone to errors. For example, for the chain-comparison, to chain two comparison operations into one comparison operation, we need to reverse compare operands for each comparison operation and consider if we need to change the comparison operation. When one comparison operation has more than one comparison operation, it is more likely to make mistakes. For example, code ③ of the RIdiom column in Figure 4.1 shows that RIdiom wrongly transforms "a != c > d" into "d < c > a". Directly using LLMs may make unexpected mistakes. For example, code ⑥ of Prompt-LLM column of Figure 4.1 shows that Prompt-LLM assumes that "h_i" is the chained comparison operand and then wrongly refactors it into "0 < y_int < h_i < 0".

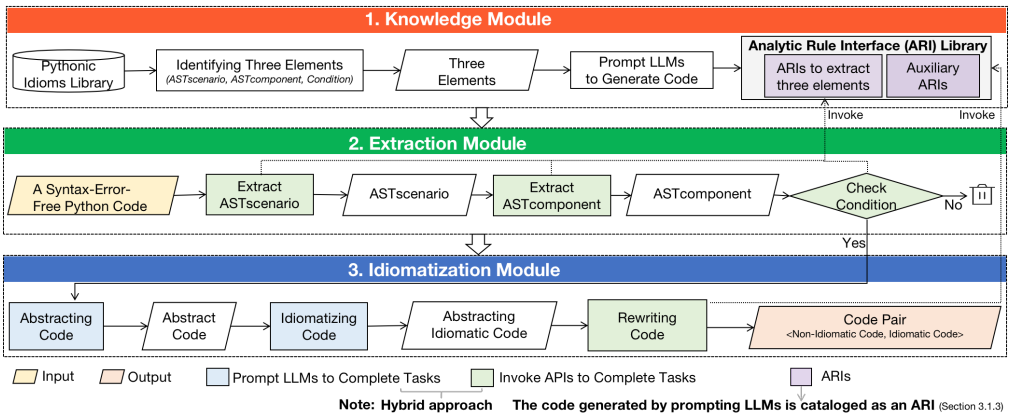


Figure 4.2: Approach overview

The three challenges shown in Figure 4.1 indicate that the rule-based approach, while deterministic, may still fall short in identifying all refactorable non-idiomatic code instances, especially those that are inherently complex or difficult to address through formulating rules (e.g., ① of Figure 4.1). Conversely, relying solely on the flexibility and adaptability of LLMs without knowledge guidance can lead LLMs to make obvious mistakes. For example, refactorable non-idiomatic code with set comprehension should contain an "add" function call. Regrettably, code ⑤ of Figure 4.1 lacks this function call. The absence of this contextual knowledge leads LLMs to misidentify it can be refactored with set comprehension. Therefore, a judicious approach emerges: initially employing code to handle deterministic and simple tasks, and then leveraging LLMs to tackle the more challenging refactoring endeavors where the rule-based approach may struggle. This hybrid approach stands poised to offer a comprehensive and effective solution.

4.3 Approach

Inspired by the motivating examples in Section 4.2, we propose a hybrid approach based on LLMs to refactor non-idiomatic code with Pythonic idioms. Figure 4.2 shows the approach overview. We first construct knowledge base of non-idiomatic code of Pythonic idioms, which consists of three elements: ASTscenario, ASTcomponent and Condition, and an ARI library consisting of ARIs to extract the three elements and ARIs to rewrite code. After that, for a given Python code, we first call ARIs to extract its ASTcomponent from the ASTscenario, and then filter out ASTcomponent that does not meet the condition. Then we input the ASTcomponent and ASTscenario from extraction module into the idiomatization module. The idiomatization module consists of three steps: abstracting code, idiomatizing code and rewriting code. To reduce the pressure on LLMs to idiomatize code, we first abstract code by abstracting expression of the code corresponding ASTcomponent. And then we write prompts to make LLMs idiomatize the abstract code. After we get the abstract idiomatic code, we need to replace the abstract expression with the original expressions, and then rewrite the non-idiomatic code with the idiomatic code. Since the rewriting operations are simple, we call ARIs from the Auxiliary ARIs to complete.

4.3.1 Knowledge Module

As investigated in Section 4.2, lacking specific knowledge, directly using LLMs to find a non-idiomatic code with a Pythonic idiom from a given Python code is like finding a needle in a haystack. For example, the non-idiomatic code of set-comprehension should have a For node containing an add function call. Without the knowledge, Prompt-LLM misses one For node as shown in ④ of Figure 4.1, and misidentifies a For node without the add function call as non-idiomatic code of set-comprehension as shown in ⑤ of Figure 4.1. We observe the non-idiomatic code that can be refactored with Pythonic idioms has deterministic knowledge. Therefore, we can construct a knowledge base to boost the ability of LLMs.

Table 4.1: Pythonic Idioms Library for Thirteen Pythonic Idioms

Source	Idiom	Explanation	Non-Idiomatic Code	Idiomatic Code
RIdiom, Farooq et al.	list/set/dict -comprehension	Use one line to append elements to an iterable	**new_cols = []** **for col in old_cols:** **new_cols.append(col + postfix)**	**new_cols = [col + postfix** **for col in old_cols]**
	chain-comparison	Chain mutliple comparison expressions into one comparison expression	**a > b and a < 1**	**b < a < 1**
	truth-test	Directly check the "truthiness" of an object	**embedding_dim % 2 == 0**	**not embedding_dim % 2**

RIdiom	loop-else	A loop statement has an else clause	while attempt < 3:
			...
			if body is not None:
			break
			if body is None:
			...
				while attempt < 3:
				...
				if body is not None:
				break
				else:
				...

	assign-multi-tar	Assign multiple values to multiple variables in an assign statement	**self._ad = device** **self._sl4a_client = None**	**self._ad, self._sl4a_client =** **device, None**

	for-multi-tar	Unpack the iterated target of a for statement	for **sample** in family.samples:
			if **sample[0]** > 2:
			...
				for **e0, *e** in family.samples:
				if **e0** > 2:
				...

	star-in-func-call	Unpack an iterable to the positional arguments in a function call	nn.Linear(**gate_channels[i]** **, gate_channels[i+1])**	nn.Linear(***gate_channels[i:** **i + 2]**)
Farooq et al.	with	Automatically close a file after it has been opened	bamfiles = [x.strip() for x in **open(bamfile)**]	**with open(bamfile) as f:** bamfiles = [x.strip() for x in f]

	enumerate	Return a tuple containing a count (from start which defaults to 0) and the values obtained from iterating over iterable.	for **i** in **range(len(text))**:
			w = text[i]
			if w in token2id:
			R[i] = token2id[w]
				for **(i, w)** in **enumerate(text)**:
				if w in token2id:
				R[i] = token2id[w]

	chain-ass-same-value	Assign a value to multiple variables.	**global_draw_name = None** **_test_name = None**	**global_draw_name =** **_test_name =None**
	fstring	Dynamically combine data from variables and other data structures into a readable string output.	log.info('**sample_num_list is %s**' % **repr(self.sample_num_list))**	log.info(f'**sample_num_list is** **repr(self.sample_num_list)**')

Pythonic idioms library

Pythonic idioms are highly valued by developers (Hettinger, 2013; Knupp, 2013; *Programming Idioms* 2022), many studies summarize Pythonic idioms and research their usage (Alexandru et al., 2018; Farooq and Zaytsev, 2021; Merchante and Robles, 2017; Zhang, Xing, Xia, Xu and Zhu, 2022). RIdiom in Chapter 3 identified nine Pythonic idioms by comparing the syntax difference between Python and Java. The nine Pythonic idioms are list/set/dict-comprehension, chain-comparison, truth-test, loop-else, assign-multi-targets, for-multi-targets and star-in-func-call. State-of-the-art of research (Farooq and Zaytsev, 2021) based on a literature review identified a total of 27 detectable idioms, of which five (list/set/dict-comprehension, chain-comparison, truth-test) overlap with those defined by RIdiom. After excluding infrequently used idioms or those with rarely corresponding non-idiomatic Python code [1], four idioms remain: with, enumerate, fstring, and chain-assign-same-value. This culminates in a total of 13 Pythonic idioms. Table 4.1 gives the explanation and code examples of the 13 Pythonic idioms.

Three elements of non-idiomatic code of Pythonic idioms

We construct the knowledge base of non-idiomatic code of Pythonic idioms as triples of <element, relation, element>. It comprises three fundamental elements: ASTscenario, ASTcomponent, and Condition. Each element focuses on a unique aspect: ASTscenario represents usage scenarios for non-idiomatic code linked to a Pythonic idiom. ASTcomponent defines the composition of such code, and Condition outlines necessary conditions for the ASTcomponents. There exist two relationships between these elements: the ASTcomponent relies on ASTscenario, and the ASTcomponent adheres to the specified conditions to be considered as non-idiomatic code of Pythonic idioms. Table 4.2 shows the three elements of non-idiomatic code of thirteen Pythonic idioms. The details are as follows:

ASTscenario: A non-idiomatic code associated with a Pythonic idiom may have restrictions on usage scenarios, corresponding to a distinct Abstract Syntax Tree (AST) node, referred to as ASTscenario. For example, chain-comparison idiom can chain two comparison operations using the "and" operator into one comparison. So the ASTscenario is a BoolOP node whose op is "and" as shown in Table 4.2. For another example, for the list comprehension idiom in Table 4.2, it allows the addition of elements to an object in just one line, as opposed to using a for statement. Since the for statement has no restrictions on the usage scenario, it does not possess an associated ASTscenario.

ASTcomponent: A non-idiomatic code associated with a Pythonic idiom has a deterministic composition, corresponding to few AST nodes, referred to as ASTcomponent. It serves as a pivotal entity in discerning and addressing non-idiomatic code patterns. Taking the chain-comparison idiom in Table 4.2 as an example, its ASTcomponent comprises two Compare nodes within a BoolOP node. These Compare nodes form essential elements of the non-idiomatic code pattern. For another example, for the list comprehension idiom in Table 4.2, which involves appending elements to an object in a for statement, its ASTcomponent is a For node and an Assign node.

Condition: A non-idiomatic code associated with a Pythonic idiom may entail specific conditions, referred to as Condition for its ASTcomponent. It serves as a guiding principle for identification of refactorable ASTcomponent and avoid LLMs from idiomatizing non-refactorable ones that does not meet the specified conditions. For example, for the chain-comparison idiom in Table 4.2, the condition stipulates that the compare operands of the two Compare nodes must intersect. For another example, for the list-comprehension in Table 4.2, its non-idiomatic code is to append elements to a list, so the For node of ASTcomponent should has a "append" function call whose function name is the assigned variable of the Assign node.

Relation: There are two relationships between the three elements. The ASTcomponent is depend on the ASTscenario, the ASTcomponent should satisfy the Condition. For example, for the chain-comparison idiom, its AST component (two Compare nodes) is depend on the ASTscenario (BoolOp node whose op is "and"), and its ASTcomponent satisfies the Condition (Compare operands of the two Compare nodes intersect). These relationships establish a clear framework for identifying non-idiomatic code. We elaborate it in Section 4.3.2.

[1]For example, @staticmethod, assert and etc. are common syntax in programming languages, a few python developers use other syntax alone to achieve the same functionality without these idioms.

Table 4.2: Three Elements of Non-Idiomatic Code of Thirteen Pythonic Idioms

Idiom	ASTscenario	ASTcomponent	Condition
list/set-comprehension	—	A For node An Assign node	1. The For node has "append/add" function call 2. The function name of "append/add" function call is the assigned variable of the Assign node
dict-comprehension	—	A For node An Assign node	1. The For node has an assign statement whose assigned variable is a Subscript node 2. The value of the Subscript node is the assigned variable of the Assign node
chain-comparison	A BoolOP node whose op is "and"	Two Compare nodes	1. Compare operands of the two Compare nodes intersect
truth-test	A test-type node	A Compare node	1. The op of the Compare node is "== "or "!=" 2. The one comparison operand is belong to EmptySet
loop-else	—	A For/While node An If node	1. The For/While node has break statements 2. The If node is the next statement of For node
assign-multi-tar	—	Consecutive Assign nodes	—
for-multi-tar	—	A For node	1. The body of the For node has a Subscript node 2. The value of the Subscript node is the iterated variable of the For node
star-in-func-call	A Call node	Consecutive Subscript nodes	1. The values of Subscript nodes are the same
with	—	A Call node	1. The function name of the Call node is "open"
enumerate	—	A For node	1. The iterated object is not a function call whose function name is "enumerate"
chain-ass-same-value	—	Consecutive Assign nodes	1. The values of consecutive Assign nodes are the same
fstring	—	A BinOP node	1. The op of the BinOp node is "%"

ARI library

The Analytic Rule Interface (ARI) library consists of ARIs to extract three elements in Table 4.2 and auxiliary ARIs. In contrast to directly relying on prompts to instruct LLMs in extracting the three essential elements from a given Python code, we employ LLMs to generate code to implement the required functionality. This approach addresses two key considerations. Firstly, within a project, it may have thousands of lines of code, and the non-idiomatic code is a small part of it. It is expensive for LLMs to handle so many codes and is difficult to make sure the non-idiomatic code is not missing and is correct from unrelated code in a given Python code. Secondly, by generating code through a single invocation of LLMs, we establish a reusable ARI library that can be leveraged consistently across different given Python code. Figure 4.3 shows examples to prompt LLMs to generate code.

Prompt LLMs to generate ARIs to extract three elements: We first create three prompt templates for the three elements: ASTscenario, ASTcomponent and Condition. Then we instantiate the prompt templates with three elements of each Pythonic idiom from Table 4.2. Finally, we instruct the LLM to generate code. Following the retrieval of the generated Python code, authors manually validate its correctness. Once verified, the code is cataloged as a reusable ARI, poised for application in subsequent any Python code [2]. This systematic approach ensures the reliability and reusability of the generated code for element extraction.

The template of ASTscenario is "Write Python method code to extract [ASTscenario] from a Python code", The [ASTscenario] is a placeholder which corresponds to the ASTscenario of a Pythonic idiom in Table 4.2. For example, for the chain-comparison idiom in Figure 4.3, the template is instantiated into "*Write Python method code to extract BoolOP nodes whose op is "and"*." When the prompt is input into the LLM, the LLM responds with an ARI called "`extract_and_boolops(code)`".

The template of ASTcomponent is "Write Python method code to extract [ASTcomponent] from [ASTscenario] / a Python code". The [ASTcomponent] and [ASTscenario] are placeholders which corresponds to the ASTscenario and ASTcomponent of a Pythonic idiom in Table 4.2. For example,

[2]We manually verify that all ARIs are correct

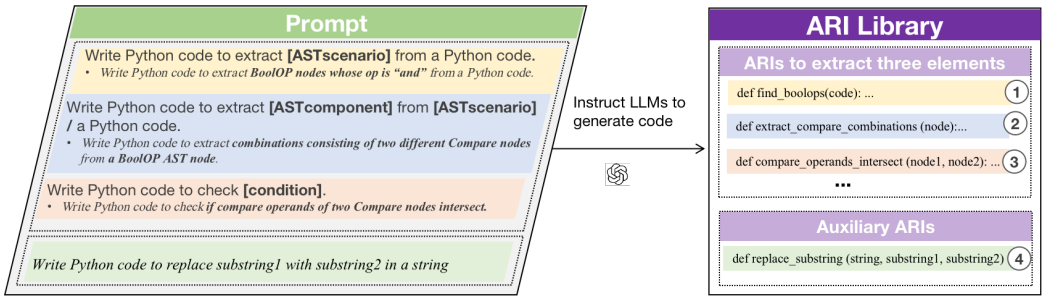


Figure 4.3: ARI library built by prompting LLMs to generate code

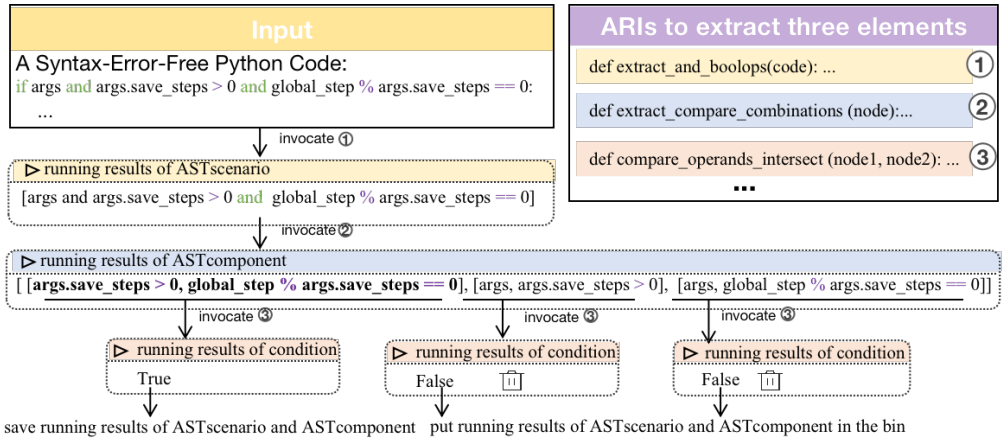


Figure 4.4: Examples of extraction module

for the chain-comparison in Figure 4.3 and Table 4.2, its ASTscenario is present, the template is instantiated into "*Write Python method code to extract combinations consisting of two different Compare nodes from a BoolOP node*". When the prompt is input into the LLM, the LLM responds with an ARI called "extract_compare_combinations(node)". For another example, for the list-comprehension, the ASTscenario is absent as shown in Table 4.2, the the template is instantiated into "*Write Python method code to extract For nodes from a Python code*". When the prompt is input into the LLM, the LLM responds with an ARI called "extract_for_nodes(code)".

The template of Condition is "Write Python method code to check [condition]". The [condition] is a placeholder which corresponds to the Condition of a Pythonic idiom in Table 4.2. For example, for the chain-comparison in Figure 4.3, the template is instantiated into "*Write Python method code to check if compare operands of two Compare nodes intersect*". When the prompt is input into the LLM, the LLM responds with an ARI called "compare_operands_intersect(node1, node2)". **Prompt LLMs to generate auxiliary ARIs:** For the auxiliary ARIs, it is a replace operation utilized in the idiomatization module. Since the replace operation is a simple task, we do not need to instruct LLMs to complete for each given Python code. And invocating the ARI can correctly and effectively replace substring1 with substring2 in a string. Specifically, we input the prompt "*Write Python method code to replace substring1 with substring2 in a string*" into the LLM, the LLM responds with "replace_substring(string, substring1, substring2)" whose body is "return string.replace(substring1, substring2)".

4.3.2 Extraction Module

After we construct the knowledge base of three elements of non-idiomatic code of Pythonic idioms and ARIs to extract three elements. We call ARIs from ARI library in order ASTscenario, ASTcomponent and Condition. To elaborate, if the ASTscenario exists, we first extract ASTscenario from a Python code, and then extract the ASTcomponent from the ASTscenario followed by filtering out components that do not satisfy the condition if condition exists. If the ASTscenario does not exist, we directly extract the ASTcomponent from a Python code followed by filtering out ASTcomponent that does not satisfy the condition if the condition exists. For example, for the chain-comparison of Figure 4.4, we first call "extract_and_boolops" ARI to get all BoolOP nodes from a Python code. Then, for each

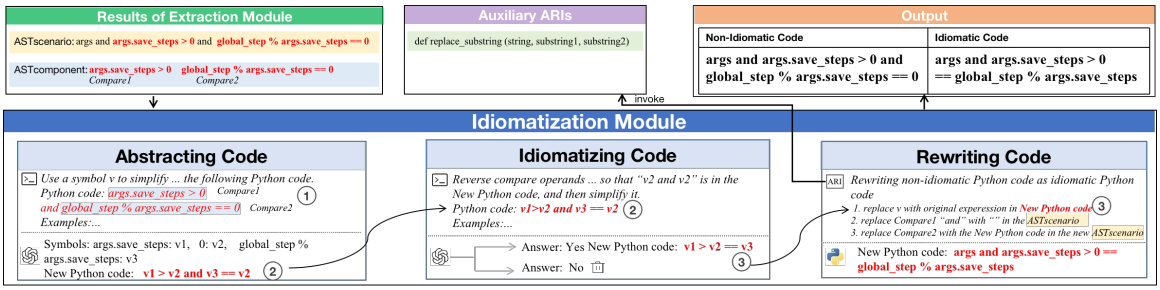


Figure 4.5: Examples of idiomatization module

BoolOP node, we call "extract_compare_combinations" ARI to extract all two different Compare nodes from the BoolOP node. Finally, for each two Compare nodes, we call "compare_operands_intersect" ARI to filter out two Compare nodes without common comparison operands. For another example, for the list-comprehension, its ASTscenario is empty, we directly call "extract_for_nodes(code)" to extract all For nodes from a Python code, and then we call "has_append(node)" filter out For nodes without "append" function call whose function name is the assigned variable of the Assign node.

4.3.3 Idiomatization Module

After extracting the code of ASTscenario and ASTcomponent, we can do the idiomatization task. Since the diversity and complexity of refactorable non-idiomatic code, it is not easy to complete the task by formulating rules. For example, consider code ① Figure 4.1. To refactor the code containing two continue statement with set-comprehension, we need to change "z is x" and "z not in df" into "z is not x" and "z in df", and then we use the "and" to connect the two conditions. Finally, we transform it with set-comprehension. Similarly, in the case of ⑥ of Figure 4.1, to refactor non-idiomatic code with chain-comparison, we need to reverse compare operands and determine whether need to change comparison operators based on the code semantic. Fortunately, LLMs trained on large corpora have rich knowledge and huge potential to complete complex tasks with natural language prompts (Brown et al., 2020; Feng and Chen, 2023; Huang, Zou et al., 2023; OpenAI, 2023; Peng et al., 2023). Therefore, we write prompts to instruct LLMs to transform non-idiomatic code into idiomatic code for Pythonic idioms. To correctly complete the refactoring task, we design three steps including abstracting code, idiomatizing code and rewriting code.

Abstracting Code

The initial step involves the abstraction of the code which corresponds to ASTcomponent, wherein we keep related code snippets with Pythonic idioms while abstract unrelated code snippets with Pythonic idioms. By distilling the code to its core elements, we enhance the clarity and simplicity of the subsequent idiomatization process. To determine the abstract form for each Pythonic idiom, we invite three external workers with more than five years Python programming experience. We first introduce Pythonic idioms to make sure they are familiar with the idioms. Each worker is then asked to independently review the non-idiomatic code dataset of Pythonic idioms from previous researches (Farooq and Zaytsev, 2021; Zhang, Xing, Xia, Xu and Zhu, 2022), and then writes a description of abstracting code and provides three examples of original non-idiomatic code and abstract non-idiomatic code. Then two authors discuss their results and give the final description of abstracting code (<*prompt*>) and three examples of original non-idiomatic code and the corresponding abstract non-idiomatic code for each Pythonic idiom (<*examples*>).

For the abstracting code for each Pythonic idiom, if the prompt has a specified object to abstract, we use ARIs from Auxiliary ARIs in Section 4.3.1 to replace a given string with another given string. Otherwise, we use prompt to instruct LLMs to complete the task. For example, for the star-in-func-call, the extracted non-idiomatic code is "feat.shape[-2], feat.shape[-1]" and the abstracted expression is a specified object ("feat.shape"). Therefore, we invoke "replace_substring" from auxiliary ARIs to replace "feat.shape"with "v", so the abstract code is "v[-2], v[-1]". For another example in the Abstracting Code of Figure 4.5, it does not have a specified object to abstract, We use prompt "*Use a symbol v to simplify each comparison operand within the following Python code. The same comparison operand is represented by the*

same symbol." to abstract represent the code "args.save_steps > 0 and global_step % args.save_steps == 0". The LLM responds with a symbol mapping and the abstract Python code "v1 > v2 and v3 == v2", facilitating subsequent idiomatizing process.

Idiomatizing code

Building upon the abstracted Python code representation, this step focuses on the actual idiomatization of the code. To determine the prompt of idomatizing code for each Pythonic idiom, following the similar process 4.3.3, we invite the same three external works to further independently write a description to idiomatize the abstract code for each Pythonic idiom and provide examples with the abstract code and the corresponding idiomatic code for each Pythonic idiom. And then two authors discuss their results to give the final description of idiomatizing code (<*prompt*>), and the abstract code and the corresponding idiomatic code for each Pythonic idiom (<*examples*>).

For example, for the chain-comparison in the Idiomatizating Code of Figure 4.5, we use prompt "*Reverse compare operands of the first comparison operation, the second comparison, or the first and the second comparison operations so that "v2 and v2" is in the new Python code, and then simplify it*" to idiomatize the abstract Python code: "v1 > v2 and v3 == v2". The LLM responds with Yes and the abstract idiomatic Python code "v1 > v2 == v3". For another example, for the abstract code of chain-comparison "v1 in v2 and v3 in v2", The LLM responds with No because reversing compare operands is invalid for the "in" operator that can change the code semantic.

Rewriting code

Following the acquisition of abstract idiomatic code from the idiomatization process, the final step involves rewriting the non-idiomatic code. It is achieved through the application of the "replace" ARI sourced from the Auxiliary ARIs. The "replace" operation serves a dual purpose: it facilitates the restoration of the abstract idiomatic code and allows for direct code rewriting by replacing the ASTcomponent with the idiomatic code in the ASTscenario, if ASTscenario exists. Therefore, it may invoke "replace" several times. To determine the process of invoking "replace", following the similiar process in Section 4.3.3, we invite the same three external workers to further independently summarize steps to use "replace" to complete rewriting non-idiomatic code into idiomatic code. And then two authors discuss their results to give the final steps of rewriting code for each Pythonic idiom.

For example, for the Rewriting Code process illustrated in Figure 4.5, we first replace abstract symbols with their corresponding original expressions within the abstract idiomatic code: "v1 > v2 ==v3", yielding the genuine idiomatic code. Subsequently, we replace the Compare1 and 'and' with an empty string in the ASTscenario code, yielding the new ASTscenario code. Finally, we replace the Compare2 with the genuine idiomatic code in the new ASTscenario code, yielding a final idiomatic code: "args and args.save_steps > 0 == global_step % args.save_steps". The rewriting step ensures the precise transformation of non-idiomatic code into its idiomatic counterpart.

4.4 Evaluation

To evaluate our approach, we study two research questions:

RQ1 (Effectiveness): What is the effectiveness of our approach in refactoring non-idiomatic Python code into idiomatic Python code with nine Pythonic idioms?

RQ2 (Scalability): Can our approach be effectively extended to new Pythonic idioms?

4.4.1 RQ1: Effectiveness of Refactoring Non-Idiomatic Python Code with Nine Pythonic Idioms

Motivation

RIdiom in Chapter 3 can automatically refactor non-idiomatic code into idiomatic code with nine Pythonic idioms by formulating detection and refactoring rules, but it causes huge human investment in formulating rules. The current success of ChatGPT (*Introducing ChatGPT* 2023) demonstrates

Table 4.3: Benchmark of Nine Pythonic Idioms

Idiom	Method	Code Pair
list-comprehension	389	512
set-comprehension	305	361
dict-comprehension	370	448
chain-comparison	391	574
truth-test	394	610
loop-else	319	360
assign-multi-targets	395	729
for-multi-targets	370	463
star-in-func-call	381	621
Total	3311	4678

remarkable ability of LLMs to comprehend human prompts and complete the corresponding tasks. Therefore, we are interested in understanding the performance of our approach based on LLMs.

Approach

To clarify the effectiveness of our approach, we perform effectiveness comparison by calculating metrics on a dataset with our approach and the state-of-the-art baselines.

DataSet. To evaluate the effectiveness of our approach, it is fundamental to have a correct and complete benchmark of code refactorings consisting of code pairs <non-idiomatic Python code, idiomatic Python code>. Manually constructing a complete and correct benchmark is unrealistic because code refactoring involves a lot of time and manpower, and inevitably comes with personal bias. Recently, RIdiom in Chapter 3 can automatically refactor non-idiomatic Python code with nine Pythonic idioms, which provides a good starting point. It provides code pairs within crawled methods for each Pythonic idiom. Considering the effort of manual verification, we randomly sample methods with a confidence level of 95% and a confidence interval of 5 from RIdiom for each Pythonic idiom. Then, to ensure the completeness of the benchmark, we run RIdiom in Chapter 3, our approach and Prompt-LLM on the sample methods for nine Pythonic idioms to collect code pairs <non-idiomatic Python code, idiomatic Python code>. To validate the correctness, we invite 18 external workers with more than five years Python programming experience. We divide them into 9 groups, and each group of two workers independently checks the correctness of the code pairs for an idiom. The Cohen's Kappa values (Viera, Garrett et al., 2005) of nine groups for their annotation results all exceeded 0.75 (substantial agreement). Finally, the two authors and external workers discuss and resolve the inconsistencies and ensure the correctness of the benchmark for nine Pythonic idioms. Table 4.3 shows the benchmark of nine Pythonic idioms. The *Method* column represents the number of sampled methods for each Pythonic idiom, and the *Code Pair* column represents the number of code pairs <non-idiomatic code, idiomatic code> for each Pythonic idiom. Since sampled methods for each Pythonic idiom are from methods of RIdiom, it is reasonable to expect that the number of code pairs is greater than the number of the corresponding methods for each Pythonic idiom.

