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

Zejun Zhang*
Australian National University & CSIRO's Data61
Canberra, Australia
zejun.zhang@anu.edu.au

Zhenchang Xing
CSIRO's Data61 & Australian National University
Canberra, Australia
zhenchang.xing@data61.csiro.au

Dehai Zhao
CSIRO's Data61
Sydney, Australia
dehai.zhao@data61.csiro.au

Qinghua Lu
CSIRO's Data61
Sydney, Australia
qinghua.lu@data61.csiro.au

Xiwei Xu
CSIRO's Data61
Sydney, Australia
xiwei.xu@data61.csiro.au

Liming Zhu
CSIRO's Data61
Sydney, Australia
liming.zhu@data61.csiro.au

ABSTRACT

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.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Pythonic Idioms, Code Transformation, Program Comprehension

*Corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639101

ACM Reference Format:
Zejun Zhang, Zhenchang Xing, Dehai Zhao, Qinghua Lu, Xiwei Xu, and Liming Zhu. 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), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639101

1 INTRODUCTION

Pythonic idioms are highly valued by the Python community [18]. Python programming books and renowned Python developers promote the usage of pythonic idiom for its benefits such as versatility, conciseness and improved performance [14, 26, 30, 32, 36, 41]. Some research has aimed to detect non-idiomatic code and recommend the use of corresponding pythonic idioms [8, 35, 50]. Zhang et al. [50, 52] developed a refactoring tool [9] for detecting and refactoring non-idiomatic code into idiomatic code for nine pythonic idioms. Furthermore, they find Python developers are concerned about the performance impact of pythonic idioms and the impact of these idioms on performance varies greatly [51].

However, there is no research for explaining pythonic idioms to enhance the correct and in-depth comprehension of pythonic idioms effectively. Program comprehension plays a crucial role in software maintenance, demanding a significant amount of time and effort [16, 47, 49]. It is the foundation of software engineering tasks, such as bug fixing, enhancement and reuse [47]. This paper focuses on explaining *nine pythonic idioms* (i.e., list/set/dict-comprehension, chain-comparison, truth-value-test, loop-else, assign-multi-targets, for-multi-targets and star) with exclusive Python syntax not present in Java that are identified by Zhang et al. [50]. This is because their uniqueness in Python may pose more challenges for programmers to comprehend them effectively and correctly.

Through analyzing questions of nine pythonic idioms on Stack Overflow [10] (see Section 2.1), we find that Python users exhibit unfamiliarity with the syntax of pythonic idioms and even misinterpretations or incorrect assumptions about the behavior of the nine pythonic idioms. Furthermore, pythonic idiom usage is frequent in the GitHub repositories (see Section 2.2). By analyzing their usage using a statistical sampling method [38], we not only find usage of pythonic idioms has various concise manifestation and may even exist potential negative effects such as decreased

performance, bugs, code redundancy and readability issues (see Section 2.4). For example, a developer wrote a code "table_name in row is False" in chain-comparison. He/she assumed that the code is the same as "(table_name in row) is False", but it is the same as the non-idiomatic code "table_name in row and row is False". The row is an iterable and it cannot be "False", so the condition will always be False and lead to a bug. Such misunderstanding was also reflected in Stack Overflow questions [4]. As an example of loop-else, Python users sometimes use else clause after a for statement without a break statement, such else clause is superfluous which leads to code redundancy [8]. Table 3 shows the percentage of such usage of loop-else is 17.8% (100%-82.2%). The results indicate the need for a tool that can enhance developers' understanding of the meaning and conciseness manifestation of pythonic idioms, as well as make them more aware of the potential negative effects of idiomatic usage.

Instead of explaining the idiomatic code of pythonic idioms with natural language, we explain them with the corresponding non-idiomatic code, which can avoid ambiguity and ensures semantic accuracy of the explanations. Meanwhile, since the nine pythonic idioms are unique syntax in Python, we use common Python syntax instead of alternative APIs to explain during the process of transforming idiomatic code of pythonic idioms into the corresponding non-idiomatic code. In this way, even developers with limited experience in Python development can understand pythonic idioms effectively in their corresponding common syntax.

