Making Python Code Idiomatic by Automatic Refactoring
Non-idiomatic Python Code with Pythonic Idioms

Zejun Zhang
Australian National University
Australia
zejun.zhang@anu.edu.au

Zhenchang Xing
Data61, CSIRO
Australia
zhenchang.xing@anu.edu.au

Xin Xia∗
Software Engineering Application Technology Lab, Huawei
China
xin.xia@acm.org

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

Liming Zhu
Data61, CSIRO
Australia
Liming.Zhu@data61.csiro.au

ABSTRACT

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. These evaluations confirm the high-accuracy, practicality and usefulness of our refactoring tool on real-world Python code.

CCS CONCEPTS

• Software and its engineering → Software evolution.

KEYWORDS

Pythonic Idioms, Abstract Syntax Grammar, Code Refactoring

∗Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

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 (ESEC/FSE ’22), November 14–18, 2022, Singapore, Singapore. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549143

1 INTRODUCTION

Programming (or code) idioms are widely present in programming languages [17]. They represent notable programming styles and features of a programming language. Python is well known for its pythonic idioms [46, 60]. Many books and online materials [29, 41, 44, 46, 54, 57] promote the use of pythonic idioms for not only concise coding styles but also improved performance (see Table 1 for examples). 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 4 and the analysis in Section 2.2).

Due to these challenges in using pythonic idioms, developers may implement a functionality in a non-idiomatic way without using pythonic idioms. Table 1 shows some examples. We study 7,638 Python projects on GitHub (see Section 2.2) 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 2). Non-idiomatic code and idiomatic code co-exist in many repositories, files or even methods. As seen in Table 1, non-idiomatic code syntax is similar to those of 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 in Table 1, Pylint [1] 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”\footnote{https://github.com/PyCQA/pylint/issues/5800}. 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. Teddy [51] 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 Table 1). 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 paper, 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 work [36, 46, 56] 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 (see Table 1), 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 [41, 44, 57]. 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) [37], 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 refactoring 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 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, this paper makes the following contributions:

- 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.

2 FORMATIVE STUDY AND MOTIVATION

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

**RQ1:** What are the benefits of pythonic idioms?

**RQ2:** What are the coding practices concerning pythonic idioms and anti-idiom code smells?

**RQ3:** What are the challenges for writing idiomatic code?

### 2.1 RQ1: The Benefits of Pythonic Idioms

Compared with other mainstream programming languages, Python supports more idioms which are highly valued by Python developers [29]. By referring to several resources (i.e., books, presentations and websites) about pythonic idioms [13, 21, 29, 41, 44, 46, 54, 57], we summarize the key benefits of pythonic idioms: conciseness (i.e., fewer lines or fewer tokens) and performance. To help readers understand these two benefits, we summarize in Table 1 the examples of idiomatic code and the corresponding non-idiomatic code (i.e., anti-idiom code smell) for each pythonic idiom excerpted from the reference resources. We also excerpt relevant performance descriptions. We omit conciseness related descriptions as code conciseness can be observed intuitively from code examples.

To confirm the performance benefit, we use the `timeit` package [23] to record the execution time of idiomatic and non-idiomatic code snippets shown in Table 1. We execute each code snippet three times repeatedly and take the average execution time. We divide the execution time of non-idiomatic code by that of corresponding idiomatic code which indicates how much speedup idiomatic code has compared with non-idiomatic code. As shown in the fifth column of Table 1, idiomatic code has about 1.09~2.07x speedup. We acknowledge this is only an anecdotal experiment. Many online resources [10, 29] argue pythonic idioms (e.g., list comprehension) may achieve significant performance advantages over non-idiomatic code, but they generally do not provide specific empirical evidences. This calls for more systematic investigation of such performance claims which is beyond the scope of this work.