Baselines. We compare our approach with two baselines. The first baseline is RIdiom that is an approach based on rules proposed by Zhang et al. (Zhang, Xing, Xia, Xu and Zhu, 2022). It detects and refactors non-idiomatic Python code into the corresponding idiomatic Python code with nine Pythonic idioms by manually formulating detection rules and refactoring steps. The second (Prompt-LLM) is to directly call the LLM to find code pairs for any given method code for each Pythonic idiom to illustrate the capability of LLM and the strengths of our proposed approach using LLM. For fairness of comparison, similar to process in Section 4.3.3, we invite three external works to independently write a prompt and provide three examples of a Python code and the corresponding code pairs. Then two authors discuss their results and give the final prompt and three examples for each Pythonic

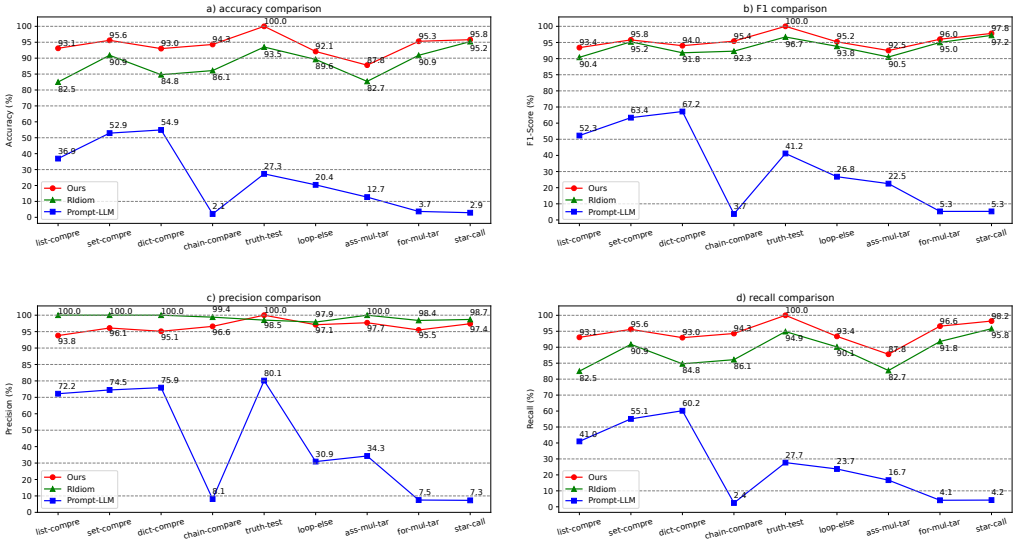


Figure 4.6: Scatter plot with straight lines of accuracy, F1-score, precision and recall of three approaches for nine Pythonic idioms

idiom. Examples are shown in Prompt-LLM column of Figure 4.1. Since the success of ChatGPT, the LLM our approach and baselines use are state-of-the-art of GPT-3.5-turbo (*GPT* 2023). And we set temperature to 0 to make the outputs mostly deterministic.

Metrics. By referring metrics using by previous researches on code refactorings (Dilhara, Dig et al., 2023; Silva and Valente, 2017; Tsantalis, Mansouri et al., 2018; Zhang, Xing, Xia, Xu and Zhu, 2022), we use four metrics accuracy, F1-score, precision and recall. To calculate the four metrics, we need to define true positives, false positives and false negatives which are represented as TP, FP and FN, respectively. We define a TP as a code pair detected by an approach is in the benchmark. We define a FP as a code pair detected by an approach is not in the benchmark. We define a FN as a code pair of the benchmark is not determined by an approach. The accuracy and F1-score represents the overall performance. We calculate accuracy, F1-score, precision and recall as follows:

$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 * P * R}{P + R} \quad Accuracy = \frac{TP}{TP + FP + FN}$$

Result

Figure 5.6 presents the accuracy, F1-score, precision and recall of our approach, RIdiom, Prompt-LLM for nine Pythonic idioms.

Comparison with RIdiom. In comparison to RIdiom, our approach consistently outperforms in three metrics: accuracy, F1-score, and recall across all Pythonic idioms. Notably, our approach exhibits a distinct advantage in both recall and accuracy over RIdiom. For eight Pythonic idioms, the accuracy and the recall exceed 90%. While our approach registers slightly below 90% in both recall and accuracy for the assign-multi-targets idiom, it is obviously close to 90% (87.8%). In contrast, RIdiom falls short of 90% in accuracy for five Pythonic idioms (list-comprehension, dict-comprehension, chain-comparison, loop-else, and assign-multi-targets), and four of these idioms (list-comprehension, dict-comprehension, chain-comparison, and assign-multi-targets) also have recall results below 90%. For the list-comprehension, the disparity in recall and accuracy between our approach and RIdiom exceeds 10%. For the other four idioms (dict-comprehension, chain-comparison, truth-test, and assign-multi-targets), these differences surpass 5%. For the remaining four idioms (set-comprehension, loop-else, for-multi-targets, and star-in-func-call), while the differences are relatively smaller, our approach maintains a consistent edge over RIdiom.

For the precision, although RIdiom exhibits a slight advantage in precision (with differences ranging from 0.8% to 6.2%) for eight of the idioms, our approach consistently achieves over 93.8% precision

for each idiom. Moreover, our approach surpasses RIdiom for the truth-test idiom. It is important to note that F1-score strikes a balance between precision and recall, and in this regard, our approach consistently outperforms RIdiom across all Pythonic idioms. The comprehensive analysis underscores the notable advantages of our approach over RIdiom for the task of refactoring non-idiomatic code with Pythonic idioms, and can make up for the shortcomings of RIdiom in recall.

Comparison with Prompt-LLM. Compared to our approach, Prompt-LLM consistently exhibits the lowest performance across four metrics for each Pythonic idiom. The disparities between our approach and Prompt-LLM in terms of accuracy, F1-score, precision, and recall for the nine Pythonic idioms are substantial, ranging from 38.1% to 92.9%, 26.8% to 92.5%, 19.9% to 90.1%, and 32.8% to 94%, respectively.

Compared to Prompt-LLM, our approach maintains stable and commendable performance across all idioms, with each metric surpassing the 90% threshold for eight of them. Even in the case of assign-multi-targets, where both accuracy and recall fall slightly below 90%, they are still notably close at 90% (87.8%). In contrast, Prompt-LLM fails to achieve a metric score of 90% for any of the idioms.

Prompt-LLM demonstrates poor performance and exhibits significant variability across different Pythonic idioms. For list/set/dict-comprehension, Prompt-LLM displays relatively better performance across all metrics, exceeding 35%. This underscores LLMs' enhanced proficiency in handling the three idioms. For truth-test, Prompt-LLM exhibits superior precision (80.1%) compared to recall (27.7%), indicating a higher likelihood of missing refactorable non-idiomatic code instances associated with the truth-test idiom. For loop-else and assign-multi-targets, Prompt-LLM's performance remains limited across all metrics, falling below 35%. Furthermore, for the remaining three idioms (chain-comparison, for-multi-targets, and star-in-func-call), Prompt-LLM's performance is markedly poorer, registering below 10% on all metrics. It suggests that Prompt-LLM struggles to effectively detect and refactor non-idiomatic code associated with the three idioms. Therefore, our approach significantly enhances the capabilities of LLMs, consistently demonstrating stable and commendable performance across all metrics.

Failure analysis of our approach. For the code pairs in the benchmarks that our approach does not find or wrongly refactor non-idiomatic Python code with nine Pythonic idioms, we summarize two reasons as follows:

(1) LLMs may produce suboptimal results when refactoring is too complicated. While LLMs have achieved success, it is reasonable that LLMs cannot handle all situations. For example, for the first example of Figure 4.7, the non-idiomatic code appends two different elements to the list "possible_mistakes" in each iteration of the for statement. Our approach finally gives the idiomatic code by concatenating two lists which independently appending two elements. The idiomatic code is wrong because the order of elements is different from the non-idiomatic code. Although the non-idiomatic code can be refactored with list-comprehension, the idiomatic code needs other statements to adjust the order of elements. For another example, for the non-idiomatic code "gnn_layer(..., n_points[idx1], n_points[idx2])", it is non-refactorable code with star-in-func-call because the "idx1, idx2" is not an arithmetic sequence. However, our approach mistakenly assumes the subscript sequence of "_points[idx1], n_points[idx2]" is an arithmetic sequence and refactor it into "gnn_layer(..., *n_points[idx1:idx2 + 1])" with star-in-func-call.

(2) LLMs may refrain from refactoring when benefits in idiomatic code appear limited: While LLM demonstrates proficiency in refactoring Python code using specific Pythonic idioms, there are instances where it abstains from doing so. For example, for the second example of Figure 4.7, the non-idiomatic code "for u in urls_results: urls.add(u)" actually can be refactored with set-comprehension. However, LLM responds with *"The given code is already simple and concise. Using set comprehension here would not make the code more readable or efficient. "*. LLM refuses to refactor it with set-comprehension because it may determine that applying set-comprehension would not notably enhance code conciseness. For another example, for the non-idiomatic code "slice2[axis] = slice(None, -1); slice1 = tuple(slice1)", LLM responds with *"The given code cannot be refactored with one assign statement as the variables being assigned are not of the same type"*. The LLMs refuses to refactor because it thinks "slice2[axis]" and "slice1" are not the same type.

Idiom	Non-idiomatic Code	Idiomatic Code	Failure Reason of Our Approach
list-comprehension	for i in range(len(unknowns) + 1): possible_mistakes.append(...) possible_mistakes.append(...)	possible_mistakes = [... for i in range(...)] + [... for i in range(...)]	Our approach gives the wrong idiomatic code of list-comprehension because it changes the order of appended elements
set-comprehenison	for u in urls_result: urls.add(u)	The given code is already simple and concise. Using set comprehension here would not make the code more readable or efficient.	Our approach abstains from refactoring with set-comprehension because LLM determines it would not enhance code conciseness

Figure 4.7: The examples that our approach wrongly refactors or refrains from refactoring

Table 4.4: Benchmark of Four New Pythonic Idioms

Idiom	Methods	Code Pairs
with	600	64
enumerate	600	565
chain-assign-same-value	600	156
fstring	600	223
Total	600	1008

4.4.2 RQ2: Scalability of Our Approach

Motivation

Although RIdiom can achieve good result on nine Pythonic idiom, it cannot handle new Pythonic idioms and is difficult to extend to new Pythonic idioms because RIdiom needs manually formulate extracting and refactoring rules. So we are interested in whether our approach can be effectively extended to new Pythonic idioms that RIdiom cannot handle.

Approach

Following the approach in Section 4.4.1, we present baselines, dataset and metrics. Baselines and metrics are the same as metrics and baselines in Section 4.4.1. Particularly, since RIdiom cannot handle the new four Pythonic idioms, the baseline is excluded. The dataset is detailed as follows:
DataSet. As Section 4.3.1 illustrates, there are four new Pythonic idioms (with, enumerate, fstring and chain-assign-same-value) that RIdiom in Chapter 3 cannot handle. Considering that we need a certain number of code pairs (<non-idiomatic Python code, idiomatic Python code>) to evaluate our approach, but manually verifying code pairs is time consuming and the current GPT-3.5 is not free, we randomly sample 600 Python methods from crawled methods by RIdiom that is a tool to refactor non-idiomatic code into idiomatic code with nine Pythonic idioms. Following the similar method in Section 4.4.1, to ensure the completeness of dataset as much as possible, we run our approach and Prompt-LLM on the sampled Python methods to collect code pairs <non-idiomatic Python code, idiomatic Python code>. To validate the correctness, we invite 8 external workers with more than five years Python programming experience. We divide them into four groups, and each group of two workers independently checks the correctness of the code pairs for an idiom. The Cohen's Kappa values (Viera, Garrett et al., 2005) of four groups for their annotation results all exceeded 0.75 (substantical agreement). Finally, the two authors and external workers discuss and resolve the inconsistencies and ensure the correctness of the benchmark for the new four Pythonic idioms. Table 4.4 shows the benchmark of new four Pythonic idioms. The *Method* column represents the number of sampled methods for each Pythonic idiom, and the *Code Pair* column represents the number of code pairs <non-idiomatic code, idiomatic code> for each Pythonic idiom. The number of code pairs is less than the number of methods is reasonable because not each method may contain non-idiomatic code. For example, in the case of the with idiom, its non-idiomatic code should include file opening operations. However, not all methods necessarily involve file opening. Given that the with idiom is widely adopted by Python users (Sakulniwat et al., 2019), the count is relatively low (64). Conversely, for the enumerate idiom, its non-idiomatic code typically corresponds to a for statement, which is more prevalent in code. As a result, the count is relatively higher (565).

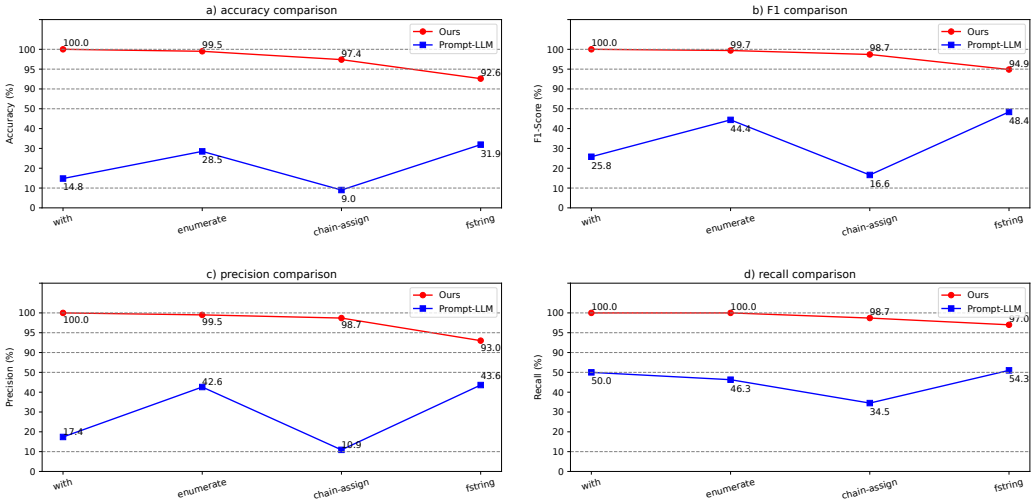


Figure 4.8: Scatter plot with straight lines of accuracy, F1-score, precision and recall of two approaches for four new Pythonic idioms

Result

Figure 4.8 presents the accuracy, F1-score, precision and recall of our approach, and Prompt-LLM for new four Pythonic idioms. For each Pythonic idioms, the metrics all are above 90%. And the accuracy, F1-score, precision and recall are above 95% for three idioms (with, enumerate and chain-assign-same-value). Compared to our approach, Prompt-LLM consistently exhibits poor performance across four metrics for each Pythonic idiom. The disparities between our approach and Prompt-LLM in terms of accuracy, F1-score, precision, and recall for the four Pythonic idioms are substantial, ranging from 60.7% to 88.4%, 46.5% to 82.1%, 49.4% to 87.8%, and 42.7% to 64.2%, respectively.

4.5 Discussion

4.5.1 Implications

Our hybrid approach in Section 4.3 demonstrates excellent performance in Section 4.4, we now delve into the scalability of our approach and future work for researchers. Currently, our approach only supports syntactically correct Python code, as it requires parsing the code into an Abstract Syntax Tree (AST). However, handling Python code with syntax errors on the hybrid framework is feasible. One solution is to incorporate a syntax-fixing module to rectify syntax errors in the Python code before it input into extraction module in Section 4.3.2. Another approach is to introduce an alternative method within the extraction module in Section 4.3.2. Specifically, prompting LLMs to extract three elements or prompting LLMs to generate AST for Python code instead of invoking APIs, which can be effective when the syntax error cannot be fixed.

Prior studies (Alexandru et al., 2018; Farooq and Zaytsev, 2021; Zhang, Xing, Xia, Xu and Zhu, 2022; Zhang, Xing, Xia, Xu, Zhu and Lu, 2023) have shown that employing Pythonic idioms may yield benefits, such as code conciseness and improved performance, but it can also have drawbacks, including potential impacts on code readability and performance degradation. Our current focus is solely on refactoring non-idiomatic code with Pythonic idioms. The results in Section 4.4 and Figure 4.7 highlight that LLMs may abstain from refactoring when benefits from idiomatic code are limited. This observation prompts researchers to consider using LLMs combined with knowledge base to generate comments explaining the positive and negative effects of refactoring non-idiomatic code with Pythonic idioms. This can enhance Python users' understanding and effective utilization of Pythonic idioms.

Furthermore, our approach combines ARIs and prompts based on LLMs to refactor non-idiomatic code with Pythonic idioms. While current Language Models (e.g., ChatGPT (OpenAI, 2023)) are not freely available, there has been a recent emergence of free alternatives (e.g., Llama 2 (Touvron et al., 2023)). Over time, it is plausible that more LLMs may become more accessible, potentially even free of

charge. This accessibility could assist developers in effectively refactoring code using Pythonic idioms to alleviate the limitation of rule-based approach.

4.5.2 Threats to Validity

Internal Validity: The one internal threat is inaccuracy when evaluating the correctness of benchmark in Section 4.4. For each Pythonic idiom, we invite two external workers with more than five Python programming experience. And two authors and external workes discuss to resolve the inconsistencies to ensure the correctness of the benchmark.

External Validity: One external threat is that our approach is limited to thirteen Pythonic idioms. RIdiom (Zhang, Xing, Xu et al., 2023b) can only handle nine Pythonic idioms, our approach contains four new Pythonic idioms which validate the scalability of our approach. When there are more Pythonic idioms, developers can determine the three elements of Pythonic idioms to extend to more Pythonic idioms. In the future, we will automatically mine undetected Pythonic idioms and automatically determine three elements of Pythonic idioms. The other external threat is the representative of the benchmark selected to evaluate our approach. To mitigate this threat, our experimented methods of benchmark are from previous research (Zhang, Xing, Xu et al., 2023b), representing an unbiased benchmark for our research.

4.6 Conclusion and Future Work

Refactoring non-idiomatic code with Pythonic idioms is not easy because of three challenges including code miss, wrong detection and wrong refactoring. Depending solely on Large Language Models (LLMs) or a rule-based approach (RIdiom) has its limitations in addressing these challenges. To alleviate the challenges, we propose a hybrid approach based on LLMs to refactor non-idiomatic code with Pythonic idioms. In detail, we first extract three elements of a Pythonic idiom and create an API library by prompting LLMs to generate code. We then invoke the APIs to extract non-idiomatic code in the extraction module. Finally, we prompt LLMs to abstract code, idiomatize the abstract code and then invoke APIs to rewrite non-idiomatic code into idiomatic code. The results of our experiments, conducted on nine Pythonic idioms from RIdiom in Chapter 3, as well as four new Pythonic idioms (Farooq and Zaytsev, 2021) not covered by RIdiom, demonstrate high levels of accuracy, F1-score, precision, and recall. This substantiates the effectiveness and scalability of our proposed approach. In the future, we will keep improving our approach and extend our approach to accommodate Python code with syntax errors. Besides, we plan to offer explanations regarding the impacts, such as enhanced readability and performance, that result from refactoring code into idiomatic Python.

Chapter 5

Hard to Read and Understand Pythonic Idioms? DeIdiom and Explain Them in Non-Idiomatic Equivalent Code

Z. Zhang, Z. Xing, D. Zhao et al. (2024). 'Hard to Read and Understand Pythonic Idioms? DeIdiom and Explain Them in Non-Idiomatic Equivalent Code'. In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ICSE 2024. , Lisbon, Portugal, Association for Computing Machinery. ISBN: 979-8-4007-0217-4/24/04. DOI: 10.1145/3597503.3639101

The Python community strives to design pythonic idioms so that Python users can achieve their intent in a more concise and efficient way. According to our analysis of 154 questions about challenges of understanding pythonic idioms on Stack Overflow, we find that Python users face various challenges in comprehending pythonic idioms. And the usage of pythonic idioms in 7,577 GitHub projects reveals the prevalence of pythonic idioms. By using a statistical sampling method, we find pythonic idioms result in not only lexical conciseness but also the creation of variables and functions, which indicates it is not straightforward to map back to non-idiomatic code. And usage of pythonic idioms may even cause potential negative effects such as code redundancy, bugs and performance degradation. To alleviate such readability issues and negative effects, we develop a transforming tool, DeIdiom, to automatically transform idiomatic code into equivalent non-idiomatic code. We test and review over 7,572 idiomatic code instances of nine pythonic idioms (list/set/dict-comprehension, chain-comparison, truth-value-test, loop-else, assign-multi-targets, for-multi-targets, star), the result shows the high accuracy of DeIdiom. Our user study with 20 participants demonstrates that explanatory non-idiomatic code generated by DeIdiom is useful for Python users to understand pythonic idioms correctly and efficiently, and leads to a more positive appreciation of pythonic idioms.

5.1 Introduction

Pythonic idioms are highly valued by the Python community (developers, 2000). Python programming books and renowned Python developers emphasize the benefits of Pythonic idioms, such as versatility, conciseness, and improved performance (Bader, 2017; Hettinger, 2013; Knupp, 2013; Martelli et al., 2005; Reitz and Schlusser, 2016; Slatkin, 2020). These idioms embody the idiomatic way of writing Python code that is both expressive and efficient, exemplifying the philosophy of the language. Additionally, some research efforts have focused on detecting non-idiomatic code and recommending the corresponding Pythonic idioms (Phan-udom et al., 2020; *Pylint* 2022; Zhang, Xing, Xia, Xu and Zhu, 2022). For instance, Zhang, Xing, Xia, Xu and Zhu (2022); Zhang, Xing, Xu et al. (2023b) developed a refactoring tool (*RIdiom* 2022) to identify and convert non-idiomatic code to its idiomatic form for nine specific Pythonic idioms. Furthermore, they found that Python developers are highly concerned about the performance impacts of these idioms, which can vary significantly (Zhang, Xing, Xia, Xu, Zhu and Lu, 2023).

Despite the clear advantages of Pythonic idioms, there is a lack of research aimed at effectively explaining these idioms to enhance correct and in-depth comprehension. Program comprehension plays a crucial role in software maintenance and demands a significant amount of time and effort (Corbi,

1989; Xia et al., 2018; Zelkowitz et al., 1979). It forms the foundation for essential software engineering tasks such as bug fixing, code enhancement, and reuse (Xia et al., 2018). This paper focuses on explaining nine Pythonic idioms (list/set/dictionary comprehension, chain comparison, truth-value test, loop-else, assignment to multiple targets, the star operator, and for loops with multiple targets) identified by Zhang, Xing, Xia, Xu and Zhu (2022), which are unique to Python and not present in Java. Their uniqueness can pose challenges for developers in comprehending them accurately and effectively.

By systematically analyzing questions about these nine idioms on Stack Overflow (*Stack Overflow* 2022) (see Section 5.2.1), we observed that Python users often exhibit unfamiliarity or misunderstandings about the syntax and behavior of these idioms. For instance, a developer wrote the code "`table_name in row is False`" using chain comparison. The intent was for the code to be equivalent to "`(table_name in row) is False`", but the correct non-idiomatic equivalent is "`table_name in row and row is False`". This misunderstanding led to a defect *Why does (1 in [1,0] == True) evaluate to False?* (2012). Similarly, for the loop-else idiom, developers sometimes use the else clause inappropriately after a `for` statement without a `break` statement, resulting in code redundancy *Pylint* (2022). Our investigations reveal that 17.8% of loop-else usages are superfluous (see Table 5.3).

Moreover, the usage of Pythonic idioms is frequent in GitHub repositories (see Section 5.2.2). By analyzing their usage via a statistical sampling method Scheaffer et al. (2011), we noted that these idioms, despite their conciseness, may also introduce negative effects such as decreased performance, bugs, code redundancy, and readability issues (see Section 5.2.4). These findings underscore the necessity of a tool to enhance developers' understanding of Pythonic idioms, their conciseness, and potential pitfalls. Instead of relying on natural language explanations, which can introduce ambiguity, we opt to use corresponding non-idiomatic code as explanations to ensure semantic accuracy. Given that these nine idioms involve syntax unique to Python, we use common Python syntax rather than alternative APIs to transform idiomatic into non-idiomatic code, ensuring that developers can understand the explanations even with limited Python experience.

To transform the idiomatic code for the nine Pythonic idioms into their corresponding non-idiomatic code, our approach follows a two-step process: detection and rewriting. Each Pythonic idiom corresponds to a unique AST (Abstract Syntax Tree) node with distinct syntactic properties. We first detect idiomatic code by examining these AST nodes and their syntactic features. For the rewriting step, we consider the need to introduce additional statements, new variables, and function declarations based on the patterns of these idioms in Section 5.2.3. Transforming rules are designed based on four atomic operations (create, insert, replace, and remove) to automatically rewrite idiomatic code into its non-idiomatic counterpart.

Our approach is then applied to detect and transform 1,708,831 idiomatic code instances of the nine Pythonic idioms across 7,577 GitHub repositories. To ensure accuracy, we verify the detection and transformation results for 7,572 idiomatic code instances using a combination of test cases and manual code review, similar to the methodology employed by Zhang et al. (Zhang, Xing, Xia, Xu and Zhu, 2022). Our approach achieves 100% detection accuracy and 99%-100% rewriting accuracy for the nine Pythonic idioms.

To further evaluate whether our tool can effectively aid Python users in understanding these idioms, we conducted a user study with 20 participants. The study involved 27 multiple-choice questions derived from 27 randomly selected idiomatic code instances from GitHub repositories. The experimental group, provided with non-idiomatic code explanations generated by our tool, demonstrated a 63.5% improvement in correctness over the control group, which only received the idiomatic code. Additionally, the experimental group completed tasks 22.6% faster. We find that the explanatory non-idiomatic code given by our tool helps build confidence in Pythonic idiom usage and promotes Python users' appreciation of the community effort in designing and developing Pythonic idioms.

In summary, the contributions of this work are as follows:

- To the best of our knowledge, this study is the first to systematically investigate and empirically evaluate the comprehensibility of Pythonic idioms.
- We have developed DeIdiom, the first tool that detects and transforms idiomatic code into non-idiomatic code for nine distinct Pythonic idioms. We also provide a web application for demonstration and practical use. The application can be accessed at the website.

- Our evaluation confirms the high accuracy of DeIdiom, achieving 100% detection accuracy and 99%-100% rewriting accuracy. User studies demonstrate that DeIdiom significantly enhances the understanding and correct usage of Pythonic idioms, improving comprehension accuracy by 63.5% and reducing task completion time by 22.6%.
- We provide actionable suggestions for Python developers to effectively use DeIdiom and for researchers to advance the understanding of Pythonic idioms. These suggestions include leveraging DeIdiom to minimize potential negative impacts such as performance issues, bugs, and readability challenges inherent in idiomatic usage. Our findings offer a foundation for further research on improving the comprehensibility and utility of idiomatic expressions in programming languages.

5.2 Empirical Study

We conduct a systematic empirical study to answer the four research questions about Pythonic idiom usage and challenges:

RQ1: What challenges do Pythonic idioms present to Python users in terms of understanding?

RQ2: How are Pythonic idioms used in real projects?

RQ3: How are conciseness of Pythonic idioms manifested?

RQ4: What are the potential negative effects of using Pythonic idioms?

5.2.1 RQ1: Challenges in Understanding Pythonic idioms

Motivation

Pythonic idioms exhibit unique syntax and semantics which are not commonly seen in programming languages. First, we want to explore what problems Python users often encounter with understanding Pythonic idioms.

Approach

Zhang, Xing, Xia, Xu and Zhu (2022) recently identified nine Pythonic idioms (list/set/dict comprehension, chain-comparison, truth-value-test, loop-else, assign-multi-targets, for-multi-targets, and star) that exhibit unique programming constructs exclusive to Python, distinguishing them from Java. We assume the uniqueness of the nine idioms making it challenging for developers to comprehend, so we focus on these 9 idioms. To understand the problems of reading and understanding these nine Pythonic idioms, we examine idiom-related questions on Stack Overflow. For each Pythonic idiom, we use the Pythonic idiom name as the keyword to search the python-tagged questions. We first collect the top-30 questions returned for each Pythonic idiom (i.e., 270 questions). Next, two authors independently read the description of question to label whether the questions returned for each Pythonic idiom are relevant to the challenges of reading and understanding the concerned idiom. The Cohen's kappa (Viera, Garrett et al., 2005) reaches 0.89 which indicates substantial agreement between the labelers. For the disagreements they discuss to reach the consensus. As result, there are 154 questions for analysis.