To transform idiomatic code of nine pythonic idioms into corresponding non-idiomatic code, our approach follows a two-step process: a detection step and a rewriting step. Since each pythonic idiom is associated with a unique AST node, characterized by distinct syntactic properties, we detect idiomatic code by examining the Abstract Syntax Tree (AST) nodes and their corresponding syntactic properties. For the rewriting step, we need to consider the necessity to creating additional statements, new variables, function declarations based on the conciseness manifestation of the nine pythonic idioms in Section 2.3. Then we design transforming rules based on four atomic operations (i.e., create, insert, replace and remove) to automatically rewrite idiomatic code into the corresponding non-idiomatic code.

We apply our approach to detect and transform 1,708,831 idiomatic code instances of nine pythonic idioms from 7,577 GitHub repositories. We verify the results of 7,572 idiomatic code instances based on test cases and code review similar to Zhang et al. [50]. Our approach achieves 100% detection accuracy and 99%~100% rewriting accuracy for nine pythonic idioms. To study if our tool can help Python users understand the nine pythonic idioms correctly and efficiently, we recruit 20 students to conduct a user study with 27 multiple-choice questions based on 27 randomly selected idiomatic code instances from 27 GitHub repositories for nine pythonic idioms. The experimental group who is given the explanatory non-idiomatic code that our tool produces for the idiomatic code has 63.5% improvement of correctness than the control group with only the idiomatic code, and the completion time was accelerated by 22.6%. We find that the explanatory non-idiomatic code given by our tool helps build confidence in pythonic idiom usage and promote Python users' appreciation of the community effort on designing and developing pythonic idioms.

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

- To the best of our knowledge, our study is the first to systematically investigate the readability of pythonic idioms.
- We develop the first tool, DeIdiom, to detect and transform idiomatic code into non-idiomatic code for nine unique pythonic idioms to explain pythonic idiom usage. We also provide a web application for demonstration and use in real-case scenarios. Please visit http://35.77.46.3:5000.
- Our evaluation confirms the high accuracy of our approach, and our user study demonstrates the effectiveness and usefulness of DeIdiom for Python users to understand pythonic idioms correctly and efficiently.
- A set of suggestions for Python developers to use DeIdiom and for researchers to conduct further exploration of pythonic idioms.

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?

2.1 RQ1: Challenges in Understanding Pythonic Idioms

2.1.1 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.

2.1.2 Approach. Zhang et al. [50] 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 [44] 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.

2.1.3 Result. We summarize two types of challenges in understanding the pythonic idioms. Table 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 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 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.

(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 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 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.

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

Table 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? [2] **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? [5] 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? [3] 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? [6] 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

Our analysis of idiom-related Stack Overflow questions reveals that Python users are often confused with and even misunderstand unusual syntax and subtle semantics of pythonic idioms.

2.2 RQ2: Pythonic Idiom Usage in the Wild

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

2.2.2 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 et al. [50] 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 4) to identify the idiomatic code instances for the nine idioms.

2.2.3 Results. Table 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%. On average 3-15 files in a repository and 1-8 idiomatic code instances for nine

Table 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

pythonic idioms. The frequent usage of pythonic idioms shows their prevalence. Such widespread utilization of pythonic idioms highlights their appeal in Python developers.

The prevalence of idioms in real projects implies the importance of correct understanding of idioms for Python developers.

2.3 RQ3: Conciseness of Pythonic Idioms

2.3.1 Motivation. Although many researches [12, 26, 50] 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.

2.3.2 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 card sorting approach [43]. 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 2.2.2 (sample size is in the *N* column of Table 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 2.1.2. The Cohen's kappa agreement is 0.75 (substantial agreement).

2.3.3 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

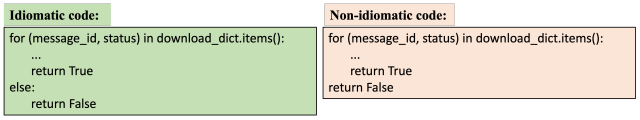


Figure 1: The idiomatic code and the corresponding non-idiomatic code of loop-else idiom

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 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 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.

• **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, 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.

• **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 [21]. 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.

• **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

Table 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%

avoids the creation of the "tmp" to make a backup for "data[-4:]" so that "the_hash" gets the old value of "data[-4:]".

• **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, 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.

Pythonic idioms achieve conciseness through both lexical changes and the introduction of additional variables, code lines, and functions. It indicates that mapping the resulting syntactic and semantic conciseness of pythonic idioms back to corresponding common program constructs in non-idiomatic code is not straightforward.