### 2.2 RQ2: Python Coding Practices

#### 2.2.1 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
We then detect the occurrence of idiomatic code and the anti-idiom we extract the test node corresponding to an object. For chain-refactoring with pythonic idioms.

For multi-targets, 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.

Table 1: Pythoric idioms: conciseness and performance

| Idiom            | Repository | File | Method | %  |
|------------------|------------|------|--------|----|
| List Comprehension | NPy | 15.20 | 49.59 | 0.01 |
| Set Comprehension | NPy | 15.20 | 49.59 | 0.01 |
| Dict Comprehension | NPy | 15.20 | 49.59 | 0.01 |
| Chain Comparison | NPy | 15.20 | 49.59 | 0.01 |
| Truth Value Test | NPy | 15.20 | 49.59 | 0.01 |
| Loop Else | NPy | 15.20 | 49.59 | 0.01 |
| Assign Multiple Targets | NPy | 15.20 | 49.59 | 0.01 |
| Star in Func Call | NPy | 15.20 | 49.59 | 0.01 |
| For Multiple Targets | NPy | 15.20 | 49.59 | 0.01 |

Table 2: Python coding practices with respect to pythonic idioms and anti-idiom code smells

| Idiom            | Repository | File | Method | %  |
|------------------|------------|------|--------|----|
| List Comprehension | NPy | 15.20 | 49.59 | 0.01 |
| Set Comprehension | NPy | 15.20 | 49.59 | 0.01 |
| Dict Comprehension | NPy | 15.20 | 49.59 | 0.01 |
| Chain Comparison | NPy | 15.20 | 49.59 | 0.01 |
| Truth Value Test | NPy | 15.20 | 49.59 | 0.01 |
| Loop Else | NPy | 15.20 | 49.59 | 0.01 |
| Assign Multiple Targets | NPy | 15.20 | 49.59 | 0.01 |
| Star in Func Call | NPy | 15.20 | 49.59 | 0.01 |
| For Multiple Targets | NPy | 15.20 | 49.59 | 0.01 |

Figure 1: The usage of pythonic idioms and anti-idiom code smells in repositories, files and methods

Table 3: Statistics of pythonic idioms and anti-idiom code smells at the statement level

| Idiom            | Non-idiomatic | Idiomatic | Sum | %  |
|------------------|---------------|-----------|-----|----|
| List Comprehension | NPy | 15.20 | 49.59 | 0.01 |
| Set Comprehension | NPy | 15.20 | 49.59 | 0.01 |
| Dict Comprehension | NPy | 15.20 | 49.59 | 0.01 |
| Chain Comparison | NPy | 15.20 | 49.59 | 0.01 |
| Truth Value Test | NPy | 15.20 | 49.59 | 0.01 |
| Loop Else | NPy | 15.20 | 49.59 | 0.01 |
| Assign Multiple Targets | NPy | 15.20 | 49.59 | 0.01 |
| Star in Func Call | NPy | 15.20 | 49.59 | 0.01 |
| For Multiple Targets | NPy | 15.20 | 49.59 | 0.01 |

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 ids can be detected by analyzing abstract syntax trees (ASTs). List/set/dict-comprehension and loop-elide 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. 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 targets of for node with elements greater than 1, respectively. For non-idiomatic code, we use our detection rules (see Section 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.

2.2.2 Result. Figure 1 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.

Table 2 summarizes the number of repositories, files and methods containing non-idiomatic code smells (the NPy column), pythonic code (the PyId column) and both non-idiomatic and idiomatic code (the Sum column) 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-multipletargets have high mix usage in methods.