To get the challenge category of understanding Pythonic idioms, we first randomly sample 111 questions with a confidence level of 95% and an error margin 5% and then two authors separately annotate issues developers encounter Pythonic idioms with a short description. Then they discuss and resolve all disagreements if their descriptions do not have the same meaning. Next, they work together to group all annotations into a challenge category with corresponding explanation. Finally, the remaining collected questions were independently annotated with challenge categories. If the remaining questions did not fit existing categories, they were annotated with new challenge descriptions. It was found that no new categories were needed. The Cohen's kappa agreement between two labels is 0.78 (substantial agreement). Then, two authors discuss the disagreements to reach an agreement.

Table 5.1: Challenges in understanding Pythonic idioms

#R	Examples
(1)	**Reason explanation:** Ignorance of existence of star idiom **Q:** Is there a way to expand a Python tuple into a function as actual parameters? **Asked** 13 years ago; **Modified** 1 year ago; **Viewed** 314k times --- **Reason explanation:** Confusion about nested for statement of list-comprehension idiom **Q:** Explanation of how nested list comprehension works? I have no problem understanding this: `b = [x for x in a]` ..., but then I found this snippet: `b = [x for xs in a for x in xs]`. The problem is I'm having trouble understanding the syntax in `[x for xs in a for x in xs]`, could anyone explain how it works? **Asked** 9 years ago; **Modified** 10 months ago; **Viewed** 28k times
(2)	**Reason explanation:** Incorrect understanding of meaning of assign-multi-targets idiom **Q:** How do chained assignments work? A quote from something: `x = y = somefunction()` is the same as `y = somefunction(); x = y;` Is `x = y = somefunction()` the same as `x = somefunction(); y = somefunction()`? Based on my understanding, they should be same. **Asked** 11 years ago; **Modified** 9 months ago; **Viewed** 20k times --- **Reason explanation:** Misattribution of unexpected behavior to the use of set-comprehension idiom **Q:** Set comprehension gives "unhashable type" (set of list) in Python? I want to collect all second elements of each tuple into a set: `my_set.add({tup[1] for tup in list_of_tuples})` But it throws the following error: TypeError: unhashable type: 'set' **Asked** 5 years ago; **Modified** 5 years ago; **Viewed** 8k times

Result

We summarize two types of challenges in understanding the Pythonic idioms. Table 5.1 presents illustrative examples. (1) or (2) in #R identifies the challenge. We excerpt and highlight relevant content in red. Among 154 questions, the corresponding numbers of list/set/dict-comprehension, chain-comparison, truth-value-test, loop-else, ass-multi-targets, for-multiple-targets and star are 25, 15, 13, 23, 14, 17, 12, 15 and 20, respectively. The two challenges have a progressive relationship. For example, when developers misunderstand the semantics of Pythonic idioms, they also are unfamiliar with the Pythonic idioms. If one question of Stack Overflow involves the (2) challenge, we do not consider it as (1) challenge. The details are as follows: **(1) Python users are often unfamiliar with the unusual syntax of Pythonic idioms.** We find Python users may not know the existence of certain Pythonic idioms, or they do not understand what the idioms mean although they know certain idioms are available, which may lead to limitations in interpreting and utilizing these idioms effectively. It occurs in all nine Pythonic idioms and accounts for 63.0% (97 of 154 questions). For the first example of the star idiom in Table 5.1, a developer did not know the star idiom is a way to expand a Python tuple into a function as actual parameters. Although it has been asked 13 years ago, it was still active within 1 year and was viewed more than 314,000 times which indicates many Python users may encounter similar problems[1]. For the second example of the list-comprehension in Table 6.1, although the Python user understands the list-comprehension syntax with one for keyword, he/she did not understand the meaning of the list-comprehension syntax with two for keywords. Hence, he/she asked if anyone can explain how it works.

[1]Among 1,944,314 python tagged questions on Stack Overflow, only 10% of questions have $>$ 3547 views.

(2) Python users often misunderstand the subtle semantics of Pythonic idioms. We find Python users can misunderstand the meaning of the idiom, or they wrongly think that the use of Pythonic idioms causes unexpected behaviors, which may lead to unintended or unexpected outcomes. It is applicable to all nine Pythonic idioms and accounts for 37.0% (57 of 154 questions). For the third example of assign-multi-targets in Table 5.1, a developer has two misunderstandings. The one is that he/she assumes that `x = y = somefunction()` is equal to `y = somefunction(); x=y`. This is not true because the `x` is the first assigned, which could cause unexpected behavior if `x` and `y` have data dependency. The another is that he/she thinks that `x = y = somefunction()` is equal to `x = somefunction(); y= somefunction()`. Actually this idiomatic code is equal to `tmp = somefunction(); x = tmp; y=tmp`. The two versions of non-idiomatic code are different. If `somefunction()` is mutable, the first version assigns `x` and `y` different values, but the second version assigns the same value to `x` and `y`. For the last example of set-comprehension in Table 6.1, a developer uses a `my_set.add()` API to add a set to `my_set` of set type, which makes the code throw the unhashable type error because items of a set in Python are immutable so a set cannot be added to `my_set` as an item. Since the user uses set-comprehension `{tup[1] for tup in list_of_tuples}` to add a set to `my_set.add()` API, he/she mistakenly thinks that the use of set-comprehension leads to the error.

5.2.2 RQ2: Pythonic idiom Usage in the Wild

Motivation

Exploring coding practices with Pythonic idioms in real projects helps us know the importance of understanding of Pythonic idioms.

Approach

To analyze coding practices with Pythonic idioms, we crawl the top 10,000 repositories using Python programming language by the number of stars from GitHub. 7,577 repositories can be successfully parsed using Python 3. For the star idiom, Zhang, Xing, Xia, Xu and Zhu (2022) is limited to the use of star in the function call AST node, we extend it to all AST nodes. We design detection rules (shown in Table 5.4) to identify the idiomatic code instances for the nine idioms.

Results

Table 5.2 shows the statistics of repositories, files and idiomatic code instances for the nine Pythonic idioms. The Total shows the total number of repositories, files and idiomatic code instances for nine Pythonic idioms. Of the 7,577 collected repositories, 6,997 repositories, 222,637 files and 1,708,831 idiomatic code instances use at least one Pythonic idiom. The percentages of repositories for nine Pythonic idioms are 13.7%-80.9%. For each of the nine Pythonic idioms, there are, on average, 3-15 files in a repository containing 4-161 idiomatic code instances. The frequent usage of Pythonic idioms shows their prevalence. Such widespread utilization of Pythonic idioms highlights their appeal in Python developers.

5.2.3 RQ3: Conciseness of Pythonic idioms

Motivation

Although many researches (Alexandru et al., 2018; Knupp, 2013; Zhang, Xing, Xia, Xu and Zhu, 2022) states that Pythonic idioms offer a concise way to achieve user intent, the specific differences between Pythonic idioms and non-idiomatic code with common syntax in many programming languages like Java and Python are not explored. These differences, are termed “conciseness manifestation". Analyzing it not only makes Python users realize the benefits that Python idioms bring to them but helps us design transformation rules for deidioming idiomatic code.

Approach

To understand where conciseness (e.g., fewer tokens) is reflected in the idiomatic code with Pythonic idioms compared to the corresponding non-idiomatic Python code, we conducted two steps. *The first step* is to determine the non-idiomatic code for idiomatic code of Pythonic idioms by using a

Table 5.2: The statistics of repositories, files and idiomatic code instances of nine Pythonic idioms

Idiom	Repositories	Files	Codes
List-Comprehension	6006	90829	313452
Set-Comprehension	1036	4694	9112
Dict-Comprehension	3109	19845	39970
Chain-Comparison	2617	10540	24764
Truth-Value-Test	2604	71043	418756
Loop-Else	1336	4048	5660
Assign-Multi-Targets	6311	119575	524777
For-Multi-Targets	5963	84286	221662
Star	4913	48344	150678
Total	6997	222637	1708831

Idiomatic code:

```
for (message_id, status) in download_dict.items():
    ...
    return True
else:
    return False
```

Non-idiomatic code:

```
for (message_id, status) in download_dict.items():
    ...
    return True
return False
```

Figure 5.1: The idiomatic code and the corresponding non-idiomatic code of loop-else idiom

card sorting approach (Spencer, 2009). We first randomly sample idiomatic codes with a confidence level of 95% and an error margin 5% for each Pythonic idiom using the data described in Section 5.2.2 (sample size is in the *N* column of Table 5.3). Then two authors with more than six years of Python programming experience independently write the corresponding non-idiomatic code for each idiomatic code. The non-idiomatic code consists of only common syntax of programming languages which allows developers in any programming language to understand the code. Finally, two authors discuss and resolve all disagreements. The Cohen's kappa agreement is 0.73 (substantial agreement). *The second step* is to analyze the categories of concise manifestation of Pythonic idioms. The process is same as the categories of challenges of reading and understanding Pythonic idioms in Section 5.2.1. The Cohen's kappa agreement is 0.75 (substantial agreement).

Result

We summarize four types of syntactic and semantic conciseness with different granularity: *lexical token, code line, variable initialization and function declaration*, which are represented by C1, C2, C3 and C4. Furthermore, C1 is a superset of C2, C2 is a superset of C3, C2 is a superset of C4. Lexical token means a sequence of characters that is the smallest unit of a Python program. When considering this conciseness manifestation of a Pythonic idiom, tokens on new lines are not taken into account; code line means the entire new line but excludes lines of variable initialization and function declaration; variable initialization means the creation of a variable not in the function declaration; function declaration means the definition of a function. Note: The conciseness manifestation of each code pair of a Pythonic idiom may fall into multiple categories simultaneously.

Table 5.3 shows the results, where *N* column means the sample size, the columns labeled *Token*, *Line*, *Variable*, *Function* mean the percentage of code pairs for each idiom belonging to respective categories of conciseness manifestation, and the *Total* column means the percentage of code pairs for each idiom that exhibit conciseness manifestation in at least one category. For nine Pythonic idioms, only the loop-else idiom may make idiomatic code more wordy, accounting for 17.8% (100%-82.2%) of sampled code pairs. Figure 5.1 shows a such example of loop-else, the non-idiomatic code removes the else: line from the idiomatic code. The other eight idioms all make code more concise.

Table 5.3: The percentages of four types of concise manifestation for nine Pythonic idioms

Idiom	N	Token	Line	Variable	Function	Total
List-Comprehension	384	-	100%	57.3%	0.5%	100%
Set-Comprehension	369	-	100%	44.7%	0.8%	100%
Dict-Comprehension	381	-	100%	48.6%	2.6%	100%
Chain-Comparison	379	100%	-	-	-	100%
Truth-Value-Test	384	100%	100%	-	100%	100%
Loop-Else	360	82.2%	82.2%	82.2%	-	82.2%
Ass-Multi-Targets	384	-	100%	35.9%	-	100%
For-Multi-Targets	384	-	100%	-	-	100%
Star	384	100%	-	-	-	100%

• **C1: Pythonic idioms reduce the use of lexical tokens**. It occurs in four Pythonic idioms: chain-comparison, truth-value-test, loop-else and star, accounting for 100%, 100%, 100% and 82.2% of all code pairs for each of these idioms, respectively. For example, in the 2nd row chain-comparison of Table 5.5, the idiomatic code is "r[0]<=line<=r[1] in r" and the corresponding non-idiomatic code is "r[0]<=line and line <=r[1] and r[1] in r". The use of chain-comparison reduces six tokens: two "and" tokens, one "line", one "r", one "[", one "1" and one "]" token. The added "and" token in the non-idiomatic code indicates that we need to create a new AST node BoolOp and replace the Compare with the BoolOp. As another example, in the truth-value-test row of Table 5.5, the idiomatic code "if fuzzy:" reduces three tokens "func", "(" and ")".

• **C2: Pythonic idioms reduce the number of logical lines**. A logical line is constructed from one or more physical lines by following explicit or implicit line joining rules (Developers, 2022b). It occurs in seven Pythonic idioms: list/set/dict-comprehension, truth-value-test, loop-else, assign-multi-targets and for-multi-targets, accounting for 100%, 100%, 100%, 100%, 82.2%, 100% and 100% of all code pairs for each of these idioms, respectively. For example, the idiomatic code of truth-value-test idiom avoids two import statements as shown in the truth-value-test row of Table 5.5. As another example, the idiomatic code in the truth-value-test row of Table 5.5 reduces two code lines with two import statements. Hence, to refactor the idiomatic code into non-idiomatic code, we need to create two "Import" AST nodes and then insert them before the original idiomatic code.

• **C3: Pythonic idioms avoid the creation of variables**, which occurs in five idioms: list/set/dict-comprehension, loop-else and assign-multi-targets. These account for 57.3%, 44.7%, 48.6%, 82.2% and 35.9% of all code pairs for each of these idioms, respectively. For example, for the idiomatic code "data, the_hash = data[:-4], data[-4:]", the corresponding non-idiomatic code is "tmp = data[-4]; data = data[:-4]; the_hash = tmp". It illustrates that assign-multi-targets avoids the creation of the "tmp" to make a backup for "data[-4:]" so that "the_hash" gets the old value of "data[-4:]". As another example, the idiomatic code of the loop-else row of Table 5.5 avoids the variable initialization "loop_flag=True".

• **C4: Pythonic idioms avoid the creation of functions**. It occurs in four idioms: list/set/dict-comprehension and truth-value-test, accounting for 0.5%, 0.8%, 2.6% and 100% of all code pairs for each of these idioms, respectively. For example, in the truth-value-test row of Table 5.5, for the idiomatic code "if fuzzy", we need to create a function to achieve the equivalent semantics because the different values and types can lead to different boolean value for the corresponding non-idiomatic code. As another example, for the idiomatic code "if fuzzy:" of truth-value-test row of Table 5.5, since the non-idiomatic code cannot get the data type and data value until the code runs, it needs to create a func function to represent the truth value of different data.

5.2.4 RQ4: Potential Negative Effects Caused by Idiom Usage

Motivation

After understanding the readability challenges (Section 5.2.1), frequent usage (Section 5.2.2) and conciseness manifestation (Section 5.2.3) of Pythonic idioms, we are interested in exploring whether their usage may cause potential negative effects.

Approach

Previous studies investigated the effects of code smells on different maintainability related aspects such as performance (Hecht et al., 2016; Leelaprute et al., 2022), redundancy (Arif and Rana, 2020; Macia Bertran et al., 2011; *Pylint* 2022), bug (D'Ambros et al., 2010; Hettinger, 2013) and readability (Abbes et al., 2011; Du Bois et al., 2006; Hettinger, 2013). Inspired by them, we explore whether the usage of nine Pythonic idioms may cause these four negative effects. Based on the samples of nine Pythonic idioms of Section 5.2.3, two authors with more than six years of Python programming experience independently analyze the code and determine whether each idiomatic code may cause certain negative effects from the four aspects. The Cohen's kappa agreement between two authors is 0.78 (substantial agreement). Finally, they work together to resolve their disagreements.

Result

• **S1: The usage of Pythonic idioms leads to extra memory allocation or more run time**, which is applicable to list/set-comprehension, star and chain-comparison whose percentages are 2%, 1.4%, 0.3% and 51%, respectively. Although the negative effect may not be directly caused by the misunderstanding or misuse of Pythonic idioms, it should be noticed by Python users and addressed by Python idiom developers. For example, for the idiomatic code "[CL.remove(m) for m in CL...]", it accumulates an iterable of meaningless values "CL.remove(m)" and then throws the iterable away, which not only takes up extra memory but makes code slower compared to non-idiomatic for statements (Developers, 2015).

• **S2: The usage of Pythonic idioms leads to redundant code**, which is applicable to loop-else whose percentage is 17.8%. We find Python users can write a for statement with the else clause but without the Break statement. Such code actually can be implemented without the else clause. Figure 1 shows an example that can be found in this project.

• **S3: The usage of Pythonic idioms leads to bugs when Python users misunderstand the meaning of Pythonic idioms**, which is applicable to chain-comparison whose percentage is 0.5%. Python users may wrongly explain the chain-comparison from left to right or based on the operator precedence rather than translating into an and-expression (*The Explanation of Chain Comparison* 2022a; *The Explanation of Chain Comparison* 2022b). For example, a Python user writes an idiomatic code, "type(destpair) in (list, tuple) == False", to express the meaning of "(type(destpair) in (list, tuple)) == False", but the code is semantically equal to "type(destpair) in (list, tuple) and (list, tuple) == False". Therefore, the code is always false. When such errors occur, the conciseness of chain-comparison makes it more difficult to debug the code.

• **S4: The idiomatic code of Pythonic idioms is too long and could lead to readability issues**, which is applicable to list/set/dict-comprehension and assign-multiple-targets, whose percentages are 38.5%, 41.6%, 56.3% and 23.8%, respectively. Python enhancement proposal 8 (PEP8) **pep8** suggests the line length should be limited to 79 characters for the readability. Since the four Pythonic idioms may use one line to implement the same functionality as multi-lines non-idiomatic code, it will make code too long to read.

5.3 Approach

In Section 4.2, we explore the challenges of understanding nine Pythonic idioms, their prevalence in usage, the diverse ways they manifest conciseness, and the potential negative effects of using these idioms. These findings underpin the importance for Python developers to acquire a thorough and precise comprehension of the meaning and specific behavior of these nine Pythonic idioms.

Table 5.4: Detection rules of the code of nine Pythonic idioms

Idiom	Detection Rules
List/Set/Dict Comprehension	P=ListComp/SetComp/DictComp
Chain Comparison	P=Compare and `Num`(P.ops)>1
Truth Value Test	P.kind=test and P $\notin$ {Compare, Call, BoolOp}
Loop Else	P $\in$ {For, While} and P.orelse $\neq \varnothing$
Assign Multi Targets	P=Assign and `Num`(P.targets)>1
For Multi Targets	P=For and `Num`(P.target)>1
Star	P=Starred

Note: P represents idiomatic code of Python idioms; `Num`(s) returns the number of elements in s. Other symbols starting with a capital letter indicates AST node types or AST node properties defined in Python language specification.

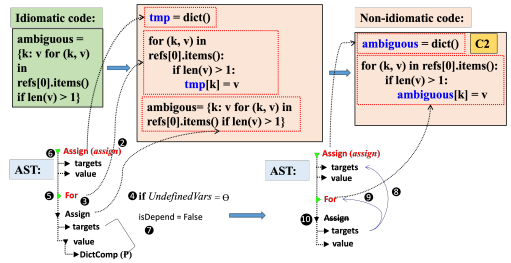
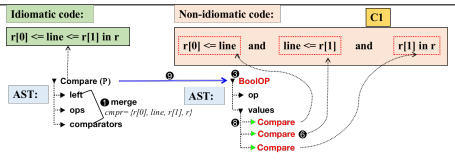
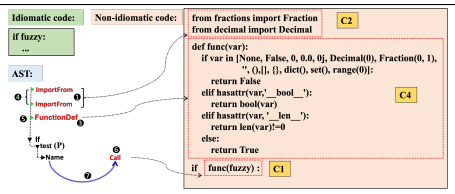
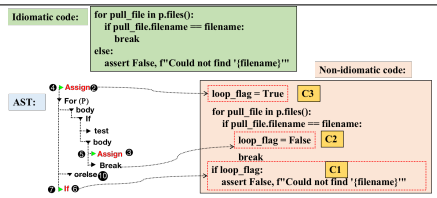
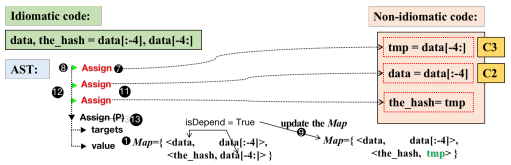
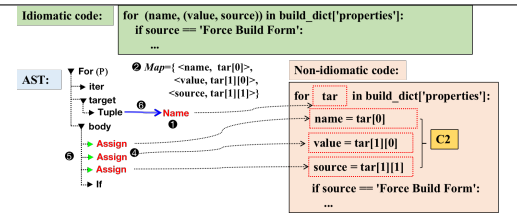
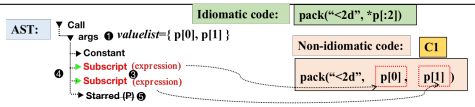
To explain the idiomatic code of Pythonic idioms, we use the corresponding non-idiomatic code. Compared to explaining idiom usage in natural language, non-idiomatic code can avoid the ambiguity of natural language and guarantee semantic correctness. Zhang, Xing, Xia, Xu and Zhu (2022) identified nine unique Pythonic idioms by comparing the exclusive syntax found in Python, not present in Java. This implies that programmers may encounter more challenges in understanding Pythonic idioms due to their unique characteristics in Python. Hence, when transforming the idiomatic code of Pythonic idioms into the corresponding non-idiomatic code, we avoid using alternative APIs for direct explanation. Instead, we employ programming syntax that is widely used in various programming languages like Java, not just Python. As a result, even developers with limited experience in Python development can comprehend the usage of Pythonic idioms.

Furthermore, such corresponding non-idiomatic code offers other potential bonuses (detailed discussion is presented in Section 5.5). It makes developers explicitly understand the conciseness and raises the awareness of the potential negative effects in using Pythonic idioms to some extent, and is also the basis for various downstream program analysis tasks.

Our approach comprises two steps. The *first step* is to design rules to detect idiomatic code of Pythonic idioms. According to the Python language specification, each Pythonic idiom corresponds to an AST node with distinct syntactic properties. We extract idiomatic code based on such AST nodes and properties, as shown in Table 5.4, which is a straightforward method.

The *second step* is to design rewriting steps to transform idiomatic code into non-idiomatic code, as shown in Table 5.5. We design transforming steps based on four atomic operations (Create, Insert, Replace and Remove). Compared to the method of transforming non-idiomatic code into idiomatic code in Zhang, Xing, Xia, Xu and Zhu (2022), transforming the idiomatic code into the corresponding non-idiomatic code has more challenges because it needs to consider whether to create additional statements, new variables, function declarations as explained in Section 5.2.3. For example, for the truth-value-test, if we transform non-idiomatic code into idiomatic code, we can get a test type AST node such as "`x != []`", then we directly refactor it into "`x`". In the contrast, if we rewrite the idiomatic code into the non-idiomatic code (see the truth-value-test row of Table 5.5), we should create a function to explain its functionality. Furthermore, we also should create two import statements to ensure Python users understand the "`Decimal`" is a class from the "`decimal`" module. We should insert the import statements before the function declaration.

Table 5.5: Rules of refactoring idiomatic code into non-idiomatic code for nine Pythonic idioms

Idiom	Examples of Code Refactoring	Transformation Steps
List/ Set/ Dict Compre-hension		1: Get variables $UndefinedVars$ from P that do not appear in the statements before P 2: $assign$ = Create("Assign", "tmp = []/set()/dict()") 3: Transform P into the for_node 4: **If** $UndefinedVars$ is ∅ **then** 5: Insert($assign$, pos(P.stmt)) 6: Insert(for_node, pos(P.stmt)) 7: **If** P.parent is Assign **and** ~ isDepend(P.parent.targets, P) **then** 8: Replace($assign.targets$, P.parent.targets) 9: traverse for_node to replace "tmp" with $assign.targets$ 10: Remove(P.stmt) 11: **Else** 12: Replace(P, $assign.targets$) 13: **Else** 14: $func$ = Create("FunctionDef", name="func", args = $UndefinedVars$, body={$assign$, for_node}) 15: ret = Create("Return", value="tmp") 16: Insert(ret, pos($func$.body)) 17: $call$ = Create("Call", name="func", args=$UndefinedVars$) 18: Replace(P, $call$)
Chain Compa-rison		1: Merge P.left and P.comparators into $cmpr$ 2: ops=P.ops 3: $boolnode$=Create("Bool", op="And") 4: ind = 0 5: **For** op in ops **do** 6: $comparenode$=Create("Compare", left= $cmpr$[ind], ops={op}, comparators= {$cmpr$[ind+1]}) 7: ind+=1 8: Insert($comparenode$, pos($boolnode$.values)) 9: Replace(P, $boolnode$)
Truth Value Test		1: $imports$=Create("ImportFrom", "from decimal import Decimal", "from fractions import Fraction") 2: Transform P into two statements $stmts$ with If statement and Return statement 3: $func$=Create("FunctionDef", args=P, name="func", body=$stmts$) 4: Insert($imports$, pos(P.stmt)) 5: Insert($func$, pos(P.stmt)) 6: $call$=Create("Call", name="func", args=P) 7: Replace(P, $call$)
Loop Else		1: **If** P exists the Break statement **then** 2: $assinit$=Create("Assign", "loop_flag=True") 3: $asschange$=Create("Assign", "loop_flag=False") 4: Insert($assinit$, pos(P)) 5: traverse the P to copy $asschange$ into the position of each Break statement 6: $ifnode$=Create("If", test="loop_flag", body=P.orelse) 7: Insert($ifnode$, pos(P.nextstmt)) 8: **Else** 9: Insert(P.orelse, pos(P.nextstmt)) 10: Remove(P.orelse)
Assign Multi Targets		1: Get a 2-tuple Map with n elements, where each element consists of a target from P.targets and a value from P.value 2: ind=0 3: **For** i from 0 to $n-1$ **do** 4: **For** j from i to n **do** 5: **If** isDepend($Map_{i,0}$, $Map_{j,1}$) and $Map_{j,1}$ has not been created as a temporary variable **then** 6: ind+=1 7: $assign$=Create("Assign", targets="tmp$_{ind}$", value=$Map_{j,1}$)) 8: Insert($assign$, pos(P)) 9: Update $Map_{j,1}$ with $assign$.targets 10: **For** tar, val in Map **do** 11: $assign$=Create("Assign", targets=tar, value=val) 12: Insert($assign$, pos(P.stmt)) 13: Remove(P)
For Multi Targets		1: $name$=Create("Name", id="e") 2: Get a variable mapping pair Map, where each element consists of a target of P.targets and a Subscript node with $name$ value 3: **For** key, val in Map **do** 4: $assign$=Create("Assign", targets=key, value=val) 5: Insert($assign$, pos(P.body.firststmt)) 6: Replace($name$, P.target)
Star		1: Unpack the P into $valuelist$ consisting of several elements 2: **For** e in $valuelist$ **do** 3: $expression$=Create(e) 4: Insert($expression$, pos(P)) 5: Remove(P)

P represents idiomatic code of Python idioms; isDepend(n_1, n_2) represents whether there is data dependence between n_1 and n_2 nodes; pos(n) represent the position of n in the abstract syntax tree.
Idiomatic code | Non-idiomatic code | Create a node | Insert a node to somewhere | Replace | XXXX Remove a node | Concise manifestation

5.3.1 List/set/dict-Comprehension

The list/set/dict-comprehension idioms are used for adding elements to an iterable. To identify such idiomatic code, we extract ListComp, SetComp and DictComp nodes for the three idioms (1st row of Table 5.4). Next, we determine whether to create functions or temporary variables to refactor idiomatic code into non-idiomatic code. For example, the idiomatic code P of the 1st row of Table 5.5 corresponds to the `DictComp` node whose parent node is an `Assign` node. Since the $UndefinedVars = \varnothing$ (line 1), the corresponding non-idiomatic code can not create a Function. We first create a variable tmp to save an empty dictionary (i.e., the *assign* node) and then transform P into a `For` node for_node (line 2 and 3). We then orderly insert the *assign* and for_node into the position of the statement corresponding to P (line 5 and 6). Since there is no data dependency between *ambiguous* and P, the temporary variable tmp is unnecessary, so we replace tmp occurring in the *assign* and the for_node with the *ambiguous*(line 8 and 9). Finally, we remove the assignment statement (line 10).

5.3.2 Chain-Comparison

The chain-comparison idiom can chain any number of comparison operators. We detect idiomatic code P that is a `Compare` node with at at least two operators in P.ops (2nd row of Table 5.4). To refactor idiomatic code into non-idiomatic code, we first merge the left comparator and comparators into *cmpr* (line 1). Then we create a `BoolOp` node with the "and" operator to conjunct several `Compare` nodes (line 3). Finally, we orderly take two comparators from *cmpr* and one operator from P.ops to create a `Compare` node *comparenode*, and then insert the *comparenode* into the values of the `BoolOp` node (line 6 and 8).