2.4 RQ4: Potential Negative Effects Caused by Idiom Usage

2.4.1 Motivation. After understanding the readability challenges (Section 2.1), frequent usage (Section 2.2) and conciseness manifestation (Section 2.3) of pythonic idioms, we are interested in exploring whether their usage may cause potential negative effects.

2.4.2 Approach. Previous studies investigated the effects of code smells on different maintainability related aspects such as performance [25, 28], redundancy [8, 13, 31], bug [17, 30] and readability [11, 22, 30]. 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 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.

2.4.3 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 [20].

• **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 [1, 7]. 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) [19] 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.

Some uses of pythonic idiom usage can cause performance degradation, code redundancy, bugs and readability issues. It indicates that helping developers notice these negative effects of pythonic idioms is important.

3 EXPLAIN PYTHONIC IDIOMS WITH NON-IDIOMATIC CODE

In Section 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.

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 et al. [50] 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.

Table 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 ∉ {Compare, Call, BoolOp}
Loop Else	P ∈ {For, While} and P.orelse ≠ ∅
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.

Furthermore, such corresponding non-idiomatic code offers other potential bonuses (detailed discussion is presented in Section 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 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. 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 et al. [50], 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 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), 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.

Due to space limitation, we list details of transforming idiomatic code into non-idiomatic code for list/set/dic-comprehension, chain-comparison and truth-value-test below. Details of the other four idioms are in of our APPENDIX A.

• **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 the `ListComp`, `SetComp` and `DictComp` nodes for the three idioms (1st row of Table 4). Next, we determine whether to create functions or temporary variables to transform idiomatic code into non-idiomatic code. For example, the idiomatic code P of the 1st row of Table 5 corresponds to the `DictComp` node whose parent node is an `Assign` node. 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). Since the $UndefinedVars = \emptyset$ (line 1), the corresponding non-idiomatic code does not need to create a function. 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).

• **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 4). To transform 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).

• **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 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. Otherwise, we will check whether the object defines a __bool__ method or a __len__ method. If neither condition is met, the object is considered True. Given this 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 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).

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?

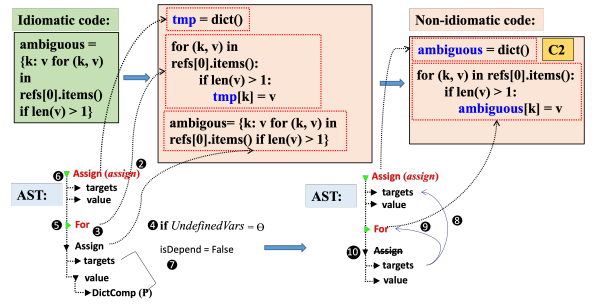
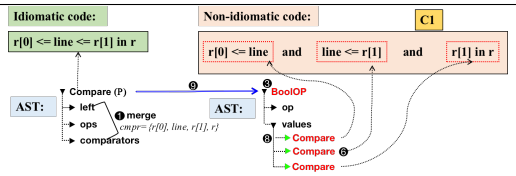
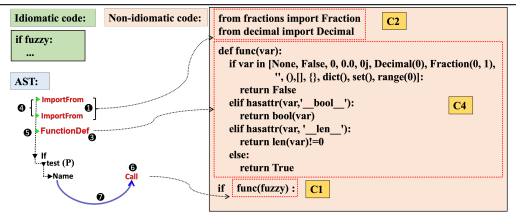
4.1 RQ1: Accuracy of Explaining Pythonic Idioms

4.1.1 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.

4.1.2 Approach. Similar to Zhang et al. [50], 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 (see in Table 2), we first use DLocator [45] to 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 the libraries required by the projects. As a result,

Table 5: Rules of transforming 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, $call$)

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

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 transformation is correct. Otherwise, if the test cases pass before code transformation but fail after transforming, we think the code transforming is wrong.

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.

Table 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

4.1.3 Result. Table 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 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.

Our approach achieves 100% detection accuracy and >99% rewriting accuracy on real-world code, confirming its reliability in rewriting idiomatic code into non-idiomatic code for nine pythonic idioms.

4.2 RQ2: Usefulness of Explaining Pythonic Idioms

4.2.1 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.