Table 3 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 in making Python code idiomatic.
more than 349K and 127K times. Although the percentage of non-
idiotic code is low, there are still large numbers of non-idiomatic code
fragments (26,510 and 8,490) that can be refactored with the
list-comprehension idiom and the for-multi-targets idiom respec-
tively. In contrast, chain-comparison and assign-multi-targets have
high non-idiomatic percentages (66.7% and 96.5% respectively). Ex-
pression 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

2.3 RQ3: 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 chal-
genques in using pythonic idioms in the discussions. We summarize
three key challenges. Table 4 shows some representative examples.
Many questions have very high view counts which indicate com-
mon 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 mean-
ing 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 4),
the developer knows list comprehension but he/she does not know
whether he/she can initialize dictionary in a similar way. The ques-
tion 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 [29], which brings challenges for developers
to learn and write idiomatic Python code.

(2) Developers know certain pythonic idioms but they do
not understand what the idioms can do. Consider the question
for using asterisk operator “*” in the function call (the second last
row in Table 4) 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 4 shows such an example. The developer wants to refactor a
list initialization 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 4: Challenges in writing idiomatic Python code

| Idiom                  | #C  | Question                                                                 |
|------------------------|-----|--------------------------------------------------------------------------|
| List Comprehension     | (3) | Questions: Here is the code I was trying to turn into a list comprehe-
|                        |     | nism... Is there a way to add the else statement to this comprehension?|
|                        |     | 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 1,000 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 1,000 times                 |
| Dict Comprehension     | (1) | Question: For using asterisk operator * in function calls, use the same
|                        |     | as discussed below.                                                     |
|                        |     | 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 1,000 times                 |
| Chain Comparison       | (1) | Question: Does Python have something like an empty string?               |
|                        |     | Answer: No, Python does not have an empty string.                       |
|                        |     | Asked 7 years; Active 2 years ago; Viewed 1,086 times                  |
| Truth Value Test       | (1) | Question: What is the meaning of the “truth value” in Python?            |
|                        |     | Answer: The “truth value” of a boolean expression is 1 for True and 0 for
|                        |     | False.                                                                  |
|                        |     | Asked 9 years; Active 4 years ago; Viewed 245k times                   |
| Loop Else              | (2) | Question: Why does Python use `else` after for and while loops?          |
|                        |     | Answer: The `else` clause provides a default value after the `for` or `while`
|                        |     | loop completes.                                                        |
|                        |     | Asked 9 years; Active 2 months ago; Viewed 245k times                   |
| Assign Multiple Targets| (1) | Question: Python assigns variables on one line.                         |
|                        |     | Answer: Yes, Python assigns variables on one line.                      |
|                        |     | Asked 8 years; Active 4 years ago; Viewed 19k times                    |
| Star as Func Call      | (2) | Question: What does the * operator mean in Python, such as in code similar
|                        |     | to zip(x) or dict(y)?                                                  |
|                        |     | Answer: The * operator can be used to unpack a tuple or list.            |
|                        |     | Asked 11 years; Active 12 months ago; Viewed 234k times                 |
| For Multiple Targets   | (2) | Question: How to use the for loop to unpack a list?                      |
|                        |     | Answer: Use a for loop to unpack a list.                                |
|                        |     | Asked 9 years; Active 1 month ago; Viewed 222k times                    |

It has been generally accepted that pythonic idioms result in concise
code and improve runtime performance. However, non-idiomatic
code that can be refactored with pythonic idioms is widely present
in real world projects and is often mix-used with idiomatic code.
This could be caused by the fact that developers are often unaware
of pythonic idioms or do not know when and how to use pythonic
idioms properly. Although there are rich learning materials about
pythonic idioms, there are no effective tools to assist developers in
writing idiomatic code and to enforce the consistent use of pythonic
idioms in practice.

3 OUR APPROACH
We now present our refactoring tool for improving idiomatic cod-
ing practices. Figure 2 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 patterns or personal pro-
gramming experience, our solution is built on the effective analysis
of Python language syntax and specification.