5.3.3 Truth-Value-Test

The truth-value-test idiom is to test the truth value for any object. Test type node is testing objects, so we extract such nodes to detect the idiomatic code. As the test type node can be any expression, we remove boolean-valued expressions (i.e., `Compare`, `BoolOp` and `Call` nodes) (3rd row of Table 5.4). If the object belongs to {None, False, '', 0, 0.0, 0j, Decimal(0), Fraction(0, 1), (), [], {}, dict(), set(), range(0)}, the object is considered False. If not, the truth-value-test will check whether the object defines a __bool__ method or a __len__ method. Otherwise, the object is considered True. Based on the workflow, we create a function to explain the idiomatic code of the truth-value-test (line 3), since "Decimal(0)" and "Fraction(0, 1)" use the "decimal" and "fractions" modules, we also need to create two `ImportFrom` nodes (line 1). After that, we insert the three statements to the position of the statement where P is located (line 4 and 5). Finally we create a Call node to replace the old node of the test type node (line 6 and 7).

5.3.4 Loop-Else

The loop-else idiom contains an else clause which is executed when the iterator is exhausted, unless the loop was ended due to a break statement. We detect the idiomatic code by extracting `For` or `While` nodes with the else clause. To explain the idiomatic code with non-idiomatic code, we replace the else clause with common syntax in Python and Java. For example, the 4th row of Table 5.5 shows the idiomatic code P that is a `For` node with an else clause. Since P has a `Break` statement, we create a temporary variable with False value to flag the state before executing the `For` statement (line 2), and then insert it to the position of P (line 4). We then create another `Assign` statement with True value to flag the state after executing the `Break` statement, and insert it to the position of each Break statement (line 3 and 5). Next, we create an If node $if node$ to save P.orelse, and then we insert the $if node$ to the end of P (line 6 and 7). Finally, we remove the P.orelse (line 10).

5.3.5 Assign-Multi-Targets

The assign-multi-targets idiom can assign values to multiple targets in an assignment statement. We detect the idiomatic code that is an Assign node with at least two targets, e.g., the P of the third last row of Table 5.5 has two targets: data and the_hash. To refactor the idiomatic code into non-idiomatic code, we first map the elements in targets and value of P to get a 2-tuple *Map* (line 1). If there is data dependence between the $i-th$ target and the value after $i-th$, we create a temporary variable to

save the value (line 5 and 7). And then we insert the assignment statement to the position of the *P* and update the *Map* with the temporary variable (line 8 and 9). Next, we create an Assign node *assign* for each mapping pair from *Map* and copy the *assign* to the position of the *P* (line 11 and 12). Finally, we remove the *P* (line 13).

5.3.6 For-Multi-Targets

The for-multi-targets idiom can use several data objects as the target of For statement. We extract a For node *P* with at least two targets as the idiomatic code. For example, the second last row of Table 5.5 shows that *P* has three targets: `name`, `value` and `source`. When refactoring the idiomatic code into non-idiomatic code, we first create a Name node to replace the target of *P* (line 1 and 6). Next, we map each object from the old target of *P* into a Subscript node and save the mapping pairs as a 2-tuple *Map* (line 2). Then we create an Assign node *assign* for each element of *Map* and copy the *assign* to the head of the body of *P* (line 4 and 5).

5.3.7 Star

The star idiom is to unpack an iterable into several elements. We detect the idiomatic code with a `Starred` node (the last row of Table 5.4). When explaining the idiomatic code with non-idiomatic code, we first unpack the value of *P* into a list *valuelist* consisting of multiple elements (line 1). Then we create an expression node *expression* for each element of *valuelist* and insert the *expression* to position of the *P* (line 3 and 4). Finally, we remove the *P* (line 5).

Table 5.5 shows the details of refactoring idiomatic code into non-idiomatic code of nine Pythonic idioms.

5.4 Evaluation

To evaluate our approach, we study two research questions:

RQ1 (Accuracy): How accurate is our approach when transforming idiomatic code of nine Pythonic idioms into non-idiomatic code?

RQ2 (Usefulness): Is the generated non-idiomatic code useful for understanding Pythonic idiom usage?

5.4.1 RQ1: Accuracy of Explaining Pythonic idioms

Motivation

Correctly transforming idiomatic code of Pythonic idioms into non-idiomatic code is important for developers to understand idioms precisely. Since we are the first to transform idiomatic code into non-idiomatic code, the high-quality code refactoring can also provide a benchmark for researchers to use.

Approach

Similar to Zhang, Xing, Xia, Xu and Zhu (2022), we use both testing and code review to evaluate the correctness of refactorings.

Testing based verification To collect executed test cases of idiomatic code instances for 1,708,831 idiomatic code instances, we first use DLocator (Wang, Li et al., 2020) statically analyze code to collect test cases that directly call the methods of the idiomatic code. Next, we execute the test cases before transforming the idiomatic code by installing required libraries of projects. As a result, there are 30,386 successfully executed test cases for 6,672 idiomatic code instances.

For test cases of idiomatic code instances that pass successfully, we test if the test cases still pass after transforming the idiomatic code into the corresponding non-idiomatic code. If test cases pass in both cases, the code transforming is correct. Otherwise, if the test cases pass before code refactoring but fail after refactoring, we think the code transforming is wrong.

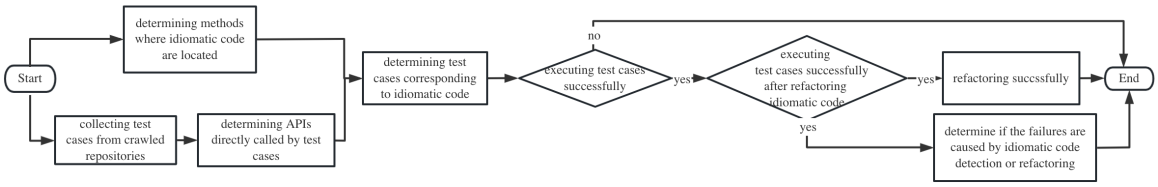


Figure 5.2: The process of testing verification

Table 5.6: Accuracy of detection (d-acc) and rewriting (r-acc) of idiomatic code for nine Pythonic idioms

Idiom	Testing				Code Review		
	#Refs	#TCs	d-acc	r-acc	#Refs	d-acc	r-acc
List-Compreh	1031	4763	1	1	100	1	1
Set-Compreh	60	281	1	1	100	1	1
Dict-Compreh	213	715	1	1	100	1	1
Chain-Compar	150	301	1	1	100	1	1
Truth-Val-Test	3717	18590	1	0.993	100	1	1
Loop-Else	68	271	1	1	100	1	1
Ass-Mul-Tar	519	2082	1	0.996	100	1	1
For-Multi-Tar	904	3370	1	1	100	1	1
Star	10	13	1	1	100	1	1
Total	6672	30386	1	0.995	900	1	1

Since our approach involves two steps, detection and rewriting, we manually identify whether the test failure is due to wrongly detecting idiomatic code instances during the detection step or by incorrectly rewriting the idiomatic code instances during the rewriting step. The complete process is shown in Figure 5.2.

Since our approach involves two steps, detection and rewriting, we manually identify whether the test failure is due to wrongly detecting idiomatic code instances during the detection step or by incorrectly rewriting the idiomatic code instances during the rewriting step. As a result, we can calculate the "*d-acc*" (detection accuracy) and "*r-acc*" (rewriting accuracy) as the percentages of correctly detecting idiomatic code instances among the collected idiomatic code instances and the percentages of correctly rewriting idiomatic code instances among the total idiomatic code instances that were correctly detected.

Code review based verification We randomly sample 100 pairs of idiomatic code and the corresponding non-idiomatic code for each Pythonic idiom. Two authors with more than six years of Python programming experience independently check whether the idiomatic code is detected and transformed correctly. Then they work together to resolve their disagreements. We use the same method as in testing-based verification to calculate the detection accuracy and rewriting accuracy according to their final review results.

Result

Table 5.6 shows the accuracy of testing and code review based verification. The #Refs and #TCs of Testing column represent the number of rewritings for successful execution of test cases and the corresponding number of test cases. And the #Refs of Code Review column are the number of rewritings we manually review. We successfully test 6,672 idiomatic code instances from 478 repositories and review 900 idiomatic code instances from 610 repositories in total. For the code

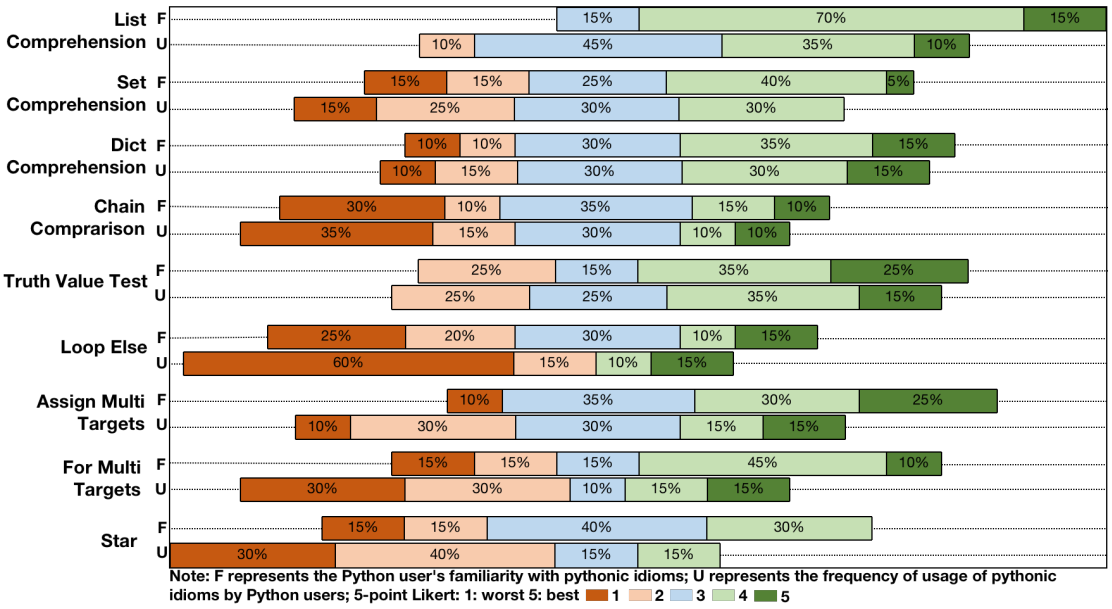


Figure 5.3: The participants' knowledge of 9 Pythonic idioms

review based verification, our approach achieves 100% detection and 100% rewriting accuracy for all nine Pythonic idioms. For the testing based verification, our approach achieves 100% detection and rewriting accuracy for seven Pythonic idioms: list/set/dict-comprehension, chain-comparison, loop-else, star and for-multi-targets. For remaining two idioms truth-value-test and assign-multi-targets, our approach achieves 100% detection accuracy and more than 99% rewriting accuracy. Therefore, our approach is robust on the real-world projects.

We summarize the reasons for the code transformation that cannot pass test cases based on the testing verification. For the truth-value-test, since we statically parse the code, we cannot get the real data type of the data. So we first check whether the data is None. We find if the data is the custom class and overloads the __eq__ method, since None has no attribute, it will raise the AttributError. For the assign-multi-targets, we default to having appropriate values to assign, but some test cases test the wrong values to assign. For example, for the idiomatic code "(f_annotation, f_value) = f_def", we transform it into "f_annotation = f_def[0]; f_value = f_def[1]". The one test case is "f_def = (1, 2, 3)", the "f_def" has three elements which will cause the ValueError because the "f_def" has too many values to unpack to cause the ValueError. However, our generated non-idiomatic code can normally run the code.

5.4.2 RQ2: Usefulness of Explaining Pythonic idioms

Motivation

After validating the high accuracy of our approach, we are interested in whether explanatory non-idiomatic code can help Python users understand idiomatic code correctly and quickly.

Approach

We conduct a controlled experiment to evaluate the impact of non-idiomatic code on understanding Pythonic idioms. Participants were assigned to read idiomatic code either with or without accompanying non-idiomatic code. The experiment involved 20 students with programming experience ranging from two to seven years in Python.

Before the experiment, we learn about the participants' familiarity and usage frequency of nine Pythonic idioms through a pre-study survey. As shown in Figure 5.3, the participants exhibit varying levels of familiarity and usage frequency for each idiom. we randomly split 20 students into two groups based on their programming experience and prior knowledge of the nine Pythonic idioms.

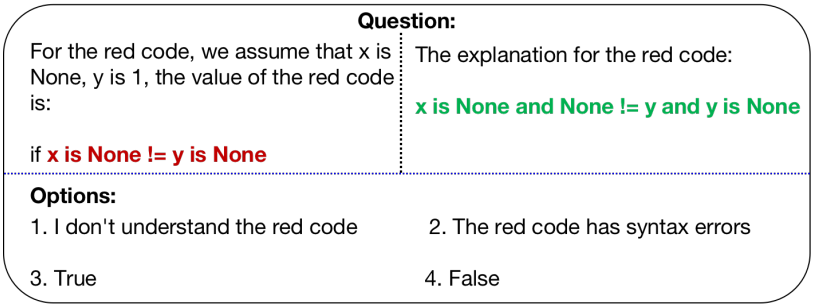


Figure 5.4: The question with idiomatic code and explanatory non-idiomatic code for the chain-comparison idiom

For the experiment, we design 27 multiple-choice questions by randomly selecting 3 idiomatic code instances for each Pythonic idiom from 27 GitHub projects. For each Pythonic idiom, we ensure selected idiomatic code instances cover different circumstances (e.g., different number of node components and negative effects of Pythonic idioms) from Section 4.2. For example, three idiomatic code instances of the chain-comparison contain different types and numbers of operators (2 operators (in and ==), 2 operators(> and >=) and 3 operators (is, is and !=)), respectively. The control group (G1) was only given the idiomatic code, and the experimental group (G2) was given the idiomatic code and the corresponding non-idiomatic code generated by our tool. Figure 5.4 gives an example of the chain-comparison provided to the experimental group with the idiomatic code and the corresponding non-idiomatic code. We collect 540 answers (27 questions×20 participants) for the study (see in the replication package).

The questions in our study are assigned consecutive natural numbers as question numbers after being shuffled. Participants can answer questions in any order they like. All questions are compulsory. When answering questions, they can search the Internet to seek relevant information. As our questions are from real projects, they cannot find the answers directly from the Internet. Our experiment setting closely resembles a real-world scenario where developers may turn to the internet to confirm their assumptions or seek further information when they use Pythonic idioms. It is helpful for us to assess whether providing explanatory code can indeed help developers enhance correctness and efficiency in real-world situations. To prevent participants from answering the questions without reading and understanding the idiomatic code, they are not allowed to run the code. As our questions are multiple choice and each code of a question is only a few lines, participants do not take long time to complete all questions. So we do not set time limits and all participants finish the task in 36 minutes. We provide a document with the above notes for them to read and understand the task before they answer questions. The notes documentation and all questions are in our replication package.

To evaluate the performance difference from participants, we compute their completion time and answer correctness. The completion time is automatically recorded during the study. Then we calculate the answer correctness with the percentage of questions answered correctly for G1 and G2 for each Pythonic idiom and all Pythonic idioms. We use Wilcoxon signed-rank test (Wilcoxon, 1945) to determine if the performance difference of 27 questions between the control and experimental group is statistically significant at the confidence level of 95%. After participants finish the task, we ask them two questions: One question asks the attitudes of all participants toward using nine Pythonic idioms with a five-point likert scale, and the reasons why they give such feedback. The other question asks G2 participants to rate the usefulness of the non-idiomatic code for understanding Pythonic idioms with a five-point likert scale.

Performance Comparison and Analysis

Table 5.7 shows the results of answer correctness and average completion time for each Pythonic idiom and all Pythonic idioms for G1 (control group) and G2 (experimental group). The last column lists the p-value of the Wilcoxon signed-rank test on the correctness difference and the completion time difference. For the answer correctness, G1 and G2 achieve 0.40~0.80 and 0.87~1 for nine Pythonic idioms, respectively. The improvement of correctness is 25%~141.7% for different idioms. For all 27 questions, the overall correctness of G1 and G2 is 0.58 and 0.94, respectively. The improvement

Table 5.7: Performance Comparison

Idioms	N	Correctness			Time (s)		
		G1	G2	Impr (%)	G1	G2	Impr (%)
List-Comprehension	3	0.80	1	25	47.9	38.6	19.4
Set-Comprehension	3	0.63	0.97	52.6	51.7	34.3	33.7
Dict-Comprehension	3	0.57	0.93	64.7	79.0	67.9	14.0
Chain-Comparison	3	0.40	0.9	125	44.2	27.8	37.3
Truth-Value-Test	3	0.70	0.97	38.1	35.3	29	17.8
Loop-Else	3	0.40	0.97	141.7	63.1	41.3	34.6
For-Mul-Target	3	0.60	0.87	44.4	50.0	47.1	5.8
Assign-Mul-Target	3	0.47	0.97	107.1	23.1	24.5	-6.1
Star	3	0.63	0.93	47.4	45.9	30.2	34.3
All	27	0.58	0.94	63.5	48.9	37.8	22.6
P-value	27	7.7×10^{-6}			2.1×10^{-4}		

is 63.5% overall. And the P-value of the *Correctness* column is less than 0.05 that shows the answer correctness of G2 is statistically significantly better than that of G1. Our results suggest that providing non-idiomatic code can improve the correct understanding of corresponding idiomatic code.

For the completion time, the total time spent on answering questions ranged from 328 seconds (about 6 minutes) to 2147 seconds (about 36 minutes) among the 20 participants. For each Pythonic idiom, only for the assign-multi-targets idiom where G2 takes about 6.1% more time than G1, which is reasonable because reading non-idiomatic code also needs extra time. We have received no complaints from the G2 participants about wasting their time reading the explanatory non-idiomatic code for the assign-multi-targets questions. For all other eight idioms, G2 takes 5.8%~37.3% less time than G1, even if they have to read both idiomatic and non-idiomatic code. And the P-value of the *Time* column is less than 0.05 that shows G2 completes tasks statistically significantly faster than G1. Our results suggest that the generated non-idiomatic code can speed up the understanding of Pythonic idioms.

• **list/set/dict-comprehension idioms:** G1 has correctness 0.81 for list-comprehension (the highest correctness score among nine pythonic idioms for G1), while G1 has much lower correctness for dict-comprehension and set-comprehension (0.57 and 0.63) respectively. According to Zhang, Xing, Xia, Xu and Zhu (2022), list-comprehension is much more frequently used than set/dict-comprehension. Our prior idiom knowledge survey in Figure 5.3 also suggests that the participants generally know better and use list-comprehension more frequently than set/dict-comprehension. With the explanatory non-idiomatic code, the correctness gap between the three comprehension idioms becomes very small (1, 0.97 and 0.93 for list/set/dict comprehension respectively). Furthermore, the explanatory non-idiomatic code can speed up the understanding of all three comprehension idioms. For list comprehension that developers generally understand well, the understanding can still be speed-up by 19.4%.

For list/set/dict-comprehension, we find that misunderstanding often occurs as the number of for, if, if-else node increases for G1. For example, for the dict-comprehension, an idiomatic code `{(x, y): 1 if y < 1 else -1 if y > 1 else 0 for x in range(1) for y in range(2) if (x + y) % 2 == 0}` consists of two for nodes, one if node and two if-else nodes. Such mixture of different node components and multiples same nodes of the dict-comprehension makes the code very difficult to understand correctly. 6 G1 participants answer wrongly (40% correctness). In contrast, two G2 participants answer wrongly (80% correctness). It indicates that G2 participants can avoid such misunderstanding with the help of the corresponding non-idiomatic code of complex dict-comprehension code.

• **truth-value-test idiom:** The improvement of correctness is 38.1%, with 17.8% speed-up. The truth-value-test involves a variety of situations, e.g, any object like Call, BinOp and Attribute can be tested for truth value. Python users need deduce the value of the object and judge whether the current value defaults to false. It is challenging for G1 to understand the meaning of the idiom correctly and efficiently because 14 constants are defaulted to false (truth-value-test row of Table 5.5). Python users generally understand that None and 0 are considered false, but grasping all other situations is hard. For example, the value of the idiomatic code if d.expression is Decimal(0). 8 G1 participants answer wrongly but no one in G2 answers wrongly. The average completion time of G1 and G2 is 35.3s and 29s, respectively. The non-idiomatic code makes the G2 spend 17.8% less time than G1. As the non-idiomatic code contains two explicit statements (see the truth-value-test row of Table 5.5): "from decimal import Decimal" and "var in [None, False, '', 0, 0.0, 0j, Decimal(0), Fraction(0, 1), (), [], {}, dict(), set(), range(0)]", G2 participants can know the Decimal is a class with value 0.0 from decimal module and the value is default to false.

• **chain-comparison idiom:** The improvement of correctness is 125%, with 37.3% speed-up. We find many Python users understand >=, <=, >, = and < chained operators correctly. However, when it involves is, is not, in and not in operators, they generally cannot understand correctly. It is because they wrongly assume that these operators have priority order or wrongly explain comparison operators from left to right. However, all comparison operations in Python have the same priority. The chain-comparison is semantically equal to union of several comparison operations. It echos well the negative effects caused by the chain-comparison idiom in Section 2.4. Non-idiomatic code by our tool can effectively clarify those common misunderstandings and thus result in faster and more correct understanding of chain comparisons.

• **loop-else idiom:** The improvement of correctness is 141.7%, with 34.6% speed-up. Most of Python users (6 of 10 participants in G1) answer that they do not understand the loop-else idiom or they think the loop-else syntax is wrong. We interview them and summarize two reasons: they assume that the else clause can only be added after the if statement or they cannot guess what the else clause does in the code. We find that the explanatory non-idiomatic code can not only help participants realize Python supports the else clause after for and while statements but also help them correctly understand the meaning of loop-else.

• **for-multi-targets idiom:** G2 has relatively lower correctness score (0.87), but it is still much higher than 0.60 for G1 on for-multi-targets. Furthermore, G2 has marginal understanding speed-up (only 5.8%). We interview participants in G2 and find participants in G2 need to read more assignment statements, and values of these assignment statements are Subscript AST nodes (i.e., element access). As the number of targets and the nesting depth increases, non-idiomatic code has more assignment statements and each value of the assignments has more Subscript nodes than idiomatic code. For example, for the idiomatic code "for (i, j), t_ij in single_amplitudes", the corresponding non-idiomatic code explains the "j" as "j = tar[0][1]" in the body of the "for tar in single_amplitudes". The for statement has three targets, the non-idiomatic code has three more assignment statements and "i" and "j" have two Subscript nodes, which makes G2 participants sometimes accidentally misread the code and spend more time.

• **assign-multi-targets idiom:** The improvement of correctness is 107.1%, but with 6.1% slow-down in the understanding time. One common misconception G1 participants have for assign-multi-targets is the evaluation order of targets. For example, for the code "dummy2 = other = ListNode(0)", all G1 participants assume that "other" is assigned first, but "dummy2" is assigned first. The explanatory non-idiomatic code "tmp=ListNode(0); dummy2 = tmp; other = tmp" makes G2 participants avoid this misunderstanding. The other common misconception is the evaluation order of targets and value. For example, for the code "uc, ud = ud - q * uc, uc", 5 G1 participants think it is equal to "uc = ud - q * uc; ud = uc". It is because they do not realize the right-hand side (value of assign-multi-targets) is always evaluated before the left-hand side (targets of the assign-multi-targets). The explanatory non-idiomatic code "tmp = uc; uc = ud - q * uc; ud = tmp" helps G2 participants realize the correct evaluation order. These evaluation order misconceptions are also reflected in some Stack Overflow questions we investigated (*Assign-Multi-Targets* 2022a; *Assign-Multi-Targets* 2022b).

• **star idiom:** The improvement of correctness is 47.4%, with 34.3% speed-up. Star can operate any iterable objects such as Subscript and Constant, and it can be used as parameters in the function call, the targets of for and targets and values of assignment statements. We find participants in

G1 generally can answer question correctly for the * before a Subscript data object, but many G1 participants cannot understand other circumstances correctly. For example, for the idiomatic code "cv2.VideoWriter_fourcc(*'mp4v')", we ask them about the number of arguments passed by the function call. 2 G1 participants think the code has syntax error and 3 G1 participants think the number of arguments is one, so the correctness is 50%. We interview them and find them do not know the star can be applied to the string constant or they think the number of elements to be unpacked is 1. The provided non-idiomatic code "cv2.VideoWriter_fourcc('m','p', '4', 'v')" help Python users avoid such misunderstandings.

Comparison of Attitude toward Using Pythonic idioms

G1 participants are disparate: 40% do not support the use of Pythonic idioms (rating 1 or 2), while 50% support the use of Pythonic idioms (rating 4 or 5). G2 participants have the same support ratio (50%), but the rest 50% are neutral. It indicates that providing non-idiomatic code for nine Pythonic idioms could mitigate negative attitudes toward these idioms.

For 10 G1 participants who are only given the idiomatic code, 4 of them express negative attitude towards Pythonic idioms. Some participants lost confidence in writing Python code. For example, a participant said "*I feel that these idioms are obscure. After I finished reading the code related to these idioms, I feel that I can't write Python code anymore.*" Some participants express that they do not understand the value of these Pythonic idioms. For example, a participant said "*These grammars are uncommon and complicated, I usually don't use such grammars and I think they are not very readable.*" Although most of G1 participants manage to answer more than half of the questions correctly, they do not master Pythonic idioms. For example, a participant said "*I didn't really understand the meaning of these idioms when I was answering the question. Fortunately, the options are relatively simple, and I basically just guessed.*"

For G2 participants who are given our explanatory non-idiomatic code, they praise that our tool is helpful for them to understand and use Pythonic idioms correctly (20%, 50% and 30% participants respectively give 3 points, 4 points, and 5 points). For example, a participant said "*When I feel confused with the idiomatic code, I just looked at the explanatory code on the right. Like idioms with loops, it becomes clear when reading the non-idiomatic version.*" Although some participants are not familiar with the idiom, our tool helps many participants learn new Pythonic idioms which prevent them from repelling the use of idioms. For example, a participant said that "*I learnt something new. I have never used the star idiom before a parameter in a function call, I understand the idiom after reading the explanatory code.*" Some developers have acknowledged the Python community's contribution to Pythonic idiom design by reading the non-idiomatic code. For example, a participant said "*The Python community has jointly developed a good specification. This is what we usually call idioms, rather than bringing in programming habits from other programming languages. The provided interpreted code reflects the benefit of the idiom.*"

5.5 Discussion

5.5.1 Implications of Explaining Pythonic idioms by Non-Idiomatic Python Code

Transforming idiomatic code of Pythonic idioms into explanatory non-idiomatic code enhances their readability and comprehension for Python users. **Such transformation may serve as a basis for various subsequent tasks** including effective debugging, programming analysis and the improvement of code quality through the mastery of the intricate syntax and nuanced semantics of Pythonic idioms. For example, when Python developers perform control flow or data flow analysis, they need to simplify syntax like list comprehension into the non-idiomatic Python code (Li, Wang et al., 2022).

Section 5.2.4 illustrates using Pythonic idioms may cause negative effects. We provide suggestions for Python developers to use our De-Idiom tool to help them use Pythonic idioms better.

Python users may consider using DeIdiom tool to avoid neglecting potential negative effects of using Pythonic idioms. For example, although list/set/dict-comprehension can be used as an individual statement, it is not recommended due to their increased memory usage and slower execution compared to loops. By providing the non-idiomatic code, introducing a variable to store the unused iterable could help Python developers realize that using a list comprehension as an individual statement is unnecessary. For example, a developer wrote a code using list comprehension [CL.remove(m) for m in CL...] to remove elements from CL and accumulating CL.remove(m) to an

iterable and subsequently discarding the iterable in Feb 2021. Upon reviewing the repository's usage, we found developers only became aware of this problem in June 2022 and submitted a pull request to remove the use of list comprehension.

Python users may consider using DeIdiom to avoid the misunderstanding of Pythonic idioms. For example, for the chain-comparison, influenced by other programming languages, many developers mistakenly believe that comparison operators have different precedences, as outlined in Section 5.2.1 and Section 5.4.2. However, all comparison operators have equal precedence in Python. Such misunderstanding cannot be ignored because it can even lead to bugs as shown in the S3 of the Section 5.2.4. The corresponding non-idiomatic code may mitigate confusion and misconceptions of Pythonic idioms.

Moreover, **Python users may consider using DeIdiom to assist them in debugging code**. Replacing the idiomatic code of Pythonic idioms into the corresponding non-idiomatic code may help them pinpoint errors more accurately. For the last example of Table 6.1, the idiomatic code "`my_set.add({tup[1] for tup in list_of_tuples})`" throws an error of unhashable type: 'set' because a set cannot be added to the "`my_set`" as an element. The Python user initially attributed the error to the incorrect usage of set comprehension and sought an explanation by asking the question: set comprehension gives "unhashable type". It may be because the code line consists of too many elements: set comprehension and `my_set.add` function call and the limited understanding of the Python user about set comprehension or set. By replacing the set comprehension with non-idiomatic code, the `my_set.add` function call and the set comprehension will be split into different statements, which may help the user realize earlier that the issue is not caused by the usage of set comprehension, but rather that the set cannot be added as elements to another set.