4.2.2 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 2, 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.

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 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 3 gives an example of the chain-comparison provided to the experimental group with

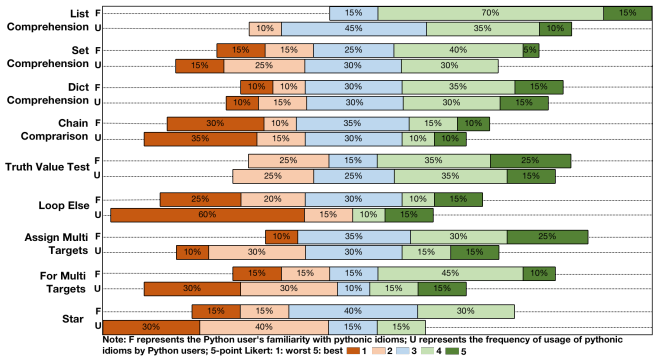


Figure 2: The participants' knowledge of 9 pythonic idioms

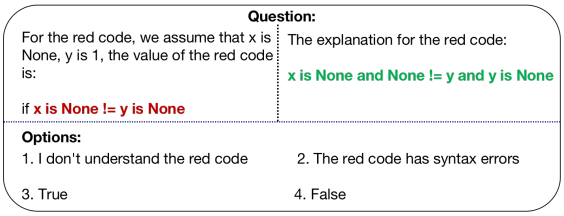


Figure 3: The question with idiomatic code and explanatory non-idiomatic code for the chain-comparison idiom

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 [46] to determine if the performance difference of 27 questions between the

Table 7: Performance Comparison

Idioms	N	Correctness			Time (s)		
		G1	G2	Impr (%)	G1	G2	Impr (%)
List-Compreh	3	0.80	1	25	47.9	38.6	19.4
Set-Compreh	3	0.63	0.97	52.6	51.7	34.3	33.7
Dict-Compreh	3	0.57	0.93	64.7	79.0	67.9	14.0
Chain-Compar	3	0.40	0.9	125	44.2	27.8	37.3
Truth-Val-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-Tar	3	0.60	0.87	44.4	50.0	47.1	5.8
Assign-Mul-Tar	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}		

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.

4.2.3 Performance Comparison and Analysis. Table 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 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.

Due to space limitation, we analyze the results for list/set/dic-comprehension and truth-value-test below. Discussions of the other five idioms are in APPENDIX B.

• **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 et al. [50], list-comprehension is much more frequently used than set/dict-comprehension. Our prior idiom knowledge survey in Figure 2 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 to 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). 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): "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.

4.2.4 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.*"

Providing explanatory non-idiomatic code for pythonic idioms can speed up code understanding and improve understanding correctness. Furthermore, it creates positive attitudes towards using pythonic idioms and prompts the appreciation of the Python community's effort to design and promote pythonic idioms.

5 DISCUSSION

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 [29].

Section 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 2.1 and Section 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 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 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.

5.2 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.

6 RELATED WORK

Studies on pythonic idioms: Pythonic idioms have been a popular topic [12, 23, 32, 33, 35, 37, 50]. Alexandru et al. [12] and Farooq et al. [23] collected pythonic idioms involving built-in methods, APIs and syntax from popular Python books. Phan-udom et al. [35] recommended pythonic code examples by searching similar code examples from 58 non-idiomatic code instances and 55 idiomatic code instances. Pattara et al. [28] explored time and memory effects of nine pythonic idioms. Zhang et al. [50] identified nine pythonic idioms by comparing the syntax differences between Python and Java, and then automatically recommended idiomatic code of nine pythonic idioms for non-idiomatic code. Different from previous works, we not only conduct an empirical study to investigate challenges in pythonic idiom comprehension, usage in real projects, conciseness manifestation, and potential negative effects, but also develop DeIdiom, a tool that explains idiomatic code as non-idiomatic code to mitigate misunderstandings and negative effects associated with pythonic idiom usage.