3.1 Identifying Unique Pythonic Idioms
A programming idiom can be regarded as a micro-level code pattern.
Mining programming idioms in code [30, 31, 46, 56] may identify a
wide range of code patterns, including not only pythonic idioms but
also many generic code patterns and API usage patterns. As online
materials [29, 41, 44, 46, 54, 57] about idiomatic code usually men-
tion only some popular pythonic idioms (e.g., list-comprehension,
truth-value-test and enumerate function) repeatedly based on per-
sonal programming experience, collecting pythonic idioms from
online materials is ad-hoc and may miss important pythonic idioms.

https://stackoverflow.com/questions/26502775/simplify-chained-comparison
We observe Python syntax adds new elements (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) **Python syntax supports more comprehensive data type.** Python and Java have the same syntactic construct, but Python supports objects of more data types. This includes the truth-value-testing idiom [26, 44]. 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 == []”.

Table 1 lists the nine pythonic idioms we identify and the code examples of idiomatic code and corresponding non-idiomatic code. For each identified pythonic idiom, we read Python language specification to confirm its validity. We also search online materials with the idiom names to support our analysis. Searching with specific idiom names can find relevant supporting documentation. However, searching “python idiom” generally return many materials which do not cover all nine types of pythonic idioms.

### 3.2 Detecting Anti-idiom Code Smells

For each pythonic idiom, we define syntactic patterns for implementing the pythonic idioms in a non-idiomatic way. Contrasting Python and Java syntax provides the basis for defining such patterns. In a sense, we try to implement a pythonic idiom in a Java-style Python code pattern. The defined syntactic patterns can detect anti-idiom code smells that can be refactored with pythonic idioms. Table 5 lists our detection rules and illustrative examples.

#### 3.2.1 List/Set/Dict Comprehension

The list-comprehension idiom is used for the list initialization (2nd row in Table 5). The rule first finds an empty assignment statement `stmt1` (e.g., “dblist = []”). Then, it finds a for statement `stmt2` which iteratively adds elements to the target (“dblist”) of `stmt1`. There cannot be other statements using the target “dblist” of `stmt2` between `stmt1` and `stmt2` to lest the “dblist” is modified (i.e., `isNotUse(stmt1, target, stmt2, stmt4)`). Since the `stmt2` corresponds to the comp node of the ListComp construct which only supports for clause and if clause, the rule checks whether `stmt2` satisfies the MatchCompre condition, i.e., if the `stmt2` 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.

#### 3.2.2 Chain Comparison

The chain-comparison "a op1 b op2 c ... y opn z" is equivalent to "a op1 b and b op2 c and ... y opn z" [19]. 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 5), the cp is the common operand of the two compare nodes, and the expression can be refactored as “183983 >= cp >= 178208”.

#### 3.2.3 Truth Value Test

The truth-value-test idiom is used for checking the “truthness” 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.
Table 5: Examples of detection and refactoring of anti-idiom code smells