Last but not least, **researchers may develop a tool to automatically identify potential negative effects** of using nine Pythonic idioms in given code for users to notice, and provide further improvement suggestions for the code. Furthermore, **the Python community may delve deeper into underlying reasons of Python users using Pythonic idioms behind the potential negative effects**, such as misconceptions or other benefits of Pythonic idioms, and continue improving the idiom's design and implementation.

5.5.2 Threats to Validity

Internal Validity: One internal threat is the errors in code implementation. We carefully checked the code and evaluated the correctness of our approach by both testing and code review. The other internal threat is the personal bias and wrong classification in challenges, concise manifestation and negative effects of Pythonic idioms. To reduce the personal bias in the manual examination, two authors with more than six years of Python programming experience independently analyze the data and then discuss to reach a consensus. To avoid the wrong classification, two authors double check their results, and since the idiomatic code of nine Pythonic idioms is much shorter than the corresponding whole method, it is not easy for them to make the same mistakes. The data is made publicly available for community evaluation.

External Validity: One external threat is the generalizability of our experimental results. To alleviate the threat, we apply our approach to 7,577 repositories and refactor 1,708,831 idiomatic code instances. And then we verify results with 6,672 refactorings from 478 repositories based on testing and 900 refactorings from 610 repositories based on code review. Another external threat is that our approach is limited to nine Pythonic idioms because they are unique in Python. In the future, we will extend our work to more Pythonic idioms.

5.6 Conclusion and Future Work

This chapter conducts a systematic empirical study on the readability of nine pythonic idioms from Stack Overflow questions, and the conciseness manifestation and potential negative effects of usage of nine pythonic idioms in GitHub projects. To mitigate readability challenges and negative effects of usage of pythonic idioms, we develop the first tool, DeIdiom, for transforming idiomatic code of nine pythonic idioms into explanatory non-idiomatic code. Our large-scale evaluation confirms the robustness of our approach, and our user study shows the usefulness of non-idiomatic code given by our tool for understanding and learning pythonic idioms. We summarize suggestions for Python

developers to use DeIdiom to comprehend and use pythonic idioms better, and for researchers to further enhance pythonic idioms. In the future, we will extend our approach to more pythonic idioms and integrate our tool as a coding assistant in the IDE to promote the adoption and correct use of pythonic idioms.

Chapter 6

Faster or Slower? Performance Mystery of Pythonic Idioms Unveiled with Empirical Evidence.

This chapter was published as

Z. Zhang, Z. Xing, X. Xia, X. Xu, L. Zhu and Q. Lu (2023). 'Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence'. In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, pp. 1495–1507. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00130

The usage of Pythonic idioms is popular among Python developers in a formative study of 101 Pythonic idiom performance related questions on Stack Overflow, we find that developers often get confused about the performance impact of Pythonic idioms and use anecdotal toy code or rely on personal project experience which is often contradictory in performance outcomes. There has been no large-scale, systematic empirical evidence to reconcile these performance debates. In the paper, we create a large synthetic dataset with 24,126 pairs of non-idiomatic and functionally-equivalent idiomatic code for the nine unique Pythonic idioms identified in Zhang, Xing, Xia, Xu and Zhu (2022), and reuse a large real-project dataset of 54,879 such code pairs provided in Zhang, Xing, Xia, Xu and Zhu (2022). We develop a reliable performance measurement method to compare the speedup or slowdown by idiomatic code against non-idiomatic counterpart, and analyze the performance discrepancies between the synthetic and real-project code, the relationships between code features and performance changes, and the root causes of performance changes at the bytecode level. We summarize our findings as some actionable suggestions for using Pythonic idioms.

6.1 Introduction

Python supports many unique idioms that are designed to make the Python code concise and improve runtime performance (Alexandru et al., 2018; Leelaprute et al., 2022; Zhang, Xing, Xia, Xu and Zhu, 2022). Zhang, Xing, Xia, Xu and Zhu (2022) recently identified nine unique Pythonic idioms (including list/set/dictionary comprehension, chain comparison, truth value test, loop else, assignment to multiple targets, the star operator in function calls, and for loops with multiple targets) by contrasting the language syntax of Python and Java. In this study, we focus on these nine Pythonic idioms to understand their performance impacts.

Although the conciseness of these Pythonic idioms is generally acknowledged, we observe that developers often do not have a clear understanding of the performance impacts associated with these idioms and frequently present contradictory information on Stack Overflow (see Section 6.2). This issue is further complicated by the fact that there has been no comprehensive large-scale systematic investigation into the performance effects of these Pythonic idioms.

Studying the performance impact of Pythonic idioms presents a significant challenge due to the lack of datasets containing pairs of idiomatic and functionally-equivalent non-idiomatic Python code. In our formative study (see Section 6.2), we find that developers often base their arguments on synthetic

toy code or personal experience with real-project code to debate whether a Pythonic idiom speeds up or slows down the code. However, both synthetic code and personal experience tend to be anecdotal and frequently contradictory. Synthetic code is typically simplistic (Chen, Yu et al., 2019; Chen, Shang et al., 2020; Developers, 2020) and does not reflect the complexity of real-project code, which may involve various Python libraries and intricate computations. While exploring the performance impact in actual projects holds more practical relevance, obtaining pairs of idiomatic and non-idiomatic code in real projects is quite challenging.

Recently, Zhang, Xing, Xia, Xu and Zhu (2022) developed a Pythonic idiom refactoring tool that can automatically transform non-idiomatic Python code into nine unique Pythonic idioms. Leveraging this tool, we create an extensive synthetic dataset and reuse a large real-project dataset provided by Zhang, Xing, Xia, Xu and Zhu (2022) to study the performance impact of these idioms. This dual approach with synthetic and real-project code allows us to reconcile the performance debates between both sides.

For the synthetic dataset, we consider two types of syntactic code features (variations of Abstract Syntax Tree (AST) nodes and variable scope) and two types of dynamic features (data properties and execution path) to construct the non-idiomatic code. Subsequently, we employ the RIdiom tool (Zhang, Xing, Xia, Xu and Zhu, 2022) to refactor the non-idiomatic code into idiomatic code. This process yields 24,126 pairs of non-idiomatic and idiomatic code for the nine Pythonic idioms, as detailed in Table 6.3. We then manually validate the correctness of the synthetic code. The real-project code dataset contains 54,879 pairs of non-idiomatic and corresponding idiomatic code from 270 successfully configured Python projects on GitHub. Zhang, Xing, Xia, Xu and Zhu (2022) construct this dataset by first detecting non-idiomatic code that their tool can refactor and then transforming it to idiomatic code. Each code pair is accompanied by a set of test cases from the project that can execute the code before and after the refactoring.

To measure and reliably compare code execution times, we execute each piece of code in 50 Virtual Machine (VM) invocations and collect the execution time of 35 iterations (excluding the first 3 warm-up executions) in each VM invocation, following procedures outlined in previous studies (*A toolkit to write, run and analyze benchmarks* 2022; Crapé and Eeckhout, 2020; Georges et al., 2007a; Laaber, Scheuner et al., 2019; Traini et al., 2021). To ensure robustness, we perform bootstrapping with hierarchical random re-sampling and replacement on both VM invocations and execution iterations, as recommended in earlier research (Davison and Hinkley, 1997; Kalibera and Jones, 2020; Ren, Lai et al., 2010; Traini et al., 2021). Our results show that the uncertainty of the performance changes across both datasets for each Pythonic idiom is less than 0.05, demonstrating the reliability of our performance measurements.

To help developers comprehend the performance impact of Pythonic idioms, we structure our study by addressing three research questions:

- *RQ1: What is the performance impact of Pythonic idioms?*
- *RQ2: How well can code features explain the performance differences caused by Pythonic idioms?*
- *RQ3: What are the root causes of performance differences caused by Pythonic idioms, and what causes the inconsistencies between synthetic and real-project code?*

Our findings reveal significant variability in the performance impact of Pythonic idioms across different idioms and between synthetic and real-project code. Although Pythonic idioms tend to speed up synthetic code, the speedup is generally small to moderate (within a factor of 2). Conversely, in real-project code, using idioms more often results in slowdowns, although the differences between using idiomatic and non-idiomatic code are typically minor. We discover that code features (such as code complexity, variable scope, data properties, and execution paths) can effectively explain the performance differences in the synthetic code but offer limited explanatory power for real-project code. On examining the bytecode differences, we find that idioms usually reduce execution time by using specialized bytecode instructions or performing specific operations (e.g., truth-value tests against Python's predefined EmptySet, variable swaps, loop-else constructs). However, the effects of these optimizations can be diminished by library objects, overloaded built-in methods, function calls, and complex computations in real-project code.

Ultimately, we offer actionable suggestions for developers to use Pythonic idioms more effectively. For instance, the loop-else idiom is beneficial as it does not degrade performance and enhances code conciseness. For other Pythonic idioms, careful consideration based on the specific needs and context of real code development is advisable. For example, while list/set/dictionary comprehensions may offer limited speedup even with a large number of added elements, they can make code harder to read due to condensed syntax.

The main contributions of this chapter are as follows:

• The first large-scale empirical study on the performance impact of Pythonic idioms on synthetic and real-project code. We leverage both synthetic datasets and real-project code datasets to address the gap in understanding the performance implications of Pythonic idioms. This dual approach provides a comprehensive view that balances the simplicity of synthetic code with the complexity of real-world applications. Our study spans across 79,005 pairs of code, making it the first of such substantial scope in examining Pythonic idioms.

• Systematic analysis of the relationships between code features and performance differences caused by Pythonic idioms and the performance-change root causes at the bytecode level. We utilize a Generalized Additive Model (GAM) to decipher the correlations between various code features and performance improvements or degradations introduced by Pythonic idioms. Our in-depth investigation at the bytecode level helps to identify critical reasons for performance variations. Notably, we categorize the impact into five primary causes, shedding light on how idiomatic expressions interact with Python virtual machine (VM) instructions.

• Systematic comparison of the performance impact of Pythonic idioms between synthetic and real-project code. While synthetic datasets reveal potential performance benefits under controlled conditions, our research reveals that these benefits often do not translate to real-world projects. Instead, complexities such as library dependencies, overloaded built-in methods, numerous function calls, and intricate calculations in real-project code tend to diminish, or even negate, the advantages purported by Pythonic idioms.

• A set of actionable suggestions to use Pythonic idioms. Based on our empirical findings, we provide developers with practical guidelines for utilizing Pythonic idioms effectively. For instance, incorporating the loop-else idiom appears beneficial as it consistently avoids performance degradation while enhancing code conciseness. However, for idioms like list/set/dictionary comprehensions, our insights suggest a more cautious approach, weighing their code readability and undisputed benefits against their limited performance gains in complex codebases.

6.2 Formative Study

To understand the opinions and arguments on the performance of Pythonic idioms, we examine Stack Overflow questions for the nine Pythonic idioms identified by Zhang, Xing, Xia, Xu and Zhu (2022). We search python-tagged questions for each Pythonic Idiom with the idiom name and performance-related keywords such as “slow”, “fast”, “performance”, “speed” or “time” by using Stack Overflow search interface (*Stack Overflow* 2022). For each idiom, we check the returned top-30 questions and finally collect 101 questions that discuss the performance of these nine Pythonic idioms.

Table 6.1 shows representative discussions about performance impact of nine Pythonic idioms on Stack Overflow. #N represents the number of questions we find for each Pythonic Idiom and blue text represents the view times that more than 1k times indicates the questions are popular.

(1) Developers are concerned with the performance of Pythonic idioms. First, developers ask whether idiomatic code is faster or slower than non-idiomatic code. For example, for the star-in-func-call in Table 6.1, developers are interested in whether the star operator affects the performance. Furthermore, developers would like to know how many times idiomatic code may be faster or slower than the non-idiomatic code. For example, for the list-comprehension in Table 6.1, a developer states that the list comprehension is 50% faster than appending to a list with for loop. Finally, developers want to know the reasons why Pythonic idioms causes the performance change. For example, for the for-multiple-targets in Table 6.1, a developer ask why accessing by index slow things down compared to for-multiple-targets.

(2) Many questions lack clear evidence of the performance impact of Pythonic idioms. Only 49.5% questions list code fragments and corresponding execution time to illustrate the performance of Pythonic idioms. For example, for the chain-comparison in Table 6.1, developers write a toy code pair of `x<y<z` and `x<y and y<z` to compare their execution time. Furthermore, for the two questions regarding loop-else and star-in-func-call, they lack code fragments or execution time, but provide only natural language descriptions for the performance of the idioms. For example, for the loop-else in Table 6.1, a developer say "I was introduced to a wonderful idiom in which you can use a for/break/else scheme with an iterator to save both time and lines of code".

(3) Developers are often confused by the controversial descriptions or evidences of the performance of Pythonic idioms. 33 out of 101 question threads present some contradictory performance results. Among the nine examples in Table 6.1, six questions discuss the contradictory performance results, annotated with a *.

6.3 Empirical Study Setup

To reconcile the performance debates around Python idioms, we conduct a large-scale empirical study. This section describes our datasets and performance measurement method.

6.3.1 Data Collection

Our study includes both synthetic and real-project code. Synthetic code is generated from a set of syntactic and dynamic code features that may affect the execution time of idiomatic and non-idiomatic code. Real-project code is from the dataset of before-after Python idiom refactorings applied to the Github projects for evaluating the Python idiom refactoring tool in Zhang, Xing, Xia, Xu and Zhu (2022). Synthetic code provides a "clean" setting to measure the performance impact of Python idioms, while real-project code covers much more complex contexts in which Python idioms are used.

Synthetic Dataset

Construction Process: We identify a set of performance-related syntactic and dynamic code features based on the literature review (Chen, Shang et al., 2020; Huang, Ma et al., 2014; Song and Lu, 2017) and our observation of Stack Overflow questions on Python idiom performance. We vary these code features to generate diverse non-idiomatic code and then refactor it using Ridiom tool (Zhang, Xing, Xia, Xu and Zhu, 2022). To validate the correctness of the synthetic code, we invite 18 external workers with more than five years Python programming experience. We divide them into 9 groups, and each group of two workers independently checks the correctness of the synthetic dataset for an idiom. The Kappa values of nine groups for their annotation results all exceeded 0.9. Finally, the two authors discuss and resolve the inconsistencies in the annotation results and ensure the synthetic dataset achieved 100% accuracy.

Code Features for Code Systhesis: As listed in Table 6.2, syntactic features include variations of AST nodes and variable scope variation (local or global) applicable to an idiom. Dynamic features include variant data properties and execution paths applicable to an idiom. The "num*" represents the number of AST nodes, the "`*Set`" represents the set of AST nodes, the "`has*`" ({1 or 0}) represents whether a Python idiom has a AST node or not, ahd the "`is*`" ({1 or 0}) represents whether a Python idiom satisfies a condition or not.

(1) **Variations of AST Nodes:** Based on the Python language syntax (*Python Abstract Grammar* 2022), we construct various valid node combinations for an idiom, and if applicable, change the number of nodes to analyze the impact of these code variations.

• The *list/set/dict comprehension* is to append elements to an iterable. They must contain at least one For node, and may have If node with else or If node without else. We set up the range of the number of For node (numFor) is 1∼4 (i.e., up to 4 nested loop), and the range of the number of If node without else (numIf) and that of If node with else (numIfElse) is 0∼5. For example, the non-idiomatic code of the list-comprehension of Table 6.2 has one numFor, zero numIf and zero numIfElse.

• The *chain-comparison* idiom is to chain multiple comparison operations, so it has at least two comparison operators. We set the range of numCompop to 2∼5. The comparison operator compop

Table 6.1: Pythonic Idiom Performance Related SO Questions

Idiom	#N	Question
List-Comprehension*	26	**Question:** How to speed up list comprehension? my understanding is that for-loop is faster than list comprehension. **Comments:** In all cases I've measured the time a list comprehension was always faster than a standard for loop. (2k times)
Set-Comprehension*	12	**Question:** How do python Set Comprehensions work? I tried timeit for speed comparisons, there is quite some difference. **Answer:** List/Dict/Set comprehensions tend to be faster than anything else. (2k times)
Dict-Comprehension*	15	**Question:** Why is this loop faster than a dictionary comprehension? **Comment:** ... I do this with a dictionary with 1000 random keys and values, the dictcomp is marginally slightly faster. (9k times)
Chain-Comparison*	6	**Question:** Is "x < y < z" faster than "x < y and y < z"? In this page, we know that chained comparisons are faster than using the "and" operator. However, I got a different result... It seems that `x < y` and `y < z` is faster than `x < y < z`. (11k times)
Truth-Value-Test*	17	**Question:** bool value of a list in Python. **Answer:** 99.9% of the time, performance doesn't matter as suggested Keith. I only mention this because I once had a scenario, using implicit truthiness testing shaved 30% off the runtime. (31k times)
Loop-Else	7	**Question:** Pythonic ways to use 'else' in a for loop. **Answer:** I was introduced to a wonderful idiom in which you can use a for/break/else scheme with an iterator to save both time and LOC. (1k times)
Assign-Multi-Targets*	8	**Question:** Python assigning two variables on one line I've been looking to squeeze a little more performance out of my code; While browsing this Python wiki page, I found this claim: Multiple assignment is slower than individual assignment. I repeated several times, but the multiple assignment snippet performed at least 30% better than the individual assignment. (3k times)
Star-in-Fun-Call	5	**Question:** What does the star mean in a function call? Does it affect performance at all? Is it fast or slow? (245k times)
For-Mul-Targets	5	**Question:** How come unpacking is faster than accessing by index? (3k times)

Table 6.2: Syntactic (AST Node and Var Scope) and Dynamic (Data Property and Execution Path) Features of Synthetic Code

Feature	List/Set/Dict-Comprehension	Chain-Comparison	Truth-Value-Test	Loop-Else	Assign-Multi-Targets	Star-in-Func-Call	For-Multi-Targets

AST Node	numFor : 1-4
	numIf : 0-4
	numIfElse : 0-4
		numComop : 2-5
		compop : `CompopSet`
			test: `TestSet`
			compop : `EqSet`
			value : `EmptySet`
				`LoopSet`
				`ConditionSet`
					numAssign : 2-30	numSubscript : 1-30
						`hasSubscript`
						`hasStep`
						`hasLower`
						`hasUpper`
							numSubscript : 1-30
							numTarget : 1-5
							`hasStarred`

Var Scope	`Local/Global`	`Local/Global`	`Local/Global`	`Local/Global`	`Local/Global`	`Local/Global`	`Local/Global`
Data Prop	`size`	`isTrue`	`isTrue`	`size`	value : `isConst` `isSwap`	index: `isConst`	`size`
Exec Path	-	-	-	`isBreak`	-	-	-
Example							

comes from the `CompopSet` ({==, !=, <, ≤, >, ≥, is, is-not, in, not-in}). For example, the non-idiomatic code of the chain-comparison of Table 6.2 has two comparison operators: != and ≥.

• The *truth-value-test* idiom tests whether an object is equal or not equal to an empty value. The parent node of the test node belongs to `TestSet`={While, Assert, If}, the comparison operator belongs to `EqSet`={==, !=}, and the value belongs to Python-predefined `EmptySet`={None, False, '', 0, 0.0, 0j, Decimal(0), Fraction(0, 1), (), [], {}, dict(), set(), range(0)}. For example, for the non-idiomatic code of the truth-value-test of Table 6.2, the test node "a == ()" belongs to the While node, the comparison operator is "==" and the empty value of the comparator is "()".

• The *loop-else* idiom is to determine whether to execute the else clause after the iterator is exhausted. It contains one For or While node (`LoopSet`) and one If node with or without else (`ConditionSet`) after the loop block. For example, the non-idiomatic code of the loop-else of Table 6.2 consists of While node and If node without else.

• The *assign-multiple-targets* idiom is to assign multiple values in one assignment statement. We set the range of the number of assign node to 2∼30. For example, the non-idiomatic code of the assign-multi-targets of Table 6.2 consists of three assignment statements (the numAssign is 3).

• The *star-in-func-call* idiom is to unpack an iterable to the positional arguments in a function call. The non-idiomatic code consists of Subscript nodes (e.g., e_list[1]). We set the range of the number of Subscript node (numSubscript) to 1∼30. The corresponding idiomatic code may or may not have the Subscript node `hasSubscript`. If it has the Subscript node, it may or may not have the step, lower and upper nodes, where are denoted by `hasStep`, `hasLower` and `hasUpper`. For example, the non-idiomatic code of the star-in-func-call of Table 6.2 consists of two Subscript nodes: e_list[1] and e_list[3]. The corresponding idiomatic code has the Subscript node with step, lower and upper nodes.

• The *for-multiple-targets* idiom is to unpack operators inside an iterable as a flattening operator. The non-idiomatic code has a For node and a Subscript node in the body of the For node to use each item. We set the range of the number of Subscript (numSubscript) node from 1∼30. The corresponding idiomatic code may have many Target nodes, so we set the range of the number of target node to 1∼5. The idiomatic code may have or not have the a Starred node (`hasStarred`) to represent remaining unused items. For example, the non-idiomatic code of the for-multiple-targets of Table 6.2 uses one item (i.e., the numSubscript is 1). The corresponding idiomatic code has two targets containing one Starred node.

(2) **Variable Scope:** In Python, the performance of variables in the local namespace is different from the variables in the global namespace (Developers, 2022a; *Python programming FAQ* 2022). To explore the impact of variable scope on the code performance, we wrap the idiomatic code in a function (i.e., `Local`) or not in the function (i.e., `Global`). For example, the code of the list-comprehension of Table 6.2 is defined the function, so the variable scope is `Local`.

(3) **Data Properties**: When constructing microbenchmarks (Costa et al., 2019; Developers, 2020) to evaluate code performance, developers usually set different data sizes. Inspired by this practice, we set `size` to {0, 1, 10, 10^2, 10^3, 10^4, 10^5, 10^6} for an iterable. For list/set/dict comprehension, it represents the number of added elements of the iterable. For the loop-else and for-multiple-targets, it represents the number of iterations. For chain-comparison and truth-value-test, we set the `isTrue` to {0 or 1}, representing the result of their expressions being evaluated to False or True. The value of assign-multiple-targets may be a temporary variable or not, which represents whether the functionality is to swap variables or not (`isSwap`=0 or 1). Besides, many researchers state that constants may be faster than variables (Barany, 2014; Ismail and Suh, 2018; Rodriguez-Cancio et al., 2016; *When and why need a constant?* 2022). As the value of assign-multiple-targets and the index of Subscript of star-in-func-call may be a constant or not, we set `isConst`=0 or 1 for these two Python idioms. For example, the code of the list-comprehension of Table 6.2 has no elements appended so the `size` is 0.

(4) **Execution Path**: A code snippet may have multiple paths whether executes the break statement or not, and different execution paths may lead to different performance. In this work, only the loop-else involves two execution paths: whether a path executes the break statement or not (`isBreak`). For example, for the code of the loop-else of Table 6.2, the size of "e_list" is 0 so the Break statement is not executed.

Real-Project Dataset

We use a publicly available dataset of nine Python idiom refactorings (Zhang, Xing, Xia, Xu and Zhu, 2022) from the 270 successfully configured GitHub Python projects. For each instance of refactoring, it provides the successfully configured project to execute the non-idiomatic code before the refactoring and the corresponding idiomatic code after the refactoring with a set of test cases with different inputs. To ensure that each test case executes the non-idiomatic and idiomatic code, we insert print statements around the code to filter out the test cases without printing the content we set up. As both refactored code and test cases are from the real projects, we cannot control their syntactic and dynamic features. However, we observe they still exhibit diverse feature variations.

(1) **Variations of AST Nodes**: For real-project code, the "`*Set`" is same as those for the synthetic code. For list-comprehension, the range of numFor, numIf, numIfElse are 1∼3, 0∼2 and 0∼2, respectively. For set-comprehension, the range of numFor, numIf, numIfElse is 1∼2, 0∼1 and 0. For dict-comprehension, the range of numFor, numIf, numIfElse is 1∼3, 0∼1 and 0∼1. For chain-comparison, the numComop is 2∼4. For assign-multiple-targets, the range of numAssign is 2∼11. For star-in-func-call, the range of numSubscript is 2∼4. For for-multiple-targets, the range of numSubscript and numTarget are 1∼6 and 1∼3, respectively. For star-in-func-call, 1%, 4%, 16% and 16% of code instances have `hasStep`, `hasLower`, `hasUpper` and `hasSubscript` being 1. For for-multi-targets, 90% of code instances have `hasStarred` being 1. The code-feature variations in the real-project dataset is smaller than those in the synthetic code.

(2) **Variable Scope**: Since the real-project code is executed by test cases, all variables of Python code are in the local scope.

(3) **Data Properties**: For list/set/dict-comprehension, loop-else and for-multiple-targets, their ranges of `size` are 0∼11,766, 0∼44, 0∼1,000, 0∼50 and 0∼583, respectively. For the chain-comparison, 45% of `isTrue` is 1. For the truth-value-test, 25% of `isTrue` is 1. For assign-multi-targets, 0.04% of `isSwap` of code instances is 1 (i.e., rarely used for swapping variables). For assign-multi-targets and star-in-func-call, 1% and 97% of code instances have `isConst` being 1.

(4) **Execution Path**: For the loop-else, the percentage of code instances with `isBreak`=1 is 89%.

Dataset Summary

Table 6.3 summarizes the number of code pairs (non-idiomatic versus idiomatic) in the synthetic dataset and the real-project dataset. For synthetic list/set/dict-comprehension code pairs, 1600 is computed by multiplying 4 numFor, 5 numIf, 5 numIfElse, 2 and 8 values (local or global) of variable scope and `size`. For synthetic chained comparison code, 11968 is computed by multiplying 2992 combinations of 2-5 comparisons from `CompopSet`, 2 and 2 values of variable scope and `isTrue`. For synthetic assign-multiple-targets, when the values are constants, the `isSwap` can only be 0, so there are only 174 code pairs instead of 232. For synthetic truth-value-test, 336 is computed by multiplying 3, 2, 14, 2 and 2 values of `TestSet`, `EqSet`, `EmptySet`, scope and `IsTrue`. For synthetic loop-else, the 128 is computed by multiplying the 2, 2, 2, 8 and 2 values of `LoopSet`, `ConditionSet`, variable scope, `size` and `isBreak`. For synthetic star-in-fun-call, the 1920 is computed by multiplying the 30 numSubscript, 2, 2, 2, 2, 2 and 2 values of `hasSubscript`, `hasStep`, `hasLower`, `hasUpper`, variable scope and `isConst`. For synthetic for-multi-targets, the 4800 is computed by multiplying 30 numSubscript, 5 numTarget, 2, 2, 8 values of `hasStarred`, scope and `size`. For real-project code, the number of code pairs are reported by the refactoring tool (accompanied by at least one test case to execute the before- and after-refactoring code).

6.3.2 Performance Measurement

The measurement of execution time of non-idiomatic and idiomatic code is far from trivial due to the non-determinism such as Python Virtual Machine (VM) and garbage collector (Georges et al., 2007b). To overcome such non-determinism, researchers repeatedly execute the code multiple times (Chen, Shang et al., 2020; Crapé and Eeckhout, 2020; Ding et al., 2020; Georges et al., 2007b; Laaber and Leitner, 2018; Laaber, Scheuner et al., 2019; Stefan et al., 2017; Traini et al., 2021). Since different VM invocations may result in different code execution time, and multiple executions of the code in a VM invocation may vary, we execute each code 35 iterations on 50 VM invocations as done in previous studies (Crapé and Eeckhout, 2020; Georges et al., 2007a; Laaber, Scheuner et al., 2019; Traini et al.,

Table 6.3: The statistics of Synthetic and Real-Project Dataset

Idiom	Synthetic	Real-Project
List Comprehension	1600	734
Set Comprehension	1600	282
Dict Comprehension	1600	194
Chain Comparison	11968	2268
Truth Value Test	336	40116
Loop Else	128	198
Assign Multiple Targets	174	10583
Star in Func Call	1920	170
For Multiple Targets	4800	334
Total	24126	54879

2021). Since the first iterations (i.e., warm-up iterations) are subject to noise caused by library loading, measurements are only collected in iterations that are subsequent to warm-up. We set the warm-up iterations to three which is enough to warm up the benchmarks (*A toolkit to write, run and analyze benchmarks* 2022; Traini et al., 2021). We run the two datasets on the CPython interpreter version 3.7.12 on an Ubuntu 18.04.4 System.