Studies on program comprehension: Program comprehension accounts for over 50% of the time allocated to software maintenance [16, 47, 49]. Many researches used different approaches (e.g., think-aloud protocols, memorization and comprehension tasks) to measure program comprehension [39, 40, 42, 47, 48]. Gopstein et al. [24] summarized code patterns that can lead to a significantly increased rate of misunderstanding versus equivalent code without the patterns. Brun et al. [15] found blindspots in Python and Java APIs result in vulnerable code and suggested to develop tools to recognize blindspots in APIs. Meszaros et al. [34] utilized LSP to improve the code comprehension experience inside code editors. Lanza et al. [27] developed CodeCrawler to visualize object-oriented software for program comprehension. Different from previous works, we are the first to explain pythonic idioms and can automatically transform the idiomatic code into the corresponding non-idiomatic code for program comprehension of nine pythonic idioms.

7 CONCLUSION AND FUTURE WORK

This paper 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.

REFERENCES

[1] 2008. *Hidden features of Python (The Explanation of Chain Comparison).* https://stackoverflow.com/questions/101268/hidden-features-of-python

[2] 2010. *Expanding tuples into arguments.* https://stackoverflow.com/questions/1993727/expanding-tuples-into-arguments

[3] 2011. *How do chained assignments work.* https://stackoverflow.com/questions/7601823/how-do-chained-assignments-work

[4] 2012. *Why does (1 in [1,0] == True) evaluate to False?* https://stackoverflow.com/questions/9284350/why-does-1-in-1-0-true-evaluate-to-false?

[5] 2013. *Explanation of how nested list comprehension works.* https://stackoverflow.com/questions/20639180/explanation-of-how-nested-list-comprehension-works

[6] 2017. *Set comprehension gives unhashable type set of list in Python.* https://stackoverflow.com/questions/42363826/set-comprehension-gives-unhashable-type-set-of-list-in-python

[7] 2019. *Strange chained comparison.* https://stackoverflow.com/questions/58084423/strange-chained-comparison

[8] 2022. *Pylint.* https://pylint.readthedocs.io/en/latest/

[9] 2022. *RIdiom.* https://plugins.jetbrains.com/plugin/20107-ridiom

[10] 2022. *Stack Overflow.* https://stackoverflow.com/help/searching

[11] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 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*. 181–190. https://doi.org/10.1109/CSMR.2011.24

[12] Carol V. Alexandru, José J. Merchante, Sebastiano Panichella, Sebastian Proksch, Harald C. Gall, and Gregorio Robles. 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* (Boston, MA, USA) *(Onward! 2018)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3276954.3276960

[13] Arooj Arif and Zeeshan Ali Rana. 2020. Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort. In *2020 14th International Conference on Open Source Systems and Technologies (ICOSST)*. 1–7. https://doi.org/10.1109/ICOSST51357.2020.9332917

[14] Dan Bader. 2017. *Python Tricks: A Buffet of Awesome Python Features.* BookBaby.

[15] Yuriy Brun, Tian Lin, Jessie Elise Somerville, Elisha M. Myers, and Natalie Ebner. 2023. Blindspots in Python and Java APIs Result in Vulnerable Code. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 76 (apr 2023), 31 pages. https://doi.org/10.1145/3571850

[16] T. A. Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306. https://doi.org/10.1147/sj.282.0294

[17] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. 2010. On the Impact of Design Flaws on Software Defects. In *Quality Software, International Conference on*. IEEE Computer Society, Los Alamitos, CA, USA, 23–31. https://doi.org/10.1109/QSIC.2010.58

[18] Python developers. 2000. *Python Enhancement Proposals.* https://peps.python.org/pep-0000/

[19] Python developers. 2001. *Python Enhancement Proposal 8 (PEP8).* https://peps.python.org/pep-0008/

[20] Python Developers. 2014. *The performance of the list-comprehension.* https://stackoverflow.com/questions/22108488/are-list-comprehensions-and-functional-functions-faster-than-for-loops

[21] Python Developers. 2022. *The definition of logical lines of Python program.* ttps://docs.python.org/3/reference/lexical_analysis.html#logical-lines

[22] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. 2006. Does god class decomposition affect comprehensibility?. In *IASTED Conf. on software engineering*. 346–355.