| Idiom | Detection Rules and Examples of Code Pairs | Refactoring Steps |
|-------|-------------------------------------------|------------------|
| List/ Set/ Dict Comprehension | $P = \{stmt_1, stmt_2\}$ where $stmt_1 = Assign \land stmt_2.value \in \{\{\}, set(), dict(),\}$ $\land$ NotUse(stmt_1.target, stmt_2, stmt_3, stmt_4) $\land$ stmt_5 = For $\land$ MatchComprehension(stmt_6) | 1. comp = Create("ListComp"/"SetComp"/"DictComp") 2. Traversing stmt_3, keep copying its children to comp 3. If stmt_1 and stmt_2 have the different parents then 4. Replace(stmt_2, stmt_3) 5. Replace(stmt_3, value, comp) 6. Else 7. Replace(stmt_3, value, comp) 8. Remove(stmt_1) |
| Chain Comparison | $P = p$ where $p.parent = test \land p = Compare \land Num(p.ops) = 1 \land \neg Op(op \in p.ops \land Op \in \{Eq, NotEq\} \land (p.left, p.comparators[0]) \cap EmptySet \neq \emptyset)$ | 1. Get empty_node which belongs to EmptySet from p.left, p.comparators[0]) 2. If p.ops is Eq then 3. node = Create("Not") 4. Copy(empty_node, Index(node.operand)) 5. Else 6. node = empty_node 7. Replace(P, node) |
| Truth Value Test | $P = [stmt_1, stmt_2, stmt_3]$ where $stmt_1 = Assign \land stmt_2 = For\lor While\lor diffSem(s, stmt_1)$ $\lor stmt_3 = Break \land stmt_1 = stmt_2.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_2, stmt_1, stmt_2, stmt_2, test) \land diffSem(s, stmt_1))$ $\lor stmt_3 = Break \land stmt_1 = stmt_1.body\to \exists s = Assign \land stmt_1.parent = s.parent$ $\lor OpposeSem(stmt_1, stmt_2, stmt_2, test) \land stmt_2, else \neq \theta \lor OpposeSem(stmt_2, stmt_1))$ $\lor stmt_3 = Break \land stmt_1 = stmt_1.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_1, stmt_1, stmt_2, test) \land diffSem(s, stmt_1, stmt_2, test, stmt_1)) \lor stmt_3 = Break \land stmt_1 = stmt_2.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_1, stmt_1, stmt_2, test) \land diffSem(s, stmt_1, stmt_2, test, stmt_1)) \lor stmt_3 = Break \land stmt_1 = stmt_2.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_1, stmt_1, stmt_2, test) \land diffSem(s, stmt_1, stmt_2, test, stmt_1)) \lor stmt_3 = Break \land stmt_1 = stmt_2.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_1, stmt_1, stmt_2, test) \land diffSem(s, stmt_1, stmt_2, test, stmt_1)) | 1. If c1 then 2. Copy(stmt_1, body, Index(stmt_2, orelse)) 3. If stmt_1, orelse is not None then 4. Copy(stmt_1, body, Index(stmt_3)) 5. If stmt_3, targets does not occur in other statements in detection rules then 6. Remove(s) 7. Remove(stmt_3) 8. Else If c2 then 9. Copy(stmt_1, orelse, Index(stmt_2, orelse)) 10. Copy(stmt_1, body, Index(stmt_3)) 11. If stmt_3, targets does not occur in other statements in detection rules then 12. Remove(s) 13. Remove(stmt_3) 14. Else If c3 then 15. Copy(stmt_1, body, Index(stmt_3)) 16. If stmt_1, orelse is not None then 17. Copy(stmt_1, orelse, Index(stmt_3)) 18. Else 19. Copy(stmt_1, orelse, Index(stmt_3)) 20. Copy(stmt_1, body, Index(stmt_3)) 21. Remove(stmt_3) |