Following previous studies surveyed in Kalibera and Jones (2013, 2020), to indicate how much speedup or slowdown idiomatic code achieves compared with non-idiomatic code, we divide the total of execution time of non-idiomatic code by that of corresponding idiomatic code. The formula is as follows: $\rho = \sum_{i=1}^{n}\sum_{j=1}^{k} m_{i,j}^{nonidiomatic} / \sum_{i=1}^{n}\sum_{j=1}^{k} m_{i,j}^{idiomatic}$ where n is the number of VM invocations, m is the number of measurement iterations, and $m_{i,j}$ is the execution time of the kth iteration in the ith invocation. The longer execution time means the slower speed. If ρ is larger than 1, the idiomatic code has a ρX speedup (i.e., the idiomatic code is 1-1/ρ% faster than non-idiomatic code). For example, if the time of non-idiomatic code is 2s and the time of idiomatic code is 1s, the idiomatic code has a 2X speedup and is 50% faster. If ρ is less than 1, the idiomatic code has a $1/\rho$X slowdown (i.e., the idiomatic code is $1/\rho - 1$ times slower than non-idiomatic code). If ρ=1, the performance of idiomatic code is the same as that of non-idiomatic code.

To validate the reliability of performance measurement, we apply the approach of quantifying performance changes with effect size proposed by Kalibera and Jones (2013, 2020). We use bootstrapping with hierarchical random re-sampling and replacement (Davison and Hinkley, 1997; Ren, Lai et al., 2010; Traini et al., 2021) on two levels (Kalibera and Jones, 2020; Traini et al., 2021): VM invocations and iterations. We run the experiments 1000 times and obtain 1000 performance change $P = \{\rho_i \mid 1 \leq i \leq 1000\}$. We obtain the lower limit ρ_l and upper limit ρ_u for 95% confidence from P. Then we compute uncertainty of performance change by the $(\rho_u - \rho_l)/(\rho)$. Our results show that all the uncertainty of the performance change on the two datasets for each Python idiom is less than 0.05, which shows the reliability of our performance measurements.

6.4 Empirical Analysis

6.4.1 RQ1: What is the performance impact of Pythonic Idioms? Is the impact consistent on synthetic and real-project code?

Motivation

Section 6.2 shows that developers are concerned with the performance of Pythonic Idioms and often experience different and even contradictory results. However, there lack of large-scale, systematic

empirical evidence of the performance differences between functionally-equivalent idiomatic and non-idiomatic code. Furthermore, there is little consensus between the performance impact on synthetic and real-project code. This RQ fills in this gap.

Approach

We consider nine unique Pythonic Idioms (Zhang, Xing, Xia, Xu and Zhu, 2022). As described in Section 6.3.1, we synthesize a large dataset of 24,126 pairs of functionally-equivalent idiomatic and non-idiomatic code with variant syntactic and dynamic code features. We also use a large dataset of 54,879 pairs of functionally-equivalent idiomatic and non-idiomatic code collected by RIdiom tool in Chapter 3. We adopt a systematic and reliable execution time benchmarking method to measure the performance differences between a pair of code. We analyze the performance differences from four aspects: performance impact (speedup, slowdown or unchanged), maximum speedup, maximum slowdown, and the variation of performance changes.

Results

Fig. 6.1 shows the distribution of performance differences between non-idiomatic and idiomatic code of nine Pythonic Idioms on the two datasets. The gray circles are the outliers. The orange line inside the box represents the median value. The upper and lower whiskers represent the maximum speedup and slowdown excluding the outliers, respectively. The box represents the 25th to 75th percentile of the dataset.

• **Performance speedup, slowdown or unchanged:** On the synthetic code, the majority of ρ for seven idioms is >1, including list/set/dict-comprehension, truth-value-test, loop-else, star-in-func-call, and for-multi-targets. That is, for these seven idioms, idiomatic code more likely results in performance speed up against non-idiomatic code. Only one idiom (i.e., assign-multi-targets) more likely results in performance slowdown (i.e., the majority of $1/\rho > 1$ or $\rho < 1$). Except for loop-else, no matter the majority is speedup (or slowdown), there is always a certain percentage of cases having the opposite effect. For chained-comparison, about half of the cases result in speedup, while the other half result in slowdown.

On the real-project code, six idioms most likely result in performance slowdown (i.e., the majority of $1/\rho > 1$), including list-comprehension, dict-comprehension, chain-comparison, assign-multi-targets, star-in-func-call, and for-multi-targets. Only one idiom (truth-value-test) more likely results in performance speedup. For the two idioms (set-comprehension, loop-else), the majority of cases with ρ very close to 1 (except for some outliers). We consider the performance difference of these two idioms as unchanged.

Across the two datasets, only truth-value-test and assign-multi-targets exhibit consistent performance impact (the majority speedup or slowdown). Four idioms exhibit the opposite impact (the majority speedup vs. the majority slowdown). The other three idioms also exhibit different impacts (the majority speedup vs. the majority unchanged for set-comprehension and loop-else, and half-half split vs. the majority slowdown for chained-comparison).

• **Maximum speedup:** On the synthetic code, six idioms (list/set/dict-comprehension, truth-value-test, star-in-func-call, for-multi-targets) have the maximum speedup >2. Three idioms (list-comprehension, truth-value-test and for-multi-targets) have relatively higher percentages (15%, 15% and 19% respectively) of cases with >2 speedup. Three idioms (set-comprehension, dict-comprehension and star-in-func-call) have a small number of cases (0.3%-5%) with >2 speedup. Three idioms (chain-comparison, loop-else, assign-multi-targets) have the maximum speedup <2. The speedup of some outliers for list-comprehension, set-comprehension, truth-value-test and for-multi-targets can be >4 times. On real-project code, although only truth-value-test has the majority speedup, it has 28% of cases with >2 speedup. The maximum speedup for truth-value-test reaches about 11 times. All other eight idioms have the maximum speedup <2 (even the outliers are not close to 2). Among these eight idioms, except for list-comprehension, star-in-fun-call and for-multi-targets, five idioms have the maximum speedup close to 1 (i.e., the same as non-idiomatic code). With the 2 times speedup as a threshold, the maximum speedup of truth-value-test, chained-comparison, loop-else and assign-multi-targets are consistent across the two datasets (both >2 or both <2). The maximum speedup of the other five idioms are inconsistent.

• **Maximum slowdown:** On the synthetic code, only for-multi-targets has the maximum slowdown >2, accounting for 15% of cases. The slowdown of some for-multi-targets outliers reaches about 8 times. For the other eight idioms, the maximum slowdown is all <2, except for a few outliers. Among these eight idioms, truth-value-test and loop-else have the maximum slowdown close to 1. On the real-project code, truth-value-test and for-multi-targets have the maximum slowdown >2, accounting for 10% and 27% of the cases of each idiom, respectively. An outlier of for-multi-targets reaches 6 times slowdown, and an outlier of truth-value-test reaches 20 times slowdown. For the other seven idioms, the maximum slowdown is all <2, except for a few outliers. Loop-else has the maximum slowdown close to 1. With the 2 times slowdown as a threshold, the maximum slowdown of eight idioms are consistent (both >2 or both <2). Only one idiom (truth-value-test) has inconsistent maximum slowdown.

• **The variation of performance changes:** We calculate the variation between the maximum speedup and the maximum slowdown. On the synthetic code, two idioms (list-comprehension and for-multi-targets) have the variation > 2. Four idioms (set/dict-comprehension, truth-value-test, star-in-func-call) have the variation between 1 and 2. The other three idioms (chained-comparison, loop-else, assign-multi-targets) have the variations <1. On the real-project code, the variation of performance changes of truth-value-test and for-multi-targets is > 2. For the other seven idioms, the variation of performance changes is < 1. The performance changes of three idioms (chained-comparison, loop-else, assign-multi-targets) are relatively stable on both datasets, and the performance changes of for-multi-targets vary largely on both datasets. Except for truth-value-testing, the variation of 8 idioms on real-project code is smaller than that on synthetic code.

6.4.2 RQ2: How well can code features explain the performance differences caused by Pythonic Idioms?

Motivation

Based on the literature review (Alexandru et al., 2018; Phan-udom et al., 2020; *Programming Idioms* 2022; Zhang, Xing, Xia, Xu and Zhu, 2022) and the observation of the performance-related Stack Overflow questions, we identify a set of syntactic and dynamic features (see Section 6.3.1) that may potentially affect code performance. However, it is unclear whether and how some or all these code features actually correlate with the performance differences caused by Pythonic Idioms. If the correlations exist, making them explicit would help developers determine when to use Pythonic Idioms and anticipate their effects.

Approach

To analyze the relationships between the code features and the performance differences reported in RQ1, we construct a Generalized additive model (GAM) (Hastie, 2017; Tan et al., 2022) using the package "mgcv" in R language. GAM can analyze complex nonlinear and non-monotonic relationships between the code features and the performance differences (Hastie, 2017). It also provides good interpretability. Similar to previous studies (Hastie, 2017; Tan et al., 2022), we log-transform highly skewed numeric code features and performance changes. We use the variance inflation factor (VIF) to analyze the multicollinearity of the two features (Mansfield and Helms, 1982). If the VIF of two features is larger than 5 (Belušić Vozila et al., 2015; Christ, 2009; Tan et al., 2022), there is significant multicollinearity between the two features, we randomly remove one of them.

The GAM model provides deviance explained (DE), p-values and the trend plot between the code features and performance differences which help us understand the positive or negative correlations. The DE value indicates the goodness-of-fit of model. The closer the DE is to 1, the better the model fits the data. Previous studies generally assume that the model can explain the data well if the DE value is greater than 40% (Belušić Vozila et al., 2015; Schoeman and Richardson, 2002; Tan et al., 2022). The p-value for each feature tests the null hypothesis that the feature has no correlation with the performance difference. If the p-value for a feature is less than 0.05, the feature is regarded as important (Schoeman and Richardson, 2002; Tan et al., 2022). Similar to the previous study (Caruana et al., 2015), we sort the features by the change of DE after removing one important feature. The larger the change, the more important the feature.

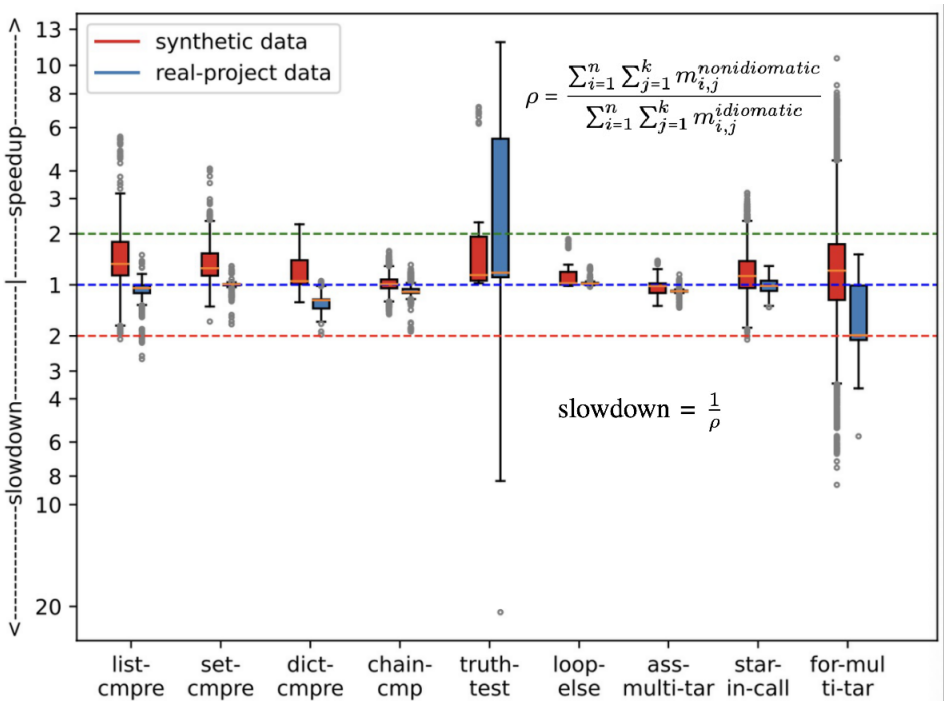


Figure 6.1: Performance Differences between Idiomatic vs. Non-idiomatic Code

Result

Table 6.4 presents our analysis results. For the synthetic code, the DE values for all nine idioms are above 40%. That is, code features can explain the performance differences caused by all nine idioms in the synthetic code. We list the important features (p-value$<$0.05) for each idiom in the descending order of importance. In contrast, only three idioms (loop-else, assign-multi-targets, for-multi-targets) have the DE values above 40% for real-project code. That is, except for these three idioms, code features alone cannot explain the performance differences caused by the other six idioms in the real-project code. For these six idioms, as feature importance becomes meaningless, their important feature cells show "-". If a feature change (value increase for num*, setting 1 for is* and has*, using a specific value in a set) incurs the larger performance speedup or smaller slowdown, we say the feature is positively correlated to the performance change (shown in blue font). Otherwise, the feature is negatively correlated to the performance change (shown in red font).

• **Analysis of Synthetic Code:** For the synthetic code, code complexity features (num* and has*) generally negatively correlate with the performance changes, while data size generally positively correlate with the performance changes. Code features specially-designed for idioms (e.g., swapping variable, assigning constant value, empty value testing) positively correlate with the performance speedup. Variable scope (local versus global) has both ways of correlations.

Code complexity: When list/set/dict-comprehension involves deeper nested loops (numFor increases) and more if-condition checking (numIf and numIfElse increases), the performance speedup by using list/set/dict-comprehension becomes smaller. For star-in-func-call, when using a subscript (has-Subscript=1), the more complex the subscript becomes (using step, lower, upper), the smaller the performance improvement becomes. For for-multi-targets, accessing more targets (numTarget increases) makes the performance speedup smaller, and using a starred node (hasStarred=1) makes slowdown even larger. There are several exceptions to this general negative correlations, including numCompop for chained-comparison, numSubscript for star-in-func-call and for-multi-targets. That is, the increase of comparisons and subscripts result in more speedup. We believe this is reasonable as these three idioms are specially designed to handle more comparisons and more subscripts efficiently.

Data size: The more elements for list/set/dict-comprehension and the more iterations for for-multi-targets (data size increases), the more performance speedup these idioms produce. However, this speedup stops or increases only marginally once the data size reaches a certain limit. Fig. 6.2 shows this observation for list/set/dict-comprehension and for-multi-targets. To see the trend clearly, we

Table 6.4: The Correlations between Code Features and Performance Differences Caused by Pythonic Idioms.

Idiom	Synthetic Code		Real-Project Code	
	DE (%)	Important Features	DE (%)	Important Features
List-Comprehension	90.3	size scope=Global numFor numIf numIfElse	28.3	-
Set-Comprehension	88.1		8.8	
Dict-Comprehension	91.2		29.3	
Chain Comparison	55.4	scope=Global numComop	19.8	-
Truth Value Test	99.6	EmptySet scope=Global	32	-
Loop Else	95.1	size isBreak=1	88.7	size isBreak=1
Assign Multi Targets	68.6	isSwap=1 scope=Global isConst=1	55.1	numAssign isConst=1
Star-in Func-Call	90.9	numSubscript scope=Global isConst=1 hasSubscript=1 hasStep=1 hasUpper=1 hasLower=1	31.9	-
For Multi Targets	88.3	numScript hasStarred=1 scope=Global numTarget size	72.8	numScript hasStarred=1 numTarget size

Note: Blue - Positive Correlation, Red - Negative Correlation

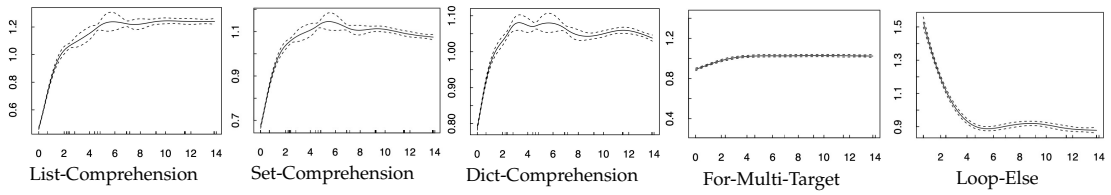


Figure 6.2: The relationship between `size` and performance changes for list/set/dict comprehension, for-multi-targets and loop-else

take the log of data size. For list-comprehension, the speedup increases fast when the number of elements increases from 1 to 55 (e^4). However, the speedup flattens when the number of elements increases over about 2981 (e^8). For the set-comprehension, the speedup increases fast when the number of elements increases from 1 to 148(e^5). However, the speedup flattens when the number of elements increases over about 2981 (e^8). For the dict-comprehension, the speedup increases fast when the number of elements increases from 1 to 20 (e^3). However, the speedup flattens when the number of elements increases over about 2981 (e^8). The impact of data size on for-multi-targets is smaller than that on list/set/dict-comprehension. In contrast, the data size for loop-size is negatively correlated with the speedup. The more iterations the loop runs, the smaller speedup the loop-else idiom produces. This is because the speedup by the special else processing accounts for smaller and smaller percentage of the overall execution as the loop runs more iterations.

Specific code features: Truth-value-test is specially designed to handle the Python pre-defined empty values in the EmptySet. So some specific empty values may produce significant speedup. For example, truth-value-testing "Fraction(0,1)" has 6 times speedup against non-idiomatic code. For other values in the EmptySet, the speedup ranges from 1.02 to 2.3 times. When assign-multi-targets is used to swap variables or to assign constant values, the idiom produces smaller slowdown than processing regular multiple variable assignments. When using the constant subscript index in star-in-func-call, the speed up is smaller than using non-constant index.

Variable scope: The impact of variable scope varies on different idioms. For list/set/dict-comprehension and chain-comparison, using global scope results in larger speedup, while for truth-value-test, assign-multi-targets, star-in-func-call and for-multi-targets, the effect is the opposite. Variable scope is not an important feature affecting loop-else.

Execution path: For the loop-else, `isBreak` positively correlate with the performance changes. Since executing the break statement means the loop run fewer iterations than not executing the break statement, the speedup when `isBreak=1` is greater than when `isBreak=0`.

• **Analysis of Real-Project Code:** For loop-else on real-project code, it important features (`size` and `isBreak=1`) and the causes of correlation are the same as loop-else on the synthetic code (seen the above discussion on data size and execution path). For assign-multi-targets on real-project code, numAssign and `isConst` are important features, which overlap only one feature with the three important features (`isSwap`, scope, `isConst`) for the idiom on synthetic code. The positive correlation of `isConst` is the same on the synthetic and real-project code. Scope variant is irrelevant to real-project code as it is always local scope. `isSwap` is not applicable to real-project code as assign-multi-targets has rarely been used to swap variables in the real-project code. For for-multi-targets, the set of important features (numSubscript, numTarget, `size`, `hasStarred`) and the impact of these features are the same on the synthetic and real-project code. Again, the scope variant is irrelevant to real-project code.

For the other six idioms (list/set/dict-comprehension, chained-comparison, truth-value-test, star-in-func-call), none of the code features have important correlations with the performance changes caused these idioms. As further studied in Section 6.4.3, many external factors (e.g., using library objects) may affect the performance change in the real-project code, other than the code features studied in the RQ2.

Table 6.5: The Statistics of Root Causes of Performance Differences

Idiom	Synthetic Code				Real-Project Code					
	#N	R1	R2	R3	#N	R1	R2	R3	R4	R5
list-comprehension	310	18.7	81.3	-	253	15.8	5.1	-	10.7	68.4
set-comprehension	310	14.8	85.2	-	282	16.1	3.1	-	32.3	48.5
dict-comprehension	310	20.3	79.7	-	194	38.2	4.6	-	3.6	53.6
chain-comparison	373	-	100	-	329	-	64.7	-	-	35.3
truth-test	336	-	-	100	381	-	-	36.2	26.8	37
loop-else	128	-	-	100	198	-	-	4.5	-	95.5
ass-multi-targets	174	50	2.5	47.5	371	16.2	-	1.6	-	82.2
star-in-call	321	-	100	-	170	-	35.1	-	-	64.9
for-multi-targets	356	24.2	-	75.8	179	67.6	-	-	3.5	28.9

6.4.3 RQ3: What are the root causes of performance differences caused by Pythonic Idioms and what causes the inconsistencies between synthetic and real-project code?

Motivation

RQ1 and RQ2 show that Pythonic Idioms have complex impact on code performance and the impact is not consistent between synthetic and real-project code. Furthermore, source code features are not sufficient to explain the performance changes in real-project code. In this RQ, we investigate how Pythonic Idioms are implemented at the bytecode level and identify the bytecode instruction differences between pairs of non-idiomatic and idiomatic code in our datasets. Analyzing these instruction differences allows us to explain the root causes of performance differences and the inconsistencies between synthetic and real-project code.

Approach

We get the bytecode instructions of idiomatic code and the corresponding non-idiomatic code for each Pythonic Idiom with dis module (*Dis module of Python* 2022). Then we compare the difference in bytecode instructions in a semi-automatic manner using code differencing tool and manual examination. For each Pythonic Idiom in the two datasets, we analyze all code pairs if the number of code pairs is less than 400. Otherwise, to make manual examination feasible, we randomly sample code pairs with a confidence level of 95% and an error margin 5% (Singh and Mangat, 2013). Two authors with more than five years Python programming experience first independently analyze the instruction differences between non-idiomatic and idiomatic code and summarize the differences into categories. Then, they discuss and reach the consensus on the final categories.

Result

Table 6.5 lists the five categories of root causes. #N represents the number of examined code instances.

Three categories (R1-R3) are shared by synthetic and real-project code, while two (R4, R5) are unique to real-project code which causes the inconsistencies between synthetic and real-project code. For

each idiom, we calculate the percentage of the examined idiomatic code instances whose performance is primarily affected by a root cause (Rx columns), resulting in either speedup or slowdown. For example, for list-comprehension, 18.7% of 310 examined idiomatic code are slower than non-idiomatic counterparts, which is primarily caused by R1, and the rest 81.3% are faster, which is primarily caused by R2.

• **R1-Pythonic Idioms add new idiom-preparation bytecode which incurs execution overhead and slows down the overall performance.** The root cause is applicable to list/set/dict comprehension, assign-multi-targets and for-multi-targets. For list/set/dict comprehension, they need to execute additional preparation instructions (green box in Fig. 6.3) before performing iteration. When there are no elements to append, the overhead of executing preparation instructions outweighs the time reduction by the special idiomatic instruction LIST_APPEND (see R2), which results in the overall slowdown performance.

For assign-multi-targets, to assign each value to the corresponding target, it needs to additionally execute the BUILD_TUPLE instruction to build a tuple and another UNPACK_SEQUENCE instruction to unpack the sequences to put values onto the stack right-to-left. Therefore, when assign-multi-targets are only to assign variables, these additional instructions slowdowns the overall performance. The more assign nodes, the higher overhead by BUILD_TUPLE and UNPACK_SEQUENCE, the slower the performance. Swapping variables and assigning constants are specially processed which can speedup the performance (see R2).

For-multi-targets needs to execute unpacking instruction and storing unpacked values for later use in the body of for statement. In real-project code, 90% of for-multi-targets access part of items, so the idiom additionally needs a Starred node to store unused items. Besides, for-multi-targets usually accesses an item few times (2∼6). Therefore, 67.6% of for-multi-targets slows down the performance in real-project code.

• **R2-Pythonic Idioms use specialized bytecode instructions to replace bytecodes of non-idiomatic codes, which could slows down or speed up overall performance.**The root cause is applicable to list/set/dict-comprehension, assign-multiple-targets and chain-comparison. List-comprehension and set-comprehension have the same mechanism. Take the list comprehension (blue box in Fig. 6.3) as an example. The non-idiomatic code executes the LOAD_FAST instruction to push data into the stack and then calls a function to append the element. In contrast, the idiomatic code only needs to execute the LIST_APPEND instruction to append the elements. Since the time of LIST_APPEND is less than that of a series of instructions especially the expensive function call in the non-idiomatic code, the time for appending elements is shortened. As the number of added elements increases, the greater the time difference between executing the idiomatic and non-idiomatic instructions becomes. When the `size` increases over a certain value, the speedup flattens because it essentially becomes the speedup ratio between the LIST_APPEND and the non-idiomatic instructions.

The dict-comprehension executes the special instruction MAP_ADD to load the dictionary iterable. Unlike the non-idiomatic code for list/set-comprehension, the non-idiomatic code for dict-comprehension does not need to execute function call. As such, MAP_ADD achieves basically the same thing as the instruction to store key-value pairs in non-idiomatic code. Therefore, the performance speedup of dict-comprehension is relatively smaller than that of list/set-comprehension.

For assign-multi-targets, when swapping two or three variables, assign-multi-targets do not execute instructions to build and unpack a tuple, but use more cheaper instructions of ROT_TWO and ROT_THREE to directly swap variables, which speed up the performance.

The chain-comparison replaces an instruction to load a comparator of non-idiomatic code with the instructions to create a reference to a comparator (DUP_TOP) and reference shuffling (ROT_THREE). Saving an extra reference to the comparator for the chain-comparison is not free, and cost more time than loading a local variable. Thus, if the comparator is a local variable, the performance decreases. Otherwise, the performance improves.

The star-in-func-call replaces instructions to load an element by index (BINARY_SUBSCR) with instructions to build a slice object and unpack the slice object into arguments. When the star-in-func-call does not have the Subscript node, it does not execute instruction to build a slice object, which cost less time than the non-idiomatic code and results in speedup. When the star-in-func-call has the Subscript node, the execution time of instructions to build and unpack a slice object is more than

that of one BINARY_SUBSCR instruction. So if the non-idiomatic code only has one Subcript node, the performance of star-in-func-call is slower. As the number of Subscript nodes of non-idiomatic code increases, the non-idiomatic code has to execute more and more BINARY_SUBSCR instructions, which takes more time than the execution time of instructions to build a slice object and unpack the slice object. In such cases, star-in-func-call results in speedup.

• **R3-Pythonic Idioms remove bytecode instructions in non-idiomatic code which speeds up the performance.** This root cause is applicable to truth-value-test, loop-else, assign-multi-targets and for-multi-targets. The truth-value-test removes the instructions for loading the object from `EmptySet` and comparison. The loop-else removes the instructions for assigning the flagging variables and comparison with the flagging variables. If assign-multi-targets is to assign constant values, it removes the instructions to load constants. If assign-multi-targets is to swap variables, it removes the instructions to load and store temporary variables. The for-multi-targets remove multiple instructions to access an element by the index.

• **R4-Non-Python built-in objects overloads Python built-in methods called by idioms, which may diminish the speedup, especially significantly affect the performance of the truth-value-test.** This root cause is applicable to list/set/dict-comprehension, truth-value-test and for-multi-targets. The relevant bytecode instructions are: iterating elements (FOR_ITER) for list/set/dict-comprehension and for-multi-targets, comparison operation (COMPARE_OP) and testing the truthiness (POP_JUMP_IF_TRUE or POP_JUMP_IF_FALSE) for truth-value-test.

FOR_ITER calls `__next__` method to get the next item. If the object type is non-Python built-in type, its overloaded `__next__` can be slow, which may diminish the speedup of idiomatic instructions. For example, for the list-comprehension: `[row for row in reader]`, the `reader` is a reader object from the CSV library. The overloaded `__next__` of `reader` cost more time than appending an element, so the bottle neck of the code is FOR_ITER. Since FOR_ITER of list comprehension and the corresponding non-idiomatic code is the same, the overall performance of idiomatic code is almost identical to the non-idiomatic code (1.008) no matter how many elements are added.

The comparison operation may be overloaded by a library. Since such overloaded comparison operation in non-idiomatic code can be costly, replacing it with truth-value-test can significantly speed up the performance. For example, the non-idiomatic code `if inter !==0` is about 11 times slower than the truth-value-test `if inter` (the maximum speedup in Fig. 6.1). `inter` is an array object of the Numpy library, which overloads the comparison operation. This overloaded comparison operation executes a series of dispatching, type conversion and wrapper object allocation. The truth-value-test does not execute this comparison operation, which results in a significant speedup.

The instruction to test the truthiness calls the `__bool__` method or `__len__` method if the object is not Boolean type. If the object is non-Python built-in data type, the implementation of `__bool__` or `__len__` may slow down the performance. For example, for the `xpath_results==[]` code, the corresponding idiomatic code is `xpath_results`. The `xpath_results` is a `HtmlElement` object from the lxml library and implements the `__bool__` method inefficiently. So when executing the truth-value-test, it costs more time than the non-idiomatic code and slowdown the performance about 20 times (the slowdown outlier in Fig. 6.1).