[23] Aamir Farooq and Vadim Zaytsev. 2021. There is More than One Way to Zen Your Python *(SLE 2021)*. Association for Computing Machinery, New York, NY, USA, 68–82. https://doi.org/10.1145/3486608.3486909

[24] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery,

New York, NY, USA, 129–139. https://doi.org/10.1145/3106237.3106264
[25] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2016. An Empirical Study of the Performance Impacts of Android Code Smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems* (Austin, Texas) *(MOBILESoft '16)*. Association for Computing Machinery, New York, NY, USA, 59–69. https://doi.org/10.1145/2897073.2897100
[26] Jeff Knupp. 2013. *Writing Idiomatic Python 3.3*. Jeff Knupp.
[27] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. 2005. CodeCrawler - an information visualization tool for program comprehension. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 672–673. https://doi.org/10.1109/ICSE.2005.1553647
[28] Pattara Leelaprute, Bodin Chinthanet, Supatsara Wattanakriengkrai, Raula Gaikovina Kula, Pongchai Jaisri, and Takashi Ishio. 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* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 575–579. https://doi.org/10.1145/3524610.3527879
[29] Li Li, Jiawei Wang, and Haowei Quan. 2022. Scalpel: The python static analysis framework. *arXiv preprint arXiv:2202.11840* (2022).
[30] Constantine Lignos. 2019. *Anti-Patterns in Python Programming*. https://lignos.org/py_antipatterns/
[31] Isela Macia Bertran, Alessandro Garcia, and Arndt von Staa. 2011. An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development* (Porto de Galinhas, Brazil) *(AOSD '11)*. Association for Computing Machinery, New York, NY, USA, 203–214. https://doi.org/10.1145/1960275.1960300
[32] Alex Martelli, Anna Ravenscroft, and David Ascher. 2005. *Python cookbook*. O'Reilly Media, Inc.
[33] José Javier Merchante and Gregorio Robles. 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*. 1–3.
[34] Mónika Mészáros, Máté Cserép, and Anett Fekete. 2019. Delivering comprehension features into source code editors through LSP. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1581–1586. https://doi.org/10.23919/MIPRO.2019.8756695
[35] P. Phan-udom, N. Wattanakul, T. Sakulniwat, C. Ragkhitwetsagul, T. Sunetnanta, M. Choetkiertikul, and R. Kula. 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 Computer Society, Los Alamitos, CA, USA, 806–809. https://doi.org/10.1109/ICSME46990.2020.00098
[36] Kenneth Reitz and Tanya Schlusser. 2016. *The Hitchhiker's guide to Python: best practices for development*. O'Reilly Media, Inc.
[37] Tattiya Sakulniwat, Raula Gaikovina Kula, Chaiyong Ragkhitwetsagul, Morakot Choetkiertikul, Thanwadee Sunetnanta, Dong Wang, Takashi Ishio, and Kenichi Matsumoto. 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, 43–435. https://doi.org/10.1109/IWESEP49350.2019.00016
[38] Richard L Scheaffer, William Mendenhall III, R Lyman Ott, and Kenneth G Gerow. 2011. *Elementary survey sampling*. Cengage Learning.
[39] Jingqiu Shao and Yingxu Wang. 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. https://doi.org/10.1109/CCECE.2003.1226146
[40] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 13–20. https://doi.org/10.1109/SANER.2016.35
[41] Brett Slatkin. 2019. *Effective python: 90 specific ways to write better python*. Addison-Wesley Professional.
[42] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 595–609. https://doi.org/10.1109/TSE.1984.5010283
[43] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
[44] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.
[45] Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/3368089.3409735
[46] Frank. Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics* 1 (1945), 196–202.
[47] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (2018), 951–976. https://doi.org/10.1109/TSE.2017.2734091
[48] Sheng Yu and Shijie Zhou. 2010. A survey on metric of software complexity. In *2010 2nd IEEE International Conference on Information Management and Engineering*. 352–356. https://doi.org/10.1109/ICIME.2010.5477581
[49] Marvin V. Zelkowitz, Alan C. Shaw, and John D. Gannon. 1979. *Principles of Software Engineering and Design*. Prentice Hall Professional Technical Reference.
[50] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 696–708. https://doi.org/10.1145/3540250.3549143
[51] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1495–1507. https://doi.org/10.1109/ICSE48619.2023.00130
[52] Zejun Zhang, Zhenchang Xing, Xiwei Xu, and Liming Zhu. 2023. RIdiom: Automatically Refactoring Non-Idiomatic Python Code with Pythonic Idioms. In *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 102–106. https://doi.org/10.1109/ICSE-Companion58688.2023.00034