| Loop Else | $P = [stmt_1, stmt_2, stmt_3]$ where $stmt_1 = Assign \land stmt_2 = For\lor While\lor diffSem(s, stmt_1)$ $\lor stmt_3 = Break \land stmt_1 = stmt_2.body\to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_2, stmt_1, stmt_2, stmt_2, test) \land diffSem(s, stmt_1))$ $\lor stmt_3 = Break \land stmt_1 = stmt_1.body\to \exists s = Assign \land stmt_1.parent = s.parent \lor OpposeSem(stmt_1, stmt_2, stmt_2, test) \land stmt_2, else \neq \theta \lor OpposeSem(stmt_2, stmt_1))$ $\lor stmt_3 = Break \land stmt_1 = stmt_1.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_1, stmt_1, stmt_2, test) \land diffSem(s, stmt_1, stmt_2, test, stmt_1)) \lor stmt_3 = Break \land stmt_1 = stmt_2.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_1, stmt_1, stmt_2, test) \land diffSem(s, stmt_1, stmt_2, test, stmt_1)) \lor stmt_3 = Break \land stmt_1 = stmt_2.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_1, stmt_1, stmt_2, test) \land diffSem(s, stmt_1, stmt_2, test, stmt_1)) \lor stmt_3 = Break \land stmt_1 = stmt_2.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_1, stmt_1, stmt_2, test) \land diffSem(s, stmt_1, stmt_2, test, stmt_1)) \lor stmt_3 = Break \land stmt_1 = stmt_2.body \to \exists s = Assign \land stmt_1.parent = s.parent \land SameSem(stmt_1, stmt_1, stmt_2, test) \land diffSem(s, stmt_1, stmt_2, test, stmt_1)) | 1. if $c_1$ then 2. Copy(stmt_1, body, Index(stmt_2, orelse)) 3. If stmt_1, orelse is not None then 4. Copy(stmt_1, body, Index(stmt_3)) 5. If stmt_3, targets does not occur in other statements in detection rules then 6. Remove(s) 7. Remove(stmt_3) 8. Else if $c_2$ then 9. Copy(stmt_1, orelse, Index(stmt_2, orelse)) 10. Copy(stmt_1, body, Index(stmt_3)) 11. If stmt_3, targets does not occur in other statements in detection rules then 12. Remove(s) 13. Remove(stmt_3) 14. Else if $c_3$ then 15. Copy(stmt_1, body, Index(stmt_3)) 16. If stmt_1, orelse is not None then 17. Copy(stmt_1, orelse, Index(stmt_3)) 18. Else 19. Copy(stmt_1, orelse, Index(stmt_3)) 20. Copy(stmt_1, body, Index(stmt_3)) 21. Remove(stmt_3) |
Table 5: Continued

| Idiom | Detection Rules and Examples of Code Pairs |
|-------|----------------------------------------|
| Assign Multiple Targets | $P = \{\text{stmt}_1, \ldots, \text{stmt}_n\}$ where $\forall \text{stmt}_i \ (n+1 > i > 0 \land \text{Num} (\text{stmt}_i, \text{targets}) = 1)$ and $\text{isDepend} (\text{stmt}_i, \text{stmt}_j) \rightarrow \text{stmt}_j (k > j > 1 \land \text{stmt}_j, \text{targets} = \text{stmt}_i, \text{value})$ |

**Refactoring Steps**
1. `value`-Create("Tuple")
2. Traversing $P$ to copy its elements to `value`
3. Replace(`stmt`, `value`, `value`)
4. `targets`-Create("Tuple")
5. Traversing $P$ to copy its elements to `targets`
6. Replace(`stmt`, `targets`, `targets`) for $i$ from 2 to $n$ do
7. Remove(`stmt`) |
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 $s$ contains the break statement $stmt_1$. 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 statement $stmt_1$ 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 $stmt_1$ is the opposite of the semantic of $stmt_1$ and the body of the if-statement $s$ contains the break statement $stmt_1$. The $c_4$ complements $c_3$, in the same vein as $c_2$ complements $c_1$.

### 3.2.5 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 $stmt_1$ does not use the result of an assignment statement $stmt_2$ before it, these assignment statements are refactorable to assign-multi-targets. When an assignment statement $stmt_1$ uses the result of the an assignment statement $stmt_2$ before $stmt_1$, the code usually is to swap variables by creating temporary variables. For such non-idiomatic code, it requires that the target of a statement $stmt_1$ between the $stmt_2$ and the $stmt_3$ is the same as the value of $stmt_2$. For example (third-to-last row in Table 5), $stmt_2$ “$d[e] = f$” uses the target “$f$” of $stmt_3$, “$f = d[0]$”, and the target “$d[0]$” of the $stmt_1$, “$d[0] = d[e]$” is the same as the value “$d[0]$” of the $stmt_1$, “$f = d[0]$”. This sequence of assignments is a temporary variable can also be refactored with the assign-multiple-targets idiom.

### 3.2.6 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 [16]. 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” (second-to-last row in Table 5). It can be refactored into “sys.argv[1:4:1]”.