• **R5-Real-project code involves more complex computations (i.e., the usage of non-Python built-in object, API calls, attribute access, compound expressions) which may diminish or change the impact of Pythonic Idioms.** The root cause applies to all nine idioms as the primary root cause for the code instances ranging from 28.9% to 95.5%. R5 accounts for over 50% for five idioms (list/dict-comprehension, loop-else, assign-multi-targets and star-in-func-call)

For the list/set/dict-comprehension, loop-else, truth-value-test, chain-comparison, assign-multi-targets and for-multi-targets, the variation of performance change is more likely to be smaller due to complex computation. For example, the set comprehension `{pow(i,2,p) for i in range(p//2+1)}` calls `pow` function that diminishes the speedup of set-comprehension (ρ=1.1). As another example, for the chain-comparison `file_size>self.max_buffer_size>0` and the non-idiomatic code `self.max_buffer_size>0 and file_size>self.max_buffer_size`. The chained comparator `self.max_buffer_size` is an attribute access which is slow, but creating and shuffling reference for the chained comparator cost less time than the time of loading `self.max_buffer_size` again. It leads to slight speedup (ρ=1.1).

The star-in-func-call with complex code is more likely to slow down the performance. Since the star-in-func-call requires additional execution of unpacking the object that accesses elements than the non-idiomatic code, if the object is non-Python built-in object, the unpacking instruction needs to construct a tuple for the object, which is costly. For example, for the non-idiomatic code `func(points[0], points[1], points[2])`, the idiomatic code is `func(*points)`. If the `points` is Python built-in object (e.g., list and tuple), the idiom speeds up the performance. However, the `points` here is an array object from the Numpy library. When unpacking the `points` for the star-in-func-call, it first construct a tuple for the `points` which cost more time than accessing elements by indices three times. As a result, the idiomatic code becomes slower (ρ=0.7).

Fig. 6.3 shows the differences in bytecode instructions between non-idiomatic code and idiomatic code for nine Pythonic Idioms.

• **List-comprehension:** From the green box of List-Comprehension in Fig. 6.3, we could see that the *list-comprehension* needs to execute additional preparation instructions to load listcomp code object and then call it as function before loop iteration. From the blue box of List-Comprehension in Fig. 6.3, we could see the non-idiomatic code executes the LOAD_* instructions to push data into the stack and then calls the `append` function to append the element. In contrast, the idiomatic code only executes the LIST_APPEND instruction to append the element.

• **Set-comprehension:** Similar to the *list-comprehension*, *set-comprehension* needs to execute additional preparation instructions to load setcomp code object and then call it as function before loop iteration (green box of Set-Comprehension in Fig. 6.3). For adding elements, its non-idiomatic code needs to execute the LOAD_* instructions to push data into the stack and then call the `add` function to add the element, but the idiomatic code only needs to execute the SET_ADD instruction to append the element (blue box of Set-Comprehension in Fig. 6.3). Besides, non-idiomatic needs to call a function to initialize an empty set, but the *set-comprehension* does not need to do it (red box of Set-Comprehension in Fig. 6.3).

• **Dict-comprehension:** Similar to the *list-comprehension* and *set-comprehension*, the *dict-comprehension* needs to execute additional preparation instructions to load dictcomp code object and then call it as function before loop iteration (green area of Dict-Comprehension in Fig. 6.3). However, different from the *list-comprehension* and *set-comprehension*, for appending the element, the non-idiomatic code of the *dict-comprehension* does not need to call a function, but only needs to load the object and store a key-value pair. In contrast, the *dict-comprehension* directly executes MAP_ADD to add the element (blue box of Dict-Comprehension in Fig. 6.3).

• **Chain-comparison:** The *chain-comparison* replaces one instruction to load the chained comparator `o` of non-idiomatic code with the instructions to create a reference to the chained comparator `o` (DUP_TOP) and reference shuffling (ROT_THREE) for the three comparators (`n, o, p`) (blue box of Chain-Comparison in Fig. 6.3).

• **Truth-value-test:** From the red box of Truth-Value—Test in Fig. 6.3, we can see The *truth-value-test* removes instructions of loading the object `0` from `EmptySet` and the comparison operation `!=`.

• **Loop-else:** The non-idiomatic code determines whether the code executes a break statement by setting a different value to the flagging variable `flag`. After the for statement is executed, if the flag is True, code executes the body of the if statement. Operations flagging variables and comparison operation is not concise for developers, so the *loop-else* removes the instructions of setting flagging variables and comparison operation of non-idiomatic code (red box in Loop-Else in Fig. 6.3).

• **Assign-multi-targets:** Compared to the non-idiomatic code, the idiomatic code additionally executes the BUILD_TUPLE instruction to build a tuple and another UNPACK_SEQUENCE instruction to unpack the sequences to put values onto the stack right-to-left (green box of Assign-Multi-Targets in Fig. 6.3).

• **Star-in-func-call:** The *star-in-func-call* replaces multiple instructions to load an element by index (BINARY_SUBSCR) with instructions to build a slice object and unpack the slice object into arguments (blue box of Star-in-Func-Call in Fig. 6.3).

• **For-multi-targets:** In the body of the for statement, the *for-multi-targets* removes instructions to access an element by index (red box of For-Multi-TargetsFig. 6.3). However, the *for-multi-targets*

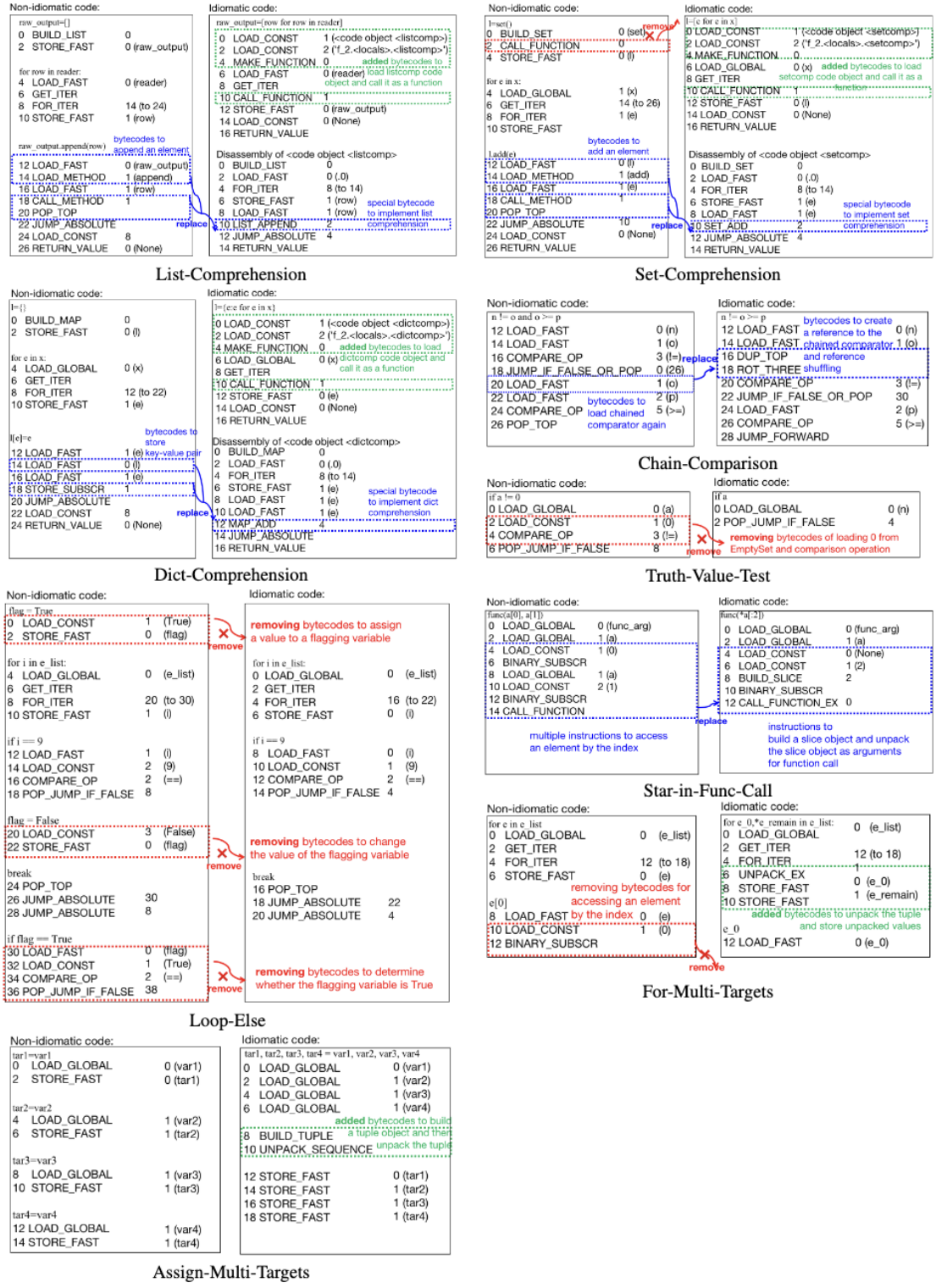


Figure 6.3: Bytecode Instructions of Pythonic Idioms (Right) and Corresponding Non-Idiomatic Codes (Left)

additionally executes instructions to unpack the object and then store the unpacked values (green box of For-Multi-TargetsFig. 6.3).

6.5 Discussion

6.5.1 Implications on Pythonic Idiom Practices

Although previous studies (Alexandru et al., 2018; Leelaprute et al., 2022; Zhang, Xing, Xia, Xu and Zhu, 2022) believe that Pythonic Idioms can improve performance, we find that developers are often confused and present contradictory opinions on the performance of nine Pythonic Idioms on Stack Overflow. Through our large-scale empirical study, we show that the outcomes of Pythonic Idioms on the synthetic code cannot be reliably transferred to the real-project code, and vice versa. This is because the the designed speedup effects of Pythonic Idioms (albeit evident in the "clean" synthetic code) diminish in the real-project code involving complex non-Python built-in objects and computations. Furthermore, even in the synthetic code, Pythonic Idioms sometimes slowdown the performance because the overhead of idiom preparation are not paid off by the reduced non-idiomatic operations.

When applying Pythonic Idioms, developers should consider code complexity, data size, variable scope and special code functions holistically. Complex code (e.g., nested loop, many if-else checking, many unpacking targets) generally diminishes the effects of Pythonic Idioms. Larger data size generally benefits more from list/set/dict-comprehension and for-multi-targets, but the speedup is capped by the time reduction ratio between the idiom-specialized processing and non-idiomatic processing. Loading global variable takes more time than loading local variable. Reducing the times of global variable loading (e.g., by the chain comparison) improves the performance, but the effect is much smaller for local variables. Special code functions (e.g., truth-value-test against the values in EmptySet, swap variables or assign constants by assign-multi-targets, loop-else) benefit from the idioms as these idioms are specially designed to handle these code functions.

When the real-project code is simple and does not involve non-Python built-in objects or complex computations (essentially similar to synthetic code), using Pythonic Idioms is generally beneficial. However, when that is not the case, developers should be cautious as Pythonic Idioms may not bring in the designed speedup. In real-project code, our results show that only truth-value-test may significantly speed up the performance, but chances are it may slow down the performance when it involves non-Python built-in objects that overload the truthiness test functions. For-multi-targets suffers from the same slowdown symptom. For other idioms, the differences between using or not using idioms are generally small. As Pythonic Idioms squeeze many code tokens in a concise (sometimes obscure) form, developers may be concerned about the readability of idiomatic code (Zhang, Xing, Xia, Xu and Zhu, 2022). It may not be worth sacrificing the readability for some trivial performance speedup or even the risk of slowdown.

We summarize the main results for Python users and the Python community in the Table 6.6 where P and R indicate whether the idiom improves performance and/or readability; "—" indicates that there is no suggestion.

6.5.2 Threats to Validity

The major **construct threat** is the performance variations due to non-determinism of code execution. To mitigate this threat, we adopt mature performance measurement methods (Georges et al., 2007a; Laaber, Scheuner et al., 2019; Traini et al., 2021), and confirm the measurement reliability by a statistical sampling method (Kalibera and Jones, 2020; Traini et al., 2021). One **internal threat** is the personal bias in analyzing the root causes of the performance changes. To reduce the bias, two authors with more than five years Python programming experience check the code instances independently, and then discuss to reach the consensus on the root causes. The other **internal threat** is the quality of our datasets. We set different parameters for code synthesis which well cover real code situations and use a large dataset of real-project code. To avoid human errors, we use a refactoring tool (Zhang, Xing, Xia, Xu and Zhu, 2022) to automatically obtain idiomatic code from non-idiomatic code. For the **external threat**, our results are limited to the nine Pythonic Idioms and CPython 3.7 interpreter. Our research method can be extended to other idioms and interpreter versions.

Table 6.6: The Implications of Pythonic Idioms for Python users and the Python community.

Idiom	We recommend Python users use the idiom (P and/or R)	We suggest the Python community may consider
List/Set/Dict-Comprehension	* when the idiomatic code length is within 79 characters **style_python**, and the code has many elements (about>1000) being added.	* optimizing the idiom-preparation instructions such as avoiding the creation and call of function objects about comprehension.
Chain-Comparison	* when chained objects are global variables, API calls, attribute accesses or compound expressions.	* improving the efficiency of the DUP_TOP instruction
Truth-Value-Test	* when the testing object is Python built-in data type (e.g., list, set and dictionary)	—
Loop-Else	* when the code needs to add a temporary variable to distinguish whether the break statement is executed or not	—
Assign-Multi-Targets	* variables swap * constant assignments	* removing BUILD_TUPLE and PACK_SEQUENCE instructions by changing orders of loading and storing multiple variables
Star-in-Func-Call	* when the function call needs to unpack all elements of an object as parameters * when the number of unpacked elements is greater than 4	* adding prediction for the <BUILD_SLICE, BINARY_SUBSCR>instruction pair
For-Multi-Targets	* when the code unpacks all elements of the iterator	* adding prediction for the <BUILD_TUPLE, UNPACK_SEQUENCE>instruction pair

6.6 Conclusion and Future Work

This chapter presents a large-scale empirical study on the performance impact of nine Pythonic idioms (list/set/dictionary comprehension, chain comparison, truth value test, loop else, assignment multiple targets, star in function calls, and for multiple targets) on both synthetic and real-project code. We systematically present the wide performance-change distributions for different idioms with large-scale empirical evidence, and show that the performance outcomes of Pythonic idioms in the synthetic code cannot be reliably transferred to the real-project code, and vice versa. Our correlation and root-cause analysis show that these discrepancies are the outcomes of the joint force exerted by not only Pythonic idiom design but also complex intrinsic and extrinsic code features. Our results help developers make a holistic consideration of these interweaving factors when using Pythonic idioms and develop a realistic understanding of the pros and cons of Pythonic idioms.

Chapter 7

Conclusions and Future Research

This chapter summarizes the research work that have been completed in the thesis, and then discuss future research direction and work.

7.1 Summary of Completed Work

In this thesis, we have presented a comprehensive investigation into the utilization and impact of Pythonic idioms on software quality. Our research has addressed critical gaps in the understanding, use, and performance impacts of Pythonic idioms within Python codebases.

7.1.1 Automated Refactoring of Non-Idiomatic Python Code

The first major contribution of this thesis is the development of RIdiom, the pioneering tool that automatically detects and refactors non-idiomatic Python code into its idiomatic form. RIdiom identifies and refactors non-idiomatic code with nine Pythonic idioms, four of which were previously unrecognized. Our tool leverages a detailed comparison of Python and Java syntax to isolate Python-specific idioms and systematically applies atomic AST rewriting operations to perform the refactoring.

The effectiveness of RIdiom was evaluated across 7,638 GitHub repositories, where it detected and refactored over 2 million instances of non-idiomatic code. The tool demonstrated high accuracy rates—100% for both detection and refactoring in most situations. Real-world validation through submitted pull requests further confirmed the tool's practical utility, with 62% of the requests receiving positive feedback from project maintainers.

7.1.2 Hybrid Knowledge-Driven Refactoring with LLMs

Addressing the limitations of purely rule-based or LLM-only approaches, we developed a hybrid methodology that combines the deterministic nature of rule-based methods with the adaptive capabilities of Large Language Models (LLMs). This method integrates Analytic Rule Interfaces (ARIs) and LLM prompts to perform precise and adaptable code idiomatization.

Our approach was tested on nine established Pythonic idioms and extended to four new ones, consistently outshining existing methods such as Prompt-LLM in terms of accuracy, F1-score, precision, and recall, all of which exceeded 90%. This hybrid method proved not only effective but also scalable, demonstrating potential for broad application across various idiomatic constructs.

7.1.3 Enhancing Comprehensibility through Equivalent Non-Idiomatic Code

Comprehension challenges associated with Pythonic idioms were tackled through the development of DeIdiom, a tool that translates idiomatic Python code into equivalent non-idiomatic code. By examining common misunderstandings reflected in Stack Overflow queries and GitHub repository usage, we developed a method for transforming complex idiomatic constructs into simpler, more universally understandable forms.

The evaluation of DeIdiom on 7,572 idiomatic code instances from real-world repositories confirmed its high accuracy, achieving detection and rewriting accuracies between 99% and 100%. A user study

with 20 participants showed a significant improvement in comprehension, with a 63.5% increase in correctness and a 22.6% reduction in completion time, highlighting the tool's effectiveness in making Pythonic idioms more accessible.

7.1.4 Performance Implications of Pythonic Idioms

The final cornerstone of this thesis was an empirical study on the performance impacts of Pythonic idioms. Using a meticulously constructed dataset of 24,126 synthetic code pairs and 54,879 real-project code pairs, we conducted a detailed performance analysis. Our findings revealed that Pythonic idioms often result in modest speedups for synthetic code but can cause slowdowns in real-project contexts due to factors like library usage, function calls, and complex computations.

This study also indicated that while code features such as code complexity, variable scope, data properties, and execution paths explain performance differences well in synthetic scenarios, they are less effective in real-world applications. We identified bytecode-level differences as key factors driving these performance discrepancies, leading to practical recommendations for developers on when and how to use Pythonic idioms for optimal performance.

Collectively, the works in this thesis bridge significant gaps in both theoretical and practical aspects of Pythonic idioms. By providing tools and insights that enhance code maintainability, correctness, and performance, we empower developers to write better Python code. Our contributions, validated through extensive real-world testing and user studies, offer a robust foundation for future research and tool development in the domain of Python programming practices.

In summary, this thesis not only advances the state-of-the-art in automated code refactoring, comprehension enhancement, and performance analysis but also provides actionable insights and tools that are readily applicable in real-world software development.

7.2 Future work

Building on the findings and methodologies of Pythonic idioms discussed throughout this thesis, several promising directions for future research emerge.

7.2.1 Expanding Idiomatic Programming Studies Across Multiple Languages

While this thesis specifically examined Pythonic idioms and their impact on software quality, including maintainability, correctness, and performance, there is significant potential for extending this research to other programming languages. The methodology used to analyze Python idioms—comparing Python and Java syntax at multiple levels of granularity—can serve as a foundation for future studies.

Future work could focus on developing an automated approach to collect and analyze idioms from a variety of programming languages such as Java, C, and C++. This expansion would enable the identification and empirical examination of a broader array of idioms, facilitating comprehensive research grounded in cross-language comparisons. Extending the methodology to other languages presents its own set of challenges, including differences in language syntax, semantics, and community best practices. For instance, Java's object-oriented nature may introduce different idiomatic patterns compared to Python's dynamic typing, while C and C++ require more careful management of memory, which could affect both the idioms used and the performance outcomes.

Moreover, our current focus on syntactic aspects of idioms can be extended to include other best practices, such as API usage patterns and general code conventions. By exploring these additional practices, we can further evaluate their impacts on software quality in terms of readability, maintainability, and performance. The core difficulty in studying and abstracting API usage patterns and code conventions across languages lies in their diversity and the context-dependent nature of these practices.

7.2.2 Software Quality Assurance Infrastructure

Building on the work detailed in Chapter 3, Chapter 4, Chapter 5, and Chapter 6, a promising direction lies in developing an integrated software quality assurance (SQA) infrastructure. This infrastructure

would encompass various dimensions of software quality, including code correctness, maintainability, and performance optimization, based on mined programming practices tailored to developers' needs.

Key areas of focus for this future work include:

- Programming Practices: Developing methods to automate the mining of beneficial programming practices, including API usage and code patterns.
- Tool Configuration: Enabling users to specify which segments of code, file names, and directories require analysis and allowing them to select programming practices to focus on.
- Code Comprehension: Creating metrics for readability, methods for explaining complex code in simpler terms, and tools for visualizing code execution step-by-step.
- Code Refactoring: Automating the refactoring processes according to specified programming practices to enhance code quality.
- Performance Reporting: Automatically generating comprehensive performance-related test cases and assessing performance impacts before and after refactoring at various levels of code granularity.
- Tool Development: Developing user-friendly and reliable tools, such as integrated development environment (IDE) plugins, continuous integration (CI) tools for platforms like GitHub, and web-based applications.

This integrated infrastructure would support a wide range of software quality enhancements, making it a valuable asset for developers.

7.2.3 Automated Compliance with Coding Standards and Regulations

Beyond automated refactoring using idioms, another crucial aspect of software quality assurance is ensuring compliance with coding standards, policies, and privacy regulations. These standards, established by organizations like Google and Airbnb, cover a wide range of domains including programming languages, user interfaces, and security protocols.

Future research could aim to automate the process of ensuring compliance with these standards through the following steps:

- Collection and Taxonomy of Standards: Gathering an extensive array of coding standards, policies, and regulations to construct a detailed taxonomy. This taxonomy would facilitate a comprehensive understanding of the standards' diversity and scope.
- Evaluation of Existing Tools: Conducting a critical assessment of existing tools that automate coding standards enforcement to identify their strengths and limitations.
- Development of a Two-Step Method ("Identify-Fix"): Designing a novel method for automated compliance that first identifies violations of standards in the code and then applies fixes. This method would streamline the compliance process, making it more efficient and less error-prone.

Bibliography

A toolkit to write, run and analyze benchmarks (2022). (cited on pp. 68, 75).

Abbes, M., Khomh, F., Gueheneuc, Y.-G. and Antoniol, G. (2011). 'An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension'. In: *2011 15Th european conference on software maintenance and reengineering*. IEEE, pp. 181–190 (cited on p. 54).

Al Dallal, J. and Abdin, A. (2017). 'Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review'. *IEEE Transactions on Software Engineering*, 44(1), pp. 44–69 (cited on p. 7).

Alcocer, J. P. S., Beck, F. and Bergel, A. (2019). 'Performance evolution matrix: Visualizing performance variations along software versions'. In: *2019 Working conference on software visualization (VISSOFT)*. IEEE, pp. 1–11 (cited on p. 9).

Alexandru, C. V., Merchante, J. J., Panichella, S., Proksch, S., Gall, H. C. and Robles, G. (2018). 'On the usage of pythonic idioms'. In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 1–11 (cited on pp. 5, 12, 16, 18, 30–33, 36, 45, 51, 67, 77, 86).

Allamanis, M., Barr, E. T., Bird, C., Devanbu, P., Marron, M. and Sutton, C. (2016). 'Mining semantic loop idioms from Big Code'. In: *Technical Report* (cited on p. 6).

Allamanis, M. and Sutton, C. (2014). 'Mining idioms from source code'. In: *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, pp. 472–483 (cited on p. 6).

Arcelli, D., Cortellessa, V. and Trubiani, C. (2015). 'Performance-based software model refactoring in fuzzy contexts'. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer, pp. 149–164 (cited on p. 8).

Arif, A. and Rana, Z. A. (2020). 'Refactoring of code to remove technical debt and reduce maintenance effort'. In: *2020 14th International Conference on Open Source Systems and Technologies (ICOSST)*. IEEE, pp. 1–7 (cited on p. 54).

Assign-Multi-Targets (2022a). (cited on p. 63).

Assign-Multi-Targets (2022b). (cited on p. 63).

Bader, D. (2017). *Python Tricks: A Buffet of Awesome Python Features*. BookBaby. ISBN: 9781775093312. (cited on pp. 18, 31, 47).

Barany, G. (2014). 'Python interpreter performance deconstructed'. In: *Proceedings of the Workshop on Dynamic Languages and Applications*, pp. 1–9 (cited on p. 73).

Beazley, D. and Jones, B. (2013). *Python Cookbook: 3rd Edition*. O'Reilly Media, Incorporated. ISBN: 9781449340377. (cited on p. 31).

Belušić Vozila, A., Bulić, I. and Klaic, Z. (2015). 'Using a generalized additive model to quantify the influence of local meteorology on air quality in Zagreb'. *Geofizika*, 32, p. 47. DOI: 10.15233/gfz.2015.32.5 (cited on p. 77).

Bhattacharjee, A., Roy, B. and Schneider, K. A. (2022). 'Supporting program comprehension by generating abstract code summary tree'. In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, pp. 81–85. ISBN: 9781450392242. DOI: 10.1145/3510455.3512793 (cited on p. 9).

Bloch, J. (2008). *Effective java (the java series)*. Prentice Hall PTR (cited on p. 5).

Bosch, J. (1996). *Design Patterns as Language Constucts* (cited on p. 5).

Brown, T. B. et al. (2020). *Language Models are Few-Shot Learners* (cited on pp. 10, 32, 39).

Brun, Y., Lin, T., Somerville, J. E., Myers, E. M. and Ebner, N. (2023). 'Blindspots in Python and Java APIs Result in Vulnerable Code'. *ACM Trans. Softw. Eng. Methodol.*, 32(3). DOI: 10.1145/3571850 (cited on p. 9).

C Sharp and Java: Comparing Programming Languages (2022). (cited on p. 19).

Caruana, R., Lou, Y., Gehrke, J., Koch, P., Sturm, M. and Elhadad, N. (2015). 'Intelligible Models for HealthCare: Predicting Pneumonia Risk and Hospital 30-Day Readmission'. In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '15. Sydney, NSW, Australia: Association for Computing Machinery, pp. 1721–1730. ISBN: 9781450336642. DOI: 10.1145/2783258.2788613 (cited on p. 77).

Chang, Y. et al. (2023). 'A Survey on Evaluation of Large Language Models'. *arXiv e-prints*, arXiv:2307.03109, arXiv:2307.03109. DOI: 10.48550/arXiv.2307.03109 (cited on p. 10).

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G. and Chen, W. (2022). *CodeT: Code Generation with Generated Tests* (cited on p. 10).

Chen, J., Yu, D., Hu, H., Li, Z. and Hu, H. (2019). 'Analyzing performance-aware code changes in software development process'. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, pp. 300–310 (cited on pp. 9, 68).

Chen, J., Shang, W. and Shihab, E. (2020). 'PerfJIT: Test-level just-in-time prediction for performance regression introducing commits'. *IEEE Transactions on Software Engineering* (cited on pp. 9, 10, 68, 70, 74).

Chen, W.-K., Liu, C.-H. and Li, B.-H. (2018). 'A Feature Envy Detection Method Based on Dataflow Analysis'. In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 02, pp. 14–19. DOI: 10.1109/COMPSAC.2018.10196 (cited on p. 7).

Christ, A. (2009). 'Mixed Effects Models and Extensions in Ecology with R'. *Journal of Statistical Software, Book Reviews*, 32(1), pp. 1–3. DOI: 10.18637/jss.v032.b01 (cited on p. 77).

Cinnéide, M. Ó. (2000). 'Automated refactoring to introduce design patterns'. In: *Proceedings of the 22nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: Association for Computing Machinery, pp. 722–724. ISBN: 1581132069. DOI: 10.1145/337180.337612 (cited on p. 8).

Code., Q. (2014). *The Little Book of Python Anti-Patterns*. (cited on pp. 18, 31).

Comparison of C Sharp and Java (2022). (cited on p. 19).

Corbi, T. A. (1989). 'Program understanding: Challenge for the 1990s'. *IBM Systems Journal*, 28(2), pp. 294–306. DOI: 10.1147/sj.282.0294 (cited on pp. 9, 47).

Costa, D., Bezemer, C.-P., Leitner, P. and Andrzejak, A. (2019). 'What's wrong with my benchmark results? studying bad practices in JMH benchmarks'. *IEEE Transactions on Software Engineering*, 47(7), pp. 1452–1467 (cited on p. 73).

Crapé, A. and Eeckhout, L. (2020). 'A rigorous benchmarking and performance analysis methodology for python workloads'. In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, pp. 83–93 (cited on pp. 68, 74).

Creswell, A., Shanahan, M. and Higgins, I. (2022). *Selection-Inference: Exploiting Large Language Models for Interpretable Logical Reasoning*. DOI: 10.48550/arXiv.2205.09712 (cited on p. 10).

Cwalina, K. and Abrams, B. (2008). *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable. NET Libraries*. Pearson Education (cited on p. 5).

D'Ambros, M., Bacchelli, A. and Lanza, M. (2010). 'On the impact of design flaws on software defects'. In: *2010 10th International Conference on Quality Software*. IEEE, pp. 23–31 (cited on p. 54).