### 3.2.7 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 5), 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 Refactoring with Pythonic Idioms

According to [37], a refactoring is a series of small behavior preserving transformations. Based on this principle, we analyze the AST transformations required to transform a piece of anti-idiom code into an idiomatic code. We identify four atomic AST-rewriting operations across all idioms, and then compose those atomic operations into the refactoring steps for each pythonic idiom. The four atomic operations are as follows:

1. **Copy** $(s, i)$ copies the node $s$ of 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, for the list-comprehension idiom (2nd row in Table 5), both the target node item and the iter node of list-comp of non-idiomatic code are copied to the corresponding target and iter position of the comprehension node respectively. For another example, for the chain-comparison idiom (3rd row in Table 5) we copy operands of compare node of non-idiomatic code into the position of operands of a new compare node.

2. **Create** $(s, *info)$ builds the node of type $s$ with information $*info$ where $*$ represents any amount of information. To refactor non-idiomatic code into pythonic idiom, it is sometimes necessary to create some new AST nodes or elements which do not have the corresponding parts in the non-idiomatic code. For example, for the truth-value-test idiom (4th row in Table 5), we need to create a “Not” node. For another example, for the star-in-function-call idiom (second-to-last row in Table 5), we need to create a Starred node with subscript information from the non-idiomatic code.

3. **Remove** $(s)$ 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 those no-longer-used nodes. For example, for the loop-else idiom (5th row in Table 5), we need to remove the initial flag assignment “good_partition = True” and the flag-update statement “good_partition = False” which are no longer needed when the loop-else idiom is used. For another example, for the assign-multi-targets idiom (6th row in Table 5), we remove assign statements from $stmt_2$ to $stmt_n$.

4. **Replace** $(s, t)$ replaces the node $s$ of non-idiomatic with the node $t$ obtained through code transformation. For example, for the chain-comparison idiom (3rd row in Table 5), we replace the original expression “$cp >= 178208$ and $cp <= 183983$” with the resulting chain-comparison “$183983 >= cp <= 178208$”. For another example, for the for-multi-targets idiom (the last row in Table 5), we replace “interval[0], interval[1]” with “interval_0, interval_1” respectively.

The 3rd column of Table 5 shows the refactoring steps to complete each pythonic idiom refactoring. The green line numbers shows the steps that are performed to refactor the examples of non-idiomatic code on the left into the idiomatic code on the right in the 2nd column of Table 5. For example, to refactor the non-idiomatic code example into a list comprehension code (2nd row in Table 5), we first create a ListComp node $comp$ and then traverse the for statement $stmt_n$ to copy its children to the $comp$ node (line 1-2), e.g., copy 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 \texttt{stmt1.value} with \texttt{comp} in and then remove \texttt{stmt1} (line 7-8). Finally, the new \texttt{stmt1} is the idiomatic code obtained through the refactoring. When \texttt{stmt1} and \texttt{stmt2} are at different scope (line 3), we do not perform the Remove operation for the \texttt{stmt1} because \texttt{stmt2} may not be executed after executing \texttt{stmt1}, so we only replace \texttt{stmt2} with \texttt{stmt1} and then update the value of \texttt{stmt2} (line 4-5).

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?

4.1 RQ1: Correctness of Refactorings

4.1.1 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.

4.1.2 Method. As described in Section 2.2.1, 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.

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 [61]. 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 [42], a popular Python unit testing framework which also supports the Python’s default unittest tool [27]. 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 refactoring, 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 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.

4.1.3 Result. Table 6 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 refactoring outcomes 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 non-refactorable non-idiomatic code fragments, 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, \texttt{stmt1: lib=} and \texttt{stmt2: tmpl=["lib"]}, “lib” of \texttt{stmt2} is a variable because Python uses curly brackets to insert variables in string. \texttt{stmt1} uses the target “lib” of \texttt{stmt1}, so

| Idiom         | Testing | Code Review |
|---------------|---------|-------------|
|               | #Refs   | #TCs d-acc r-acc | #Refs d-acc r-acc |
| List-Compre   | 132     | 391 1 1   | 100 1 1              |
| Set-Compre    | 21      | 39 1 1   | 100 1 1              |
| Dict-Compre   | 102     | 297 1 1   | 100 1 1              |
| Chain-Compa   | 309     | 837 1 1   | 100 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      |
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 lose the information of data dependency and mistakenly identify the statements as refactorable.

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_circleArc(p[-1], p[0], p[1])", `p[-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 the resulting code encounters the runtime error.

**Performance.** 3 reject responses are concerned about the performance or memory usage. For example, the developers reject the pull request because they are not sure that the performance improvement would be significant in their project. For the list-comprehension refactoring which refactors "ss[0], ss[1], ss[2], ss[3], ss[4], ss[5], ss[6]" into "ss[0:7:1]", the developer worries that refactoring may lose 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(...)".

**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 lose 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(...)".

**Systematic refactoring.** 3 reject responses indicate 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.”

Our pythonic idiom refactorings have been well received by the developers. The developers also raise concerns about readability and performance of pythonic idioms which deserve further study.

5 RELATED WORK

5.1 Pythonic Idioms

Current studies mainly mine Python idioms (commonly referred to as more pythonic [46, 60]) by searching for Python books or online resources. Alexandru et al. [29] identify 19 pythonic idioms (list-comprehension and dict-comprehension overlap with our work) from several books. They manually classify the idioms based on performance and readability. Phan-udom et al. [51] collect 10 Python idioms from presentations given by renowned Python developers [44]. They develop a tool to recommend pythonic code examples by searching similar code examples from 113 code snippets of three repositories for the ten pythonic idioms. The Python idioms collected by previous work covers language syntax, magic methods, lambda functions and API calls, which are not unique to Python. For example, the finally, assert, lambda functions are also common in other programming languages such as Java. Different from previous works, our work identifies nine unique pythonic idioms by comparing the syntax of Java and Python, and find 4 idioms (star-in-func-call, for-multi-targets, assign-multi-targets, loop-else) that have not been identified before. Furthermore, we also develop an automatic tool to refactor anti-idiom code into idiomatic code.

5.2 Code refactoring

Martin Fowler proposes code refactoring [37] about 30 years ago. An active line of research is to recover refactorings made in the code [33, 34, 40, 43, 48, 52, 55, 58, 62, 63]. Prete et al. [43, 52] detect the largest number of refactoring types based on the Fowler’s catalog. Tsantalis et al. [58] identify 15 Java refactoring types by statement matching algorithm and refactoring detection rules. Dilhara et al. [35] identify 18 Python refactoring kinds (e.g., rename, move, pull up methods), which do not overlap with our refactoring type. Another active line of research is to refactor code [1, 32, 38, 45, 47, 49, 53, 59]. Pylint [1] is a static code analysis tool, which could give refactoring suggestions for chain-comparison and truth-value-test but does not support automatic refactorings. To the best of our knowledge, our tool is the first automatic refactoring tool which covers 9 types of pythonic idioms.

6 DISCUSSION

6.1 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. 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.

6.2 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 [39, 50, 64]. We release our tool and data in GitHub1 for public evaluation.

7 CONCLUSION AND FUTURE WORK

This paper designs and implements the first automatic refactoring tool for nine 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, defines 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.

ACKNOWLEDGMENT

This research was partially funded by Data61-ANU Collaborative Research Project CO19314.

1https://github.com/idiomaticrefactoring/pythonidiomsrefactor
[64] Neng Zhang, Qiao Huang, Xin Xia, Ying Zou, David Lo, and Zhenchang Xing. 2020. Chatbot4qr: Interactive query refinement for technical question retrieval. *IEEE Transactions on Software Engineering* (2020).