Daly, D., Brown, W., Ingo, H., O'Leary, J. and Bradford, D. (2020). 'The use of change point detection to identify software performance regressions in a continuous integration system'. In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pp. 67–75 (cited on p. 9).

David, C., Kesseli, P. and Kroening, D. (2017). 'Kayak: Safe semantic refactoring to java streams'. *arXiv preprint arXiv:1712.07388* (cited on pp. 6, 8).

Davison, A. C. and Hinkley, D. V. (1997). *Bootstrap methods and their application*. 1. Cambridge university press (cited on pp. 68, 75).

Demeyer, S. (2005). 'Refactor conditionals into polymorphism: what's the performance cost of introducing virtual calls?' In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, pp. 627–630 (cited on p. 9).

Developers, J. (2020). *OpenJDK: Java Microbenchmark Harness*. ttps://openjdk.java.net/projects/code-tools/jmh/. Accessed Dec. 19, 2018 (cited on pp. 68, 73).

Developers, P. (2013). *The grammar of the star idiom*. (cited on p. 20).

Developers, P. (2015). *The performance of the list-comprehension*. (cited on p. 54).

Developers, P. (2022a). *Python Performane Tips*. https://pyperf.readthedocs.io. Accessed March. 10th, 2022 (cited on p. 73).

Developers, P. (2022b). *The definition of logical lines of Python program*. (cited on p. 53).

Developers, P. (2023). *The grammar of the truth-value-test*. (cited on p. 20).

developers, P. (2000). *Python Enhancement Proposals*. (cited on pp. 30, 47).

Dilhara, M., Bellur, A., Bryksin, T. and Dig, D. (2024). 'Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example'. *ArXiv*, abs/2402.07138. (cited on pp. 8, 10).

Dilhara, M., Dig, D. and Ketkar, A. (2023). 'PYEVOLVE: Automating Frequent Code Changes in Python ML Systems'. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, pp. 995–1007 (cited on pp. 8, 42).

Dilhara, M., Ketkar, A., Sannidhi, N. and Dig, D. (2022). 'Discovering Repetitive Code Changes in Python ML Systems'. In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 736–748. DOI: 10.1145/3510003.3510225 (cited on pp. 7, 19).

Ding, Z., Chen, J. and Shang, W. (2020). 'Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet?' In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, pp. 1435–1446 (cited on p. 74).

Dis module of Python (2022). (cited on p. 81).

Dong, Y., Jiang, X., Jin, Z. and Li, G. (2023). *Self-collaboration Code Generation via ChatGPT* (cited on p. 31).

Du Bois, B., Demeyer, S., Verelst, J., Mens, T. and Temmerman, M. (2006). 'Does god class decomposition affect comprehensibility?' In: *IASTED Conf. on software engineering*, pp. 346–355 (cited on p. 54).

Farooq, A. and Zaytsev, V. (2021). 'There is More than One Way to Zen Your Python'. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, pp. 68–82 (cited on pp. 5, 13, 18, 30, 33, 36, 39, 45, 46).

Feng, S. and Chen, C. (2023). *Prompting Is All You Need: Automated Android Bug Replay with Large Language Models* (cited on pp. 10, 11, 32, 39).

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional (cited on pp. 6, 13, 24).

Franklin, L., Gyori, A., Lahoda, J. and Dig, D. (2013). 'LAMBDAFICATOR: from imperative to functional programming through automated refactoring'. In: *2013 35th international conference on software engineering (ICSE)*. IEEE, pp. 1287–1290 (cited on pp. 6, 8).

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L. and Lewis, M. (2023). *InCoder: A Generative Model for Code Infilling and Synthesis* (cited on p. 31).

Fu, M., Tantithamthavorn, C., Le, T., Nguyen, V. and Phung, D. (2022). 'VulRepair: A T5-Based Automated Software Vulnerability Repair'. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022.

Singapore, Singapore: Association for Computing Machinery, pp. 935–947. ISBN: 9781450394130. DOI: 10.1145/3540250.3549098 (cited on p. 10).

Gao, Z., Xia, X., Lo, D., Grundy, J. and Zimmermann, T. (2021). 'Automating the removal of obsolete TODO comments'. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 218–229 (cited on p. 29).

Georges, A., Buytaert, D. and Eeckhout, L. (2007a). 'Statistically rigorous java performance evaluation'. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, pp. 57–76. ISBN: 9781595937865. DOI: 10.1145/1297027.1297033 (cited on pp. 9, 68, 74, 86).

Georges, A., Buytaert, D. and Eeckhout, L. (2007b). 'Statistically rigorous java performance evaluation'. *ACM SIGPLAN Notices*, 42(10), pp. 57–76 (cited on p. 74).

Gopstein, D., Iannacone, J., Yan, Y., DeLong, L., Zhuang, Y., Yeh, M. K.-C. and Cappos, J. (2017). 'Understanding Misunderstandings in Source Code'. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, pp. 129–139. ISBN: 9781450351058. DOI: 10.1145/3106237.3106264 (cited on p. 9).

Gosling, J., Holmes, D. C. and Arnold, K. (2005). *The Java programming language* (cited on p. 19).

GPT (2023). (cited on p. 42).

Gupta, A., Suri, B., Kumar, V., Misra, S., Blažauskas, T. and Damaševičius, R. (2018). 'Software Code Smell Prediction Model Using Shannon, Rényi and Tsallis Entropies'. *Entropy*, 20(5). DOI: 10.3390/e20050372 (cited on p. 7).

Hastie, T. J. (2017). 'Generalized additive models'. In: *Statistical models in S*. Routledge, pp. 249–307 (cited on p. 77).

He, S., Liu, T., Lama, P., Lee, J., Kim, I. K. and Wang, W. (2021). 'Performance Testing for Cloud Computing with Dependent Data Bootstrapping'. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 666–678. DOI: 10.1109/ASE51524.2021.9678687 (cited on p. 9).

He, S., Manns, G., Saunders, J., Wang, W., Pollock, L. and Soffa, M. L. (2019). 'A statistics-based performance testing methodology for cloud applications'. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, pp. 188–199. ISBN: 9781450355728. DOI: 10.1145/3338906.3338912 (cited on p. 9).

Hecht, G., Moha, N. and Rouvoy, R. (2016). 'An empirical study of the performance impacts of android code smells'. In: *Proceedings of the international conference on mobile software engineering and systems*, pp. 59–69 (cited on p. 54).

Hettinger, R. (2013). *Transforming code into beautiful, idiomatic Python*. (cited on pp. 12, 13, 18, 31, 36, 47, 54).

Huang, P., Ma, X., Shen, D. and Zhou, Y. (2014). 'Performance Regression Testing Target Prioritization via Performance Risk Analysis'. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, pp. 60–71. ISBN: 9781450327565. DOI: 10.1145/2568225.2568232 (cited on pp. 9, 70).

Huang, Q., Yuan, Z., Xing, Z., Xu, X., Zhu, L. and Lu, Q. (2023). 'Prompt-Tuned Code Language Model as a Neural Knowledge Base for Type Inference in Statically-Typed Partial Code'. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery. ISBN: 9781450394758. DOI: 10.1145/3551349.3556912 (cited on pp. 10, 31).

Huang, Q., Zhu, J., Xing, Z., Jin, H., Wang, C. and Xu, X. (2023). *A Chain of AI-based Solutions for Resolving FQNs and Fixing Syntax Errors in Partial Code* (cited on p. 31).

Huang, Q., Zou, Z., Xing, Z., Zuo, Z., Xu, X. and Lu, Q. (2023). *AI Chain on Large Language Model for Unsupervised Control Flow Graph Generation for Statically-Typed Partial Code* (cited on pp. 10, 31, 32, 39).

Hunt, J. (2019). 'PyTest Testing Framework'. In: *Advanced Guide to Python 3 Programming*. Springer, pp. 175–186 (cited on p. 25).

Introducing ChatGPT (2023). (cited on p. 40).

Ismail, M. and Suh, G. E. (2018). 'Quantitative overhead analysis for Python'. In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, pp. 36–47 (cited on p. 73).

Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S. and Sharma, R. (2022). 'Jigsaw: Large Language Models Meet Program Synthesis'. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, pp. 1219–1231. ISBN: 9781450392211. DOI: 10.1145/3510003.3510203 (cited on p. 10).

Jin, G., Song, L., Shi, X., Scherpelz, J. and Lu, S. (2012). 'Understanding and detecting real-world performance bugs'. *ACM SIGPLAN Notices*, 47(6), pp. 77–88 (cited on p. 9).

Kalibera, T. and Jones, R. (2013). 'Rigorous benchmarking in reasonable time'. In: *Proceedings of the 2013 international symposium on memory management*, pp. 63–74 (cited on pp. 9, 75).

Kalibera, T. and Jones, R. (2020). 'Quantifying performance changes with effect size confidence intervals'. *arXiv preprint arXiv:2007.10899* (cited on pp. 9, 68, 75, 86).

Kaur, A. and Kaur, M. (2016). 'Analysis of code refactoring impact on software quality'. In: *MATEC Web of Conferences*. Vol. 57. EDP Sciences, p. 02012 (cited on p. 7).

Kim, M., Zimmermann, T. and Nagappan, N. (2012). 'A field study of refactoring challenges and benefits'. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: Association for Computing Machinery. ISBN: 9781450316149. DOI: 10.1145/2393596.2393655 (cited on p. 6).

Knupp, J. (2013). *Writing Idiomatic Python 3.3*. Jeff Knupp (cited on pp. 5, 12, 13, 18, 20, 31, 36, 47, 51).

Köhler, M. and Salvaneschi, G. (2019). 'Automated refactoring to reactive programming'. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 835–846 (cited on pp. 6, 8).

Kuhlman, D. (2009). *A python book: Beginning python, advanced python, and python exercises*. Dave Kuhlman Lutz (cited on p. 19).

Laaber, C. and Leitner, P. (2018). 'An evaluation of open-source software microbenchmark suites for continuous performance assessment'. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, pp. 119–130 (cited on p. 74).

Laaber, C., Scheuner, J. and Leitner, P. (2019). 'Software microbenchmarking in the cloud. How bad is it really?' *Empirical Software Engineering*, 24(4), pp. 2469–2508 (cited on pp. 68, 74, 86).

Laaber, C., Würsten, S., Gall, H. C. and Leitner, P. (2020). 'Dynamically reconfiguring software microbenchmarks: reducing execution time without sacrificing result quality'. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, pp. 989–1001. ISBN: 9781450370431. DOI: 10.1145/3368089.3409683 (cited on p. 9).

Langer, A. (2001). 'Java programming idioms'. In: *Technology of Object-Oriented Languages, International Conference on*. Vol. 1. IEEE Computer Society, pp. 197–197 (cited on p. 5).

Lanza, M., Ducasse, S., Gall, H. and Pinzger, M. (2005). 'CodeCrawler - an information visualization tool for program comprehension'. In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* Pp. 672–673. DOI: 10.1109/ICSE.2005.1553647 (cited on p. 9).

Leelaprute, P., Chinthanet, B., Wattanakriengkrai, S., Kula, R. G., Jaisri, P. and Ishio, T. (2022). 'Does coding in pythonic zen peak performance? preliminary experiments of nine pythonic idioms at scale'. In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pp. 575–579 (cited on pp. 10, 32, 54, 67, 86).

Lemieux, C., Inala, J. P., Lahiri, S. K. and Sen, S. (2023). 'CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models'. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 919–931. DOI: 10.1109/ICSE48619.2023.00085 (cited on p. 10).

Li, L., Wang, J. and Quan, H. (2022). 'Scalpel: The python static analysis framework'. *arXiv preprint arXiv:2202.11840* (cited on p. 64).

Li, Y., Choi, D. et al. (2022). 'Competition-level code generation with AlphaCode'. *Science*, 378(6624), pp. 1092–1097. DOI: 10.1126/science.abq1158 (cited on p. 10).

Lin, B., Nagy, C., Bavota, G. and Lanza, M. (2019). 'On the impact of refactoring operations on code naturalness'. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 594–598 (cited on p. 7).

Liu, Z., Chen, C., Wang, J., Che, X., Huang, Y., Hu, J. and Wang, Q. (2023). 'Fill in the Blank: Context-Aware Automated Text Input Generation for Mobile GUI Testing'. In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, pp. 1355–1367. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00119 (cited on p. 10).

Luo, Q., Poshyvanyk, D. and Grechanik, M. (2016). 'Mining performance regression inducing code changes in evolving software'. In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, pp. 25–36 (cited on p. 9).

Macia Bertran, I., Garcia, A. and Staa, A. von (2011). 'An exploratory study of code smells in evolving aspect-oriented systems'. In: *Proceedings of the tenth international conference on Aspect-oriented software development*, pp. 203–214 (cited on p. 54).

Mansfield, E. R. and Helms, B. P. (1982). 'Detecting multicollinearity'. *The American Statistician*, 36(3a), pp. 158–160 (cited on p. 77).

Martelli, A., Ravenscroft, A. and Ascher, D. (2005). *Python cookbook*. " O'Reilly Media, Inc." (cited on p. 47).

Mayrhauser, A. von and Vans, A. (1993). 'From program comprehension to tool requirements for an industrial environment'. In: *[1993] IEEE Second Workshop on Program Comprehension*, pp. 78–86. DOI: 10.1109/WPC.1993.263903 (cited on p. 8).

Merchante, J. J. and Robles, G. (2017). 'From Python to Pythonic: Searching for Python idioms in GitHub'. In: *Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution*, pp. 1–3 (cited on pp. 12, 13, 36).

Mészáros, M., Cserép, M. and Fekete, A. (2019). 'Delivering comprehension features into source code editors through LSP'. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, pp. 1581–1586 (cited on p. 9).

Midolo, A. and Tramontana, E. (2021). 'Refactoring Java Loops to Streams Automatically'. In: *2021 4th International Conference on Computer Science and Software Engineering (CSSE 2021)*, pp. 135–139 (cited on p. 6).

Mumtaz, H., Beck, F. and Weiskopf, D. (2018). 'Detecting Bad Smells in Software Systems with Linked Multivariate Visualizations'. In: *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 12–20. DOI: 10.1109/VISSOFT.2018.00010 (cited on p. 7).

Nguyen, T. H., Adams, B., Jiang, Z. M., Hassan, A. E., Nasser, M. and Flora, P. (2012). 'Automated detection of performance regressions using statistical process control techniques'. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pp. 299–310 (cited on p. 9).

Nguyen, T. H., Nagappan, M., Hassan, A. E., Nasser, M. and Flora, P. (2014). 'An industrial case study of automatically identifying performance regression-causes'. In: *Proceedings of the 11th working conference on mining software repositories*, pp. 232–241 (cited on p. 9).

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S. and Xiong, C. (2022). 'CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis'. In: *International Conference on Learning Representations*. (cited on p. 31).

OpenAI (2023). *GPT-4 Technical Report* (cited on pp. 10, 32, 39, 45).

OpenAI Codex (2023). (cited on pp. 10, 31).

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K. and Deb, K. (2016). 'Multi-criteria code refactoring using search-based software engineering: An industrial case study'. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3), pp. 1–53 (cited on p. 8).

Paixão, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J. and Arvonio, E. (2020). 'Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review'. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. Seoul, Republic of Korea: Association for Computing Machinery, pp. 125–136. ISBN: 9781450375177. DOI: 10.1145/3379597.3387475 (cited on p. 7).

Pan, M., Huang, A., Wang, G., Zhang, T. and Li, X. (2020). 'Reinforcement learning based curiosity-driven testing of Android applications'. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 153–164 (cited on p. 29).

Peldszus, S., Kulcsár, G., Lochau, M. and Schulze, S. (2016). 'Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching'. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE '16. Singapore, Singapore: Association for Computing Machinery, pp. 578–589. ISBN: 9781450338455. DOI: 10.1145/2970276.2970338 (cited on p. 7).

Peng, Y., Wang, C., Wang, W., Gao, C. and Lyu, M. R. (2023). *Generative Type Inference for Python* (cited on pp. 10, 11, 31, 32, 39).

PEP 202-List comprehension (2023). (cited on p. 19).

PEP 274-Dict comprehension (2023). (cited on p. 19).

PEP 448-Additional Unpacking Generalizations (2023). (cited on p. 23).

Perlis, A. J. and Rugaber, S. (1979). 'Programming with idioms in APL'. *ACM SIGAPL APL Quote Quad*, 9(4-P1), pp. 232–235 (cited on p. 5).

Phan-udom, P., Wattanakul, N., Sakulniwat, T., Ragkhitwetsagul, C., Sunetnanta, T., Choetkiertikul, M. and Kula, R. G. (2020). 'Teddy: Automatic Recommendation of Pythonic Idiom Usage For Pull-Based Software Projects'. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 806–809 (cited on pp. 5, 6, 13, 31, 47, 77).

Programming Idioms (2022). (cited on pp. 12, 36, 77).

Pylint (2022). (cited on pp. 6, 13, 47, 48, 54).

Python Abstract Grammar (2022). (cited on pp. 19, 70).

Python programming FAQ (2022). (cited on p. 73).

Radoi, C., Fink, S. J., Rabbah, R. and Sridharan, M. (2014). 'Translating imperative code to MapReduce'. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pp. 909–927 (cited on pp. 6, 8).

Reitz, K. and Schlusser, T. (2016). *The Hitchhiker's guide to Python: best practices for development*. " O'Reilly Media, Inc." (cited on pp. 12, 47).

Ren, S., Lai, H., Tong, W., Aminzadeh, M., Hou, X. and Lai, S. (2010). 'Nonparametric bootstrapping for hierarchical data'. *Journal of Applied Statistics*, 37(9), pp. 1487–1498 (cited on pp. 68, 75).

Ren, X., Ye, X., Zhao, D., Xing, Z. and Yang, X. (2023). 'From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining'. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 976–987. DOI: 10.1109/ASE56229.2023.00143 (cited on pp. 10, 11, 32).

Replication Package (2023). (cited on p. 32).

RIdiom (2022). (cited on p. 47).

Rodriguez-Cancio, M., Combemale, B. and Baudry, B. (2016). 'Automatic microbenchmark generation to prevent dead code elimination and constant folding'. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 132–143 (cited on p. 73).

Romano, S. and Scanniello, G. (2018). 'Exploring the Use of Rapid Type Analysis for Detecting the Dead Method Smell in Java Code'. In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 167–174. DOI: 10.1109/SEAA.2018.00035 (cited on p. 7).

Sackman, H., Erikson, W. J. and Grant, E. E. (1968). 'Exploratory experimental studies comparing online and offline programming performance'. *Commun. ACM*, 11(1), pp. 3–11. DOI: 10.1145/362851.362858 (cited on p. 8).

Sakulniwat, T., Kula, R. G., Ragkhitwetsagul, C., Choetkiertikul, M., Sunetnanta, T., Wang, D., Ishio, T. and Matsumoto, K. (2019). 'Visualizing the usage of pythonic idioms over time: A case study of the with open idiom'. In: *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, pp. 43–435 (cited on p. 44).

Scheaffer, R. L., Mendenhall III, W., Ott, R. L. and Gerow, K. G. (2011). *Elementary survey sampling*. Cengage Learning (cited on p. 48).

Schoeman, D. and Richardson, A. (2002). 'Investigating biotic and abiotic factors affecting the recruitment of an intertidal clam on an exposed sandy beach using a generalized additive model'. *Journal of Experimental Marine Biology and Ecology*, 276(1), pp. 67–81. DOI: https://doi.org/10.1016/S0022-0981(02)00239-3 (cited on p. 77).

Shaft, T. M. and Vessey, I. (1995). 'Research Report—The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension'. *Information Systems Research*, 6(3), pp. 286–299. DOI: 10.1287/isre.6.3.286 (cited on p. 8).

Shao, J. and Wang, Y. (2003). 'A new measure of software complexity based on cognitive weights'. In: *CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No.03CH37436)*. Vol. 2, 1333–1338 vol.2. DOI: 10.1109/CCECE.2003.1226146 (cited on p. 9).

Shin, J., Tang, C., Mohati, T., Nayebi, M., Wang, S. and Hemmati, H. (2023). 'Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks'. *arXiv e-prints*, arXiv:2310.10508, arXiv:2310.10508. DOI: 10.48550/arXiv.2310.10508 (cited on p. 10).

Shneiderman, B. (1976). 'Exploratory experiments in programmer behavior'. *International Journal of Computer & Information Sciences*, 5, pp. 123–143 (cited on p. 8).

Siegmund, J. (2016). 'Program Comprehension: Past, Present, and Future'. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5, pp. 13–20. DOI: 10.1109/SANER.2016.35 (cited on pp. 8, 9).

Silva, D. and Valente, M. T. (2017). 'RefDiff: Detecting refactorings in version histories'. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, pp. 269–279 (cited on p. 42).

Singh, R. and Mangat, N. S. (2013). *Elements of survey sampling*. Vol. 15. Springer Science & Business Media (cited on p. 81).

Sivaraman, A., Abreu, R., Scott, A., Akomolede, T. and Chandra, S. (2022). 'Mining idioms in the wild'. In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pp. 187–196 (cited on pp. 6, 13).

Slatkin, B. (2020). *Effective Python : 90 specific ways to write better Python / Brett Slatkin.* Second edition. Effective software development series. Place of publication not identified: Addison-Wesley. ISBN: 0-13-485471-3 (cited on pp. 12, 13, 30, 31, 47).

Soloway, E. and Ehrlich, K. (1984). 'Empirical Studies of Programming Knowledge'. *IEEE Transactions on Software Engineering*, SE-10(5), pp. 595–609. DOI: 10.1109/TSE.1984.5010283 (cited on p. 9).

Song, L. and Lu, S. (2017). 'Performance diagnosis for inefficient loops'. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, pp. 370–380 (cited on pp. 9, 70).

Spencer, D. (2009). *Card sorting: Designing usable categories*. Rosenfeld Media (cited on p. 52).

Stack Overflow (2022). (cited on pp. 48, 69).

Stefan, P., Horky, V., Bulej, L. and Tuma, P. (2017). 'Unit testing performance in java projects: Are we there yet?' In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pp. 401–412 (cited on p. 74).

Tan, X., Gao, K., Zhou, M. and Zhang, L. (2022). 'An exploratory study of deep learning supply chain'. In: *Proceedings of the 44th International Conference on Software Engineering*, pp. 86–98 (cited on p. 77).

The Explanation of Chain Comparison (2022a). (cited on p. 54).

The Explanation of Chain Comparison (2022b). (cited on p. 54).

Top Programming Languages (2023). (cited on p. 18).

Touvron, H. et al. (2023). *Llama 2: Open Foundation and Fine-Tuned Chat Models* (cited on p. 45).

Traini, L., Di Pompeo, D., Tucci, M., Lin, B., Scalabrino, S., Bavota, G., Lanza, M., Oliveto, R. and Cortellessa, V. (2021). 'How Software Refactoring Impacts Execution Time'. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2), pp. 1–23 (cited on pp. 10, 68, 74, 75, 86).

Trubiani, C., Bran, A., Hoorn, A. van, Avritzer, A. and Knoche, H. (2018). 'Exploiting load testing and profiling for performance antipattern detection'. *Information and Software Technology*, 95, pp. 329–345 (cited on p. 9).

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D. and Dig, D. (2018). 'Accurate and efficient refactoring detection in commit history'. In: *Proceedings of the 40th international conference on software engineering*, pp. 483–494 (cited on pp. 7, 42).

Tsantalis, N., Mazinanian, D. and Rostami, S. (2017). 'Clone refactoring with lambda expressions'. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, pp. 60–70 (cited on p. 8).

Vaithilingam, P., Zhang, T. and Glassman, E. L. (2022). 'Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models'. In: *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022, Extended Abstracts*. Ed. by S. D. J. Barbosa, C. Lampe, C. Appert and D. A. Shamma. ACM, 332:1–332:7. DOI: 10.1145/3491101.3519665 (cited on p. 31).

Viera, A. J., Garrett, J. M. et al. (2005). 'Understanding interobserver agreement: the kappa statistic'. *Fam med*, 37(5), pp. 360–363 (cited on pp. 41, 44, 49).

Wang, J., Li, L., Liu, K. and Cai, H. (2020). 'Exploring how deprecated python library apis are (not) handled'. In: *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 233–244 (cited on pp. 25, 58).

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S. and Wang, Q. (2024). *Software Testing with Large Language Models: Survey, Landscape, and Vision* (cited on p. 10).

Wang, Y., Yu, H., Zhu, Z., Zhang, W. and Zhao, Y. (2018). 'Automatic Software Refactoring via Weighted Clustering in Method-Level Networks'. *IEEE Transactions on Software Engineering*, 44(3), pp. 202–236. DOI: 10.1109/TSE.2017.2679752 (cited on p. 8).

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q. and Zhou, D. (2023). *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models* (cited on p. 10).

When and why need a constant? (2022). (cited on p. 73).

Why does (1 in [1,0] == True) evaluate to False? (2012). (cited on p. 48).

Wilcoxon, F. (1945). 'Individual Comparisons by Ranking Methods'. *Biometrics*, 1, pp. 196–202 (cited on p. 61).

Wu, T., Terry, M. and Cai, C. J. (2022). *AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts* (cited on p. 10).

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E. and Li, S. (2018). 'Measuring Program Comprehension: A Large-Scale Field Study with Professionals'. *IEEE Transactions on Software Engineering*, 44(10), pp. 951–976. DOI: 10.1109/TSE.2017.2734091 (cited on pp. 8, 9, 48).

Yang, K., Peng, N., Tian, Y. and Klein, D. (2022). 'Re3: Generating Longer Stories With Recursive Reprompting and Revision'. In: *Conference on Empirical Methods in Natural Language Processing*. (cited on p. 10).

Yu, S. and Zhou, S. (2010). 'A survey on metric of software complexity'. In: *2010 2nd IEEE International Conference on Information Management and Engineering*, pp. 352–356. DOI: 10.1109/ICIME.2010.5477581 (cited on p. 9).

Zelkowitz, M. V., Shaw, A. C. and Gannon, J. D. (1979). *Principles of Software Engineering and Design*. Prentice Hall Professional Technical Reference. ISBN: 013710202X (cited on pp. 9, 48).

Zhang, N., Huang, Q., Xia, X., Zou, Y., Lo, D. and Xing, Z. (2020). 'Chatbot4qr: Interactive query refinement for technical question retrieval'. *IEEE Transactions on Software Engineering* (cited on p. 29).

Zhang, Z., Xing, Z., Xia, X., Xu, X. and Zhu, L. (2022). '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. New York, NY, USA: Association for Computing Machinery, pp. 696–708. ISBN: 9781450394130. DOI: 10.1145/3540250.3549143 (cited on pp. 3, 5, 6, 12, 30–32, 36, 39, 41, 42, 45, 47–49, 51, 55, 58, 62, 67–70, 74, 76, 77, 86).

Zhang, Z., Xing, Z., Xia, X., Xu, X., Zhu, L. and Lu, Q. (2023). 'Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence'. In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, pp. 1495–1507. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00130 (cited on pp. 4, 32, 33, 45, 47, 67).

Zhang, Z., Xing, Z., Xiao, X., Lu, Q. and Xu, X. (2024). 'Refactoring to Pythonic Idioms: A Hybrid Knowledge-Driven Approach Leveraging Large Language Models'. In: *2024 ACM International Conference on the Foundations of Software Engineering (FSE '24)*. FSE 2024. , Brazil, Brazil, , Association for Computing Machinery. DOI: 10.1145/3597503.3639101 (cited on pp. 3, 30).

Zhang, Z., Xing, Z., Xu, X. and Zhu, L. (2023a). 'RIdiom: Automatically Refactoring Non-Idiomatic Python Code with Pythonic Idioms'. In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 102–106. DOI: 10.1109/ICSE-Companion58688.2023.00034 (cited on p. 3).

Zhang, Z., Xing, Z., Xu, X. and Zhu, L. (2023b). 'RIdiom: Automatically Refactoring Non-Idiomatic Python Code with Pythonic Idioms'. In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, pp. 102–106 (cited on pp. 5, 6, 31, 46, 47).

Zhang, Z., Xing, Z., Zhao, D., Lu, Q., Xu, X. and Zhu, L. (2024). 'Hard to Read and Understand Pythonic Idioms? DeIdiom and Explain Them in Non-Idiomatic Equivalent Code'. In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ICSE 2024. , Lisbon, Portugal, Association for Computing Machinery. ISBN: 979-8-4007-0217-4/24/04. DOI: 10.1145/3597503.3639101 (cited on pp. 4, 47).

Zheng, Z., Ning, K., Chen, J., Wang, Y., Chen, W., Guo, L. and Wang, W. (2023). *Towards an Understanding of Large Language Models in Software Engineering Tasks* (cited on p. 10).